A Parallel PARAFAC Implementation & Scalability Testing for Large-Scale Dense Tensor Decomposition

Kareem S. Aggour
Knowledge Discovery Lab
GE Global Research
Niskayuna, NY 12309
518-387-4047
aggour@ge.com

Bülent Yener
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
518-276-6907
yener@cs.rpi.edu

ABSTRACT
Parallel Factor Analysis (PARAFAC) is used in many scientific disciplines to decompose multidimensional datasets into principal factors in order to uncover relationships in the data. While quite popular, the common implementations of PARAFAC are single server solutions that do not scale well to very large datasets. To address this limitation, a Parallel PARAFAC algorithm has been designed and implemented in C using MPI. The end-to-end pipeline includes a parallel read of the input data from a file, the execution of the parallel algorithm, and concludes with a parallel write of the results to a file. The implementation has been evaluated using a strong scaling study on an IBM Blue Gene/Q supercomputer. The compute time, as well as the communication, file read, and file write bandwidths were each captured across multiple scenarios to evaluate the overall system performance and scalability. Results indicate the implementation scales well—with a 128x increase in the number of parallel processes, the system executed 200x faster. Further, the communication time at its peak accounted for only 12% of the total processing time, indicating the implementation is currently CPU bound and thus should continue to scale well across more and more nodes.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent programing – *parallel programming.*

G.1.0 [**Numerical Analysis**]: General – *parallel algorithms.*

General Terms
Algorithms, Performance, Design

Keywords
Parallel factor analysis; tensor; multiway data analysis; parallel computing; MPI; scalability testing; performance testing

1. INTRODUCTION
Parallel Factor Analysis (PARAFAC) [1][2] is a powerful technique for uncovering hidden relationships in datasets that have three or more dimensions. Specifically, PARAFAC is used to decompose multidimensional datasets into principal factors to uncover relationships between variables. Three-or-more dimensional datasets are typically referred to as 'tensors' or 'multiway datasets,' and are used in many scientific disciplines including chemometrics (e.g., for identifying the chemical components within a mixture [3]), social network analysis (e.g., for identifying hidden structures within a network [4]), and neuroscience (e.g., for studying the effects of drug treatments on brain activity [5]), to name a few.

An industrial example of a 3-way tensor is a set of sensor readings generated from industrial equipment over time, in which one dimension represents the physical units (e.g., gas turbines), a second dimension represents the sensors on each unit (e.g., temperature sensors, pressure sensors, air flow sensors), and the third dimension is time. In this example, PARAFAC could be used to characterize relationships within and across the dimensions, which in turn could be used to identify normal and abnormal functioning of the equipment.

The canonical approach to PARAFAC decomposition is to use an Alternating Least Squares (ALS) algorithm that performs a sequence of matrix operations to decompose a tensor into its principal factors (plus some residual error). The objective of PARAFAC-ALS is to iteratively improve the principal factor decomposition of a tensor in order to minimize the residual error, and thus requires the iterative execution of the matrix operations until either a convergence or other stopping criteria is met [6].

The two most commonly used tensor analysis packages are both written in MATLAB [7][8], and are limited in their ability to scale to large datasets. Now that we are in the era of "Big Data," datasets are growing ever larger and a greater amount of data-driven analysis is being enabled within almost every scientific discipline [9][10]. Thus, the need to be able to run PARAFAC and similar analytic techniques at scale is becoming increasingly important. To address the need for a PARAFAC implementation able to scale to very large, dense datasets, a Message Passing Interface (MPI)-based parallel implementation of PARAFAC has been designed and implemented to decompose three-dimensional tensors. This implementation includes the ability to: (1) read in large datasets using parallel I/O and then formulate that data into a tensor distributed across a collection of parallel processes, (2) execute a parallel version of the PARAFAC-ALS algorithm, and finally, (3) use parallel I/O to write the principal factor matrices to files. This MPI-based Parallel PARAFAC implementation has been designed and implemented such that the full tensor is never collected on a single node, allowing it to analyze tensors that are too large to fit in the main memory of a single machine. This implementation has been tested on an IBM Blue Gene/Q supercomputer to evaluate its runtime performance and scalability.

This paper is organized as follows: Section 2 summarizes prior art in the space of parallel implementations of PARAFAC. Section 3 outlines the sequential PARAFAC-ALS algorithm. Section 4 details the design of our Parallel PARAFAC algorithm, and Section 5 describes the MPI-based implementation. Section 6 presents the results of performance and scalability testing on a dense dataset, and Section 7 outlines conclusions and future work.

2. PRIOR ART

The first documented explorations of parallel implementations of PARAFAC appeared independently in 2009. In an HPC minisymposium, Sears et al. [11] presented efforts to develop an MPI-based parallel implementation of PARAFAC. However, they explicitly stated their solution was implemented for sparse matrices, which limits its applicability to only certain multiway datasets. Further, they did not provide any details on their parallelism approach nor is their source code publicly available, preventing a meaningful evaluation of their approach.

Around the same time Zhang et al. [12] published a detailed description of a PARAFAC parallelization approach applied to global climate data (they focus on Nonnegative Tensor Factorization, an extension of PARAFAC such that all of the factors are nonnegative). Their approach relies on a few observations of the inherent parallel properties of the matrix operations that were rediscovered during this research effort. While we use similar properties as that of Zhang et al., their use of a gradient descent algorithm results in different operations to perform the decomposition. Their use of C++ and MPI to implement the parallel algorithm makes this work similar to our own research, however, they limited their scalability testing to 10 parallel processes and achieved only a modest sub-linear peak speedup of 6.8x (comparing runtime on 8 nodes vs. 1 node) [12].

Phan and Cichocki [13] also explored parallel implementations of PARAFAC, including developing a grid-based approach to parallelizing the algorithm that divides a tensor into an arbitrary number of sub-tensors split across all dimensions. Their approach allows a finer-grained level of parallelism than our design; however, their approach introduces some error with more parallel processes, resulting in a tradeoff between runtime and accuracy.

Beyond the above efforts, recent research in large-scale, parallel PARAFAC implementations has originated primarily from a group at Carnegie Mellon University. Starting in 2012, Kang et al. [14] describe GigaTensor, which appears to be the first Hadoop-based implementation of PARAFAC. GigaTensor is implemented using three separate MapReduce jobs executed in sequence for each iteration of the PARAFAC algorithm. While interesting, GigaTensor's parallelism approach requires the passing of the tensor elements between the Map and Reduce stages, and is efficient only because they assume the tensor is sparse.

Papalexakis et al. [15] describe ParCube, which generates many small tensors using random sampling of a single large tensor, parallelizes the analysis of each small tensor, and finally combines the resulting small decompositions. With this approach, the actual PARAFAC-ALS algorithm is executed in an embarrassingly parallel manner, and only at the completion of the small decompositions are the results synthesized into a single solution. Thus, ParCube is performing an approximation of a traditional PARAFAC decomposition. ParCube is implemented in MATLAB, and utilizes one of the established tensor toolboxes [8] for the PARAFAC algorithm implementation.

