MPI-LiFE: Designing High-Performance Linear Fascicle Evaluation of Brain Connectome with MPI

Shashank Gugnani*, Xiaoyi Lu*, Franco Pestilli†, Cesar F. Caiafa†‡, and Dhabaleswar K. (DK) Panda*
*Department of Computer Science and Engineering, The Ohio State University
*Email: {gugnani.2, lu.932, panda.2}@osu.edu
†Psychological and Brain Sciences, Engineering, Computer Science, Neuroscience and Cognitive Science, Indiana University
†Email: franpest@indiana.edu, ccaiafa@iu.edu
‡Instituto Argentino de Radioastronomía (IAR), CONICET, CCT - La Plata, ARGENTINA

***Abstract*—In this paper, we combine high-performance computing science with computational neuroscience methods to show how to speed-up cutting-edge methods for mapping and evaluation of the large-scale network of brain connections. More specifically, we use a recent factorization method of the Linear Fascicle Evaluation model (i.e., LiFE [1], [2]) that allows for statistical evaluation of brain connectomes. The method called ENCODE [3], [4] uses a Sparse Tucker Decomposition approach to represent the LiFE model. We show that we can implement the optimization step of the ENCODE method using MPI and OpenMP programming paradigms. Our approach involves the parallelization of the multiplication step of the ENCODE method. We model our design theoretically and demonstrate empirically that the design can be used to identify optimal configurations for the LiFE model optimization via ENCODE method on different hardware platforms. In addition, we co-design the MPI runtime with the LiFE model to achieve profound speed-ups. Extensive evaluation of our designs on multiple clusters corroborates our theoretical model. We show that on a single node on TACC Stampede2, we can achieve speed-ups of up to 8.7x as compared to the original approach.**

***Keywords*-Brain Connectome, LiFE, MPI, Multiway Array, OpenMP, Tensor Decomposition**

I. Introduction

In this paper, we present work that bridges two communities of researchers, namely computer scientists working on high-performance computing and computational neuroscientists working on methods for mapping the network of human brain connections. The work exemplifies the modern needs of scientific progress, where trans-disciplinary efforts help advance understanding faster by allowing the fastest computing methods to support basic research. In our example, we combine cutting-edge computational neuroscience methods with high-performance computing approaches. We demonstrate major speedups that allow solving the structure of the human brain in less than one-eighth of the original time.

The Linear Fascicle Evaluation (LiFE) method [1], [2] is a method based on convex optimization that uses a large set of fascicles generated using multiple tractography methods and identifies the subset of fascicles that best predict the measured dMRI signal. The LiFE method has been extended into a flexible framework for encoding dMRI, brain anatomy, and evaluation methods using multidimensional arrays [3]. This framework called ENCODE allows implementing the convex optimization algorithm [4] used to identify fascicles that effectively predict the measured dMRI signal by means of multidimensional arrays manipulations. The multidimensional organization of ENCODE allows us to exploit parallelization methods such as OpenMP and MPI to speedup the process of fascicle evaluation.

The original LiFE model [1] operated on arrays which were too large to fit in main memory (50 - 100 GB). The extended LiFE model [3], [5] uses a decomposition method to compress the data so that it can easily fit in memory (1 GB). This extension poses challenges for designing parallel algorithms for LiFE, particularly owing to the sparse nature of the resulting compressed data matrices. In addition, even with the compression technique, the data sizes involved are much larger than typical message sizes used in MPI. In this paper, we propose parallel algorithms using the MPI and OpenMP programming models which can solve these challenges. We theoretically model our proposed algorithms by calculating the runtime complexity and maximum speedup. Extensive evaluation on three modern clusters demonstrates that our proposed parallel LiFE model (called **MPI-LiFE**) can achieve a speed-up of up to 8.7x. We also show that our model works well on different platforms and by using our theoretical analysis, the speed-up can be approximately estimated. Hereafter, we first describe a recent model for statistical evaluation of brain connectome (the large-scale network of brain connections), we then demonstrate how to use MPI-based methods to reduce the computing time necessary to optimize the model and map the human brain.

The rest of this paper is organized as follows. Section II discusses the background of our work, Section III presents a description of the parallelization problem, and Section IV presents our proposed algorithms to parallelize the LiFE model. Section V discusses the theoretical analysis of our proposed model, Section VI demonstrates a performance eval-

*This research is supported in part by National Science Foundation grants #CNS-1419123, #IIS-1447804, #ACI-1450440, #CNS-1513120, #IIS-1636846, #IIS-1636893, and #BCS-1734853, the National Institute of Health Grant #ULT-TR001108 and the Indiana University Areas of Emergent Research initiative Learning: Brains, Machines, Children to Franco Pestilli. Data provided in part by the Human Connectome Project (NIH 1U54MH091657).

0-7695-6326-0/17/$31.00 ©2017 IEEE
DOI 10.1109/HiPC.2017.00033

uation of our proposed design, and Section VII discusses related work. Finally, Section VIII concludes the work.

II. Background

A. Statistical Evaluation of Brain Connectomes

Diffusion-weighted MRI (dMRI) data and computational tractography methods allow the estimation of the macroscopic structure of the brain connections in living human brains. dMRI measures the diffusion of water molecules along different spatial directions within the brain tissue. Diffusion is strongest in the direction of the neuronal fiber bundles, this signal can be used to reconstruct the three-dimensional structure of the neuronal axons within the brain tissue by using computational tractography. Brain connections are reconstructed as sets of *fascicles*, namely the Brain Connectome (see Figure 1), describing the putative position and orientation of the neuronal axons bundles wrapped by myelin sheaths traveling within the brain [6], [7], [8], [9]. Since dMRI provides only indirect measurements of the brain tissue organization and tractography is a stochastic computational method, the anatomical properties of the putative fascicles estimated with these methods can depend on data type, tractography algorithm, and parameters settings [1], [2], [10]. For this reason, investigators have been developing methods for results validation based on statistical and computational approaches [11], [8], [12], [13].

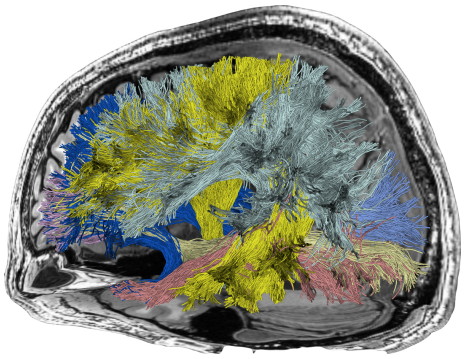


Figure 1. **The Brain Connectome.** Illustration of a set of fascicles (white matter bundles) obtained by using a tractography algorithm. Fascicles are grouped together conforming white matter tracts (shown with different colors here) connecting different cortical areas of the human brain.

