SC20: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-7281-9998-6/20/$31.00 ©2020 IEEE | DOI: 10.1109/SC41405.2020.00078

# Distributed-Memory Parallel Symmetric Nonnegative Matrix Factorization

Srinivas Eswar*, Koby Hayashi*, Grey Ballard†, Ramakrishnan Kannan‡, Richard Vuduc*, and Haesun Park*
*Dept. of Computational Science and Engineering,
Georgia Institute of Technology, Atlanta, USA
Emails: {seswar3,khayashi9}@gatech.edu, {richie,hpark}@cc.gatech.edu
†Dept. of Computer Science,
Wake Forest University, Winston-Salem, USA
Email: ballard@wfu.edu
‡Computational Data Analytics Group
Oak Ridge National Laboratory, Oak Ridge, USA
Email: kannanr@ornl.gov

*Abstract*—**We develop the first distributed-memory parallel implementation of Symmetric Nonnegative Matrix Factorization (SymNMF), a key data analytics kernel for clustering and dimensionality reduction. Our implementation includes two different algorithms for SymNMF, which give comparable results in terms of time and accuracy. The first algorithm is a parallelization of an existing sequential approach that uses solvers for nonsymmetric NMF. The second algorithm is a novel approach based on the Gauss-Newton method. It exploits second-order information without incurring large computational and memory costs. We evaluate the scalability of our algorithms on the Summit system at Oak Ridge National Laboratory, scaling up to 128 nodes (4,096 cores) with 70% efficiency. Additionally, we demonstrate our software on an image segmentation task.**

*Index Terms*—**High performance computing, Newton method, Parallel algorithms, Symmetric Matrices**

## I. Introduction

This paper concerns the *symmetric nonnegative matrix factorization* (SymNMF) problem, which arises in unsupervised learning and data mining applications [1]–[5]. In this problem, the input data is naturally represented by a symmetric, nonnegative data matrix $\mathbf{A} \in \mathbb{R}_+^{n \times n}$ where $\mathbf{A} = \mathbf{A}^\mathsf{T}$ and $\mathbf{A} \geqslant 0$. For instance, in graph clustering $\mathbf{A}$ might be an unweighted adjacency matrix; in image segmentation $\mathbf{A}$ represents pixel-to-pixel positive similarity scores [6]; and in topic modeling $\mathbf{A}$ stores normalized co-occurrence counts [7]. Data analysis tasks such as dimension reduction, clustering, embedding, or imputation of missing values, may be formulated as low-rank matrix factorization problems. That is, given $\mathbf{A}$ and an integer parameter, $k \ll n$, SymNMF seeks a nonnegative rank-$k$ matrix $\mathbf{H} \in \mathbb{R}_+^{n \times k}$ such that $\mathbf{A} \approx \mathbf{H}\mathbf{H}^\mathsf{T}$ and $\mathbf{H} \geqslant 0$. Preserving nonnegativity in $\mathbf{H}$ helps the end-user analyst interpret the results more readily.

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

Sequential algorithms to compute SymNMF are summarized by Kuang et al. [1]. Following methods for the nonsymmetric NMF case [8], [9], it formulates SymNMF as the optimization problem,

$$\min_{\mathbf{H} \geq 0} \|\mathbf{A} - \mathbf{H}\mathbf{H}^\mathsf{T}\|_F^2. \tag{1}$$

Equation (1) is nonlinear and nonconvex, which makes it hard to solve to optimality.

Solution algorithms may be broadly classified into two groups: block coordinate descent and direct optimization. Block coordinate descent algorithms, including Alternating Nonnegative Least Squares (ANLS) [1] and Cyclic Coordinate Descent (CCD) [2], partition the solution variables into blocks and alternate among solving subproblems with all but one block of a variables fixed, each of which can be solved exactly. ANLS duplicates the solution matrix $\mathbf{H}$, solves the problem as a nonsymmetric problem, and uses regularization to converge to a unified symmetric solution (discussion in Section III). Direct optimization techniques such as Projected Gradient Descent (PGD) and Newton-like algorithms [1] iteratively update all the variables at once. First-order direct optimization uses only gradient information (PGD) and can suffer slow convergence, while second-order methods incorporate Hessian information to improve convergence at the price of higher per-iteration costs. Previously developed second-order methods suffer from a particularly high computational complexity of $O(n^3k^3)$ per iteration, making them infeasible for even moderate data sizes. Regarding solution quality, neither method is clearly superior to the other and both are expensive to compute sequentially even at moderate input sizes [1]. As such, we are motivated to consider both classes of methods and to improve both scalability via parallelism and work-efficiency via algorithmic techniques.

A critical advantage of parallelizing the ANLS variant of SymNMF is that one can reuse existing algorithms and software developed for the nonsymmetric NMF case.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.

Our approach is to adopt the communication-optimal ANLS method due to Kannan et al. [10]. We present our algorithm in section III. It enjoys the same advantages as the nonsymmetric method, and is the first distributed-memory parallel algorithm for SymNMF.

Our second algorithm is a novel Newton-like algorithm for SymNMF based on the Gauss-Newton method (section IV). It efficiently uses second-order information *implicitly* without incurring prohibitive memory or computational costs. The proposed Gauss-Newton method for SymNMF finds low-rank approximations that are competitive with existing algorithms in quality (see § V-D). Surprisingly, our parallel Gauss-Newton method using Conjugate Gradient (GNCG) can even be twice as fast as the ANLS variant in practice.

We evaluate these methods on the Summit supercomputer[1] using a variety of datasets (section V). We are able to scale GNCG up to 128 nodes (4,096 cores) with 70% efficiency. In general, GNCG runs $2\times$ faster than ANLS for most problems where gradient computation is the major bottleneck (see § V-E). In cases with extremely large $k$ ANLS becomes competitive again (§ V-F).

Overall, the end results constitute the first distributed-memory parallel methods for the SymNMF problem. The significance is to enable practical use of SymNMF on real-world problems. As an example, we present a case study in which we apply SymNMF to the segmentation of large satellite images [11]. The pixel embeddings produced by SymNMF are used to detect boundaries and partition images into regions [1], [6].

## II. Preliminaries

### A. Notation

Scalars will be denoted as lowercase characters (e.g. $x$), vectors as lowercase boldface characters (e.g. $\mathbf{x}$), and matrices as uppercase boldface characters (e.g. $\mathbf{X}$). A specific element of a matrix $\mathbf{X}$ is denoted $x_{ij}$. To denote submatrices, we use subscripts with an uppercase boldface character: $\mathbf{X}_{ij}$ denotes the $(i,j)$th submatrix of $\mathbf{X}$. The comparison $\mathbf{X} \geq 0$ is interpreted element-wise. A vector $\mathbf{x}$ and a matrix $\mathbf{X}$ that change during the course of an iterative algorithm will be distinguished using bracketed superscripts. For example at iteration $t$ these variables are $\mathbf{x}^{(t)}$ and $\mathbf{X}^{(t)}$.

The matrix $\mathbf{A}$ will refer to a square, nonnegative $n \times n$ data matrix, and we reserve $\mathbf{H}$ to represent the factor matrix of the symmetric approximation. We use $[\mathbf{X}]_+$ to denote projection to the nonnegative orthant.

### B. Parallel Nonsymmetric NMF

We build our implementations on top of an existing open-source library designed for nonsymmetric NMF called MPI-FAUN [10]. MPI-FAUN is written in C++, uses MPI for the interprocessor communication, and uses Armadillo for local matrix operations (interfacing to a BLAS implementation for dense matrix operations). It is designed to solve the problem

$$\min_{\{\mathbf{W},\mathbf{H}\}\geq 0} \|\mathbf{A} - \mathbf{W}\mathbf{H}^{\mathsf{T}}\|_F^2 \tag{2}$$

[1]The system hosted at Oak Ridge National Laboratory.

for dense or sparse, rectangular, nonnegative matrices $\mathbf{A}$ using algorithms that alternate between updating $\mathbf{W}$ for fixed $\mathbf{H}$ and updating $\mathbf{H}$ for fixed $\mathbf{W}$. The available updating algorithms (which can be specified by the user) include block principal pivoting, which is what we use exclusively. Block principal pivoting is an active-set-like method for nonnegative least squares problems that solves them exactly [12].

MPI-FAUN uses a 2D distribution of $\mathbf{A}$ over a grid of processors whose dimensions are specified by the user. Since the matrices considered in this work are symmetric (square), we use a square grid of processors in our algorithms. The data distribution of the factor matrices is 1D, so that each processor owns a subset of the rows of each tall-skinny factor, and the 1D distribution is forced to be conformal to the distribution of the data matrix. We discuss these distributions in more detail in Section III, where we describe how the factor matrix distributions must be tailored for the symmetric case.

### C. Gauss-Newton Method using Conjugate Gradient

The Gauss-Newton (GN) method is a technique for minimizing a sum of squares of residual functions. That is, it can be used to solve optimization problems involving multivariate functions of the form $f(\mathbf{x}) = \sum_l r_l(\mathbf{x})^2$. Here the functions $r_l(\mathbf{x})$ are called the residual functions and are generally nonlinear. GN proceeds to minimize $f(\mathbf{x})$ by starting with an initial guess $\mathbf{x}^{(0)}$ and following the iteration:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} + \operatorname*{argmin}_{\mathbf{p}} \left\| \mathbf{J}^{(t)}\mathbf{p} + \mathbf{r}^{(t)} \right\|_2^2. \tag{3}$$

Here $\mathbf{J}^{(t)}$ is the Jacobian matrix defined as $\mathbf{J}_{lq}^{(t)} = \frac{\partial r_l}{\partial \mathbf{x}_q}(\mathbf{x}^{(t)})$ and $\mathbf{r}^{(t)}$ is a vector of the residual function values $r_l(\mathbf{x}^{(t)})$. This iteration is performed until some stopping criteria is met. We solve the linear least squares problem via the normal equations. The GN method is a Quasi-Newton or second-order optimization method where the matrix $\mathbf{J}^{\mathsf{T}}\mathbf{J}$ acts as an approximate Hessian matrix for $f$ [13].

Thus, the main task at each iteration is to find a solution to the linear system:

$$\left(\mathbf{J}^{(t)\mathsf{T}}\mathbf{J}^{(t)}\right)\mathbf{p} = -\mathbf{J}^{(t)\mathsf{T}}\mathbf{r}^{(t)}. \tag{4}$$

For this one can use an iterative method for symmetric positive semi-definite matrices such as the Conjugate Gradient method (CG) [14]. The right-hand-side vector $\mathbf{J}^{(t)\mathsf{T}}\mathbf{r}^{(t)} = \mathbf{g}^{(t)}$ is the gradient evaluated at $\mathbf{x}^{(t)}$. The CG algorithm maintains for each iteration an approximate solution vector, a search/update direction vector, and a residual vector, and it requires the multiplication of the coefficient matrix with an approximate solution vector as well as several vector operations. While the size of the Gramian of the Jacobian is large, CG iterations can be performed efficiently if the matrix-vector multiplication can exploit structure in the Jacobian and avoid the explicit formation of the Gramian.

For constrained optimization problems, the Gauss-Newton step is not guaranteed to maintain the constraint satisfaction. The updated solution can be projected back onto the feasible set. We apply the projected version of this method and show how to implement it efficiently for Symmetric NMF in § IV-A.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.