In a similar vein to ParCube, Sidiropoulos et al. [16] propose a parallel tensor decomposition approach they call PARACOMP, which operates by generating a large number of random compressed sub-tensors from a single large tensor, decomposing each in an embarrassingly parallel manner and then aggregating the results, again approximating a complete decomposition. It appears that an implementation of PARACOMP is underway.

Outside of the research efforts at Carnegie Mellon, Zhe et al. [17], a combined team from Purdue University and IBM's T.J. Watson Research Center, developed DinTucker (Distributed Infinite Tucker), which utilizes an alternate algorithm for tensor decomposition called Tucker [18]. While not based on PARAFAC, DinTucker is still of interest because it addresses the core challenge of decomposing a large tensor in a highly parallel manner. DinTucker was built on top of Hadoop, and uses a distributed stochastic gradient descent algorithm they built as stages within MapReduce jobs in a vein similar to that of GigaTensor.

Overall, the recent parallel tensor decomposition approaches address the challenge of decomposing a large tensor either, in the case of ParCube and PARACOMP, by approximating the solution through the creation of many small tensors and decomposing them in an embarrassingly parallel manner, or, in the case of GigaTensor and DinTucker, by utilizing Hadoop with its consequent performance overheads.

Our approach returns to the early work of parallel PARAFAC decomposition through MPI, but attempts to (a) achieve significantly better performance than what has been reported, (b) without the assumption of a sparse tensor limiting the applicability of our solution.

3. PARAFAC-ALS ALGORITHM

As stated previously, an alternating least squares algorithm is traditionally used to decompose a tensor X into R factors, with each dimension of X decomposed into R distinct vectors, as shown in Figure 1.

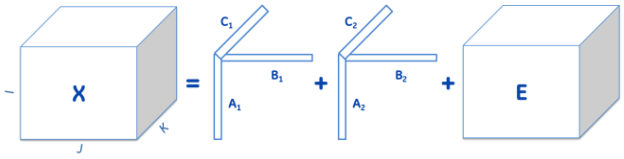

Figure 1. 3D tensor decomposition into the sum of two rank-one tensors and a tensor of residual error terms

For a 3-way tensor $X \in \mathbb{R}^{IxJxK}$, the tensor is decomposed into matrices $A \in \mathbb{R}^{IxR}$, $B \in \mathbb{R}^{JxR}$, and $C \in \mathbb{R}^{KxR}$. Figure 1 can be alternately represented by the equation:

$$x_{ijk} = \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr} + e_{ijk} \quad (1)$$

The objective of PARAFAC-ALS is to find A, B, and C matrices that minimize the total residual error in Equation 1.

PARAFAC-ALS includes a sequence of matrix operations, including the Khatri-Rao matrix product, matricization of X (projection of a 3D tensor onto a 2D plane), matrix-matrix multiplication, matrix transposition, and matrix inversion. Brief descriptions of the Khatri-Rao product, matricization, and the overall sequential ALS algorithm are provided next.

3.1 Khatri-Rao Product

The Khatri-Rao product is represented by the symbol ⊙ and takes two matrices with the same number of columns and performs column-wise multiplication of each row in the first matrix with each column in the second. For matrices A ∈ $\mathbb{R}^{nxk}$ and B ∈ $\mathbb{R}^{mxk}$, the Khatri-Rao product A⊙B produces an (n*m)xk matrix P, such that the first column of P equals:

$$P[:,1] = [a_{1,1}b_{1,1}\ a_{2,1}b_{1,1}\ ..\ a_{n,1}b_{1,1}\ \ldots\ldots\ a_{n,1}b_{1,1}\ a_{n,1}b_{2,1}\ ..\ a_{n,1}b_{m,1}]^T \quad (2)$$

The remaining columns of the Khatri-Rao product are derived in the exact same pattern as the first column, with the second column of the product calculated as the product of the second column of matrix A multiplied by the second column of matrix B in the format shown in Equation 2.

3.2 Matricization

Matricization is the process of unfolding a tensor into a 2D matrix, with different modes representing the different dimensions of unfolding. For example, $X_{(1)}$ represents the unfolding of tensor X ∈ $\mathbb{R}^{IxJxK}$ in the 1st dimension, creating a matrix ∈ $\mathbb{R}^{IxJK}$. For the other modes, $X_{(2)} \in \mathbb{R}^{JxIK}$ and $X_{(3)} \in \mathbb{R}^{KxIJ}$.

Figure 2 shows a representative example of a 3rd order tensor of dimensions 4x3x2 unfolded in all three modes.

$$A = \left[\begin{array}{ccc|ccc} 1 & 5 & 9 & 13 & 17 & 21 \\ 2 & 6 & 10 & 14 & 18 & 22 \\ 3 & 7 & 11 & 15 & 19 & 23 \\ 4 & 8 & 12 & 16 & 20 & 24 \end{array}\right]$$

$$A_{(1)} = \left[\begin{array}{ccc|ccc|ccc|ccc} 1 & 5 & 9 & 13 & 17 & 21 \\ 2 & 6 & 10 & 14 & 18 & 22 \\ 3 & 7 & 11 & 15 & 19 & 23 \\ 4 & 8 & 12 & 16 & 20 & 24 \end{array}\right]$$

$$A_{(2)} = \left[\begin{array}{cccc|cccc|cccc} 1 & 2 & 3 & 4 & 13 & 14 & 15 & 16 \\ 5 & 6 & 7 & 8 & 17 & 18 & 19 & 20 \\ 9 & 10 & 11 & 12 & 21 & 22 & 23 & 24 \end{array}\right]$$

$$A_{(3)} = \left[\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \end{array}\right]$$

Figure 2. Example 3D tensor A ∈ $\mathbb{R}^{4x3x2}$ unfolded in three modes, $A_{(1)}$, $A_{(2)}$, and $A_{(3)}$. The lines within the matrices $A_{(i)}$ highlight where the IxJx1 and IxJx2 matrices in tensor A are located within the 2D matrices

3.3 Alternating Least Squares Algorithm

The PARAFAC alternating least squares algorithm for tensor decomposition is shown in Figure 3. The algorithm iterates until it either: (a) converges upon A, B, and C matrices that no longer improve the residual error above a threshold ε, or (b) has executed for a maximum number of iterations.

4. PARALLEL PARAFAC

Through this effort, we have designed and developed an approach to parallelizing the PARAFAC-ALS algorithm that does not rely upon sampling or approximations of the canonical ALS algorithm shown in Figure 3. Further, our approach implements the PARAFAC-ALS algorithm in a highly scalable manner, limited only by the largest dimension of the tensor.

To parallelize the algorithm, we first make the following assumptions: dimensions I and J of matrix X are expected to be relatively small (on the order of 10's → 1,000's), while dimension K is expected to be very large (on the order of 10K's → 1MM's or more). This assumption reflects use cases wherein dimensions I and J represent physical assets or entities, such as industrial equipment and sensors, respectively, and dimension K represents an unbounded variable such as time. Given these assumptions, it is most practical to divide the tensor X into *r* distinct subsets by splitting the tensor across the K dimension, resulting in IxJx(K/*r*) sub-tensors located on each rank (parallel process) *r*. A visual representation of this split can be seen in Figure 4.