Recently, linear methods based on convex optimization have been proposed for connectome evaluation [1], [2] and simultaneous connectome and white matter microstructure estimation [14]. The Linear Fascicle Evaluation method (LiFE; [1], [2]), uses a large set of putative fascicles generated by using multiple tractography methods (called candidate connectome) and identifies the subset of fascicles that best predict the measured dMRI signal using convex optimization. The LiFE method was recently extended into a flexible framework for encoding dMRI, brain anatomy and evaluation methods using multidimensional arrays [3]. The framework called ENCODE allows to implement the convex optimization algorithm [4] used to identify fascicles that effectively predict the measured dMRI signal by means of multidimensional arrays manipulations. As a result of the multidimensional organization of ENCODE, we can exploit parallelization methods such as OpenMP and MPI to speedup the process of Fascicle Evaluation. Hereafter we demonstrate striking results with speedups up to 8.7x that combine ENCODE and MVAPICH2. Before that, we briefly introduce the MVAPICH2 MPI library.

B. MVAPICH2

MVAPICH2 [15] is an open-source implementation of the MPI-3.1 specification over modern high-end computing systems and servers using InfiniBand, Omni-Path, Ethernet/iWARP, and RDMA over Converged Ethernet (RoCE) networking technologies. The MVAPICH2 software packages are being used by more than 2,825 organizations worldwide in 85 countries and are powering some of the top supercomputing centers in the world, including the 1st, 10,649,600-core (Sunway TaihuLight) at National Supercomputing Center in Wuxi, China, the 15th Pleiades at NASA, the 20th Stampede at TACC, and the 44th Tsubame 2.5 at Tokyo Institute of Technology. MVAPICH2 is also being distributed by many InfiniBand, Omni-Path, iWARP, and RoCE vendors in their software distributions. MVAPICH2 has multiple derivative packages, which provide support for hybrid MPI + PGAS (CAF, UPC, and OpenSHMEM) programming models with unified communication runtime (i.e., MVAPICH2-X), optimized MPI communication for clusters with NVIDIA GPUs (i.e., MVAPICH2-GDR) and Intel MIC (i.e., MVAPICH2-MIC), high-performance and scalable MPI communication for hypervisor- and container-based HPC cloud (i.e., MVAPICH2-Virt), and energy aware and high-performance MPI communication (i.e., MVAPICH2-EA). In this paper, we primarily use the standard MVAPICH2 library to parallelize the LiFE code with the MPI programming model to obtain optimized performance on supercomputers with InfiniBand and Omni-Path networks.

Symbol	Description
N_θ	Number of diffusion directions
N_v	Number of voxels
N_f	Number of fascicles
N_n	Number of nodes in connectome
p	Number of MPI processes

Table I
Notation

III. Description of Parallelization Problem

Table I describes the commonly used symbols in this paper and this section in particular. The LiFE model [1] can be expressed using the following equation:

$$y \approx Mw, \tag{1}$$

where $y \in \mathbb{R}^{N_\theta N_v}$ is a vector which contains the demeaned signals for all white-matter voxels (v) across all diffusion directions (θ). $M \in \mathbb{R}^{N_\theta N_v \times N_f}$ is a matrix, which at column f contains the signal contribution given by fascicle f at voxel

v across all directions θ, and $w \in \mathbb{R}^{N_f}$ contains the weights for each fascicle in the connectome. The estimation of these weights requires solving the the following non-negative least-square constrained optimization problem:

$$\min_{w} \quad \left(\frac{1}{2}\|y - Mw\|^2\right) \text{ subject to } w_f \geq 0, \forall f. \tag{2}$$

Recently, a multidimensional encoding of brain connectomes, namely, the ENCODE model [3] was proposed, which employs sparse multiway decomposition to reduce the size of the data that needs to be stored. Data compression is achieved using the Sparse Tucker Decomposition (STD) model for multiway arrays [16], [17]. Using this model, a 3^{rd} array $\underline{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, is approximated by the following decomposition:

$$\underline{X} \approx \underline{G} \times A_1 \times A_2 \times A_3, \tag{3}$$

where $\underline{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ is the sparse core array and $A_n \in \mathbb{R}^{I_n \times R_n}$ are the factor matrices. The decomposition does not guarantee a good approximation. However, it compresses the data because the core array is very sparse.

M is a very large block-sparse matrix ($M \in \mathbb{R}^{N_\theta N_v \times N_f}$) which can be converted to a multiway array by using diffusion directions (θ), voxels (v), and fascicles (f) as the dimensions of a 3D multiway array $\underline{M} \in \mathbb{R}^{N_\theta \times N_v \times N_f}$. By representing M as a multiway array, we can use the STD to compress its size. The original LiFE equation can then be rewritten as follows:

$$Y \approx \underline{M} \times_3 w^T, \tag{4}$$

where $Y \in \mathbb{R}^{N_\theta \times N_v}$ is the matrix version of vector y. By employing the Sparse Tucker Decomposition, we get

$$\underline{M} = \underline{\Phi} \times_1 D \times_2 S_0, \tag{5}$$

where the 3-way array $\Phi \in \mathbb{R}^{N_a \times N_v \times N_f}$ has matrices $\Phi_v \in \mathbb{R}^{N_a \times N_f}$ as lateral slices and $S_0 = diag(S_0(1), S_0(2), ..., S_0(N_v)) \in \mathbb{R}^{N_v \times N_v}$ is a diagonal matrix with values $S_0(v)$ along the main diagonal. By combining Equations 4 and 5, the following decomposition is obtained:

$$Y \approx \underline{\Phi} \times_1 D \times_2 S_0 \times_3 w^T. \tag{6}$$

$\underline{\Phi}$ is sparse, which results in strong data compression.

Solving the optimization problem (Equation 2) requires the use of Non-Negative Least Squares (NNLS) optimization algorithms. These algorithms require the iterative computation of two basic operations ($y = Mw$ and $w = M^T y$). Since we never explicitly store the matrix M, but rather its STD, the computation of the two basic operations requires the use of multiway arrays. $y = Mw$ can be computed using Equation 6, while $w = M^T y$ can be computed as follows:

$$w = \Phi_{(3)} vec(D^T Y S_0) \tag{7}$$

An analysis of the STD-based LiFE implementation on a single Xeon node of cluster A (see Table II), reveals that about 92% of the total time is spent in the two multiway array multiplication operations. 3% of the time is spent in loading the data, and 5% of the time is spent solving the optimization algorithm. A similar trend is observed on cluster C. Thus, it is clear that multiway array multiplication is the main bottleneck in the LiFE model and offers the most opportunity for parallelization. This parallelization poses a challenge due to the large sizes and sparse nature of the arrays as well as data dependencies in the computation of the array multiplication. For example, with the dataset we used for evaluation, the size of matrix D is 94.65 MB and matrix Y is 149.46 MB. Given these sizes, the MapReduce [18] paradigm seems to be a good parallelization choice, however, given the iterative nature of the optimization algorithm, this choice is clearly the wrong way to proceed. While MPI is usually not used for message sizes of this nature, we argue that it fits our requirements. MPI is designed for performance and flexibility. The unique computational nature of the LiFE model can be accurately captured and parallelized using MPI. We thus select the MPI programming paradigm to design a parallel algorithm for the LiFE model. We describe our proposed design to parallelize the LiFE model in the next section.