### D. Related Work

SymNMF has been extensively studied and a number of effective sequential algorithms have been proposed for solving Equation (1). Vandaele et al. [2] propose an algorithm using a cyclic coordinate descent (CCD) in which elements of the matrix $\mathbf{H}$ are iteratively updated in a cyclic fashion. This algorithm is not readily parallelizable because of its dependencies among elements. Kuang et al., [1], [15], present three algorithms for computing SymNMF: 1) a method based on projected gradient descent (PGD), 2) a Newton-type method which utilizes second-order information, and 3) a regularized block coordinate descent method based on alternating nonnegative least squares (ANLS). The authors of [1] focus on the Newton-type and ANLS algorithms as they both tend to outperform the PGD algorithm. They conclude that the ANLS algorithm performs most effectively in terms of quality of solution and run time. We parallelize the ANLS algorithm in this work, as described in Section III. Though we do not develop distributed memory parallel algorithms for the PGD algorithm from [1] or the CCD algorithm from [2], we provide convergence experiments from sequential implementations in § V-D to compare these algorithms.

The Gauss-Newton method has been used before in the context of low-rank approximation. For example, Gauss-Newton methods are effective for the computation of the Canonical Polyadic (CP) tensor decomposition. Vervliet and de Lathauwer [16] detail how GN with CG can be used to efficiently compute the CP decomposition. In the context of high performance computing, Singh et al. [17] compare the scalability of the Alternating Least Squares (ALS) and GN algorithms for computing a CP decomposition. The authors demonstrate efficient weak scaling results for both algorithms but less compelling strong scaling results, particularly for the GN algorithm. The authors attribute this to the fact that the ALS algorithm contains more easily parallelizable computations and is more dominated by computation.

## III. Nonsymmetric Alternating-Updating NMF with Symmetric Regularization

### A. SymNMF via Alternating Nonnegative Least Squares

One approach we consider for solving Equation (1) is to compute a nonsymmetric NMF with a regularization term that drives the two nonsymmetric factors towards each other to encourage convergence to a symmetric solution. As proposed by Kuang et al. [1], we can use the following surrogate optimization problem:

$$\min_{\{\mathbf{W},\mathbf{H}\}\geq 0} \|\mathbf{A}-\mathbf{W}\mathbf{H}^{\mathsf{T}}\|_F^2 + \gamma\|\mathbf{W}-\mathbf{H}\|_F^2. \tag{5}$$

The symmetry constraint is dropped and in its place we add the regularizer $\gamma\|\mathbf{W}-\mathbf{H}\|$, where $\gamma \geq 0$. Note that if $\gamma = 0$ Equation (2) is recovered. This encourages the algorithm to find a solution such that $\mathbf{W} \approx \mathbf{H}$.

In this approach, we can use existing solvers for nonsymmetric NMF, including those available in MPI-FAUN.

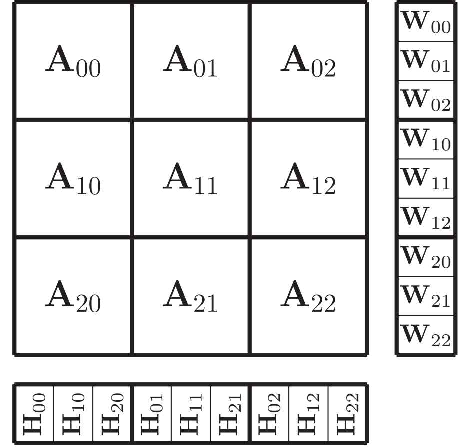



Fig. 1. Nonsymmetric data distribution for a $3\times 3$ processor grid [10].

For alternating-updating algorithms, the subproblem for updating $\mathbf{H}$ is a nonnegative least squares problem of the form

$$\min_{\mathbf{H}\geqslant 0} \left\| \begin{bmatrix} \mathbf{W} \\ \sqrt{\gamma}\mathbf{I}_k \end{bmatrix} \mathbf{H}^{\mathsf{T}} - \begin{bmatrix} \mathbf{A} \\ \sqrt{\gamma}\mathbf{W}^{\mathsf{T}} \end{bmatrix} \right\|_F^2,$$

and the subproblem for updating $\mathbf{W}$ is similar. Following Kuang et al. [1], we will refer to this approach as Alternating Nonnegative Least Squares (ANLS).

The bulk of the computation for any alternating-updating algorithm is the computation of the matrices involved in the gradients: $\mathbf{W}^{\mathsf{T}}\mathbf{W}+\gamma\mathbf{I}_k$ and $\mathbf{A}\mathbf{W}+\gamma\mathbf{W}$ (for updating $\mathbf{H}$) and $\mathbf{H}^{\mathsf{T}}\mathbf{H}+\gamma\mathbf{I}_k$ and $\mathbf{A}\mathbf{H}+\gamma\mathbf{H}$ (for updating $\mathbf{W}$), assuming $\mathbf{A}$ is symmetric, which are the matrices appearing in the gradients of the subproblems. More detailed analysis is given in § III-D.

### B. Parallel Nonsymmetric ANLS Algorithm

We parallelize the ANLS method using the MPI-FAUN software framework [10]. For completeness, we describe the parallel matrix multiplication algorithms used for the nonsymmetric case in this section and then describe how we modify them for the symmetric case in § III-C. Figure 1 illustrates the data distribution of the data and factor matrices for a square matrix and a $3\times 3$ processor grid. The matrices are oriented to emphasize the conformal distributions of $\mathbf{W}$ to processor rows and $\mathbf{H}$ to processor columns, which is designed to support a particular parallel algorithm for matrix multiplication.

In the nonsymmetric case, the matrix products involving the data matrix are $\mathbf{W}^{\mathsf{T}}\mathbf{A}$ and $\mathbf{A}\mathbf{H}$. The parallel matrix multiplication algorithm used in MPI-FAUN does not communicate any data matrix elements, so it must communicate factor matrix elements. Consider processor 01: to compute its contribution to $\mathbf{W}^{\mathsf{T}}\mathbf{A}$ (i.e., any multiplications involving $\mathbf{A}_{01}$) it must access submatrices $\mathbf{W}_{00}$, $\mathbf{W}_{01}$, and $\mathbf{W}_{02}$, only one of which it owns. Note that processors 00 and 02 (other processors in the processor row) also need those same submatrices, so they perform an all-gather collective communication operation to receive all the data they need. All processor rows independently perform similar all-gathers. After computing the local contribution, processor 01 must sum

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.

its results with processors 11 and 21 (other processors in the processor column). Each column of the result $\mathbf{W}^\mathsf{T}\mathbf{A}$ will be used to update the corresponding row of $\mathbf{H}$, so the processor column performs a reduce-scatter collective communication operation to simultaneously sum the local results and distribute them to match the distribution of $\mathbf{H}$ (note in Figure 1 that the middle 3 blocks of $\mathbf{H}$ are owned by the processors in the middle processor column). All processor columns independently perform similar reduce-scatters. Computing $\mathbf{AH}$ works the other way: all-gathers are performed over processor columns and reduce-scatters are performed over processor rows.

With this factor matrix distribution, the Gramian matrices $\mathbf{W}^\mathsf{T}\mathbf{W}$ and $\mathbf{H}^\mathsf{T}\mathbf{H}$ are computed by local computation followed by an all-reduce collective communication operation over all processors. At the end of the collective, all processors own a copy of the Gramian matrix.

### C. Parallel ANLS

We now describe how to adapt the nonsymmetric parallel ANLS algorithm to the symmetric case. We use the same 2D distribution of the data matrix, always with a square $\sqrt{p}\times\sqrt{p}$ processor grid, storing both upper and lower triangles of the matrix explicitly. We also use the same 1D distribution of the factor matrices, so the parallel algorithm for computing Gramian matrices does not change.

The difference arises in the computation of $\mathbf{AW}+\gamma\mathbf{W}$ and $\mathbf{AH}+\gamma\mathbf{H}$. In particular, we note that the distributions of $\mathbf{W}$ and $\mathbf{H}$ are not identical. As shown in Figure 1, the second block of $\mathbf{W}$ is owned by processor 01, while the second block of $\mathbf{H}$ is owned by processor 10. The nonsymmetric matrix multiplication algorithm is designed so that $\mathbf{AW}$ has the same distribution of $\mathbf{H}$. In the symmetric case, to incorporate the regularization, we must add $\mathbf{AW}$ to $\gamma\mathbf{W}$, but these matrices are not identically distributed and so their addition requires communication. Because the result will be used to update $\mathbf{H}$, and the distribution of $\mathbf{AW}$ matches $\mathbf{H}$, we communicate the necessary block of $\mathbf{W}$ to complete the addition. This communication can be performed via pairwise exchanges between symmetric partners (processors $ij$ and $ji$ for $i\neq j$), as depicted in Figure 2. A similar technique is used to communicate $\mathbf{H}$ for the matrix addition with $\mathbf{AH}$ when updating $\mathbf{W}$.

The rest of the nonsymmetric NMF algorithm can be applied directly: multiple algorithms can be used to solve the local nonnegative least squares problem. After convergence, either factor or their average may be used as the symmetric result; to average $\mathbf{W}$ and $\mathbf{H}$ requires using a temporary local copy of the matrix that was communicated for the final update step. We present the ANLS algorithm with symmetric regularization in Algorithm 1, which is adapted from [10, Alg. 3].

### D. Analysis

The computation and communication costs of ANLS for symmetric NMF are nearly identical to the nonsymmetric case [10]. The dominant computation costs are due to multiplications between the data matrix and each factor matrix, computing Gramian matrices of the factor matrices, and evaluating the local Update function. Computing the products

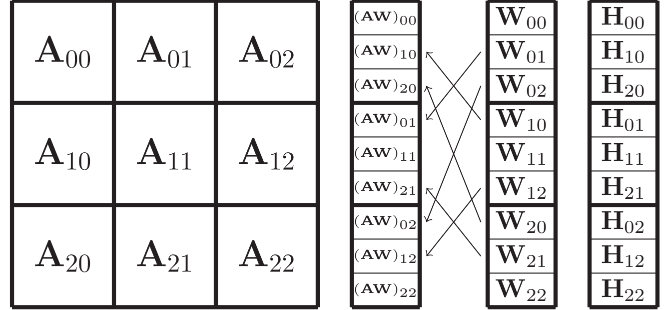



Fig. 2. Data distribution of symmetric ANLS and communication pattern of $\mathbf{W}$ to compute $\mathbf{AW}+\gamma\mathbf{W}$ for a $3\times 3$ processor grid

involving the matrix $\mathbf{A}$ costs $4n^2k/p$ when $\mathbf{A}$ is dense and $4k\text{nnz}(\mathbf{A})/p$ when $\mathbf{A}$ is sparse, the Gramian matrix computations cost $O(nk^2/p)$ arithmetic operations, and the matrix additions are $O(nk/p)$. We use the block principal pivoting method for the local update, which can take $k!$ iterations each costing $O(k^3+nk^2/p)$ but in practice takes much fewer [12].

We analyze the communication costs assuming the use of efficiently implemented collective communication operations [18], [19]. The size of the data involved in the all-gather and reduce-scatters is $O(nk/\sqrt{p})$, so the costs are $O(nk/\sqrt{p})$ words and $O(\log p)$ messages. The extra cost of the pairwise exchange does not affect the leading order communication costs of the overall algorithm, because the size of the messages is $O(nk/p)$. The communication cost of the Gram all-reduces is $O(\log p)$ messages of size $O(k^2)$ . Additionally, the algorithm never communicates the data matrix so these communication costs are the same for both the dense and sparse cases. We compare the costs of GNCG with ANLS (for the dense case) in Table I.

