

DOI:

<https://doi.org/10.1016/j.jpdc.2017.10.013>

Access to this work was provided by the University of Maryland, Baltimore County (UMBC) ScholarWorks@UMBC digital repository on the Maryland Shared Open Access (MD-SOAR) platform.

Please provide feedback

Please support the ScholarWorks@UMBC repository by emailing scholarworks-group@umbc.edu and telling us what having access to this work means to you and why it's important to you. Thank you.

Performance Considerations for Scalable Parallel Tensor Decomposition

Thomas B. Rolinger^{a,*}, Tyler A. Simon^a, Christopher D. Krieger^a

^aLaboratory for Physical Sciences - College Park, Maryland 20740, United States

Abstract

Tensor decomposition, the higher-order analogue to singular value decomposition, has emerged as a useful tool for finding relationships in large, sparse, multidimensional data. As this technique matures and is applied to increasingly larger data sets, the need for high performance implementations becomes critical. A better understanding of the performance characteristics of tensor decomposition on large and sparse tensors can help drive the development of such implementations. In this work, we perform an objective empirical evaluation of three state of the art parallel tools that implement the Canonical Decomposition/Parallel Factorization tensor decomposition algorithm using alternating least squares fitting (CP-ALS): SPLATT, DFacTo, and ENSIGN. We conduct performance studies across a variety of data sets and evaluate the tools with respect to total memory required, processor stall cycles, execution time, data distribution, and communication patterns.

Furthermore, we investigate the performance of the implementations on tensors with up to 6 dimensions and when executing high rank decompositions. We find that tensor data structure layout and distribution choices can result in differences as large as 14.6x with respect to memory usage and 39.17x with respect to execution time. We provide an outline of a distributed heterogeneous CP-ALS implementation that addresses the performance issues we observe.

Keywords: sparse tensors, decomposition, parallel, performance analysis, Canonical Decomposition/Parallel Factorization, alternating least squares

1. Introduction

In the era of Big Data, corporations and scientists alike have a need to analyze large amounts of multi-dimensional data to find interrelations. Examples of higher-dimensional data sources include social network data, chemometric and pharmacological data, neuroscience signals such as EEGs, business transaction logs, and unstructured text from many sources. In all of these cases, analysts wish to discover previously unknown relationships between elements in data that is often sparse and multivariate.

Tensor decomposition is emerging as a useful tool

for extracting patterns from these types of data. Tensor decomposition operates on a *tensor*, or multi-dimensional array, and performs “multi-way” matrix decomposition. The original tensor is decomposed into a weighted sum of several rank-1 tensors. Each of these component tensors is represented as the outer product of n vectors, where n represents the number of dimensions of the original tensor. The components capture correlations found in the original tensor. This is similar to how a singular value decomposition (SVD) decomposes a two dimensional matrix. One algorithm for conceptually extending SVD to higher dimensions is the Canonical Decomposition/Parallel Factorization algorithm (known as CP for CANDECOMP/PARAFAC) using alternating least squares fitting (CP-ALS) [1].

CP-ALS is iterative and contains repeated high

*Corresponding author

Email addresses: tbrolin@lps.umd.edu (Thomas B. Rolinger), tasimon@lps.umd.edu (Tyler A. Simon), krieger@lps.umd.edu (Christopher D. Krieger)

level tensor operations, such as Kronecker and Khatri-Rao products. These operations contain indirect and irregular data accesses that prevent static optimization and challenge prefetching and caching hardware. Tensor products can be reduced to multiple sparse matrix operations and the manner in which CP-ALS is executed using such operations has a significant practical impact on code performance, total memory usage, and parallelizability. Similarly, the choice of data structures used to hold tensors affects memory access patterns, which in turn have a bearing on data locality and bandwidth requirements. The need for high performance implementations and efficient data structures becomes imperative as tensors become very large and sparse, which is often the case for real-world data.