Platform	Data Loading	NNLS	$y = Mw$	$w = M^T y$
Cluster A	2.57%	4.91%	63.31%	29.21%
Cluster C	5.02%	2.47%	72.95%	19.56%

Table II
TIME BREAKUP OF LIFE. RESULTS ARE TAKEN ON A SINGLE NODE OF CLUSTER A (XEON NODES) AND CLUSTER C (KNL NODES). PLEASE REFER TO TABLE III FOR A DESCRIPTION OF THE CLUSTERS.

IV. PROPOSED DESIGN

In this section, we propose high-performance parallel designs to accelerate the computation of the optimization algorithm used in the LiFE model. We specifically use the MPI and OpenMP programming models to compute the multiplication of large sparse multiway arrays. The two main operations to be parallelized are $y = Mw$ and $w = M^T y$. We present designs to accelerate both of these operations. We theoretically analyze our proposed designs and show how this analysis can be used to predict the optimal configuration to run LiFE on different platforms.

A. MPI-based LiFE Model

The proposed MPI-based design follows a master-slave architecture. The master rank runs the main optimization algorithm and is responsible for distribution of computation among slave ranks. Distribution of data and gathering of results is also handled by the master rank and is done with the help of MPI collectives. The iterative computation required by the optimization algorithm results in the repeated calculation of y and w. However, M remains constant throughout the process. Thus, M only needs to be sent to the slave ranks in the beginning of the application execution. Since M is itself never stored, but rather its STD, the sub-components of M, i.e. $\underline{\Phi}$, D, and S_0 are sent to the slave ranks in the beginning itself.

The computation of y and w requires iterating over all the possible nodes within the brain connectome (N_n) and all possible diffusion directions (N_θ). Nodes are like (x, y, z) spatial coordinates within the brain. $N_n \approx 10^6 - 10^7$, depending on the dMRI resolution and is typically $\gg N_\theta$. Thus, we divide the computation based on the nodes. Each MPI process gets an appropriate chunk of the sub-components of M so that it can compute the values of y and w corresponding to the nodes which it has been assigned. This is done using MPI_Scatter. In addition to M, the master rank also needs to send the matrices Y and w to the slave ranks so that the two operations can be computed. Y needs to be sent to compute w, and vice-versa. This needs to be done each time when one of the two operations requires computation. In addition, the computation of each index of Y can depend on any index of w and vice-versa. This is because any fascicle can pass through any voxel, thus the diffusion signal at a voxel can depend upon any number of fascicles. Similarly, the weight of a fascicle can depend upon the diffusion signal at any voxel. This means that to compute w, the entire Y matrix needs to be sent to all processes and vice-versa. While this might seem to be computationally inefficient and expensive, as we will see later, the time required for the broadcast of these matrices constitutes only a small portion of the overall application time.

We now describe the algorithm for computing the two operations.

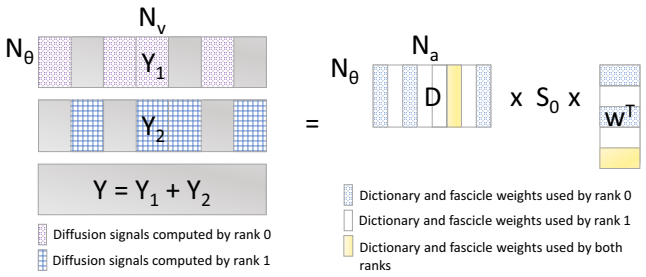


Figure 2. **Computation of $y = Mw$ using 2 MPI processes.** Each MPI process uses some of the columns of D, and some of the fascicle weights. The partial diffusion signals are added to obtain the final weights at rank 0 using MPI_Reduce. It should be noted that only non zero entries in Φ for D and w are multiplied to save CPU cycles.

1) $y = Mw$: For this operation, w is broadcasted to all processes using MPI_Bcast. As mentioned before, each MPI process computes the demeaned signal for a fraction of the nodes in the brain, for all diffusion directions. Thus each MPI process will have the partial demeaned signals for a subset of the voxels. These subsets are not guaranteed to be mutually exclusive. Thus, to compute the overall diffusion signal for all voxels in all diffusion directions, we need to add the computed signals for the corresponding indices across all MPI processes. To this end, we use MPI_Reduce to get the final result, which conveniently provides the final result at the master rank as desired. Figure 2 shows the working of this algorithm across two MPI processes and Algorithm 1 provides a formal representation of this operation.

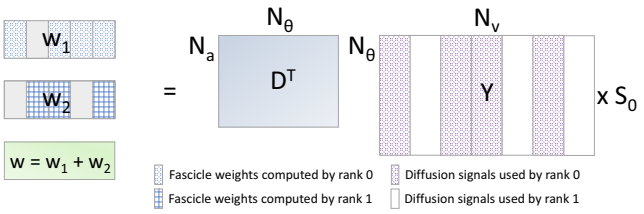


Figure 3. **Computation of $w = M^T y$ using 2 MPI processes.** Each MPI process uses the entire D matrix and a fraction of the columns of Y to compute some of the fascicle weights. The fascicle weight values are added to obtain the final weights at rank 0. It should be noted that only non zero entries in Φ for D^T and Y are multiplied to save CPU cycles and w is the vectorized output of the multiplication.

2) $w = M^T y$: The matrix Y is broadcasted to all processes using MPI_Bcast. Each MPI process computes the weight of a subset of fascicles for a subset of nodes in the brain and for all diffusion directions. For each node, we compute the weight contribution for a particular fascicle. To calculate the overall weight for each fascicle, we must add the weight contributions for individual fascicles. To compute this result, we gather the computed values from all MPI processes using MPI_Gather at the master rank and iterate over the values for all nodes and compute the combined weight for each fascicle. MPI_Reduce could also have been used to compute the final result, however, given the sparse nature of the weight array at each process, it would have resulted in a lot of unnecessary computation. Figure 3 shows the working of this algorithm across two MPI processes and Algorithm 2 provides a formal representation of this operation.