## IV. Gauss Newton based Distributed Symmetric Nonnegative Matrix Factorization

### A. Gauss-Newton for Symmetric NMF

Here we derive the Gauss-Newton method for solving the Symmetric NMF objective function in Equation (1) and show how to employ CG efficiently for each GN step. Since we are using the Frobenius norm, our Symmetric NMF objective function is a sum of squares. The GN method is described for general objective functions in § II-C. In this case, the residual functions are of the form

$$r_{ij}(\mathbf{H})=a_{ij}-\sum_{\ell=1}^{k}h_{i\ell}h_{j\ell}.$$

Note that we use two indices for the residual functions because they correspond to matrix entries, though they must be vectorized before corresponding to rows of the Jacobian. Similarly, we will use two indices to index into the solution vector $\mathbf{x}$, which is the matrix $\mathbf{H}$ in this case, though the indices must be vectorized before corresponding to columns of the Jacobian. If the input matrix $\mathbf{A}$ is $n\times n$ and the desired low-rank factor

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.

**Algorithm 1** $[\mathbf{W},\mathbf{H}] = \text{SymANLS}(\mathbf{A},k,\gamma)$

**Require:** $\mathbf{A} \in \mathbb{R}_+^{n\times n}$ is distributed across a $\sqrt{p} \times \sqrt{p}$ grid of processors, $k>0$ is rank of approximation, $p$ divides $n$

**Require:** Local matrices: $\mathbf{A}_{ij}$ is $n/\sqrt{p}\times n/\sqrt{p}$, $\mathbf{W}_i$ and $\mathbf{H}_j$ are $n/\sqrt{p}\times k$, $\mathbf{W}_{ij}$ and $\mathbf{H}_{ij}$ are $n/p\times k$

1: Proc $p_{ij}$ initializes $\mathbf{H}_{ij}$
2: **while** stopping criteria not satisfied **do**
  **/* Compute W given H */**
3: $p_{ij}$ computes $\mathbf{U}_{ij} = \mathbf{H}_{ij}^{\mathsf{T}}\mathbf{H}_{ij}$
4: compute $\mathbf{H}^{\mathsf{T}}\mathbf{H}=\sum_{i,j}\mathbf{U}_{ij}$ using all-reduce across all procs
5: $p_{ij}$ collects $\mathbf{H}_j$ using all-gather across proc columns
6: $p_{ij}$ computes $\mathbf{V}_{ij} = \mathbf{A}_{ij}\mathbf{H}_j$
7: compute $(\mathbf{AH})_i=\sum_j \mathbf{V}_{ij}$ using reduce-scatter across proc row to achieve row-wise distribution of $(\mathbf{AH})_i$
8: $p_{ij}$ sends $\mathbf{H}_{ij}$ to $p_{ji}$ and receives $\mathbf{H}_{ji}$ from $p_{ji}$
9: $p_{ij}$ computes $\mathbf{W}_{ij}^{\mathsf{T}} = \text{Update}(\mathbf{H}^{\mathsf{T}}\mathbf{H}+\gamma\mathbf{I},(\mathbf{AH})_{ij}+\gamma\mathbf{H}_{ji})$
  **/* Compute H given W */**
10: $p_{ij}$ computes $\mathbf{X}_{ij} = {\mathbf{W}_{ij}}^T\mathbf{W}_{ij}$
11: compute $\mathbf{W}^T\mathbf{W}=\sum_{i,j}\mathbf{X}_{ij}$ using all-reduce across all procs
12: $p_{ij}$ collects $\mathbf{W}_i$ using all-gather across proc rows
13: $p_{ij}$ computes $\mathbf{Y}_{ij} = \mathbf{W}_i^{\mathsf{T}}\mathbf{A}_{ij}$
14: compute $(\mathbf{AW})_j = \sum_i \mathbf{Y}_{ij}$ using reduce-scatter across proc columns to achieve row-wise distribution of $(\mathbf{AW})_j$
15: $p_{ij}$ sends $\mathbf{W}_{ij}$ to $p_{ji}$ and receives $\mathbf{W}_{ji}$ from $p_{ji}$
16: $p_{ij}$ computes $\mathbf{H}_{ij}^{\mathsf{T}} = \text{Update}(\mathbf{W}^T\mathbf{W}+\gamma\mathbf{I},(\mathbf{AW})_{ij}+\gamma\mathbf{W}_{ji})$
17: **end while**

**Ensure:** $\mathbf{W},\mathbf{H} \approx \underset{\tilde{\mathbf{W}},\tilde{\mathbf{H}}\geqslant 0}{\text{argmin}} \|\mathbf{A}-\tilde{\mathbf{W}}\tilde{\mathbf{H}}^{\mathsf{T}}\|_F^2 + \gamma\|\tilde{\mathbf{W}}-\tilde{\mathbf{H}}\|_F^2$

**Ensure:** $\mathbf{W},\mathbf{H}$ are $n\times k$ row-wise distributed across processors

is $n\times k$, then the Jacobian matrix, $\mathbf{J}$, will be of size $n^2\times nk$. We note that residuals $r_{ij}\equiv r_{ji}$ due to the symmetry of $\mathbf{A}$.

In order to uncover the structure of the Jacobian matrix, we first consider the individual residuals $r_{ij}(\mathbf{H})$. The partial derivatives needed to define the elements of the Jacobian are $\frac{\partial r_{ij}}{\partial h_{i'\ell}}$. Observing that this partial derivative is only nonzero for elements $h_{i'l}$ that lie in the $i$th and $j$th rows of $\mathbf{H}$, we end up with 4 cases:

$$\frac{\partial r_{ij}}{\partial h_{i'\ell}} = \begin{cases} -2h_{i\ell} & \text{if } i'=i=j \\ -h_{j\ell} & \text{if } i'=i\neq j \\ -h_{i\ell} & \text{if } i'=j\neq i \\ 0 & \text{else} \end{cases}. \tag{6}$$

With this observation we see that there are at most $2k$ nonzero values per row of the Jacobian.

We use the convention of row-wise vectorizations of the matrices $\mathbf{A}$ and $\mathbf{H}$ to correspond to rows and columns of the Jacobian. Using these conventions the Jacobian can be written as

$$\mathbf{J} = -(\mathbf{H}\otimes\mathbf{I}_n) - \mathbf{P}_{n,n}(\mathbf{H}\otimes\mathbf{I}_n), \tag{7}$$

where $\mathbf{P}_{n,n}$ is the perfect shuffle or "vec" permutation matrix, defined so that $\text{vec}(\mathbf{Y}) = \mathbf{P}_{n,n}\text{vec}(\mathbf{Y}^{\mathsf{T}})$ for an $n\times n$ matrix $\mathbf{Y}$ [20]. To verify this matrix expression for the Jacobian, we use the notation $(ij)$ to denote the row-wise linearization of indices $i$ and $j$, so that $(ij) = in+j$ when $i,j$ range from 0 to $n-1$. Then by eq. (7),

$$\begin{aligned}\mathbf{J}_{(ij),(i'\ell)} &= -(\mathbf{H}\otimes\mathbf{I}_n)_{(ij),(i'\ell)} - [\mathbf{P}_{n,n}(\mathbf{H}\otimes\mathbf{I}_n)]_{(ij),(i'\ell)} \\ &= -(\mathbf{H}\otimes\mathbf{I}_n)_{(ij),(i'\ell)} - (\mathbf{H}\otimes\mathbf{I}_n)_{(ji),(i'\ell)} \\ &= -h_{jl}\delta_{ii'} - h_{il}\delta_{ji'},\end{aligned}$$

where $\delta_{ij}$ is the Kronecker delta function, matching eq. (6).

*1) Gradient computation:* In order to take a GN step, we must solve a linear system with coefficient matrix $\mathbf{J}^{\mathsf{T}}\mathbf{J}$ and right-hand-side $\mathbf{J}^{\mathsf{T}}\mathbf{r}$. The vectorized residual of our problem is $\mathbf{r} = \text{vec}(\mathbf{A}-\mathbf{H}\mathbf{H}^{\mathsf{T}})$. The right-hand-side vector (gradient, see § II-C) $\mathbf{g} = \mathbf{J}^{\mathsf{T}}\mathbf{r}$ can be efficiently formed via

$$\begin{aligned}\mathbf{g} &= -((\mathbf{H}\otimes\mathbf{I}_n) + \mathbf{P}_{n,n}(\mathbf{H}\otimes\mathbf{I}_n))^{\mathsf{T}}\text{vec}(\mathbf{A}-\mathbf{H}\mathbf{H}^{\mathsf{T}}) \\ &= -(\mathbf{H}^{\mathsf{T}}\otimes\mathbf{I}_n)\text{vec}(\mathbf{A}-\mathbf{H}\mathbf{H}^{\mathsf{T}}) - \\ &\qquad (\mathbf{H}^{\mathsf{T}}\otimes\mathbf{I}_n)\mathbf{P}_{n,n}^{\mathsf{T}}\text{vec}(\mathbf{A}-\mathbf{H}\mathbf{H}^{\mathsf{T}}) \\ &= -2*\text{vec}(\mathbf{AH}-\mathbf{H}(\mathbf{H}^{\mathsf{T}}\mathbf{H})),\end{aligned}$$

because applying $\mathbf{P}_{n,n}^{\mathsf{T}}$ to the vectorization of a symmetric matrix does not change the vector. The last equality is due to the identity $(\mathbf{B}^{\mathsf{T}}\otimes\mathbf{A})\text{vec}(\mathbf{X}) = \text{vec}(\mathbf{AXB})$.

*2) Applying Gramian of Jacobian:* In order to solve the linear system for the GN step using CG, we need to apply the coefficient matrix $\mathbf{J}^{\mathsf{T}}\mathbf{J}$ to a vector. Because the perfect shuffle permutation for the square case is both orthogonal and symmetric, the Gramian of the Jacobian, see Equation (3), takes the form

$$\mathbf{J}^{\mathsf{T}}\mathbf{J} = 2\big(\mathbf{H}^{\mathsf{T}}\mathbf{H}\otimes\mathbf{I}_n + (\mathbf{H}\otimes\mathbf{I}_n)^{\mathsf{T}}\mathbf{P}_{n,n}(\mathbf{H}\otimes\mathbf{I}_n)\big). \tag{8}$$

Thus, to apply the Gramian to a vector $\mathbf{x}$, which we will consider to be a vectorization of an $n\times k$ matrix $\mathbf{X}$, we use

$$\begin{aligned}\mathbf{J}^{\mathsf{T}}\mathbf{J}\mathbf{x} &= 2((\mathbf{H}^{\mathsf{T}}\mathbf{H}\otimes\mathbf{I}_n)\mathbf{x} + (\mathbf{H}\otimes\mathbf{I}_n)^{\mathsf{T}}\mathbf{P}_{n,n}(\mathbf{H}\otimes\mathbf{I}_n)\mathbf{x}) \\ &= 2(\text{vec}(\mathbf{X}\mathbf{H}^{\mathsf{T}}\mathbf{H}) + (\mathbf{H}\otimes\mathbf{I}_n)^{\mathsf{T}}\mathbf{P}_{n,n}\text{vec}(\mathbf{X}\mathbf{H}^{\mathsf{T}})) \\ &= 2(\text{vec}(\mathbf{X}\mathbf{H}^{\mathsf{T}}\mathbf{H}) + (\mathbf{H}\otimes\mathbf{I}_n)^{\mathsf{T}}\text{vec}(\mathbf{H}\mathbf{X}^{\mathsf{T}})) \\ &= 2(\text{vec}(\mathbf{X}(\mathbf{H}^{\mathsf{T}}\mathbf{H})) + \text{vec}(\mathbf{H}(\mathbf{X}^{\mathsf{T}}\mathbf{H}))),\end{aligned}$$

which can be computed using 4 dense matrix multiplications.