Analyzing the performance of CP-ALS on large, sparse tensors is an active area of research. Our goal in this paper is to perform an objective, empirical study that identifies the key characteristics of CP-ALS that contribute to significant gaps in performance and scalability. Such results can help drive the development of future high performance CP-ALS implementations. To this end, we present a performance evaluation of three different current CP-ALS implementations, namely DFacTo, SPLATT, and ENSIGN, using several metrics across a range of data sets. At the time of this writing, these tools represent the state of the art in shared and/or distributed-memory sparse tensor decomposition. The contributions of this paper are as follows:

1. We present an extensive analysis of parallel CP-ALS implementations on large, sparse tensors with respect to memory usage, processor stall cycles, execution time and scalability. To the best of our knowledge, this is the first study of its kind to look at such metrics together.
2. We provide insight for future implementations of CP-ALS through a discussion of key implementation choices that lead to significant reductions in performance. We find that certain choices can result in differences as large as 14.6x with respect to memory usage and 39.17x with respect to decomposition runtime. We then present an outline for a distributed heterogeneous CP-ALS implementation that would address some of the performance challenges that

we identified.

3. We investigate the performance side-effects of executing high-rank decompositions using CP-ALS, as well as performing CP-ALS on tensors with up to 6 dimensions. As far as we know, only limited performance analysis exists regarding the scalability of CP-ALS implementations for high-rank decompositions and tensors with more than three dimensions.
4. We evaluate the performance of data distribution schemes and communication patterns for distributed memory implementations of CP-ALS, focusing on scalability with respect to the number of processes used, the size of the tensor, and the rank of the decomposition performed.

The rest of this paper is organized as follows. In Section 2, we provide an overview of the CP-ALS algorithm and briefly discuss the known performance bottlenecks and implementation issues. We describe the three different CP-ALS implementations and their approaches to tensor storage and decomposition in Section 3. In Section 4, we detail our experimental setup and present the performance results for each approach. In Section 5, we discuss our observations and findings from the performance results, highlighting key characteristics of the tools that have significant impact across the performance metrics. We present an outline for an implementation of CP-ALS that is both distributed and heterogeneous in Section 6. Lastly, Section 7 provides concluding remarks.

2. Tensor Decomposition Using CP-ALS

Spectral based matrix decompositions such as SVD have been applied to a large variety of applications such as image compression, signal processing, and statistical analysis. The CP algorithm extends two dimensional SVD to higher order tensors. As shown in Algorithm 1, CP constructs an approximation to the original tensor \mathcal{X} using the sum of R components that are rank-1 tensors, $\mathbf{A}^{(1)} \dots \mathbf{A}^{(R)}$. The quality of the approximation, which is expressed as a *fit* measure, depends on R as well as the rank of \mathcal{X} . The component tensors can be determined using several iterative methods. In this paper, we focus on

the most common method, alternating least squares (ALS).

Algorithm 1, drawn from the work of Kolda and Bader [1], is a basic sketch of the CP-ALS algorithm. \mathbf{V}^\dagger denotes the Moore-Penrose pseudoinverse. Observe on line 6 that most of the computational and storage complexity stems from multiplying a reformatted, or *matricized*, tensor $\mathcal{X}^{(n)}$ with the Khatri-Rao product (written as \odot) of the factor matrices, $\mathbf{A}^{(n)}$. This operation is often referred to as the matricized tensor times Khatri-Rao product (MTTKRP). The MTTKRP is computed once per dimension of the tensor for each iteration of CP-ALS. Because each $\mathbf{A}^{(n)}$ has dimensions $I_n \times R$, where I_n is the size of the n th dimension, the Khatri-Rao product requires $O(R \times \prod_{n=1}^N I_n)$ storage space. Unless the factors are very sparse, this product is totally dense due to high fill-in and can easily require many times more memory than \mathcal{X} . This considerable increase in required memory is called the *intermediate data explosion problem* [2]. Methods for addressing this problem have been proposed for related tensor decompositions [3, 4]. Some of the approaches examined in this paper propose novel data structures and algorithmic improvements to address this issue.

Algorithm 1 General Form of CP-ALS

```

1: Procedure CP-ALS( $\mathcal{X}$ ,  $R$ )
2: initialize  $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$  for  $n = 1, \dots, N$ , where
    $N = \text{order}(\mathcal{X})$ 
3: repeat
4:   for  $n = 1, \dots, N$  do
5:      $\mathbf{V} \leftarrow \mathbf{A}^{(1)T} \mathbf{A}^{(1)} * \dots * \mathbf{A}^{(n-1)T} \mathbf{A}^{(n-1)} * \mathbf{A}^{(n+1)T} \mathbf{A}^{(n+1)} * \dots * \mathbf{A}^{(N)T} \mathbf{A}^{(N)}$ 
6:      $\mathbf{A}^{(n)} \leftarrow \mathcal{X}^{(n)} (\mathbf{A}^{(N)} \odot \dots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \dots \odot \mathbf{A}^{(1)}) \mathbf{V}^\dagger$ 
7:     normalize columns of  $\mathbf{A}^{(n)}$  (storing norms as  $\lambda$ )
8:   end for
9: until fit ceases to improve or maximum iterations reached
10: return  $\lambda, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(N)}$ 