Algorithm 1 $y =$ M_times_w$(\underline{\Phi}, D, S_0, w)$

procedure M_TIMES_W$(\underline{\Phi}, D, S_0, w)$ ▷ Predict demeaned diffusion signals
  c_size $\leftarrow \frac{N_n}{p}$
  $[a, v, f, c] = get\_nonzero\_entries(\underline{\Phi})$ ▷ $a(n), v(n), f(n), c(n)$ indicate the atom, the voxel, the fascicle, and coefficient associated with node n, respectively, with $n = 1, 2, ..., N_n$
  MPI_Bcast(w)
  for $n = 1$ to c_size **do**
    $Y(:, v(n)) = Y(:, v(n)) + D(, : a(n))w(f(n))c(n)$ ▷ Compute partial Y signals
  MPI_Reduce(Y) ▷ Aggregate partial demeaned signals
  $y \leftarrow vec(Y)$ ▷ Only rank 0
  return y

B. Hybrid MPI + OpenMP-based LiFE Model

While the MPI-based design is useful in parallelizing the LiFE model across multiple processors (on the same or different nodes), it adds communication cost which limits the speedup that can be attained. Thread-based parallelism (like OpenMP and pthreads) offers the opportunity to scale applications on multiple cores within a single node without adding additional communication cost. However, limitation to a single node restricts its applicability. A hybrid design with

Algorithm 2 w = Mtransp_times_y($\underline{\Phi}, D, S_0, y$)

procedure MTRANSP_TIMES_Y($\underline{\Phi}, D, S_0, y$) ▷ Predict fascicle weights
  c_size $\leftarrow \frac{N_n}{p}$
  $[a, v, f, c] = get\_nonzero\_entries(\underline{\Phi})$ ▷ $a(n), v(n), f(n), c(n)$ indicate the atom, the voxel, the fascicle, and coefficient associated with node n, respectively, with $n = 1, 2, ..., N_n$
  MPI_Bcast(y)
  for $n = 1$ to c_size **do**
    $k(n) = D^T(:, a(n))Y(:, v(n))c(n)$ ▷ Compute partial weights
  MPI_Gather(k) ▷ Gather partial weigths
  for $n = 1$ to N_n **do** ▷ Only rank 0
    $w(f(n)) \leftarrow w(f(n)) + k(n)$
  return w

MPI and OpenMP has the potential to be much more scalable than its constituents.

We thus extend the MPI-based design with OpenMP, such that each MPI process uses OpenMP threads to parallelize the multiway array multiplication. For both multiplication operations, each MPI process iterates over a fraction of nodes and all diffusion directions as part of the multiplication computation. Adding threads to the equation works in a similar fashion. Each thread iterates over a fraction of the nodes assigned to each MPI process and all diffusion directions.

By leveraging OpenMP, we can achieve speedup without adding any communication cost. With MPI + OpenMP, we can run the LiFE model across multiple nodes allowing access to a larger set of cores, and thus higher speedup.

C. MPI Collective Performance

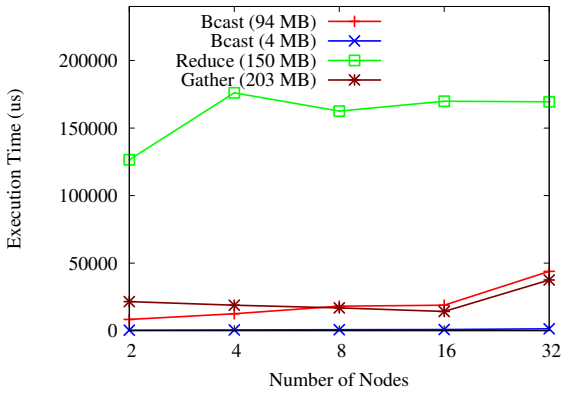


Figure 4. **MPI Collective Performance.** Result are taken on Cluster A with 1 PPN. The message sizes for the collectives are the exact message sizes that are used in our experimental evaluation dataset.

With the knowledge that the collective communication message sizes involved in our implementation are very large ($4 - 200$ MB), the performance of MPI collectives comes into question. We look at the performance of bcast, gather, and reduce in particular because these collectives are used in each iteration of the optimization algorithm. We evaluate the latency of these collectives for the exact message size as used in our implementation for different number of MPI processes. This analysis is presented in Figure 4. It is clear that the performance of reduce is poor as compared to other collectives. This is a result of the large message size (150 MB) for reduce as well as the compute portion of reduce. Reduce performance is a bottleneck when running a large number of MPI processes. After analyzing the reduce performance runs, we found that the computation part of reduce is the main bottleneck. For the MPI + OpenMP-based design, each MPI process utilizes some OpenMP threads to parallelize the portion of multiway array multiplication it is allocated. These threads can also be used to parallelize the reduce operation at each MPI process. Since the reduce operation is embarrassingly parallel, its parallelization is straightforward using OpenMP. This optimization is implemented in the MVAPICH2 library and is used for all performance evaluations.

D. Implementation

The main optimization algorithm is implemented in MATLAB while the multiway matrix multiplication is implemented in C for performance and to allow integration with MPI. The entire MATLAB implementation is compiled as a C shared library with the help of the MATLAB compiler SDK. This shared library uses the MATLAB runtime internally to call and execute the MATLAB functions. This allows us to create a C wrapper for the entire application, thus converting it into a binary executable. In addition, this binary can be packaged along with the MATLAB runtime so that the entire application can be run without any dependencies or MATLAB license. Containerization of the entire application is possible, although is beyond the scope of this paper, and is thus left as future work.

V. ANALYSIS

In this section, we analyze the runtime complexity of the default, MPI-based, and MPI + OpenMP-based LiFE models. We first consider an ideal scenario in which the MPI communication cost is zero and computational speedup scales linearly with the number of cores used. Then we consider the scenario when collective communication cost is non-zero, but its complexity is that of its best possible implementation.

A. MPI Collectives

We use bcast, scatter, gather, and reduce in our LiFE models. Scatter is only used in the beginning to distribute the components of the multiway array M. Since a large number of iterations of the optimization algorithm are run (50 for evaluation; at least 500 for real use cases), the scatter cost can be said to be amortized, and is thus not considered as part of the model complexity. For modeling the cost of the other collectives, we employ the *Hockney* model [19] to estimate point-to-point communication cost. According to the Hockney model, the time to send a message of m bytes between two nodes is $\alpha + \beta m$, where α is the latency for each message,

and β is the time required to transfer a byte of data. Using this model, as demonstrated in [20], the cost of each collective can be estimated as follows:

- **Bcast**: $O\left(\lceil \log_2 p \rceil \times (\alpha + \beta m)\right)$
- **Gather**: $O\left(\lceil \log_2 p \rceil \times (\alpha + \beta m)\right)$
- **Reduce**: $O\left(\lceil \log_2 p \rceil \times (\alpha + \beta m + \gamma m)\right)$,

where m is the size of each segment, γ is the computation time per byte of data, and p is the number of MPI processes.

B. LiFE Model

The original LiFE model runs the SBB NNLS optimization algorithm [21], which runs multiple iterations, each time converging further towards an optimal solution. Each iteration requires the computation of the two multiplication operations at least once. Assuming that data loading and running the optimization algorithm take a constant amount of time and the model is run for n iterations, the total complexity of the LiFE model is $O\left(N_n N_\theta n\right)$.