Random initialization of B and C
While (old error – new error) > ε and have not exceeded maximum number of iterations do
/* fix B and C, solve for A */
$Z = C \odot B$
$A = X_{(1)}Z(Z^TZ)^{-1}$
/* fix A and C, solve for B */
$Z = C \odot A$
$B = X_{(2)}Z(Z^TZ)^{-1}$
/* fix A and B, solve for C */
$Z = B \odot A$
$C = X_{(3)}Z(Z^TZ)^{-1}$
calculate new residual error
end while

Figure 3. PARAFAC Alternating Least Squares algorithm

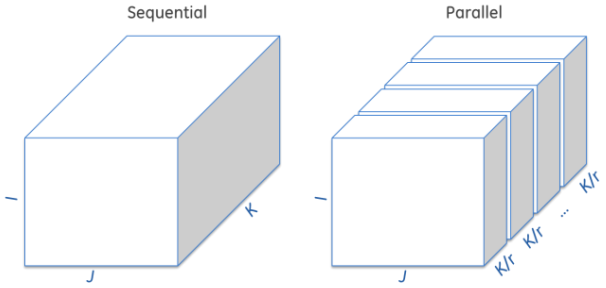

Figure 4. Single tensor for single-server sequential processing, and split into *r* blocks for parallel processing

Further, we assume that because each process will have a complete set of IxJ matrices (K/*r* of them, to be precise), each process will require a complete A and B matrix, and a K/*r* subset of the C matrix, C_i. Note that in practice, any dimension can be viewed as the 'K' dimension for splitting the tensor across the parallel processes. However, it makes the most sense to always split on the largest dimension, to maximize the opportunity for parallelism.

Given the above assumptions, we then designed an approach to solve for matrices A, B, and C in parallel. The approach is built upon three theorems of the transformations used in the algorithm.

4.1 Theorem 1: Khatri-Rao product decomposition

When solving for A, if a process has only a subset of the C matrix but has the full B matrix available, then the Khatri-Rao product is fully decomposable. In practice, one can apply the Khatri-Rao product to every row in the C matrix in parallel and assemble the correct output. For example, assuming the C matrix is divided into two parts, this can be expressed as:

$$C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} \quad (3)$$

$$\begin{aligned} Z_1 &= C_1 \odot B \\ Z_2 &= C_2 \odot B \end{aligned} \quad (4)$$

$$Z = \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} \equiv C \odot B \quad (5)$$

From Equation 4, if the Khatri-Rao product is parallelized for each C_i block of C, then each process will generate a distinct subset of the Z matrix, Z_i.

Proof:

For $B \in \mathbb{R}^{JxR}$ and $C \in \mathbb{R}^{KxR}$,

if $Z = C \odot B$

then $\forall i \in [1, I], j \in [1, J], k \in [1, K]$

$$Z[i * K + j, k] = C[i, j] * B[j, k]. \square \quad (6)$$

Therefore, each row of Z (indexed by i*K + j) can be solved for with only the i^{th} row of C and the j^{th} row of B.

After the Khatri-Rao product, when solving for A the next operation in the algorithm requires solving for $X_{(1)}Z(ZTZ)^{-1}$. The remaining two theorems give insight into how this operation can be parallelized using the Z_i subsets of Z available within each parallel process.

4.2 Theorem 2: Z^TZ sum of parts

If the Z matrix is divided into *r* blocks, then the Z^TZ operation is equal to the sum of the $Z_i{}^TZ_i$ of the parts. For our 2-process example, this can be represented as:

$$Z^TZ = Z_1{}^TZ_1 + Z_2{}^TZ_2 \quad (7)$$

Proof:

For $Z \in \mathbb{R}^{IxJ}$,

$$Z^TZ[i, j] = \sum_{k=1}^{W} Z[i, k] * Z[k, j] \quad (8)$$

$$= \sum_{k=1}^{n-1} Z[i, k] * Z[k, j] + \sum_{k=n}^{W} Z[i, k] * Z[k, j]. \square \quad (9)$$

Therefore, if Z is divided into *r* blocks, then Z^TZ can be derived from the sum of the $Z_i{}^TZ_i$ of the parts. Consequently, we can perform parallel $Z_i{}^TZ_i$ operations and sum the results across all of the processes to generate a complete Z^TZ matrix. This is a particularly attractive property because while the full Z matrix is of dimension JKxR (when solving for A) and is expected to be extremely large, the Z^TZ matrix is extremely small; of dimension RxR (where R is typically between 2 and 4). Therefore, the collective operation is performed over an extremely small matrix.

4.3 Theorem 3: $X_{(1)}Z$ sum of parts

If the Z matrix is divided into blocks of equal dimensions, then the $X_{(1)}Z$ operation is equal to the sum of the $X_{(1)r}Z_i$ of the parts. For our 2-process example, this can be expressed as:

$$X_{(1)} = [X_{(1)1} \quad X_{(1)2}] \quad (10)$$

$$X_{(1)}Z = X_{(1)1}Z_1 + X_{(1)2}Z_2 \quad (11)$$

Proof:

If we split $X_{(1)}$ and Z into *r* blocks as shown in Figure 4, $X_{(1)}Z$ can be expressed as:

$$X_{(1)}Z = [X_{(1)1} \quad X_{(1)2} \quad \dots \quad X_{(1)K}] \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_K \end{bmatrix} \quad (12)$$

$$X_{(1)}Z = \sum_{k=1}^{K} X_{(1)}Z_k. \square \quad (13)$$

As in the previous theorem, this also requires a collective operation to sum for the complete $X_{(1)}Z$ matrix, but this matrix is of dimension IxR (when solving for A), which is expected to be reasonably small.

From these three theorems, we conclude that we can completely parallelize the derivation of decomposition matrix A with a collection of large matrix operations performed independently in each process, and the transmission of a relatively small amount of data (*r* RxR and IxR matrices).

The calculation of matrix B follows an identical pattern to that of matrix A, and so the same steps can be performed to solve for B in parallel. When solving for B, we again require a collective operation to sum the Z^TZ RxR matrix, but because the B matrix is likely of a different dimension from the A matrix, in this instance we will require a collective operation to sum a JxR matrix.

The same parallel approach cannot be taken when solving for C, however, because each parallel process contains only a portion of the matrices along the K dimension of tensor X. Fortuitously, solving for C is even simpler than solving for A and B, because C can be updated independently within each process based on its local data and does not require any collective operations. This is a result of the matricization of X on the 3rd mode, which causes each block of X_i to be divided into blocks of rows instead of columns (as previously highlighted in Figure 2). Therefore, the parallel operations can be expressed as:

$$C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} \quad (14)$$

$$X_{(3)} = \begin{bmatrix} X_{(3)1} \\ X_{(3)2} \end{bmatrix} \quad (15)$$

$$C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} X_{(3)1} \\ X_{(3)2} \end{bmatrix} Z(Z^TZ)^{-1} \quad (16)$$

$$C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} X_{(3)1}Z(Z^TZ)^{-1} \\ X_{(3)2}Z(Z^TZ)^{-1} \end{bmatrix} \quad (17)$$

As shown in Equation 12, the full C matrix can be updated by updating the individual C_i components within each parallel process. These updates can occur independently on each node because the Z matrix in this instance is dependent on matrices A and B only, which are identical on each node and therefore no information passing is required to solve for each local C_i component.