```

Basic knowledge of this central algorithm can inform our study of concrete CP-ALS implementations. We observe that the MTTKRP is the critical

routine for CP-ALS, requiring the majority of the computation as well as posing memory growth issues. Furthermore, careful consideration needs to be given to the storage of the tensor, \mathcal{X} , and factor matrices, $\mathbf{A}^{(n)}$, as they contribute a significant amount to the overall memory consumption. Their design also affects memory access patterns, which in turn can affect execution time. When parallelizing CP-ALS for distributed memory systems, the way in which the tensor and factor matrices are distributed across multiple processes will impact the overall performance and scalability of the implementation.

3. Parallel Implementations of CP-ALS Tensor Decomposition

In this work, we examine three different parallel tools for performing CP-ALS tensor factorization: DFacTo, SPLATT, and ENSIGN. While all three perform the CP-ALS decomposition, they each approach both the computation and the supporting data structures in significantly different ways. A computational and memory complexity analysis of each approach is outside the scope of this paper, but can largely be found in the SPLATT and DFacTo publications [5, 6]. These codes use shared memory parallelization using OpenMP or distributed memory parallelism using MPI. Tensor decomposition implementations running on Cloud or MapReduce architectures, such as GigaTensor [2] or HaTen2 [7], are excluded from this study. Below, we highlight a few of the key features of each approach and provide references containing further details.

3.1. DFacTo

DFacTo¹ is a distributed CP-ALS implementation developed by researchers at Purdue University [6]. DFacTo computes the MTTKRP column by column, where each column requires two sparse matrix by vector multiplications (SpMV). This provides the opportunity to use an optimized SpMV routine from a vendor or open source library to improve performance. DFacTo also reuses the index arrays of a

¹DFacTo code is available at <http://web.ics.purdue.edu/~choi240/>

Compressed Sparse Row (CSR) data structure to reshape results into sparse matrices with the same non-zero pattern. Although the algorithm is general, current implementations of DFacTo are limited to 3-dimensional tensors.

DFacTo is designed for distributed execution over MPI and uses a coarse-grained approach to data distribution. Each MPI process is assigned a set of contiguous slices for each dimension of the tensor and is responsible for the corresponding rows in the factor matrices. This coarse-grained approach has the advantage of simplifying the MTTKRP, only requiring communication at the end of each iteration to exchange updated rows in the factor matrices. However, full copies of the factor matrices are held by each process since a given process may be assigned non-zeros that span all of the dimensions of the tensor. DFacTo also allocates a dense matrix for each dimension of the tensor to store the output of the MTTKRP corresponding to that dimension. Since DFacTo is limited to 3-dimensional tensors, this equates to three dense matrices that have the same dimensions as the factor matrices. Each MPI process also has a full copy of these three matrices.

DFacTo is implemented in C++ and uses the Eigen library [8] for the storage of dense and sparse matrices as well as for basic linear algebra subprograms (BLAS). The dense matrices are stored in row-major order and the sparse matrices are stored in CSR format.

3.2. SPLATT

SPLATT² is an open source software toolbox for sparse tensor factorization and related kernels developed at the University of Minnesota [5, 9, 10]. SPLATT includes routines for computing least-squares CP, as well as constrained CP (e.g., non-negative or sparse CP) and CP with missing values (i.e., tensor completion) [11]. In this study, we focus on SPLATT’s implementation of CP-ALS. Version 2.0.0, used in this evaluation, takes advantage of both OpenMP shared memory parallelization [5, 9] and MPI-based distributed memory parallelization [10]. In this paper, SPLATT using MPI

will be denoted as Distributed Memory SPLATT (DMS), while the OpenMP version will be referenced simply as SPLATT.