### *B. Parallel GNCG*

We present the parallel algorithm for Gauss-Newton using Conjugate Gradient in Algorithm 2. Nearly all of the pseudocode is devoted to the "vector" operations of CG, which in our case corresponds to matrix additions and matrix inner products because we maintain all of the CG vectors as matrices with the same distribution as the factor matrix $\mathbf{H}$. The comments in the pseudocode show the standard vector notation of CG. We encapsulate the expensive operations that are unique to GN for SymNMF in function calls to Compute-Gradient (Algorithm 3, described in § IV-B1) and Apply-Gramian (Algorithm 4, described in § IV-B2).

Because of the nesting of iterative algorithms, there is overloaded terminology and a clash of standard notation. We use the matrix $\mathbf{H}$ to represent the factor matrix, which is the solution vector of the Gauss-Newton iteration. We use the matrix $\mathbf{X}$ to represent the step direction for the Gauss-Newton iteration, which is also the solution vector of the linear system

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.

**Algorithm 2** $[\mathbf{W},\mathbf{H}] = \text{SymGNCG}(\mathbf{A},k,s_{\max})$

**Require:** $\mathbf{A} \in \mathbb{R}_+^{n\times n}$ is distributed across a $\sqrt{p}\times\sqrt{p}$ grid of processors, $k>0$ is rank of approximation, $p$ divides $n$
**Require:** Local matrices: $\mathbf{H}_{ij},\mathbf{X}_{ij},\mathbf{P}_{ij},\mathbf{R}_{ij},\mathbf{Y}_{ij}$ are $n/p\times k$
1: Proc $p_{ij}$ initializes $\mathbf{H}_{ij}$
2: **while** stopping criteria not satisfied **do**
3: $\quad\mathbf{X}=\mathbf{0}$ % *Initialize* $\mathbf{x}_0=\mathbf{0}$
4: $\quad\mathbf{R}=$Compute-Gradient$(\mathbf{A},\mathbf{H})$ % $\mathbf{r}=\mathbf{b}-\mathbf{J}^\mathsf{T}\mathbf{J}\mathbf{x}_0$
5: $\quad p_{ij}$ sets $\mathbf{P}_{ij}=\mathbf{R}_{ij}$ % $\mathbf{p}=\mathbf{r}$
6: $\quad p_{ij}$ computes $\epsilon_{ij}^{\text{old}}=\langle\mathbf{R}_{ij},\mathbf{R}_{ij}\rangle$
7: $\quad$compute $\epsilon^{\text{old}}=\sum_{i,j}\epsilon_{ij}^{\text{old}}$ using all-reduce across all procs
8: $\quad$**for** $s=1$ to $s_{\max}$ **do**
9: $\quad\quad\mathbf{Y}=$Apply-Gramian$(\mathbf{H},\mathbf{P})$ % $\mathbf{y}=\mathbf{J}^\mathsf{T}\mathbf{J}\mathbf{p}$
10: $\quad\quad p_{ij}$ computes $\alpha_{ij}=\epsilon^{\text{old}}/\langle\mathbf{P}_{ij},\mathbf{Y}_{ij}\rangle$
11: $\quad\quad$compute $\alpha=\sum_{i,j}\alpha_{ij}$ using all-reduce across all procs
12: $\quad\quad p_{ij}$ computes $\mathbf{X}_{ij}=\mathbf{X}_{ij}+\alpha\mathbf{P}_{ij}$ % $\mathbf{x}=\mathbf{x}+\alpha\mathbf{p}$
13: $\quad\quad p_{ij}$ computes $\mathbf{R}_{ij}=\mathbf{R}_{ij}-\alpha\mathbf{Y}_{ij}$ % $\mathbf{r}=\mathbf{r}-\alpha\mathbf{y}$
14: $\quad\quad p_{ij}$ computes $\epsilon_{ij}=\langle\mathbf{R}_{ij},\mathbf{R}_{ij}\rangle$
15: $\quad\quad$compute $\epsilon=\sum_{i,j}\epsilon_{ij}$ using all-reduce across all procs
16: $\quad\quad p_{ij}$ computes $\mathbf{P}_{ij}=\mathbf{R}_{ij}+(\epsilon/\epsilon^{\text{old}})\mathbf{P}_{ij}$ % $\mathbf{p}=\mathbf{r}+\beta\mathbf{p}$
17: $\quad\quad\epsilon^{\text{old}}=\epsilon$
18: $\quad$**end for**
19: $\quad p_{ij}$ computes $\mathbf{H}_{ij}=[\mathbf{H}_{ij}-\mathbf{X}_{ij}]_+$ % *projected GN step*
20: **end while**
**Ensure:** $\mathbf{H}\approx\underset{\tilde{\mathbf{H}}\geqslant 0}{\operatorname{argmin}}\|\mathbf{A}-\tilde{\mathbf{H}}\tilde{\mathbf{H}}^\mathsf{T}\|_F^2$
**Ensure:** $\mathbf{H}$ is $n\times k$ row-wise distributed across processors

solved approximately by CG. The matrices $\mathbf{P}$ and $\mathbf{R}$ follow the standard notation of CG and correspond to the step direction and residual of the linear system. Matrix $\mathbf{Y}$ is a temporary variable needed within CG, the output of the single matrix-vector product (computed by Apply-Gramian in our case).

The right-hand-side of the linear system, $\mathbf{b}$ in standard notation, is the gradient of the GN step, which is computed and stored in the residual matrix $\mathbf{R}$ in Line 4. The GN step is taken in Line 19, initially in the direction of $-\mathbf{X}$ and then projected onto the set of nonnegative matrices. The negative sign appears since we work with $\mathbf{J}^\mathsf{T}\mathbf{r}$ as $\mathbf{b}$ instead of $-\mathbf{J}^\mathsf{T}\mathbf{r}$.

*1) Gradient computation:* For each GN step, we compute the gradient, which is the right-hand-side vector of the linear system solved approximately by CG. Algorithm 3 shows the parallel algorithm for evaluating the gradient expression derived in § IV-A1. The algorithm consists of three matrix multiplications: $\mathbf{AH}$, $\mathbf{H}^\mathsf{T}\mathbf{H}$, and $\mathbf{H}(\mathbf{H}^\mathsf{T}\mathbf{H})$. The first two multiplications appear in the ANLS algorithm (Algorithm 1). To compute $\mathbf{H}^\mathsf{T}\mathbf{H}$, with $\mathbf{H}$ row-distributed, each processor performs a local (symmetric) multiplication and then performs an all-reduce collective. Afterwards, the third multiplication between $\mathbf{H}$ and $\mathbf{H}^\mathsf{T}\mathbf{H}$ can be performed locally. Because the 1D distribution of $\mathbf{H}$ conforms to the 2D distribution of $\mathbf{A}$, the first multiplication involves an all-gather collective, local multiplication, and a reduce-scatter collective. We note that the output matrix $\mathbf{AH}$ is 1D distributed, but in a different order than $\mathbf{H}$, as shown in Figure 3. In order to perform the matrix subtraction we must redistribute $\mathbf{AH}$ to match $\mathbf{H}$, which can be done via pairwise exchanges between symmetric partners (Line 4).

*2) Applying Gramian of Jacobian:* Algorithm 4 shows the parallel algorithm for applying the Gramian of the Jacobian

**Algorithm 3** $\mathbf{G}=\text{Compute-Gradient}(\mathbf{A},\mathbf{H})$

**Require:** $\mathbf{A}\in\mathbb{R}^{n\times n}$ is distributed across a $\sqrt{p}\times\sqrt{p}$ grid of processors, $\mathbf{H}\in\mathbb{R}^{n\times k}$ is row-wise distributed
**/* Compute $\mathbf{AH}$ */**
1: $p_{ij}$ collects $\mathbf{H}_j$ using all-gather across proc columns
2: $p_{ij}$ computes $\mathbf{V}_{ij}=\mathbf{A}_{ij}\mathbf{H}_j$
3: compute $(\mathbf{AH})_i=\sum_j\mathbf{V}_{ij}$ using reduce-scatter across proc row to achieve row-wise distribution of $(\mathbf{AH})_i$
4: $p_{ij}$ sends $(\mathbf{AH})_{ij}$ to $p_{ji}$ and receives $(\mathbf{AH})_{ji}$ from $p_{ji}$
**/* Compute $\mathbf{H}^\mathsf{T}\mathbf{H}$ */**
5: $p_{ij}$ computes $\mathbf{U}_{ij}=\mathbf{H}_{ij}^\mathsf{T}\mathbf{H}_{ij}$
6: compute $\mathbf{U}=\sum_{i,j}\mathbf{U}_{ij}$ using all-reduce across all procs
**/* Compute $-2(\mathbf{AH}-\mathbf{H}(\mathbf{H}^\mathsf{T}\mathbf{H}))$ */**
7: $p_{ij}$ computes $\mathbf{G}_{ij}=-2((\mathbf{AH})_{ji}-\mathbf{H}_{ij}\mathbf{U})$
**Ensure:** $\mathbf{G}=-2(\mathbf{AH}-\mathbf{HH}^\mathsf{T}\mathbf{H})$ distributed row-wise

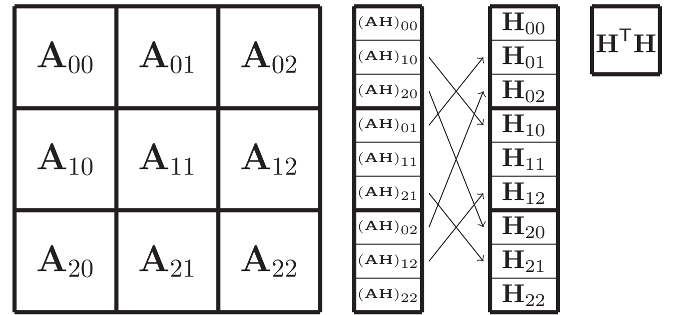



Fig. 3. Data distribution of GNCG and communication pattern of $\mathbf{AH}$ to compute $\mathbf{AH}-\mathbf{H}(\mathbf{H}^\mathsf{T}\mathbf{H})$ for a $3\times3$ processor grid.

to a vector (reshaped into a matrix for our case), which is required of every CG iteration. Using identical row-wise distributions of $\mathbf{H}$, the linear-system solution vector $\mathbf{X}$, and the output vector $\mathbf{Y}$, we can perform the four matrix multiplications derived in § IV-A2 efficiently in parallel, using only two collective communication operations. Computing the $k\times k$ matrices $\mathbf{H}^\mathsf{T}\mathbf{H}$ and $\mathbf{X}^\mathsf{T}\mathbf{H}$ requires reducing results across all processors. By using all-reduce, we can obtain those matrices on all processors in order to multiply the local blocks of $\mathbf{X}$ and $\mathbf{H}$ with those $k\times k$ matrices and perform the addition with no further communication.