4.4 Parallel Alternating Least Squares Algorithm

Figure 5 shows the parallel ALS algorithm we designed to decompose a 3rd-order tensor. The operations ensure that each parallel process has the same local A and B matrix (which are both small), and their own local component C_i of the extremely large C matrix. Two collective operations occur when solving for matrices A and B, and then a collective operation is performed to calculate the total residual error at the end of each iteration.

This parallel algorithm is particularly attractive because it reduces the dimensions of many large-scale matrix operations (such as matrix multiplication, which is of order $O(n^3)$), and so a reduction in the size of *n* on each node (by adding more nodes) can result in super-linear speedups. The limiting factor of the speedup will come from the overhead induced through the message passing required in the parallel implementation. A critical question is whether reductions in the compute time will be overshadowed by commensurate increases in the communication overhead from more and more nodes participating in the parallel computation.

Random initialization of B and C_i on each node
while (sum(old error) – sum(new error)) > ε and not exceeded maximum number of iterations do
/* fix B and C, solve for A */
$Z_i = C_i \odot B$
solve for $Z_i^T Z_i$
$Z^T Z = \sum_{i=0}^{r} Z_i^T Z_i$ // MPI all reduce matrix sum
solve for $X_{(1)i} Z_i$
$X_{(1)} Z = \sum_{i=0}^{r} X_{(1)i} Z_i$ // MPI all reduce matrix sum
$A = X_{(1)} Z (Z^T Z)^{-1}$
/* fix A and C, solve for B */
$Z_i = C_i \odot A$
solve for $Z_i^T Z_i$
$Z^T Z = \sum_{i=0}^{r} Z_i^T Z_i$ // MPI all reduce matrix sum
solve for $X_{(2)i} Z_i$
$X_{(2)} Z = \sum_{i=0}^{r} X_{(2)i} Z_i$ // MPI all reduce matrix sum
$B = X_{(2)} Z (Z^T Z)^{-1}$
/* fix A and B, solve for C */
$Z = B \odot A$
$C_i = X_{(3)i} Z (Z^T Z)^{-1}$
calculate new error local e_i
$new\ e = \sum_{i=0}^{r} e_i$ // MPI all reduce error sum
end while

Figure 5. Parallel PARAFAC Alternating Least Squares algorithm for 3rd-order tensors

5. IMPLEMENTATION

The Parallel PARAFAC-ALS algorithm has been implemented in C using MPI compiled with the IBM XL C compiler [19]. The code first initializes MPI for the user-defined number of ranks (parallel processes) *r*, and then each rank allocates memory for the A[I][R], B[J][R], and C_i[K/*r*][R] matrices, and randomly initializes its local copies of the B and C_i matrices. A is not initialized because the first step in the algorithm will derive A from the B and C_i matrices. The code then allocates memory for the local block of the tensor X[I][J][K/*r*], and performs a parallel block read of the tensor from a single input file using the MPI_File_read_at_all command, such that each rank reads in an equal-sized portion of the file into its local tensor object. The system captures the max file read time across the ranks, and this time is used to calculate the file read bandwidth.

Once the tensor has been fully loaded across all ranks, the Parallel PARAFAC-ALS algorithm can begin. A 'while' loop is used, inside of which A is first solved for, and then B, and then each C_i, as outlined in the algorithm in Figure 5. As mentioned previously, these solves require the use of the Khatri-Rao product, matrix transpose, inverse, multiply, and unfolded matrix multiply, as described in the previous section. Solving for matrices A and B requires two matrix synchronization (sum) operations, which execute via the MPI_Allreduce function for each element of the two matrices. At the end of each iteration the local error is calculated for each rank, and then the cumulative error is summed across all ranks using another MPI_Allreduce sum operation. If the change in total error from one run to the next is less than the minimum error change ε, or the loop has executed more than N times, the computation completes. In this implementation, $\varepsilon=10^{-4}$ and N=50, but both values are easily changed.

During the code execution, the compute time is captured from before the while loop begins to after the while loop ends (minus the communication time), and the MPI communication time is captured as the time spent performing each of the many MPI_Allreduce operations. The compute time is reported as part of the final results, and the communication time is used to calculate the communication bandwidth.

At the completion of the algorithm, the final step is to write the three matrices to three separate files. Because each rank contains an identical copy of the A and B matrices there is no need for parallel writes, and the C command 'fprintf' is used to write these two matrices to files, with rank 0 writing the A matrix and rank 1 writing the B matrix. As described previously, these matrices are both quite small and take a negligible amount of time to write. In these experiments the A matrix is 180 bytes and the B matrix is 5.2KB.

The C matrix is more complicated, as it is considerably larger and divided across the *r* ranks. In these experiments the compact C matrix file is 0.5MB. The code uses two methods to perform parallel writes of the C matrix: one approach uses a compact write of the matrix to a single contiguous file, and the other approach writes the C matrix in 2KB blocks per C_i matrix. The code dynamically sets the write block size as some multiple of 2KB based on the size of the C_i matrix. In most instances one 2KB block is sufficient, but in one instance the C_i matrix requires two 2KB blocks. The command MPI_File_write_at_all is used to execute the parallel writes, and the code separately captures the time spent performing the compact and block file writes. These two write times are each used to calculate the file write bandwidths.

5.1 Computational Complexity

Through this algorithm, tensor $X \in \mathbb{R}^{IxJxK}$ is decomposed into matrices $A \in \mathbb{R}^{IxR}$, $B \in \mathbb{R}^{JxR}$, and $C \in \mathbb{R}^{KxR}$. The Big-O computational complexity of traditional implementations of the matrix operations used in the Parallel PARAFAC algorithm assuming all matrices are NxN are in Table 1.

Table 1. Computational complexity of matrix operations

Operation	Symbol	O(·)
Khatri-Rao product	$\odot$	$O(N^3)$
matrix multiply	$*$	$O(N^3)$
matrix inverse	$^{-1}$	$O(N^3)$

From the Big-O of the general operations defined in Table 1, we can identify the Big-O computational complexity for each operation in the Parallel PARAFAC algorithm. The Big-O complexity and the dimensions of the resulting matrices are found in Table 2.

As can be seen in Table 2, the majority of the operations are dependent on the size of the tensor split, K/r, indicating that the more parallel processes *r* we can use, the faster the overall calculation will execute. The few operations that do not change based on the size of the tensor split are of the order R^3, I*R*R and J*R*R, which are each expected to be reasonably small given the assumptions outlined prior to the development of the parallel algorithm, and therefore are not expected to dominate the calculations.