SPLATT uses a novel data structure for the storage of sparse tensors, *compressed sparse fiber* (CSF), that addresses the memory/computation trade-off of computing tensor-matrix products [9]. CSF achieves a small memory footprint by operating on a single compressed tensor instead of a compressed tensor for each dimension of the original data. To reduce the amount of additional floating point operations that are introduced by using a single compressed tensor, a novel shared-memory parallelized algorithm was developed to perform tensor-matrix multiplication on a tensor stored in CSF. Furthermore, CSF allows SPLATT to store and operate on tensors with an arbitrary number of dimensions. SPLATT employs partial tiling to the tensor modes in order to expose parallelism, giving priority to the longest modes. This technique was first introduced along with the CSF data structure [9], but recent improvements allow SPLATT to only tile one of the tensor modes for parallelism [12]. The CSF data structure is general and is not specifically tailored towards CP-ALS, as it has been used for stochastic gradient descent (SGD) and coordinate descent (CCD++) in tensor completion algorithms.

DMS uses a m -dimensional decomposition over the tensor, where m is the number of dimensions in the tensor. This decomposition is then used to determine one-dimensional partitions of the factor matrices. Groups of MPI processes that share a non-zero in the tensor are referred to as a *layer*. The rows of the factor matrices are divided into chunks and are collectively owned by all processes in a layer. This medium-grained approach avoids the costs of pre-processing steps required by fine-grained approaches, such as hypergraph partitioning [13], as well as avoids the need to store full copies of the factor matrices on each MPI process that is required for coarse-grained approaches like DFacTo. The sub-tensor held by each MPI process is stored in a CSF data structure. This gives DMS the ability to utilize SPLATT’s shared memory parallelized MTTKRP algorithm that operates on tensors in CSF, achieving a hybrid of shared and distributed memory parallelization. However, we defer this OpenMP+MPI capabil-

²SPLATT source code is available from <https://github.com/ShadenSmith/splatt>

ity for a future study.

SPLATT/DMS is written in C and allows for external BLAS/LAPACK libraries to be specified during the build process. The version of SPLATT/DMS evaluated in this work has been improved from the version used in our prior evaluation [14]. Details about those improvements can be found in [12]. Note that the version of the code used here does not include the AVX2 intrinsics or improvements regarding hub slices.

3.3. ENSIGN

The ENSIGN Tensor Toolbox³ is a commercially available tensor decomposition package. It is a stand-alone package containing C implementations of a variety of tensor decomposition methods, namely, CP-ALS (used in this study), variants of CP for non-negative decompositions (Alternate Poisson Regression with Multiplicative Update, Alternate Poisson Regression with Projected Damped Newton based solver for Row subproblems), CP-OPT-Joint (optimized coupled CP tensor decompositions based on gradient descent), Tucker-ALS, and low-rank updates to Tucker decompositions.

ENSIGN tensor methods are implemented using optimized sparse tensor data structures, called mode-specific sparse (MSS) tensor and mode-generic sparse (MGS) tensor data structures [4]. The data structures used in ENSIGN were developed to provide a foundation for a range of decomposition methods and tensor sizes and are not specifically optimized for CP-ALS. ENSIGN tensor methods are parallelized and optimized for load balancing and data locality using techniques developed by Baskaran et al. [15]. However, these optimizations are not enabled for CP-ALS and are therefore not active in this study.

ENSIGN’s CP-ALS implementation is written in C and uses OpenMP-based shared memory parallelism. It also contains customized BLAS routines that are parallelized. ENSIGN tensor decomposition methods, including CP-ALS, support decomposition of tensors of arbitrary order and are not limited to three dimensional tensors. Version 4.0 of ENSIGN was evaluated in this study.

³<https://www.reservoir.com/support/ensign/>

4. Performance Evaluation

The key contribution of this paper is an extensive performance evaluation of parallel CP-ALS tools. The purpose of the evaluation is to better understand CP-ALS from the perspective of data structure efficiency, execution time, memory usage, and efficacy of parallelization techniques. Our evaluation incorporates a suite of data sets that vary widely in the total number of non-zeros, density, and number of dimensions. We also investigate the performance of the tools as they perform increasingly higher rank decompositions.

Comparisons of multithreaded tools running on a multicore, shared memory machine to distributed memory MPI codes running on multiple machines are inherently problematic. To mitigate this difference between shared and distributed approaches somewhat, we present results from running the tools on a single multicore machine in Sections 4.2 – 4.4. In the multithreaded cases, we bind one thread to one core, while for the MPI based codes, we assign one MPI rank to each core. This allows us to discuss the performance of all of the tools together, knowing each tool had the same hardware resources, rather than attempting to account for some implementations running on more hardware nodes. We present a study of the distributed codes, DMS and DFacTo, running on multiple machines and report those results in Section 4.5.