C. MPI-based LiFE Model

For both the multiplication operations, each MPI process operates on a fraction of nodes. Thus, assuming the ideal scenario, the complexity of $y = Mw$ will be $O\left(\frac{N_n N_\theta}{p}\right)$. For $w = M^T y$, the complexity will be the same except that the final updating of the weights is done by only the master rank. Its complexity is thus $O\left(\frac{N_n N_\theta}{p} + N_n\right)$. The overall complexity can be calculated out to be $O\left(\frac{N_n N_\theta}{p} n + N_n n\right)$. For the scenario with non-zero communication cost, the overall complexity can be calculated by using the equations for MPI collectives listed before as

$$O\Bigg(\frac{N_n N_\theta}{p} n + N_n n + \lceil \log_2 p \rceil n \times \left(\alpha + \beta\left(N_v N_\theta + N_f + \frac{N_n}{p}\right) + \gamma\left(N_v N_\theta\right)\right)\Bigg), \quad (8)$$

D. Hybrid MPI + OpenMP-based LiFE Model

In this model, the computation at each MPI process is further divided among the OpenMP threads. The overall complexity can be calculated out to be $O\left(\frac{N_n N_\theta}{p \times t} n + N_n n\right)$. While for the non-zero communication time scenario, this cost is

$$O\Bigg(\frac{N_n N_\theta}{p \times t} n + N_n n + \lceil \log_2 p \rceil n \times \left(\alpha + \beta\left(N_v N_\theta + N_f + \frac{N_n}{p}\right) + \gamma\left(N_v N_\theta\right)\right)\Bigg),$$

where t is the number of OpenMP threads per MPI process.

E. Speed Up

In this sub-section, we analyze the maximum theoretical speedup using Amdahl's law. According to Amdahl's law, each program consists of a parallel and a sequential portion. The speedup of the program is limited by the sequential portion. If we analyze the LiFE model with this ideology, we can conclude that the two multiway array multiplication functions constitute the parallel portion of the model, while data loading and execution of optimization algorithm constitute the sequential portion. Assuming that t_d is the time taken for data loading, t_o is the time taken by the optimization algorithm, t_{mm1} and t_{mm2} are the overall times spent in the two matrix multiplication functions, the theoretical maximum speedup when using c cores, $s_{max}@c$, can be calculated out as

$$s_{max}@c = \frac{t_d + t_o + t_{mm1} + t_{mm2}}{t_d + t_o + \frac{t_{mm1} + t_{mm2}}{c}}. \quad (9)$$

The absolute theoretical maximum speedup, s_{max}, is $\frac{t_d + t_o + t_{mm1} + t_{mm2}}{t_d + t_o}$. This speedup can practically never be achieved because of the memory wall phenomena, as we shall see in Section VI-C. The maximum speedup is in fact limited by the maximum memory bandwidth achievable on a single node. The actual maximum speedup is

$$s_{max} = \frac{t_d + t_o + t_{mm1} + t_{mm2}}{t_d + t_o + \frac{t_{mm1} + t_{mm2}}{\frac{m_{max}}{m_s} \times N}}, \quad (10)$$

where m_{max} is the maximum memory bandwidth achievable, m_s is the memory bandwidth required by the application for a single core, and N is the number of nodes.

VI. Performance Evaluation

A. Experimental Testbed

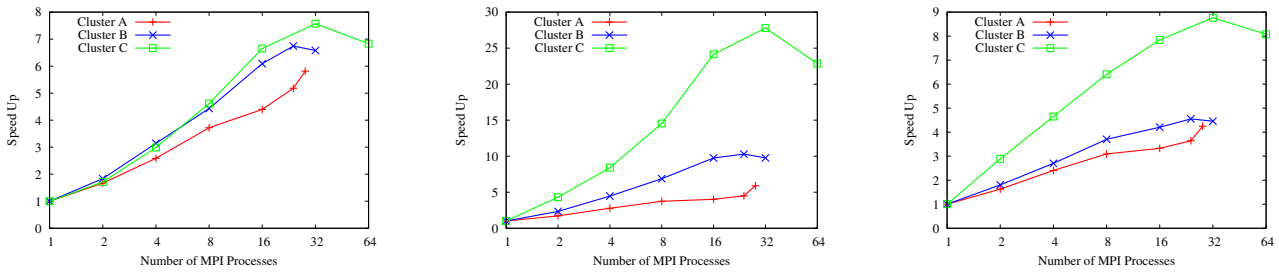
We evaluate our LiFE model on three separate platforms. The configuration of these platforms is detailed in Table III. Cluster A [22] is a local cluster at OSU with Broadwell processors and EDR InfiniBand. Clusters B and C are the Stampede [23] and Stampede2 [24] supercomputers at TACC. In Cluster B, we use the large memory nodes which provide Sandy Bridge processors with 32 cores per node. Cluster C has the latest Xeon Phi nodes (KNL) from Intel which have 68 cores per node. We choose such diverse platforms to understand the performance characteristics of our designs on different architectures and to show that they can work well in any environment.

Parameter	Cluster A (OSU RI2)	Cluster B (TACC Stampede)	Cluster C (TACC Stampede2)
Processor	Xeon E5-2680 v4	Xeon E5-4650	Xeon Phi 7250
Cores	28	32	68
Clock Speed	2.40 GHz	2.70 GHz	1.40 GHz
Memory	128 GB	1 TB	96 GB
Interconnect	EDR InfiniBand (100 Gbps)	FDR InfiniBand (56 Gbps)	Omni-Path (100 Gbps)

Table III
HARDWARE SPECIFICATION OF CLUSTERS USED FOR EVALUATION

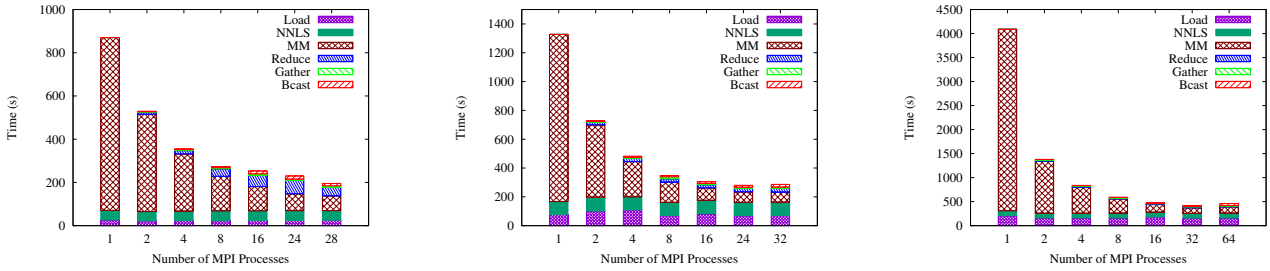
We use MATLAB R2016a for writing and compiling the optimization algorithm and MVAPICH2 v2.3a as the base for our reduce designs. In addition, we perform manual tuning to find the collective algorithms which perform the best for the message sizes involved in our implementation.