Because the Gramian of $\mathbf{H}$ does not change over CG iterations (and it's also used in the computation of the gradient), it does not need to be recomputed for each CG iteration. We employ this optimization in our implementation but leave it out of the pseudocode for simpilicity. This avoids a computation step (Line 1) and a communication step (Line 2).

## C. Analysis

We now analyze the cost of single GN iteration, which involves $s_{\max}$ CG iterations. Aside from the CG iterations, the most expensive operation is the call to Compute-Gradient (Line 4), shown in Algorithm 3. As analyzed in the case of ANLS (§ III-D), the cost of computing $\mathbf{AH}$ is $2n^2k/p$ flops (if the matrix is dense), $O(nk/\sqrt{p})$ words, and $O(\log p)$

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.

**Algorithm 4** $\mathbf{Y} = \text{Apply-Gramian}(\mathbf{H},\mathbf{X})$

**Require:** $\mathbf{H},\mathbf{X} \in \mathbb{R}^{n\times k}$ are distributed row-wise (identically) across processors
1: $p_{ij}$ computes $\mathbf{U}_{ij} = \mathbf{H}_{ij}^{\mathsf{T}}\mathbf{H}_{ij}$
2: compute $\mathbf{U} = \sum_{i,j}\mathbf{U}_{ij}$ using all-reduce across all procs
3: $p_{ij}$ computes $\mathbf{V}_{ij} = \mathbf{X}_{ij}^{\mathsf{T}}\mathbf{H}_{ij}$
4: compute $\mathbf{V} = \sum_{i,j}\mathbf{V}_{ij}$ using all-reduce across all procs
5: $p_{ij}$ computes $\mathbf{Y} = \mathbf{X}_{ij}\mathbf{U}$
6: $p_{ij}$ computes $\mathbf{Y} = 2\mathbf{Y} + 2\mathbf{H}_{ij}\mathbf{V}$
**Ensure:** $\mathbf{Y} = 2(\mathbf{X}\mathbf{H}^{\mathsf{T}}\mathbf{H} + \mathbf{H}\mathbf{X}^{\mathsf{T}}\mathbf{H})$ distributed row-wise

TABLE I
PER-ITERATION PER-PROCESSOR COSTS FOR DENSE CASE

| Algorithm | flops | words | messages |
|---|---|---|---|
| ANLS | $\frac{4n^2k}{p} + O(\frac{nk^2}{p})$ | $O(\frac{nk}{\sqrt{p}} + k^2)$ | $O(\log p)$ |
| GNCG | $\frac{2n^2k}{p} + O(\frac{s_{\max}nk^2}{p})$ | $O(\frac{nk}{\sqrt{p}} + s_{\max}k^2)$ | $O(s_{\max}\log p)$ |

messages. If the matrix is sparse, the flop cost is $2\text{nnz}(A)k/p$, assuming perfect load balance of the nonzeros. The cost of the extra step of redistribution of $\mathbf{AH}$ is dominated by the costs of the multiplication. As before, computing $\mathbf{H}^{\mathsf{T}}\mathbf{H}$ requires $O(nk^2/p)$ flops, $O(k^2)$ words, and $O(\log p)$ messages, and the final multiplication requires another $O(nk^2)$ flops but no more communication.

The cost of each CG iteration is dominated by the call to Apply-Gram (Line 9), shown in Algorithm 4. All 4 local matrix multiplications involve $O(nk^2/p)$ flops, and the two collectives cost $O(k^2)$ words and $O(\log p)$ messages.

Thus, the cost of each GN iteration, assuming a fixed number $s_{\max}$ of CG iterations and a dense matrix, is $2n^2k/p + O(s_{\max}nk^2/p)$ flops, $O(nk/\sqrt{p} + s_{\max}k^2)$ words, and $O(s_{\max}\log p)$ messages. We compare the costs of GNCG with ANLS (for the dense case) in Table I.

## V. EXPERIMENTS

### A. Experimental Setup

All our experiments were conducted on Summit, a supercomputer at the Oak Ridge Leadership Computing Facility [21]. It is an IBM system comprising 4,600 compute nodes. Each Summit node contains 2 IBM POWER9 processors on separate sockets. Sockets are connected via dual NVLINK capable of transferring at 25 GB/s between each other. Nodes are connected to an InfiniBand network providing a node injection bandwidth of 23 GB/s. Each node contains 512 GB of DDR4 memory. Additionally Summit nodes have 6 NVIDIA Volta V100 accelerators but they are not used by our implementation [2].

MPI-FAUN uses the Armadillo library [22] for matrix operations. Armadillo stores dense matrices in column major order and sparse matrices in the Compressed Sparse Column (CSC) format. We link Armadillo (version 9.900) with OpenBLAS (version 0.3.9) and IBM Spectrum MPI (version 10.3.1.2-20200121) for dense BLAS and LAPACK operations and compile using the GNU C++ compiler version 6.4.0.

All the scaling experiments are conducted with flat MPI scaling. By flat we mean that each core is assigned to a different MPI process. This is in contrast to assigning individual MPI processes to an entire node/socket and enabling multithreading within a node/socket. We found the flat setting to run faster and use it as the basis for our scaling experiments.

Beyond reported speedups, we also examined the absolute performance of our implementation by assessing the performance of Armadillo on a single Summit node. In particular, we ran matrix multiply kernels in a manner similar to the flat MPI setting by launching multiple matrix multiply kernels, each bound to a core, on one node. For dense GEMM on large square matrices, Armadillo achieved 63% of the peak FLOPS. However, if one the matrices has a small dimension, as in our experiments, Armadillo instead achieved 43% of peak. This is a reasonable fraction of the peak performance and is close to the 75% achieved in most systems [23].

In the sparse setting, we compared Armadillo with Eigen [24] for a dense-matrix-times-sparse-matrix kernel. Both libraries performed similarly and we use Armadillo in our experiments. This kernel is expected to be bound by memory bandwidth, especially if the dense matrix is of low-rank as in SymNMF. We computed a conservative lower bound on the bandwidth achieved by this kernel and compared that to the sustainable bandwidth reported by the Stream triad [25] benchmark. By "conservative" we mean that we consider only compulsory load traffic, which is the sum of the bytes need to store the input and output matrices, and divide this value by the kernel runtime [26], [27]. This value was found to be 24% of the peak Stream triad bandwidth which is comparable to 10-35% of peak performance cited in earlier studies [28], [29] [3].

Our implementation is constrained to run on square processor grids. Summit nodes have sockets with 21 cores on each socket and the scheduler requires tasks to be divided uniformly across sockets. This forces us to limit the number of processor grid configurations to either use 16 or 18 processors per socket as we scale across nodes. Our experiments scale up to 128 nodes (256 sockets) with MPI processor grid sizes of $1\times 1, 2\times 2, 3\times 3, 4\times 4, 6\times 6, 8\times 8, 12\times 12, 24\times 24, 32\times 32, 48\times 48$ and $64\times 64$. Here, 16 processors is the largest configuration that can fit in a single socket and 36 processors is the largest that can fit in a node.

### B. Datasets

*1) Pixel Similarity Data [30]:* The Pixel Similarity matrix is generated using the Berkeley Segmentation Engine [6]. Each image is flattened to a vector of pixels and a similarity matrix is generated between pixels. The similarity value can be computed based on various factors including brightness, color and textural cues. We compute similarities only between spatially near-by pixels. This neighborhood is defined by a disk of radius 20 pixels around every pixel [1]. We used 3 satellite images from the Functional Map of the World

[2] https://github.com/ramkikannan/planc

[3] Hong et al. [28] studied this kernel in the context of GPUs.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.

TABLE II
PIXEL SIMILARITY DATA

| Image | Image Size | Matrix Size ($n$) | nonzeros |
|---|---|---|---|
| lighthouse_61_9 | 1525×1419 | 2,163,975 | 32,406,651 |
| amusement_park_186_6 | 2865×2535 | 7,262,775 | 108,844,443 |
| shipyard_11_1 | 5584×4304 | 24,033,536 | 360,325,074 |

(fMoW) dataset [11] to generate these matrices. We randomly permute the matrices for load balancing. Some salient features of the images are described in Table II.

*2) Synthetic:* Our synthetic datasets are constructed in two ways depending on whether the input is dense or sparse. For the dense case we generate $\mathbf{A} = \mathbf{H}\mathbf{H}^\mathsf{T}$ where $\mathbf{H}$ is a random low-rank and nonnegative matrix. This is an exact SymNMF model, and we can confirm that the residual error of our algorithm with a random start converges to zero. For benchmarking we run a fixed number of iterations of the SymNMF algorithms rather than till convergence. For sparse inputs, we specify a fixed density of 0.005.

### C. Performance Breakdown

We breakdown the running time of our algorithms into the following categories.

*a) Matrix Multiply:* This is the application of the data matrix to the factor matrices for computing $\mathbf{AH}$ (or $\mathbf{W}^\mathsf{T}\mathbf{A}$). These products are needed for the RHS in the nonnegative least squares subproblems in the ANLS version and the gradient in the GNCG. This is further broken into the computation and communication phases. The communication phases is the all-gather, reduce-scatter and the sendrecv of the factor matrices. Only the local matrix multiplication call is considered for the computation phase and the reduce-scatter computation is counted towards the communication phase. There is only a single Matrix Multiply phase per outer iteration in GNCG versus two per outer iteration for the ANLS version.

*b) Solve:* This is the rest of the work needed to complete an inner iteration. Primarily, this is constructing the Gramian matrices $\mathbf{W}^\mathsf{T}\mathbf{W}$ and $\mathbf{H}^\mathsf{T}\mathbf{H}$ and performing the nonnegative least squares solves in the ANLS variant. For GNCG we include the calculations involved in the CG section of the algorithm which include the matrix multiplies involved in Apply-Gram. There are all-reduce communications involved in this phase but only involve smaller $k \times k$ matrices.

*c) Other:* This includes smaller computations like norm checks and applying regularizations which are not explicitly timed in the above phases.

### D. Convergence

We first test the sequential performance of our proposed Gauss-Newton algorithm against other SymNMF variants in Figure 4. We run SymNMF ANLS (ANLS) [1], Projected Gradient Descent (PGD) [1] and Cyclic Coordinate Descent (CCD) [2] on 3 different inputs: random symmetric matrices, low-rank symmetric positive semi-definite (SPSD) matrices, and the Soybean dataset [31]. The SPSD case is exactly low-rank input and all algorithms converge quickly. For the other inputs, which need not have exact low-rank, all algorithms perform similarly. Figure 4 shows that the Gauss-Newton method is competitive with the other algorithms. The Soybean data set is taken from the UC Irvine Machine Learning Repository[4]. We apply various standard pre-processing steps, e.g. removing small clusters, resulting in a $200 \times 200$ similarity matrix.

Figure 5 and Figure 6 shows the convergence of the parallel SymNMF algorithms on large dense synthetic low-rank matrices and the Pixel Similarity data. Since the synthetic matrices are exactly low-rank we expect to see good convergence and small relative errors $\|\mathbf{A} - \mathbf{H}\mathbf{H}^\mathsf{T}\|_F^2 / \|\mathbf{A}\|_F^2$. We test GNCG with different number of inner CG iterations allowed per outer iteration. We specify 3 different settings where we allow 5, $k$, or 1000 inner CG iterations per outer iteration.