Table 2. Computational complexity of matrix operations

Operation	O(·)	Output Matrix Dimensions
$Z_i = C_i \odot B$	K/r * J * R	(K/r*J) x R
$Z_i^T Z_i$	(K/r * J) * R^2	R x R
$X_{(1)i} Z_i$	I * (K/r * J) * R	I x R
$(Z^T Z)^{-1}$	R^3	R x R
$A = X_{(1)} Z (Z^T Z)^{-1}$	I * R^2	I x R
$Z_i = C_i \odot A$	K/r * I * R	(K/r*I) x R
$Z_i^T Z_i$	(K/r * I) * R^2	R x R
$X_{(2)i} Z_i$	J * (K/r * I) * R	J x R
$(Z^T Z)^{-1}$	R^3	R x R
$B = X_{(2)} Z (Z^T Z)^{-1}$	J * R^2	J x R
$Z = B \odot A$	J * I * R	(J*I) x R
$Z^T Z$	(J * I) * R^2	R x R
$X_{(3)i} Z$	K/r * (J * I) * R	K/r * R
$(Z^T Z)^{-1}$	R^3	R x R
$C_i = X_{(3)i} Z (Z^T Z)^{-1}$	K/r * R^2	K/r * R

Overall, the complexity analysis confirms that the algorithm should scale well with more parallelism. A central question explored in the performance testing is how the message passing impacts the overall performance as the number of parallel tasks grows. A plot comparing the upper bound of the runtime calculated from the Big-O analysis to the actual runtime is provided in the next section.

6. PERFORMANCE TESTING

While PARAFAC-ALS is usually executed until a convergence criteria is met (e.g., the change in error is less than some constant ε), for performance testing we required the system to execute exactly 50 iterations of the algorithm. This ensured that the performance comparison between different numbers of nodes and ranks per core were comparable. For testing purposes, we have assumed the tensor will be decomposed into R=2 factors, but this is easily modified.

6.1 Tensor Dataset

A tensor of dimension 10x295x32,768 was generated for performance testing, resulting in a 0.98 GB comma-separated text file. The 10x295 dimensions arose from a small dataset that was used for initial experimentation, which came from a biomedical study of 295 patients with 10 biometrics captured per patient. The tensor was generated to simulate this same size of population and biometrics, with the biometrics simulated over 32,768 seconds (over 9 hours). This number represents the maximum number of parallel processes that can be used. This number was chosen because it would allow us to run up to 4 ranks/core across 512 nodes in parallel.

6.2 AMOS Blue Gene/Q

The MPI-based Parallel PARAFAC implementation was run on AMOS (Advanced Multiprocessing Optimized System), an IBM Blue Gene/Q supercomputer that is a part of the Center for Computational Innovations at Rensselaer Polytechnic Institute (RPI). AMOS is named in homage to Amos Eaton, one of the founders and first teachers at RPI.

In the November 2014 Top500 ranking of supercomputers (www.Top500.org), AMOS ranked #43 in the world. AMOS is comprised of 5 Blue Gene/Q racks and contains 5,120 nodes of 16-core 1.6 GHz A2 processors for a total of 81,290 cores, with 81,920 GB of RAM. It also contains 160 nodes for I/O. AMOS uses a 5D torus network with a 56 Gbit/s FDR Infiniband backbone, and includes 32 Intel servers for disk storage with 24TB disk per server, and can achieve a peak I/O bandwidth of 5 to 24GB/sec.

A strong scaling study was performed on AMOS to understand how the performance of the algorithm for a fixed problem size (in this case, tensor size) changes with the number of parallel processes. The following metrics were captured during the simulation: file read time, total computation time, MPI communication time, block file write time, and compact file write time. The algorithm was run with as few as 128 parallel processes across 8 nodes to 16,384 parallel processes across 256 nodes. At this scale, each process is managing an IxJx2 tensor.

To understand how the performance was impacted by the number of nodes and ranks/core, we executed the algorithm on a total of 18 unique configurations using 6 different numbers of nodes (8, 16, 32, 64, 128, and 256) and 3 scenarios of ranks/core (1, 2, and 4). Each configuration was run 3 times for a total of 54 runs, and the median of each metric was used for plotting and analysis in the figures and tables below. The median was chosen to limit the impact of outlier readings, which occurred solely in a small percentage of the file read and write measurements. It is natural that outliers may occur for file I/O operation times, as these can be noticeably impacted by other jobs on the supercomputer.

6.3 Compute Time & MPI Communication Bandwidth

The compute time is defined as the amount of time spent performing the many matrix operations. Table 3 shows the compute time versus the number of nodes for the three different ranks/core scenarios.

Table 3. Compute time vs. number of nodes for 1, 2, and 4 ranks/core

Num. Nodes	*1 rank/core (sec)*	*2 ranks/core (sec)*	*4 ranks/core (sec)*
8	1,014.85	380.94	193.02
16	353.15	148.74	82.95
32	137.95	63.96	38.05
64	59.30	29.40	18.27
128	27.17	14.00	8.95
256	12.94	6.87	4.45

The columns in Table 3 have been plotted versus the number of nodes. Figure 6 shows all values and Figure 7 shows only the values between 64 and 256 nodes.

From Table 3, Figure 6, and Figure 7, we observe very consistent behavior across the three lines. The total execution time drops dramatically when increasing from 8 to 32 nodes, and then drops less and less substantially when increasing from 32 to 256 nodes. This demonstrates that doubling the number of nodes has less of

an impact the more nodes we are already using, i.e., we are witnessing a 'law of diminishing returns' effect as we approach the maximum possible number of parallel processes.

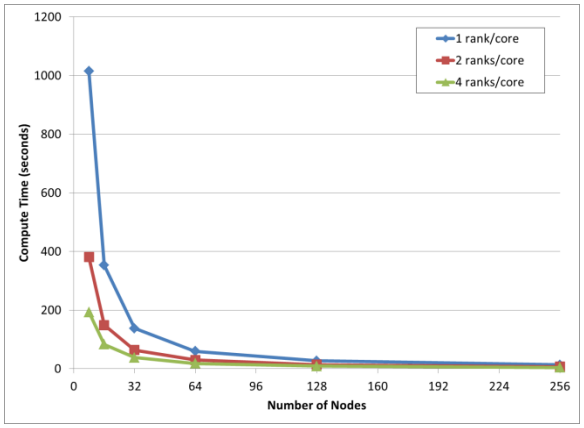

Figure 6. Compute Time (sec) vs. Number of Nodes

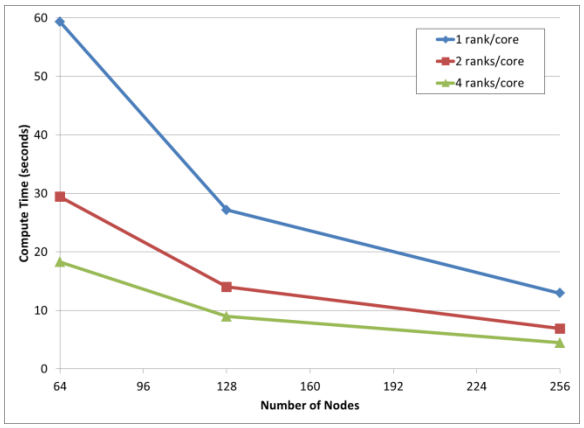

Figure 7. Compute Time (sec) vs. Number of Nodes for 64 to 256 nodes only

Further, comparing one line to the next we see an average 53% decrease in compute time moving from 1 rank/core to 2 ranks/core, and a smaller average 41% decrease moving from 2 ranks/core to 4 ranks/core. This also highlights the law of diminishing returns effect when doubling the number of ranks/core for a fixed number of nodes, but may also be a result of resource contention as we overcommit the cores.