Throughout this section, we present the results of various performance measurements of DFacTo, SPLATT, DMS, and ENSIGN. We measure memory usage, processor stall cycles, runtime, and scalability. We defer a more thorough, in-depth analysis of key patterns and findings until Section 5.

4.1. Experimental Setup

Performance of each run was measured on a single node within a 24 node cluster. The node had 512 GB of DDR4 DRAM and consisted of two 10 core E5-2650v3 Xeon “Haswell” processors, each running at 2.3 GHz with 25MB of shared last level cache. A 56 Gbit/s FDR Infiniband network connected each node within the cluster, which was utilized in the multiple machine runs.

All tools were built using the same version of OpenMP (OpenMP 4.0 standard, GOMP implementation) and the same compiler (gcc 5.2.0, -O3 optimization level). DMS and DFacTo use MVAPICH 2.2b for MPI. SPLATT and DMS were built using the same parallel BLAS library (OpenBLAS 0.2.15) and LAPACK (version 3.6.0). However, for the results presented in this section, DMS did not make use of multiple threads in OpenBLAS and instead ran serially on each MPI rank. DFacTo’s build process is intended to use the Intel compiler and can make use of the parallelized BLAS found in Intel’s Math Kernel Library. However, we built all of the tools using gcc 5.2.0. In this case, DFacTo defaults to using Eigen’s BLAS (version 3.3), which we run serially within each MPI rank. This makes it comparable to DMS, which is likewise running a single thread in OpenBLAS per MPI rank. ENSIGN uses its own custom parallel BLAS routines rather than those provided by any external library.

The data sets used in this work are summarized in Table 1. These data sets were chosen because they are generally available and many have been widely used in prior work. The Never Ending Language Learner (NELL) data [16] contains subject-verb-object relationships between words. The Yelp Phoenix Academic Data set, from the Yelp Dataset Challenge⁴, contains reviews of businesses. The VAST data set [17] is from the 2015 VAST Mini-Challenge 1 and contains information about visitor movement in an amusement park. We have included two different versions of the VAST data set: the original 5-dimensional data set (VAST 5D) and a version with the last two dimensions removed (VAST 3D). The Netflix data set contains movie review data from the Netflix Prize Competition [18]. CELLAR TRACKER and RATE BEER consist of reviews of beverages and were drawn from the Stanford Network Analysis Project (SNAP) [19], but have subsequently been removed from SNAP at the request of the data owners. The Flickr data set [20] contains tags from user images on Flickr in the form of *user-image-tag-date*. The Chicago Crimes data set [21] contains information about crimes committed in the city of Chicago from January 2001 to

March 2017. As mentioned previously, DFacTo is limited to 3-dimensional tensors, so the higher dimensional tensors in Table 1 will only be evaluated by ENSIGN, SPLATT and DMS. The VAST, Flickr and NELL data sets were retrieved from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [22].

For the 3-dimensional data sets, we performed runs with each tool on 1, 2, 4, 8, 16 and 20 cores, computing CP-ALS decompositions (denoted as CPDs) of rank 35, 50, 75, and 100. For the higher dimensional data sets, we performed the same runs but excluded DFacTo. For each test, the tools performed iterations of CP-ALS until convergence. The tolerance value used for all tests was 1E-5 and the maximum number of iterations was set to 100. For each metric excluding memory usage, the results we present represent the average from 5 runs of the tools

When presenting the higher rank decomposition results, we will focus on the data sets that highlight key performance results. To this end, we present results either for the NELL-1 or VAST 3D data sets. The NELL-1 data set is the largest 3-dimensional data set with respect to the number of non-zeros and emphasizes the performance scalability of the tools when increasing the decomposition rank. The VAST 3D data set is the most dense data set in our suite and has a small third dimension, being only two in size. Such features tend to affect parallel performance.

For SPLATT and ENSIGN, the number of cores used was adjusted by varying the number of OpenMP threads on a single machine. In order to create a fair comparison to the shared memory tools, we varied the number of MPI ranks used on a single machine for DMS and DFacTo for the results presented in Sections 4.2 – 4.4, binding each rank to a core. For the results presented in Section 4.5, we ran DMS and DFacTo on up to 20 machines, assigning 1 MPI rank per machine and one core per rank.