To evaluate LiFE, we use one of the pre-processed dMRI brain scans provided by the STN dataset [25]. We run 50



(a) CW (b) CDS (c) MPI-based LiFE model

Figure 5. **Single Node Evaluation.** CW represents the time taken for computing the fascicle weights ($w = M^T y$) and CDS represents the time taken for computing demeaned diffusion signals ($y = Mw$). The final graph shows the speedup for the entire LiFE model. Cluster A and B have Xeon nodes, while Cluster C has KNL nodes.



(a) Cluster A (b) Cluster B (c) Cluster C

Figure 6. **Time Breakup Evaluation.** Load represents time spent on loading dataset into memory, NNLS represents time spent in executing the optimization algorithm, and MM represents the overall execution time of the two multiway array multiplication operations. Reduce, Gather, and Bcast represent the time spent executing the respective MPI collectives.

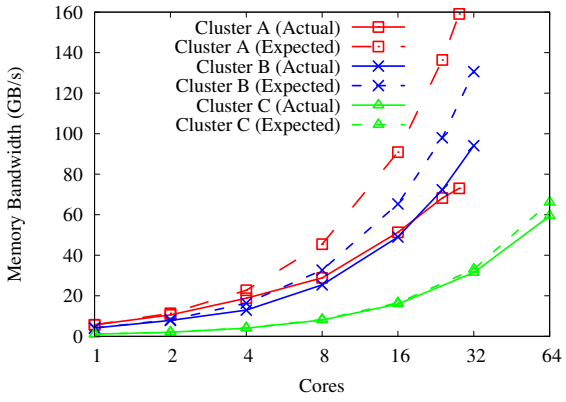


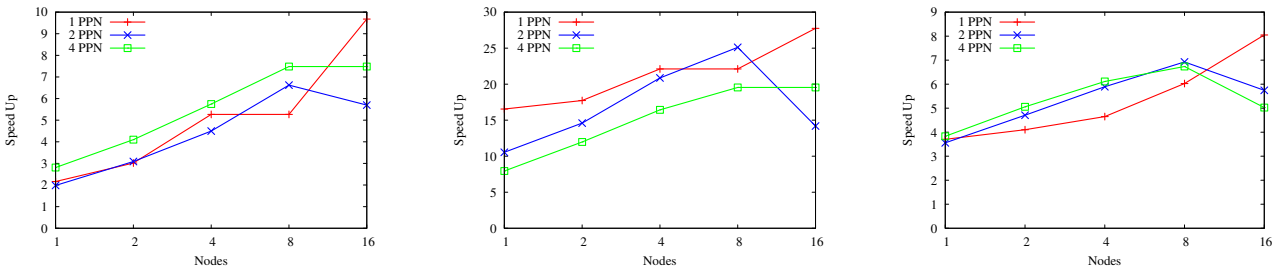
Figure 7. **Memory Bandwidth Evaluation.** Actual bandwidth represents the memory bandwidth achieved with the MPI-based LiFE model. Expected bandwidth represents the single core memory bandwidth scaled linearly. These results were obtained by removing the computation part of MPI-LiFE and measuring the time taken for the memory accesses.

iterations of the optimization algorithm so as to ensure that the application runs in a reasonable amount of time. We also use a validation phase in our evaluation which verifies that the output of the model is the same as that obtained from the default sequential version.

B. Evaluation Methodology

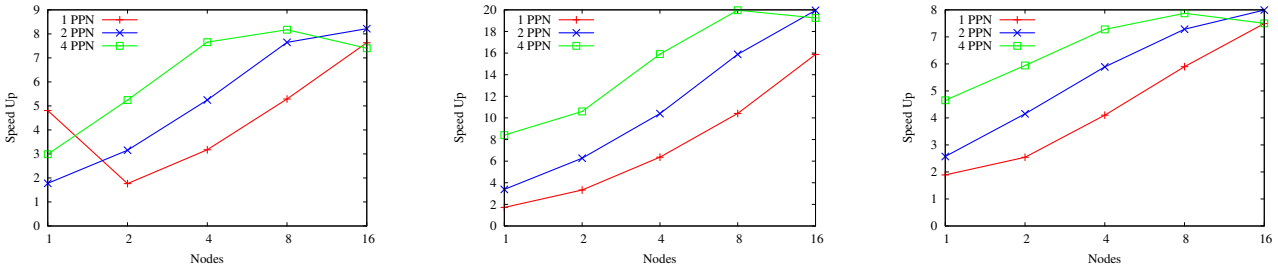
Our main goal is to evaluate the performance of our design on multiple architectures with different configurations. We also want to show that the evaluation results corroborate with our theoretical analysis. We first evaluate on a single node with the MPI-based LiFE design. We then move on to multi-node evaluation using the MPI + OpenMP-based LiFE design. We show both the speedup as well as the overall execution time. speedup on a platform is calculated with respect to the execution of the LiFE model on a single node on the same platform. We also present the speedup of the two operations which we have parallelized. CW (computing weights) is used to represent the $w = M^T y$ operation and CDS (computing diffusion signals) is used to represent the $y = Mw$ operation. To differentiate the communication and computation costs, we present the time breakup of various tasks in the proposed designs. The total time is broken down into the following tasks:

- **MM**: computing the multiway matrix multiplication on local data
- **NNLS**: solving the SBB NNLS optimization problem
- **Load**: loading pre-processed data into memory via MATLAB
- **Bcast**: executing MPI_Bcast
- **Gather**: executing MPI_Gather



(a) CW (b) CDS (c) MPI + OpenMP-based LiFE model

Figure 8. **Multi Node Evaluation on Cluster A.** Each MPI process uses 28, 14, and 7 OpenMP threads for 1, 2, and 4 PPN cases, respectively. CW represents the time taken for computing the fascicle weights ($w = M^T y$) and CDS represents the time taken for computing demeaned diffusion signals ($y = Mw$). The final graph shows the speedup for the entire LiFE model.



(a) CW (b) CDS (c) MPI + OpenMP-based LiFE model

Figure 9. **Multi Node Evaluation on Cluster C.** Each MPI process uses 64, 32, and 16 OpenMP threads for 1, 2, and 4 PPN cases, respectively. CW represents the time taken for computing the fascicle weights ($w = M^T y$) and CDS represents the time taken for computing demeaned diffusion signals ($y = Mw$). The final graph shows the speedup for the entire LiFE model.

- **Reduce**: executing MPI_Reduce

Through our evaluation, we seek to answer the following questions:

- What are the performance characteristics of our designs on different architectures?
- What factors influence the speedup on different architectures?
- Do the evaluation results match with our theoretical model?