Figure 5 shows good convergence across all the algorithm settings on synthetic matrices with $n = 442{,}368$ and $k = 100$. The relative error is calculated as the average of 5 different initializations. The different algorithms solve the same problem from the same starting point. The number of CG iterations does not significantly affect the final relative error. The difference between 100 and 1000 CG iterations is imperceptible in the figure. However, it does affect the execution time: the 5 CG iteration is the fastest of the different settings, though the difference is slight when running time is dominated by Matrix Multiply (performed once per GN iteration). Since this does not greatly affect the relative error we set $s_{\max} = 5$ for the rest of the experiments.

Figure 6 shows the convergence for 30 iterations on the Pixel Similarity matrices with $k = 16$. This is the rank used to generate embeddings from images in prior work [1], [6]. The relative error is large but decreasing over time. In this case, we seek only to discover embeddings in this task rather than to factor the matrix exactly. Once again we can see similar performance with both the ANLS and GNCG variants.

### E. Scaling Studies

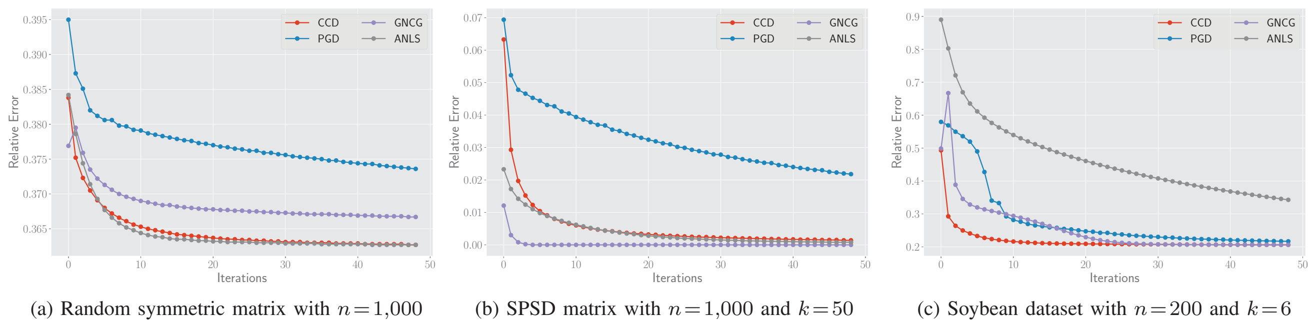
*1) Strong Scaling:* We present the strong scaling performance for both dense and sparse inputs in Figures 8 and 9. Our matrix sizes are chosen to fill up a single socket's memory on Summit. We use $n = 156{,}401$ for the dense case and $n = 884{,}736$ for the sparse case with density 0.005. We use a low-rank of 48 which is a reasonable size for embeddings.

Figure 8a and Figure 9a shows the average time per outer iteration of the algorithms as the number of processors is scaled up. In general the performance scales gracefully up to 4096 cores, but we can see a noticeable bump when we first span a socket (i.e. 36 processes). Figure 8b and Figure 9b clearly show that the Matrix Multiply time is the dominant cost for both cases. GNCG takes advantage of this fact and is approximately $2\times$ faster than the ANLS variant.

Figure 7 shows the scaling efficiency of both cases. The dense case is able to scale gracefully up to 4096 processes with 55% efficiency for ANLS and 70% efficiency for GNCG. The sparse case behaves more erratically and is able to scale at $\approx$ 70% efficiency till 576 processes. It displays slight

[4]https://archive.ics.uci.edu/ml/index.php

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.




(a) Random symmetric matrix with $n = 1{,}000$ (b) SPSD matrix with $n = 1{,}000$ and $k = 50$ (c) Soybean dataset with $n = 200$ and $k = 6$

Fig. 4. Convergence comparison of the sequential SymNMF algorithms. The GNCG relative error is comparable to other SymNMF variants.

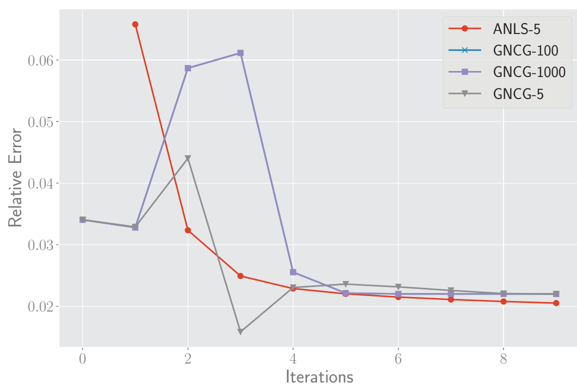



Fig. 5. Synthetic with $n = 442{,}368$ and $k = 100$ with different number of inner CG iterations. Inner CG iterations does not seem to affect the final relative error.

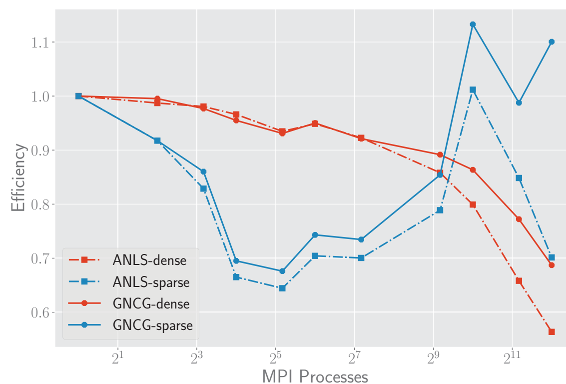



Fig. 7. Strong scaling efficiency upto 128 nodes of Summit. Initial efficiency drop is noticed once we scale out a socket (16 processes) and cache effects are observed for the sparse case after 576 processes.

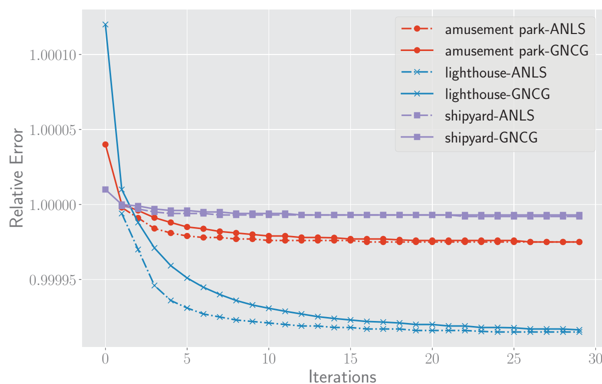


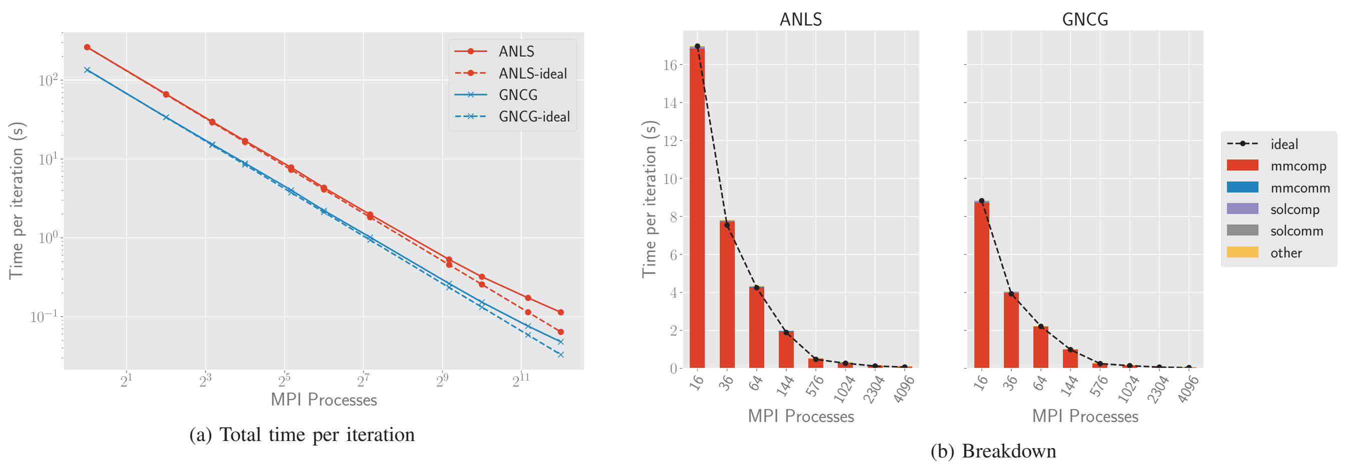

Fig. 6. Convergence on the Pixel Similarity matrices. Both ANLS and GNCG perform similarly in terms of final relative error.

superlinear tendency at the larger grid dimensions which we attribute to caching effects as the problem size decays. The problem size of the input matrix is $27{,}648 \times 27{,}648$ for 1024 processors and is only expected to occupy $\approx$ 58 MB in memory per process. The SymNMF problem is bandwidth bound for these input dimensions. This is seen clearly in the sparse case as efficiency drops within a socket but stabilizes as more sockets are added. This is because bandwidth doesn't increase when we scale within a socket as more cores are used.

*2) Weak Scaling:* Figure 10 demonstrates the weak scaling performance of our algorithms up to 4096 processes. the memory per node is kept constant and scales from an initial size of $n_0 = 156{,}401$ for dense inputs and $n_0 = 625{,}603$ for sparse inputs. Matrix dimensions are increased proportionally to the square root of the number of nodes as we scale up. This keeps the local $\mathbf{A}$ matrix dimensions constant per processor. Since we expect the computation to be bottlenecked by the matrix multiplication call we expect to observe flat runtimes. Figure 10 confirms this prediction and we see roughly the same performance on all processor grids. We can see two distinct sets of bars in the graphs with one set running slightly faster than the other. In the dense case the faster runs corresponds to MPI grids of size 16, 64, 1024, and 4096. All these configurations have 16 MPI processes per socket whereas the others have 18 per socket. Interestingly, the opposite effect is seen in the sparse case with the 18 core per socket configurations running slightly faster.