We modified each tool to ensure that the same initial factor matrices would be generated given the same random seed. This results in each tool requiring the same number of iterations of the CP-ALS algorithm to arrive at the same final fit. The random seed used for all runs was 9058628.

While SPLATT is capable of finding cache-friendly reorderings and utilizing them with a novel

⁴https://www.yelp.com/dataset_challenge/

Table 1: Properties of Data Sets

Name	Dimensions	Non-Zeros	Density
CHICAGO CRIMES	2M x 81 x 396 x 172 x 74k x 124k	5M	4.87E-17
YELP	41k x 11k x 75k	8M	2.2E-7
CELLAR TRACKER	34k x 363k x 163k	20M	1.1E-8
VAST 3D	165k x 11k x 2	26M	6.91E-3
VAST 5D	165k x 11k x 2 x 100 x 89	26M	7.77E-7
RATE BEER	27k x 105k x 294k	62M	8.3E-8
NELL-2	12k x 9k x 29k	77M	2.4E-5
NETFLIX	480k x 17k x 2k	100M	5.4E-6
FLICKR	320K x 28M x 2M x 731	112M	1.07E-14
NELL-1	3M x 2M x 25M	143M	9.05E-13

form of cache tiling [5], this capability is not enabled in the version we used. However, we make use of SPLATT’s partial tensor tiling on data sets with small modes (both VAST data sets, CHICAGO CRIMES and FLICKR). The effects of this tiling feature are discussed more in Section 5.4.

It should be noted that none of the tools have been optimized for serial performance. Therefore, some overhead and inefficiencies are incurred in such a case. We call out these issues where relevant in the following sections. The purpose of evaluating the serial performance of the tools is not only to establish a baseline for scalability, but also to investigate the quality and efficiency of the data structures used. Furthermore, observing the serial performance allows us to uncover key differences between the tools that would otherwise be overshadowed by parallelization effects.

4.2. Memory Usage

One indirect measure of the efficiency of both algorithms and data structures is the total amount of memory required to execute the algorithm for a given data set. This is particularly important in light of the intermediate data explosion problem mentioned in Section 2. Also, poor choices in data distribution schemes can lead to scalability problems for MPI-based implementations. To investigate these potential issues, we measured the largest resident set size (RSS) needed by each tool and data set combination. We sampled the RSS usage once per second over the CPD execution when running the tools on a single core as well as on 20 cores on a single machine.

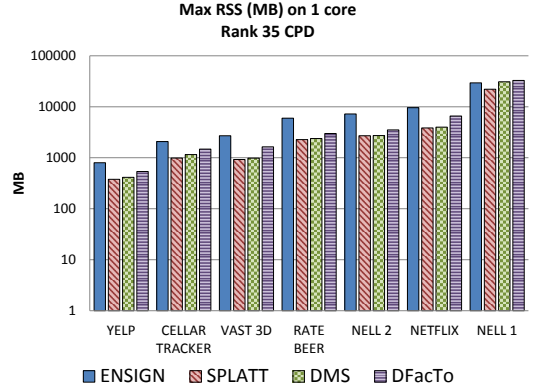


Figure 1: Maximum RSS on 1 core when computing a rank-35 CPD. The vertical axis is logarithmic and represents memory usage in MB. The data sets shown here represent only 3-dimensional tensors. Shorter bars represent better performance.

This excludes the data preparation and initialization phase, which takes a tensor data file and builds up the internal data structures, as well as the final output phases.

4.2.1. Serial Memory Usage

The maximum RSS seen by each tool on the 3-dimensional data sets when using a single core (ENSGN and SPLATT) and process (DMS and DFacTo) are presented in Figure 1. These results are for a rank-35 CPD. SPLATT uses the least amount of memory on all of the data sets while ENSGN consumes the most memory on average. The reason for this difference is due to the choice of data structures and is discussed further in Section 5.4. DMS uses slightly more memory on average than SPLATT despite the fact that SPLATT and DMS are virtually