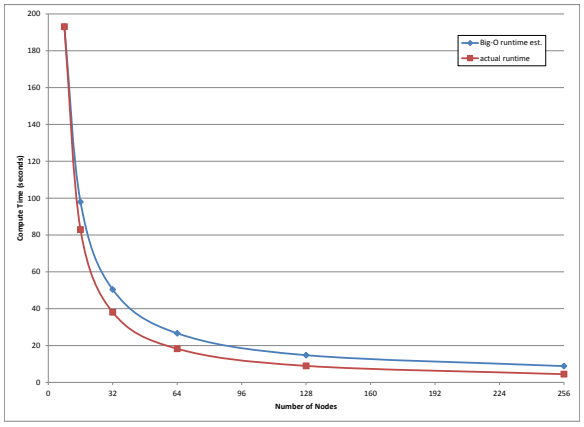

Figure 8. Big-O Upper Bound Runtime and Actual Runtime (sec) vs. Number of Nodes for 4 ranks/core

A plot comparing the upper bound of the runtime calculated from the Big-O analysis to the actual runtime is shown in Figure 8. Clearly, the actual runtime follows a very consistent pattern to the Big-O analysis, as expected.

The MPI communication bandwidth is the amount of data transmitted per second during the MPI reduce operations. Table 4 shows the communication bandwidth versus the number of processes for the three ranks/core scenarios.

Table 4. MPI communication bandwidth vs. number of nodes for 1, 2, and 4 ranks/core

Num. Nodes	*1 rank/core (GB/sec)*	*2 ranks/core (GB/sec)*	*4 ranks/core (GB/sec)*
8	0.07	0.05	0.03
16	0.20	0.19	0.13
32	0.51	0.54	0.44
64	1.02	1.24	1.05
128	2.05	2.69	2.48
256	4.49	5.89	5.99

Figure 9 plots the MPI communication bandwidth versus the number of nodes from Table 4.

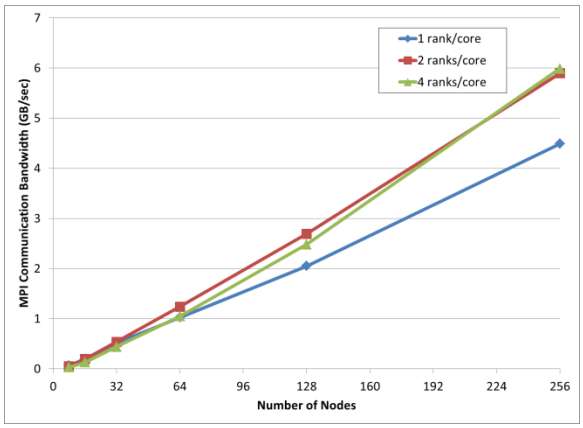

Figure 9. MPI Communication Bandwidth (GB/sec) vs. Number of Nodes

In Figure 9 we observe a consistent pattern across all three scenarios. The communication bandwidth is quite low initially, when there are only 8 nodes used. Above 8 nodes, we see an approximately linear increase in the bandwidth. This indicates that we are not saturating the communication backbone, and that as we transmit more and more data (due to more and more parallel processes executing), we are able to use more and more network bandwidth. While the bandwidth across the three scenarios starts out similar, as the number of nodes increases the 1 rank/core bandwidth grows less than the other two, most likely because it benefits the least from the fastest channel—inter-process communication within a node.

The total processing time, defined as the sum of the compute and MPI communication time, is shown in Table 5.

Table 5. Total processing time vs. number of nodes for 1, 2, and 4 ranks/core

Num. Nodes	*1 rank/core (sec)*	*2 ranks/core (sec)*	*4 ranks/core (sec)*
8	1,015.30	382.05	196.81
16	353.45	149.35	84.75
32	138.18	64.40	39.12
64	59.53	29.78	19.17
128	27.40	14.35	9.71
256	13.15	7.19	5.08

Figure 10 plots the total processing time versus the number of nodes. We see very consistent behavior between this and Figure 6, indicating that the processing time is dominated by the compute time. In fact, at its peak (for 256 nodes, 4 ranks/core), the communication time only accounts for 12% of the total processing time. (The total communication time can be calculated as the difference between Table 5 and Table 3, and thus is not included in a separate table here.)

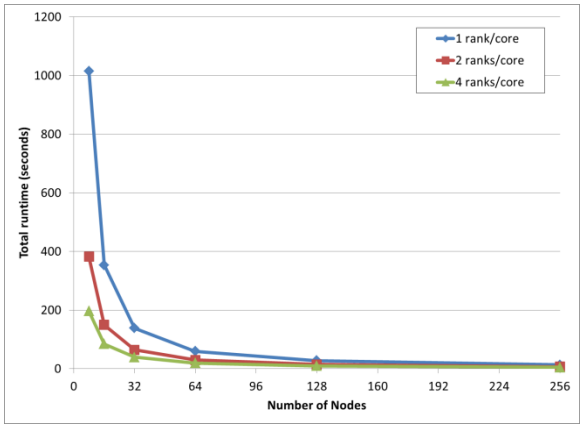

Figure 10. Total Processing Time (sec) vs. Number of Nodes

For the slowest runtime of 1,015.30 sec we had 8 nodes with 1 rank/core, resulting in 128 (2^7) parallel tasks. For the fastest runtime at 5.08 sec we had 256 nodes with 4 ranks/core, resulting in 16,384 (2^{14}) parallel tasks. The ratio of the number of parallel tasks between the best and worst runtimes was 128, and the ratio of the associated runtimes was 200. Therefore, for 128x the number of processes we achieved a 200x runtime performance improvement. As mentioned previously, this super-linear speedup was expected, because operations such as matrix multiply are of $O(n^3)$, and so any reduction to the order n of the matrices on a single node will result in a greater than n reduction in the computation time.

While the super-linear compute time speedup could have been hampered by commensurate increases in the communication time, from the consistent growth in the MPI communication bandwidth indicating a lack of network saturation and the small percentage of time attributed to MPI communication within the total processing time, we conclude that the system is CPU bound at the current scale (up to 256 nodes, 214 processes). If we were able to increase the number of parallel processes even further, it is expected that eventually we would saturate the network and see the communication bandwidth level off. At that point the communication time would come to dominate the total processing time indicating that the system had become network bound, but this did not become an issue at the current scale.