C. Single Node Evaluation

The comparison of speedup for different clusters and LiFE models on a single node is presented in Figure 5. It can be observed that we can achieve up to 4.2x speedup on cluster A, 4.5x on cluster B, and 8.7x on cluster C for the MPI-based design. If we take a look at the speedup we can achieve for CW and CDS, it is clear that CDS is much more scalable. The maximum speedup achievable for CDS is 28x, while for CW is 8x. As seen in Algorithm 2, the final updating of weights is done sequentially (due to data dependencies) by rank 0, which limits the parallel performance of the operation. Figure 6 shows the time breakup of the entire application run for the three clusters. The cost for data loading (Load) and running the SBB NNLS optimization algorithm (NNLS) stays constant for all runs of the application. We do, however, see some variations on cluster B and C. These are mostly due to variation in the load on the NFS present on the clusters. The time for multiway array multiplication (MM) is significantly reduced with increasing number of MPI processes. The total communication cost is only a small portion of the entire application time. Although it increases with increasing number of MPI processes, even at full subscription, it only represents a small fraction of the execution time.

Memory Wall. With the large number of cores available in modern processors, the available memory bandwidth per core is extremely low. Parallelization of a memory-bound application within a node will be limited by the available memory bandwidth on the node. This phenomenon is commonly known as *Memory Wall* [26]. This is the reason that the speedup within a node does not scale linearly with increasing number of cores. To demonstrate that this phenomenon is the reason for the sub-linear speedup within a node, we modify Algorithms 1 and 2 to eliminate the compute portion while keeping the memory accesses. We then measure the total time taken by this portion of the algorithms and vary the number of cores used, like in Figure 5. By analyzing the code and calculating the total memory accesses, we calculate the aggregate memory bandwidth achieved. The required bandwidth is calculated out to be memory bandwidth obtained on a single core scaled linearly. The required bandwidth signifies the bandwidth required to achieve linear scalability of the application. This analysis

is presented in Figure 7. We can observe that cluster C offers the best memory bandwidth scalability, while cluster A offers the worst memory bandwidth scalability. While both clusters A and C use DDR4 RAM, the low clock speed of cluster C implies that the memory bandwidth will not be saturated easily. Thus, cluster C offers a higher opportunity of parallelism, which is exactly the trend we observe for speedup. In fact, the speedup achievable on each cluster is directly correlated with the memory bandwidth scalability of that cluster.

D. Multi Node Evaluation

The comparison of speedup for different clusters on multiple nodes with the MPI + OpenMP-based model is presented in Figures 8 and 9. It can be observed that we can achieve up to 8.1x speedup on cluster A and 8.1x on cluster C. The speedup on cluster C is much lower than expected (considering the results in Figure 5). The poor OpenMP performance on the KNL nodes of cluster C is the reason for this observation. In general, increasing the number of processes on each node improves the speedup for a small number of nodes. However, beyond a certain number of nodes, the performance with multiple PPN deteriorates rapidly, owing to the increase in communication cost. For CW and CDS, we can achieve a speedup of up to 9.7x and 27.7x, respectively. The speedup for CDS is particularly impressive, while that of CW is limited because of the sequential updating of weights.

E. Discussion

The results we have obtained clearly demonstrate the ability of our designs to work across different architectures and gain major speedup. From Figure 6, it can be observed that bcast constitutes less than 8% of the total execution time for any configuration. This solidifies our initial claim and use of bcast in both Algorithms 1 and 2. In addition, the total communication cost is only a small fraction ($< 15\%$) of the total execution time in most cases. This highlights the high-performance collective designs in MVAPICH2, even for large message sizes.

The maximum speedup achievable is bound by the memory wall. This phenomena is clearly realized if we look at the MM time in Figure 6. The MM time is nearly the same for the last two cases on each cluster. In fact the MM time speedup can be approximately predicted using Equation 10. For example, with cluster A, s_{max} (MM) $= \frac{799.53}{68.21} = 11.7 \approx 12.85 = \frac{73.03}{5.68} = \frac{t_{mm1}+t_{mm2}}{\frac{m_{max}}{m_s}\times N}$. In addition, the overall complexity of our design (Equation 8) can be used to predict changes to the the execution time based on the changes to evaluation configuration and dataset.

Apart from MM, Load and NNLS are also scalability bottlenecks. The runtime for these two tasks remains nearly constant on increasing the numbers of processes. The data loading and NNLS optimization algorithm portions are currently written in MATLAB, which prevents any possible parallelization. We are in the process of porting the code from MATLAB to C/Python to allow additional parallelism. To address the scalability of Reduce, we are working on co-designing with the MPI runtime by creating a chunked and pipelined version of MPI_Reduce. These optimizations should help in improving the scalability of our proposed designs.

VII. Related Work

In the parallel computing domain, there are several studies focused on optimizing application performance with different parallel programming models, such as MPI, OpenMP, PGAS, and MapReduce. For bioinformatics applications, Zhang et al. [27] proposed a fine-grained approach to parallelize Basic Local Alignment Search Tool for searching Protein sequences (i.e., BLASTP), where each individual phase of sequence search is mapped to many threads on a GPU and data-access patterns are reordered to reduce divergent branches of the most time-consuming phases. MR-MSPolygraph [28] is a MapReduce-based implementation for parallelizing peptide identification from mass spectrometry data. MSPolygraph used a novel hybrid approach to match an experimental spectrum against a combination of a protein sequence database and a spectral library. Hou [29] proposed a framework called AAlign to automatically vectorize pairwise sequence alignment algorithms for bioinformatics applications. Not only for bioinformatics applications, Li et al. [30] took advantage of the new features from MPI-3 Remote Memory Access model to re-design a scalable Graph500 application kernel. The authors in [31] proposed a hybrid MPI + OpenSHMEM approach to optimize the original MiniMD [32] application, which is a simple proxy for the force computations in a typical molecular dynamics applications. Kannan et al. [33] proposed a parallel algorithm to compute the non-negative factorization of a matrix. They present a distributed-memory parallel algorithm based on MPI to compute sparse matrix-matrix (SpMM) multiplication. This approach, however, cannot be applied to LiFE since it uses STD to decompose large matrices. In addition, their approach does not work well for very sparse matrices, which is the case in LiFE.

VIII. Conclusion and Future Work

In this paper, we presented designs to parallelize the multiplication of very large and sparse multiway arrays. These designs are used to accelerate the performance of the LiFE method which is part of the ENCODE framework. This method is useful in encoding dMRI, brain anatomy, and evaluation methods using multidimensional arrays. We proposed MPI and MPI + OpenMP-based LiFE models (collectively known as **MPI-LiFE**) and evaluated their performance on multiple clusters. We theoretically analyzed the complexity and speedup of our proposed models and correlated them with the performance results. On a single node on Stampede2, we were able to achieve a speedup of 8.7x. Overall, the maximum speedup achievable was 8.1x on RI2, 8.7x on Stampede2, and 4.5x on Stampede. We also showed that the speedup within a node is limited by the memory wall, which can be theoretically modeled and used for performance prediction.