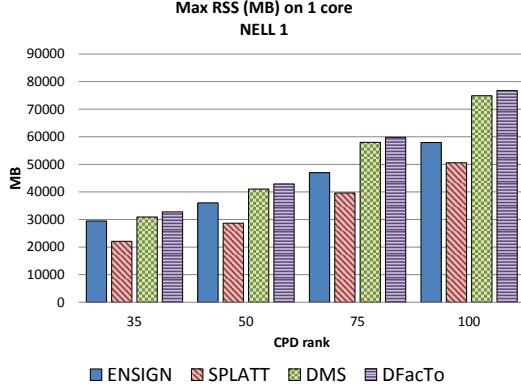


Figure 2: Maximum RSS on 1 core when computing rank 35, 50, 75 and 100 CPDs on the NELL-1 data set.

identical algorithmically and both use the CSF data structure. This is due to DMS creating communication data structures even in the single MPI rank case, as well as a full copy of the factor matrices that would otherwise be distributed across multiple MPI ranks. The difference in memory consumption between SPLATT and DMS is greater on the NELL-1 data set because the factor matrices are very large, leading to a significant increase in memory usage from the full copy created by DMS.

We then ran the tools across the same data sets but increased the rank of the CPD that was computed. Figure 2 shows the maximum RSS values on the NELL-1 data set when computing CPDs of rank 50, 75 and 100, as well as the rank-35 results shown in Figure 1. It is clear that DMS and DFacTo’s memory usage are increasing much faster than that of ENSIGN and SPLATT as the CPD rank grows larger. The reason for this behavior is explained in Sections 5.1 and 5.2.

We also evaluated the tools’ memory usage on tensors with more than three dimensions, with the exception of DFacTo since it is restricted to 3-dimensional tensors. These results are shown in Figure 3. The tools exhibit the same memory usage trend that we observe on the 3-dimensional data sets in Figure 1, namely that SPLATT uses the least amount of memory while DMS uses slightly more except on the larger data sets (FLICKR), where it uses significantly more memory.

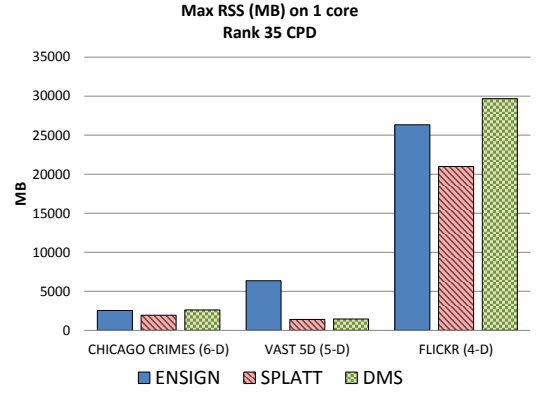


Figure 3: Maximum RSS on 1 core for the higher dimensional data sets when computing a rank-35 CPD.

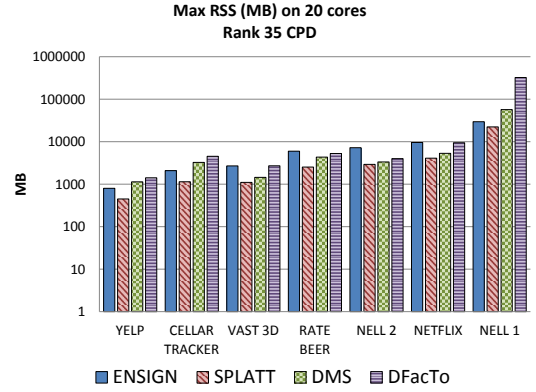


Figure 4: Maximum RSS on 20 cores when computing a rank-35 CPD. The vertical axis is logarithmic and shown in MB. The data sets shown here represent only 3-dimensional tensors.

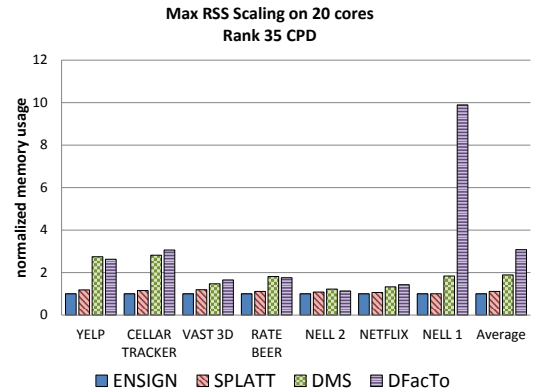


Figure 5: Maximum RSS scaling on 20 cores when computing a rank-35 CPD. The y-axis represents how many times more memory is used when compared to the single core results from Figure 1. The average scaling of the tools is shown on the far right. Shorter bars represent better performance.