6.4 File Read and Write Bandwidth

As described previously, the file read bandwidth is calculated as the tensor file size divided by the time to load the file by the parallel processes. Table 6 shows the read bandwidth and Figure 11 plots the same.

Table 6. Tensor file read bandwidth vs. number of nodes for 1, 2, and 4 ranks/core

Num. Nodes	*1 rank/core (GB/sec)*	*2 ranks/core (GB/sec)*	*4 ranks/core (GB/sec)*
8	0.49	0.43	0.24
16	0.26	0.25	0.25
32	0.46	0.44	0.85
64	1.86	1.20	1.17
128	2.99	2.14	1.43
256	0.28	1.57	0.99

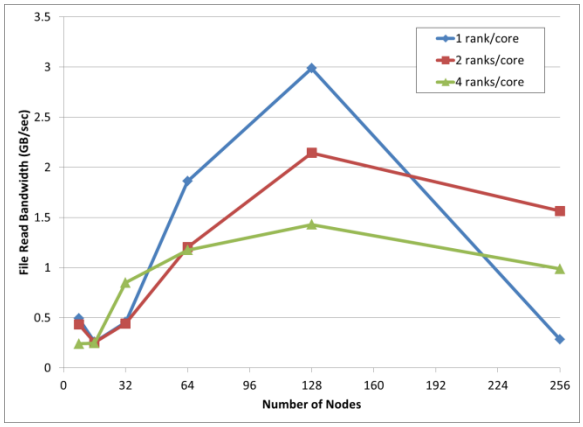

Figure 11. Tensor File Read Bandwidth (GB/sec) vs. Number of Nodes

From the above table and figure, we see that the three scenarios exhibit very similar behavior. The read bandwidth starts out quite low on 8 nodes, and actually gets a bit worse for all three scenarios as we scale to 16 nodes. However, the bandwidth then improves as we grow from 16 to 128 nodes, where the bandwidth peaks for all three scenarios. Moving from 128 to 256 nodes, the bandwidth experiences a drop, which is significant in all three cases but most dramatic for the 1 rank/core scenario. The 1 rank/core scenario has the best bandwidth at 128 nodes (2.99 GB/sec), and the worse at 256 nodes (0.28 GB/sec).

The consistent behavior across all three scenarios implies that the MPI read performance is contingent less on the number of processes, and more on the number of nodes involved in the read operations, and that 128 appears to be the optimal number of nodes for reading the tensor file. It appears that for the 0.98 GB file being read, above 128 nodes MPI file metadata management overhead begins to dominate the read performance, resulting in the bandwidth drops we observe across all three scenarios [20].

The file write bandwidth is calculated as the compact C matrix file size divided by the time to write the file in either a compact or

block mode. Due to space considerations, only the block file write bandwidth is provided in table form, in Table 7. The block bandwidth is included because it consistently outperformed the compact write, and would almost certainly be the preferred approach in future efforts.

Table 7. C matrix block file write bandwidth vs. number of nodes for 1, 2, and 4 ranks/core

Num. Nodes	*1 rank/core (MB/sec)*	*2 ranks/core (MB/sec)*	*4 ranks/core (MB/sec)*
8	6.25	4.17	2.38
16	3.57	2.78	1.61
32	3.57	2.38	1.43
64	3.33	2.00	1.14
128	3.13	1.79	0.93
256	1.85	0.62	0.47

Figure 12 plots the block write bandwidth (from Table 7), and Figure 13 plots the compact write bandwidth, for the three scenarios. The same axes are used in both figures to facilitate visual comparisons between the two.

From these two figures, we see roughly the same behavior, in that the file write time starts out reasonably high for 8 nodes, and then decreases as the number of nodes grows to 256. This is most pronounced for the block file writes, which start out at 6.25 MB/sec for 1 rank/core on 8 nodes, and then decreases nearly by half to 3.57 MB/sec at 16 nodes. After the initial dramatic drop for each scenario, the write bandwidth then gradually declines as the number of nodes increases, with the 1 rank/core scenario reaching a low of 1.85 MB/sec at 256 nodes. The compact write bandwidth shows less consistency across the scenarios, other than the fact that it performs consistently worse than the block file writes and the performance also tends to get worse as the number of nodes increases.

The write bandwidth decreases as the number of nodes increases most likely because the C matrix file is very small (0.5 MB). Therefore, each of the parallel processes is writing an extremely small amount of data (as few as 32 bytes to at most 4KB in these experiments), and so the blocks are too small for AMOS to efficiently parallelize the writes and we are not actually benefiting from truly parallel write operations.

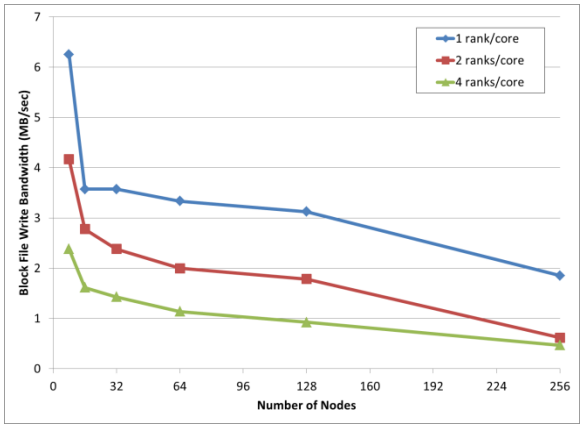

Figure 12. C Matrix Block Write Bandwidth (MB/sec) vs. Number of Nodes

One interesting observation between the read and write bandwidths is that the 1 rank/core scenario exhibits both the best read and write rate. While each of the three ranks/core scenarios achieve the best write rate for different numbers of nodes, the absolute peak read bandwidth of 2.99 GB/sec (on 128 nodes) is experienced by the 1 rank/core scenario, as is the absolute peak write bandwidth of 6.25 MB/sec (on 8 nodes). Unlike the read bandwidth, however, for the block write bandwidth the 1 rank/core scenario consistently outperforms the other two scenarios across all node counts. These imply that for the peak number of nodes involved in a parallel read or write operation, it may be preferable to minimize the number of processes executing on each node.

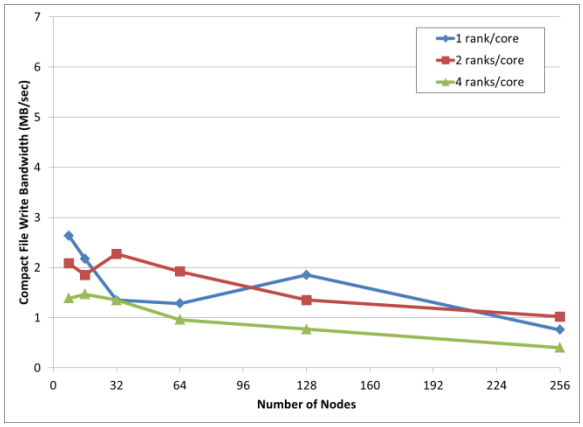

Figure 13. C Matrix Compact Write Bandwidth (MB/sec) vs. Number of Nodes

7. CONCLUSIONS & FUTURE WORK

PARAFAC is a popular technique for discovering hidden relationships in multidimensional datasets. As data volumes grow within many scientific disciplines, researchers and scientists will require new systems to enable PARAFAC (and similar analyses) to execute on data that is too large to be manipulated on a single server with traditional tools.