In the future, we plan to further improve the performance and scalability of our designs by co-designing with the MPI

library. We also plan to explore the application of our designs for other applications using multiway arrays. Evaluation with more datasets and configurations is also left as future work.

REFERENCES

[1] F. Pestilli, J. D. Yeatman, A. Rokem, K. N. Kay, and B. A. Wandell, "Evaluation and Statistical Inference for Human Connectomes," *Nature Methods*, vol. 11, no. 10, pp. 1058–1063, Sep. 2014.

[2] H. Takemura, C. F. Caiafa, B. A. Wandell, and F. Pestilli, "Ensemble Tractography." *PLoS Computational Biology*, vol. 12, no. 2, pp. e1 004 692–, Feb. 2016.

[3] C. F. Caiafa and F. Pestilli, "Multidimensional Encoding of Brain Connectomes," *Scientific Reports*, Sep. 2017.

[4] D. Kim, S. Sra, and I. S. Dhillon, "A non-monotonic method for large-scale non-negative least squares," *Optimization Methods and Software*, vol. 28, no. 5, pp. 1012–1039, Oct. 2013.

[5] C. F. Caiafa and F. Pestilli, "Sparse Multiway Decomposition for Analysis and Modeling of Diffusion Imaging and Tractography," *arXiv preprint arXiv:1505.07170*, 2015.

[6] J. Li, Y. Shi, and A. W. Toga, "Mapping Brain Anatomical Connectivity Using Diffusion Magnetic Resonance Imaging: Structural connectivity of the human brain," *IEEE Signal Processing Magazine*, vol. 33, no. 3, pp. 36–51, Apr. 2016.

[7] B. A. Wandell, "Clarifying Human White Matter," *Annual Review of Neuroscience*, vol. 39, no. 1, pp. 103–128, Jul. 2016.

[8] A. Rokem, H. Takemura, A. S. Bock, K. S. Scherf, M. Behrmann, B. A. Wandell, I. Fine, H. Bridge, and F. Pestilli, "The Visual White Matter: The Application of Diffusion MRI and Fiber Tractography to Vision Science." *Journal of Vision*, vol. 17, no. 2, p. 4, Feb. 2017.

[9] Y. Shi and A. W. Toga, "Connectome Imaging for Mapping Human Brain Pathways." *Mol. Psychiatry*, vol. 340, p. 1234, May 2017.

[10] M. Bastiani, N. J. Shah, R. Goebel, and A. Roebroeck, "Human Cortical Connectome Reconstruction from Diffusion Weighted MRI: The Effect of Tractography Algorithm," *Human Brain Mapping Journal*, vol. 62, no. 3, pp. 1732–1749, 2012.

[11] M.-A. Cote, G. Girard, A. Boré, E. Garyfallidis, J.-C. Houde, and M. Descoteaux, "Tractometer: Towards Validation of Tractography Pipelines," *Medical Image Analysis*, vol. 17, no. 7, pp. 844–857, Oct. 2013.

[12] R. L. Goldstone, F. Pestilli, and K. Börner, "Self-portraits of the Brain: Cognitive Science, Data Visualization, and Communicating Brain Structure and Function," *Trends in Cognitive Sciences*, pp. 1–14, Jul. 2015.

[13] F. Pestilli, "Test-retest Measurements and Digital Validation for in vivo Neuroscience." *Scientific Data*, vol. 2, p. 140057, 2015.

[14] A. Daducci, A. D. Palù, A. Lemkaddem, and J.-P. Thiran, "COMMIT: Convex Optimization Modeling for Microstructure Informed Tractography." *Medical Imaging, IEEE Transactions on*, vol. 34, no. 1, pp. 246–257, Jan. 2015.

[15] MVAPICH2: High Performance MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE, http://mvapich.cse.ohio-state.edu/.

[16] C. F. Caiafa and A. Cichocki, "Computing Sparse Representations of Multidimensional Signals using Kronecker Bases." *Neural Computation*, pp. 186–220, Dec. 2012.

[17] A. Cichocki, D. Mandic, L. De Lathauwer, G. Zhou, Q. Zhao, C. Caiafa, and A. H. Phan, "Tensor Decompositions for Signal Processing Applications: From Two-way to Multiway Component Analysis," *IEEE Signal Processing Magazine*, vol. 32, pp. 145–163, Mar. 2015.

[18] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. USENIX Association, 2004, pp. 10–10.

[19] R. W. Hockney, "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Computing*, vol. 20, no. 3, pp. 389–398, 1994.

[20] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance Analysis of MPI Collective Operations," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 2005, pp. 8–pp.

[21] D. Kim, S. Sra, and I. S. Dhillon, "A Non-Monotonic Method for Large-Scale Non-Negative Least Squares," *Optimization Methods and Software*, vol. 28, no. 5, pp. 1012–1039, 2013.

[22] "OSU RI2 Cluster," http://ri2.cse.ohio-state.edu/.

[23] "TACC Stampede," https://www.tacc.utexas.edu/systems/stampede.

[24] "TACC Stampede2," https://www.tacc.utexas.edu/systems/stampede2.

[25] "STN96 Dataset," https://purl.stanford.edu/rt034xr8593.

[26] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[27] J. Zhang, H. Wang, and W. c. Feng, "cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on CPU+GPU," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. PP, no. 99, pp. 1–1, 2017.

[28] A. Kalyanaraman, W. R. Cannon, B. Latt, and D. J. Baxter, "MapReduce Implementation of a Hybrid Spectral Library-database Search Method for Large-scale Peptide Identification," *Bioinformatics*, vol. 27, no. 21, p. 3072, 2011. [Online]. Available: +http://dx.doi.org/10.1093/bioinformatics/btr523

[29] K. Hou, H. Wang, and W. C. Feng, "AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 780–789.

[30] M. Li, X. Lu, S. Potluri, K. Hamidouche, J. Jose, K. Tomko, and D. K. Panda, "Scalable Graph500 design with MPI-3 RMA," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2014, pp. 230–238.

[31] M. Li, J. Lin, X. Lu, K. Hamidouche, K. Tomko, and D. K. Panda, "Scalable MiniMD Design with Hybrid MPI and OpenSHMEM," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, Eugene, OR, USA, October 6-10, 2014*, 2014, pp. 24:1–24:4. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676893

[32] "Mantevo Project." [Online]. Available: https://mantevo.org/

[33] R. Kannan, G. Ballard, and H. Park, "A High-Performance Parallel Algorithm for Nonnegative Matrix Factorization," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, p. 9.