*3) Scaling on Pixel Similarity Matrices:* Next we consider scaling performance on the Pixel Similarity matrices in Figure 11. All the experiments were run with $k = 16$. Figure 11a shows that all the matrices scale similarly. The algorithms steadily lose efficiency till they scale out of a socket and then stabilize at 50% efficiency. This is because bandwidth does not scale with cores within a socket and our algorithm is

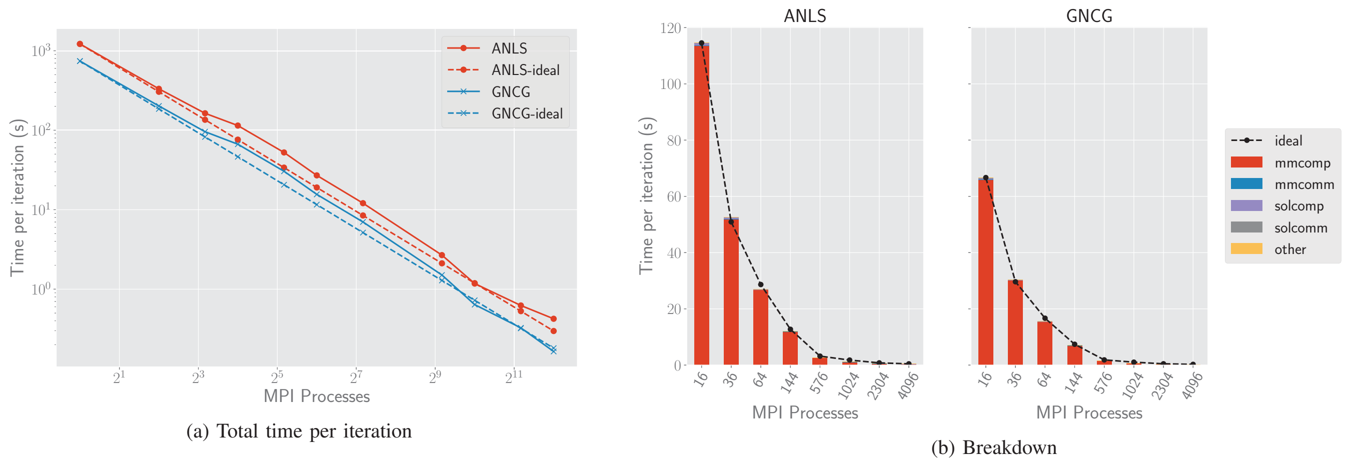
Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.




(a) Total time per iteration

(b) Breakdown

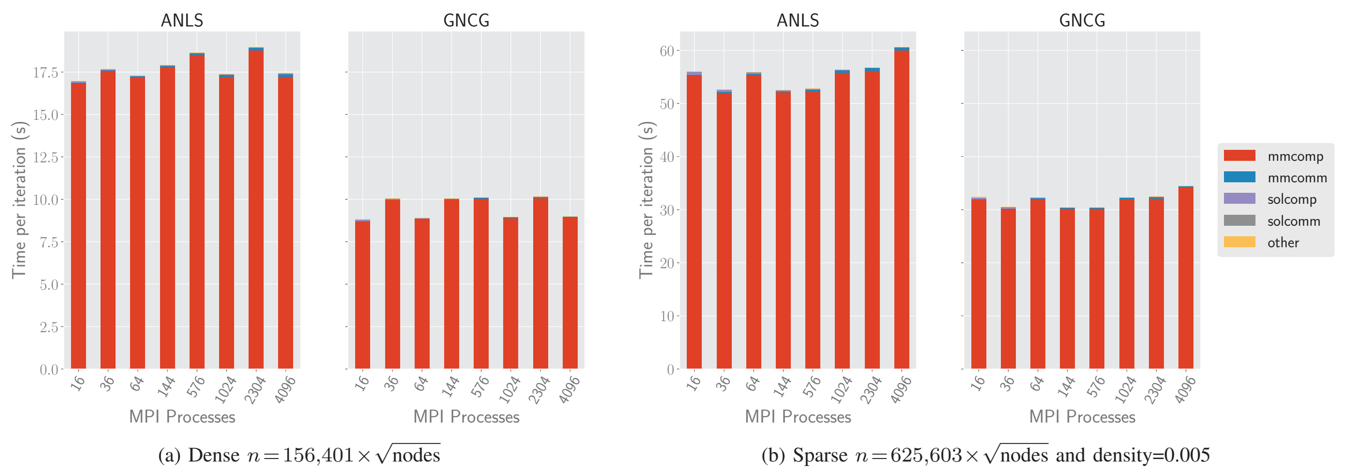
Fig. 8. Dense strong scaling with $n = 156{,}401$ and $k = 48$




(a) Total time per iteration

(b) Breakdown

Fig. 9. Sparse strong scaling with $n = 884{,}736$, density = 0.005 and $k = 48$

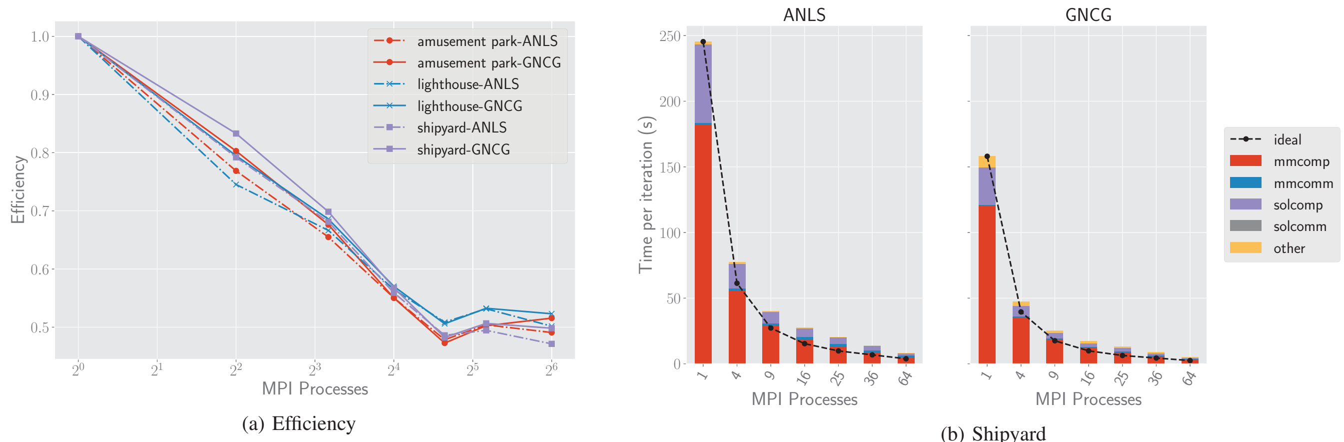



(a) Dense $n = 156{,}401 \times \sqrt{\text{nodes}}$

(b) Sparse $n = 625{,}603 \times \sqrt{\text{nodes}}$ and density=0.005

Fig. 10. Weak scaling with $k = 48$. Efficient configurations for the dense and sparse are different.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.




(a) Efficiency

(b) Shipyard

Fig. 11. Strong scaling on the Pixel Similarity matrices with $k=16$

bandwidth bound. Bandwidth is only increased when scaling out to multiple sockets. Since the algorithm seems to perform similarly on all 3 inputs we only show the breakdown for the shipyard image. Unlike the previous scaling runs we can see other components of the algorithm apart from just the matrix multiply. One can see that the matrix multiply is scaling in $O(p)$ but it is not as clear for the solver times. GNCG is still the faster algorithm due to performing fewer matrix multiplies.

### F. Low-rank Sweep

The $k$-sweep experiment describes the variation in running time when larger low-rank parameters are chosen. For larger matrix dimensions we are completely dominated by the matrix multiply time and these variations do not affect the overall run time. Therefore, we choose a relatively small matrix to conduct this experiment. Figure 12a shows how SymNMF runtimes vary for a dense synthetic matrix of size $14000 \times 14000$. We see a linear increase in runtime as $k$ increases. Figure 12b shows the breakdown of this linear increase. The proportion of solver time increases more rapidly for GNCG than ANLS. This indicates that for cases with extremely large $k$ it might be better to use ANLS than GNCG. The runtimes for a sparse input with the same memory footprint is similar.

### G. Image Segmentation

We recreate the image segmentation experiments described in earlier works [1], [6] albeit on much larger images. The task is to cluster the pixels of an image into a nonoverlapping set of closed regions. Once these regions are discovered we can determine "boundary" pixels which segment the image. The Berkeley Segmentation Engine (BSE) [6] is one of the classical segmentation algorithms used for this task. It represents the pixels in the image as nodes in a graph and defines the segmentation task as a graph partitioning problem. We refer the reader to earlier works for the details on generating such a graph [1], [6]. Spectral clustering [6] is used as the graph partitioning algorithm. In this method, an eigendecomposition is used to generate embeddings for each pixel. We replace those embeddings with the ones produced by the SymNMF algorithm and leave the rest of the pipeline intact. Figure 13 displays the boundary and regions discovered for the lighthouse image.

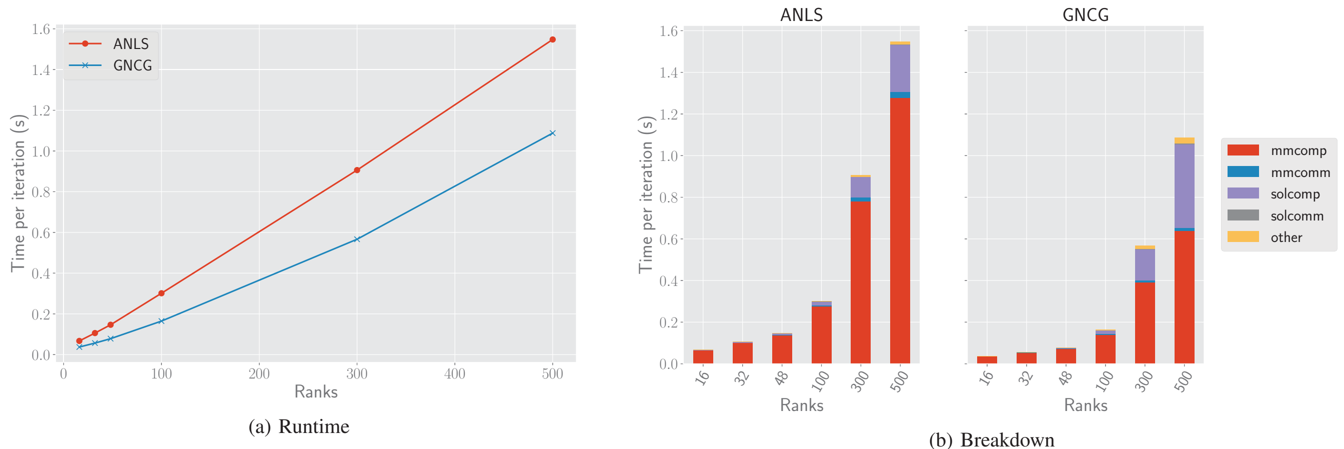
## VI. Discussion and Future Work

The experiments in the preceding sections show that the proposed SymNMF algorithms perform well both in terms of scaling and low-rank approximations. In comparing ANLS and GNCG, our results show both a block coordinate descent and a second-order method can be parallelized efficiently. We did not observe large deviations in convergence between the two methods, so the relative efficiency depends mostly on the per-iteration costs. As seen from our scaling experiments, GNCG runs about twice as fast when the matrix multiply time is dominant, which we expect for large $n$ and small $k$. However, when the other parts of the algorithm come into play (small $n$ and relatively large $k$), as is the case for the pixel similarity matrices, ANLS is more competitive, depending on the number of CG iterations used by GNCG.

We also mention some of the limitations of the current work which we hope to address in the future. The first limitation is the use of square processor grids of the form $\sqrt{p} \times \sqrt{p}$, ignoring the symmetry in the input data. Generalizing the approach to triangular grids could avoid the redundant storage of $\mathbf{A}$ and possibly enable better load balancing and finding effective communication patterns [32]. Another approach to reducing the constraints on mapping a square grid to the architecture is to develop a hybrid implementation, assigning only 1 MPI process to each node (or socket) and employing shared-memory parallel subroutines locally.