We have designed an approach to parallelize the execution of the PARAFAC alternating least squares algorithm, and have implemented it using C and MPI. A dataset was simulated and used to test the performance and scalability of this implementation on an IBM Blue Gene/Q supercomputer. The compute time, MPI communication bandwidth, tensor file read bandwidth, and C matrix file write bandwidth were all captured for three scenarios (1 rank/core, 2 ranks/core, and 4 ranks/core) across 6 different node counts.

This study found that the implementation scaled quite well and specifically that increasing the number of parallel processes by a factor of 128 resulted in a 200x speedup in the overall execution time. The MPI bandwidth grew consistently as the size of the study grew, indicating that the communication network was never saturated during execution. At its peak the communication time only accounted for 12% of the total processing time with a peak bandwidth of 6 GB/sec, indicating the implementation is CPU bound at the current scale. The peak bandwidth achieved when performing a parallel read of the tensor from a file was approximately 3 GB/sec, and the peak write bandwidth achieved was 6.25 MB/sec.

This MPI-based Parallel PARAFAC implementation will allow researchers to scale to considerably larger datasets, taking

advantage of more disk, memory, and CPU available on supercomputers such as AMOS. In the future, we will explore how the performance is impacted using substantially larger datasets, starting in the multi-terabyte range. We also expect to begin analyzing a real industrial multiway dataset from GE Power & Water's Remote Monitoring and Diagnostics (RM&D) Center. This RM&D Center has over 20TB of time series data from sensors on gas turbines captured over more than a 7 year period. To our knowledge no one has ever attempted to execute PARAFAC on this type of data, nor at this scale.

Beyond new and larger datasets, we will also explore alternative platforms on which to implement the Parallel PARAFAC algorithm. In particular we will focus on Apache Hadoop and Apache Spark. Hadoop is a well-established Big Data platform for distributed data storage and parallel task execution using the MapReduce computing paradigm running on clusters of commodity hardware [21][22]. Spark is a comparatively new project originating from UC Berkeley that allows for iterative MapReduce-style operations to be run across large commodity clusters [23]. Spark keeps working sets of data entirely in memory between jobs to optimize the performance of iterative operations, and claims to be 100x faster than traditional Hadoop as a result. We plan to explore both Hadoop and Spark as alternative platforms on which to implement our Parallel PARAFAC algorithm.

8. REFERENCES

[1] Harshman, R.A., 1970. Foundations of the parafac procedure: models and conditions for an 'explanatory' multi-modal factor analysis. UCLA working papers in phonetics 16, 1-84.

[2] Harshman, R.A. and Lundy, M.E., 1994. PARAFAC: Parallel factor analysis. Computational Statistics & Data Analysis 18, 39-72

[3] Andersen, C.M. and Bro, R., 2003. Practical aspects of parafac modelling of fluorescence excitation-emission data. J. of Chemometrics 17, 4, 200-215

[4] Bader, B.W., Harshman, R.A. and Kolda, T.G., 2006. Temporal analysis of social networks using three-way dedicom. Technical Report SAND2006-2161, Sandia National Laboratories

[5] Estienne, F., Matthijs, N., Massart, D.L., Ricoux, P. and Leibovici, D., 2001. Multi-way modelling of high-dimensionality electroencephalographic data. Chemometrics Intell. Lab. Systems 58, 1, 59-72

[6] Acar, E. and Yener, B., 2009. Unsupervised Multiway Data Analysis: A Literature Survey. IEEE Transactions on Knowledge and Data Engineering, 21, 1

[7] Andersson, C.A. and Bro, R., 2000. The n-way toolbox for matlab. Chemometrics and Intelligent Laboratory Systems, 52, 1, 1-4

[8] Bader, B.W. and Kolda, T.G., 2007. Efficient MATLAB Computations with sparse and factored tensors. SIAM Journal on Scientific Computing, 30, 1, 205-231

[9] McKinsey & Company, 2011. Big data: The next frontier for innovation, competition, and productivity, http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation

[10] Fan, J., Han, F. and Liu, H., 2014. Challenges of Big Data analysis, National Science Review, Oxford University Press, doi: 10.1093/nsr/nwt032

[11] Sears, M., Bader, B.W. and Kolda, T.G., 2009. Presentation: Parallel Implementation of Tensor Decompositions for Large Data Analysis. SIAM Annual Meeting (AN09), Minisymposium on High Performance Computing on Massive Real-World Graphs

[12] Zhang, Q., Berry, M., Lamb, B. and Samuel, T., 2009. A parallel nonnegative tensor factorization algorithm for mining global climate data. Computational Science, International Conference on Computational Science (ICCS), 405-415

[13] Phan, A.H. and Cichocki, A., 2011, PARAFAC algorithms for large-scale problems, Neurocomputing, 74, 1970-1984

[14] Kang, U., Papalexakis, E.E., Harpale, A. and Faloutsos, C., 2012. GigaTensor: Scaling Tensor Analysis Up By 100 Times - Algorithms and Discoveries, ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), Beijing, China

[15] Papalexakis, E.E., Faloutsos, C. and Sidiropoulos, N.D., 2012. ParCube: Sparse Parallelizable Tensor Decompositions, European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD), Bristol, United Kingdom

[16] Sidiropoulos, N.D., Papalexakis, E.E. and Faloutsos, C., 2014. A Parallel Algorithm for Big Tensor Decomposition using Randomly Compressed Cubes (PARACOMP), IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Florence, Italy, 1-5

[17] Zhe, S., Qi, Y., Park, Y., Molloy, I.M. and Chari, S. N., 2013. DinTucker: Scaling up Gaussian process models on multidimensional arrays with billions of elements, Computing Research Repository (CoRR) arXiv:1311.2663

[18] Tucker, L.R., 1964. The extension of factor analysis to three-dimensional matrices. In Contributions to Mathematical Psychology. Holt, Rinehart and Winston, New York, 110-182

[19] Gilge, M., 2014. IBM System Blue Gene Solution Blue Gene/Q Application Development, IBM Redbooks, ISBN 0738438235

[20] Alam, S.R., El-Harake, H.N., Howard, K., Stringfellow, N. and Verzelloni, F., 2011. Parallel I/O and the metadata wall. In Proceedings of the sixth workshop on Parallel Data Storage (PDSW '11). ACM, New York, NY, USA, 13-18

[21] Dean, J. and Ghemawat, S., 2008, MapReduce: Simplified Data Processing on Large Clusters, Communications of ACM, 51, 1, 107-113

[22] Shvachko, K., Kuang, H., Radia., S. and Chansler, R., 2010, The Hadoop Distributed File System, Proc. of IEEE Symposium on Mass Storage and Technologies (MSST), 1-10

[23] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S. and Stoica, I., 2010. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on hot topics in cloud computing (HotCloud'10). USENIX Association, Berkeley, CA, USA