Though outside the scope of this work, we see many opportunities for further performance optimization for different classes of sparse matrices from applications outside image segmentation, such as text data and other undirected relationship graphs. For example, in the extremely sparse case we should switch from collective communication to a point-to-point communication scheme [33]. Because the algorithms iteratively apply the matrix, it may also be worth partitioning the matrix data more carefully using graph or hypergraph partitioners to achieve both load balance and low communication costs. As

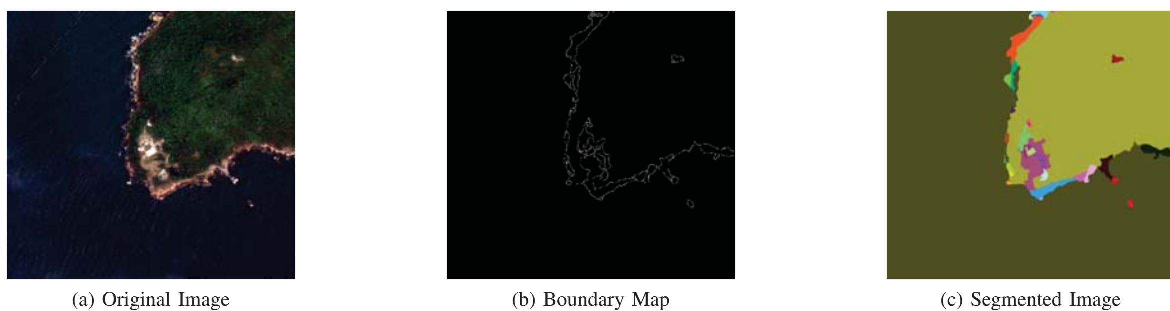
Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.




(a) Runtime

(b) Breakdown

Fig. 12. K-Sweep with a dense synthetic matrix with $n = 14{,}000$ on a single node with $4 \times 4$ processor grid



(a) Original Image

(b) Boundary Map

(c) Segmented Image

Fig. 13. Boundary detection and image segmentation using features generated by SymNMF.

discussed earlier and shown in Figure 11, for large sparse matrices, the matrix multiply times decrease in proportion to the solver times. In light of this, optimizations in the nonnegative least squares solver such as those discussed in prior work [34] could become important to the runtime of the ANLS method.

## VII. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Nos. OAC-1642385, CCF-1942892, OAC-1642410, CCF-1533768, and OAC-1710371. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. Koby Hayashi acknowledges support from the United States Department of Energy through the Computational Sciences Graduate Fellowship (DOE CSGF) under grant number: DE-SC0020347. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

## References

[1] D. Kuang, S. Yun, and H. Park, "SymNMF: nonnegative low-rank approximation of a similarity matrix for graph clustering," *Journal of Global Optimization*, vol. 62, no. 3, pp. 545–574, 2015. [Online]. Available: https://doi.org/10.1007/s10898-014-0247-2

[2] A. Vandaele, N. Gillis, Q. Lei, K. Zhong, and I. Dhillon, "Efficient and non-convex coordinate descent for symmetric nonnegative matrix factorization," *IEEE Transactions on Signal Processing*, vol. 64, no. 21, pp. 5571–5584, 2016. [Online]. Available: https://ieeexplore.ieee.org/iel7/78/4359509/07513400.pdf

[3] F. Moutier, A. Vandaele, and N. Gillis, "Off-diagonal symmetric nonnegative matrix factorization," arXiv, Tech. Rep. 2003.04775, 2020. [Online]. Available: https://arxiv.org/abs/2003.04775

[4] R. Du, D. Kuang, B. Drake, and H. Park, "Hierarchical community detection via rank-2 symmetric nonnegative matrix factorization," *Computational social networks*, vol. 4, no. 1, p. 7, 2017. [Online]. Available: https://doi.org/10.1186/s40649-017-0043-5

[5] R. Du, B. Drake, and H. Park, "Hybrid clustering based on content and connection structure using joint nonnegative matrix factorization," *Journal of Global Optimization*, vol. 74, no. 4, pp. 861–877, 2019. [Online]. Available: https://doi.org/10.1007/s10898-017-0578-x

[6] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik, "Contour detection and hierarchical image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 5, pp. 898–916, 2010. [Online]. Available: https://ieeexplore.ieee.org/iel5/34/5735600/05557884.pdf

[7] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990. [Online]. Available: https://www.crss.ucsc.edu/Papers/deerwester-jasis90.pdf

[8] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Proceedings of the 13th International Conference on Neural Information Processing Systems*, ser. NIPS '00. Cambridge, MA, USA: MIT Press, 2000, pp. 535–541. [Online]. Available: https://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf

[9] J. Kim, Y. He, and H. Park, "Algorithms for nonnegative matrix and tensor factorizations: a unified view based on block coordinate descent framework," *Journal of Global Optimization*, vol. 58, no. 2, pp. 285–319, February 2014. [Online]. Available: https://ideas.repec.org/a/spr/jglopt/v58y2014i2p285-319.html

[10] R. Kannan, G. Ballard, and H. Park, "MPI-FAUN: An MPI-based framework for alternating-updating nonnegative matrix factorization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 3, pp. 544–558, March 2018. [Online]. Available: https://www.computer.org/csdl/trans/tk/2018/03/08089433-abs.html

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.

[11] G. Christie, N. Fendley, J. Wilson, and R. Mukherjee, "Functional map of the world," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6172–6180. [Online]. Available: http://openaccess.thecvf.com/content_cvpr_2018/papers/Christie_Functional_Map_of_CVPR_2018_paper.pdf

[12] J. Kim and H. Park, "Fast nonnegative matrix factorization: An active-set-like method and comparisons," *SIAM Journal on Scientific Computing*, vol. 33, no. 6, pp. 3261–3281, 2011. [Online]. Available: https://doi.org/10.1137/110821172

[13] D. Bertsekas, *Nonlinear Programming*. Athena Scientific, 1999.

[14] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[15] D. Kuang, C. Ding, and H. Park, "Symmetric nonnegative matrix factorization for graph clustering," in *Proceedings of the 2012 SIAM International Conference on Data Mining*, 2012, pp. 106–117. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611972825.10

[16] N. Vervliet and L. De Lathauwer, "Numerical optimization-based algorithms for data fusion," in *Data Handling in Science and Technology*. Elsevier, 2019, vol. 31, pp. 81–128. [Online]. Available: https://doi.org/10.1016/B978-0-444-63984-4.00004-1

[17] N. Singh, L. Ma, H. Yang, and E. Solomonik, "Comparison of accuracy and scalability of Gauss-Newton and alternating least squares for CP decomposition," arXiv, Tech. Rep. 1910.12331, 2019. [Online]. Available: https://arxiv.org/abs/1910.12331

[18] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1206

[19] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, p. 4966, Feb. 2005. [Online]. Available: https://doi.org/10.1177/1094342005051521

[20] H. V. Henderson and S. R. Searle, "The vec-permutation matrix, the vec operator and Kronecker products: a review," *Linear and Multilinear Algebra*, vol. 9, no. 4, pp. 271–288, 1981. [Online]. Available: http://dx.doi.org/10.1080/03081088108817379

[21] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell *et al.*, "The design, deployment, and evaluation of the CORAL pre-exascale systems," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 661–672. [Online]. Available: https://dl.acm.org/doi/pdf/10.5555/3291656.3291726

[22] C. Sanderson and R. Curtin, "Armadillo: a template-based C++ library for linear algebra," *Journal of Open Source Software*, vol. 1, no. 2, p. 26, 2016. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.00026.pdf

[23] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer, and H. Meuer, "Top500." [Online]. Available: https://www.top500.org/lists/top500/2020/06/

[24] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," 2010. [Online]. Available: http://eigen.tuxfamily.org

[25] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995. [Online]. Available: http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps

[26] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1213–1222. [Online]. Available: https://doi.org/10.1109/IPDPS.2014.125

[27] S. E. Kurt, V. Thumma, C. Hong, A. Sukumaran-Rajam, and P. Sadayappan, "Characterization of data movement requirements for sparse matrix computations on GPUs," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 2017, pp. 283–293. [Online]. Available: https://doi.org/10.1109/HiPC.2017.00040

[28] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, Ü. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient sparse-matrix multi-vector product on GPUs," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 66–79. [Online]. Available: https://dl.acm.org/doi/10.1145/3208040.3208062

[29] B. C. Lee, R. W. Vuduc, J. W. Demmel, K. A. Yelick, M. de Lorimier, and L. Zhong, "Performance optimizations and bounds for sparse symmetric matrix - multiple vector multiply," University of California, Berkeley, Tech. Rep. UCB/CSD-03-1297, 2003. [Online]. Available: https://bebop.cs.berkeley.edu/pubs/csd-03-1297.pdf

[30] S. Eswar, K. Hayashi, G. Ballard, R. Kannan, R. Vuduc, and H. Park, "Pixel similarity," August 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3877635

[31] M. Tan and L. Eshelman, "Using weighted networks to represent classification knowledge in noisy domains," in *Machine Learning Proceedings 1988*, J. Laird, Ed. San Francisco (CA): Morgan Kaufmann, 1988, pp. 121–134. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780934613644500189

[32] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Improving the scalability of a symmetric iterative eigensolver for multi-core platforms," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 16, pp. 2631–2651, 2014. [Online]. Available: https://doi.org/10.1002/cpe.3129

[33] O. Kaya, R. Kannan, and G. Ballard, "Partitioning and communication strategies for sparse non-negative matrix factorization," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3225058.3225127

[34] M. H. Van Benthem and M. R. Keenan, "Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems," *Journal of Chemometrics*, vol. 18, no. 10, pp. 441–450, 2004. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cem.889

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore. Restrictions apply.

# Appendix: Artifact Description/Artifact Evaluation

## SUMMARY OF THE EXPERIMENTS REPORTED

We ran scaling experiments for our parallel SymNMF algorithm on OLCF's Summit supercomputer using Spectrum MPI and OpenBLAS as described in the paper.

## ARTIFACT AVAILABILITY

*Software Artifact Availability:* All author-created software artifacts are maintained in a public repository under an OSI-approved license.

*Hardware Artifact Availability:* There are no author-created hardware artifacts.

*Data Artifact Availability:* All author-created data artifacts are maintained in a public repository under an OSI-approved license.

*Proprietary Artifacts:* None of the associated artifacts, author-created or otherwise, are proprietary.

*Author-Created or Modified Artifacts:*

```
Persistent ID: https://github.com/ramkikannan/planc
Artifact name: PLANC

Persistent ID: https://doi.org/10.5281/zenodo.3877635
Artifact name: Pixel Similarity Dataset
```

## BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

*Relevant hardware details:* Summit, IBM Power9 (1/socket, 2/node), dual NVLINK between sockets, 512 GB RAM, InfiniBand network, IBM Spectrum Scale filesystem

*Operating systems and versions:* RHEL

*Compilers and versions:* GCC 6.4.0

*Applications and versions:* None

*Libraries and versions:* Spectrum MPI 10.3.1.2-20200121, OpenBLAS (version 0.3.9)

*Key algorithms:* Gauss-Newton

*Input datasets and versions:* Functional Map of the World: lighthouse 61 9, amusement park 186 6, shipyard 11 1

*URL to output from scripts that gathers execution environment information.*

```
https://www.dropbox.com/s/sd79sdmtzig7z2c/summitenv.
↪  txt?dl=0
```

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on August 30,2021 at 06:27:08 UTC from IEEE Xplore.  Restrictions apply.