4.2.2. Parallel Memory Usage

To evaluate how the tools’ memory usage is affected by parallelization, we performed runs on the 3-dimensional data sets using 20 cores (ENSIGN and SPLATT) and 20 processes on a single machine (DMS and DFacTo). The total memory usage required by each tool’s CPD is shown in Figure 4 and the scaling of memory usage from 1 to 20 cores is shown in Figure 5. Note that the difference in memory usage between the tools is so large that the vertical axis in Figure 4 is logarithmic. Indeed, DFacTo uses 14.6 times more memory than SPLATT on the NELL-1 data set. The memory usage for ENSIGN is virtually the same as in Figure 1, indicating that there is a minimal amount memory overhead for thread-local structures. However, SPLATT does exhibit a slight increase in memory due to the replication of parts of the CSF data structure that are used for thread-local computations. On the other hand, DMS uses an average of 1.9 times more memory than it did when running on 1 process and DFacTo uses an average of 3.08 times more memory. These large increases are due to the MPI data distribution schemes employed by DMS and DFacTo. More details on these schemes and their impact on performance is provided in Section 5.2.

We also increased the rank of the CPD when running on 20 cores. Figures 6 and 7 show the maximum RSS usage on 20 cores when computing CPDs of rank 35, 50, 75 and 100 on the VAST 3D and NELL-1 data sets, respectively. Ideally, each tool’s memory increase would only include the extra storage required for the factor matrices as they grow larger with the increasing CPD rank. For the VAST data set, this would be an increase of roughly 86 MB from a rank-35 CPD to a rank-100 CPD. ENSIGN is the closest to the ideal memory increase, requiring about 90 MB more memory when performing a rank-100 CPD, while DFacTo requires an additional 1957 MB of memory and DMS requires 525 MB of additional memory. DFacTo and DMS’ extra memory usage can be attributed to their approaches to data distribution among MPI ranks. SPLATT requires 370 MB more memory when performing a rank-100 CPD, which is largely due to the allocation of thread-local factor matrices that are used to avoid locking in cases where the factor matrices are

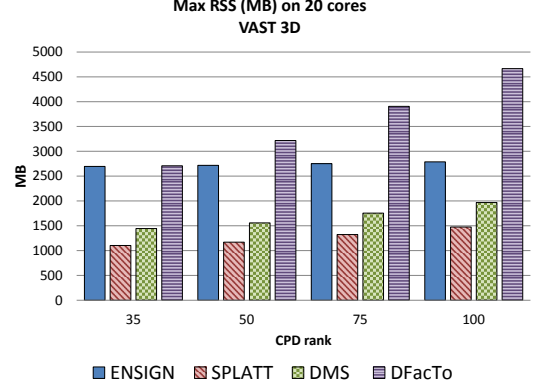


Figure 6: Maximum RSS usage on 20 cores when computing rank 35, 50, 75 and 100 CPDs on the VAST 3D data set.

sufficiently small. It should be noted that this replication is a tunable parameter for SPLATT. While both SPLATT and DMS require more extra memory than ENSIGN as the CPD rank increases on the VAST 3D data set, ENSIGN consumes a significant amount more total memory than SPLATT and DMS. For the NELL-1 data set, DFacTo was unable to run on our 512 GB system when computing a CPD of rank-75 and higher with 20 processes and was omitted from Figure 7. When computing a rank-75 CPD with a single MPI rank on NELL-1, DFacTo consumed 59.8 GB, as shown in Figure 2. DFacTo’s memory usage increases by a factor of 9.9 on NELL-1 when using 20 MPI ranks versus a single MPI rank, as shown in Figure 5. Therefore, DFacTo would be predicted to use roughly 592 GB of memory when computing a rank-75 CPD on 20 MPI ranks and would be unable to run on single 512 GB machine. For the higher dimensional data sets, the memory usage on 20 cores for ENSIGN, SPLATT and DMS follow the same trend as the single core case shown in Figure 3. We present these results in Figure 8.

4.3. Processor Stall Cycles

In addition to total memory footprint, we also looked at the efficiency of the data accesses in each implementation. To this end, we used the CPU’s performance monitoring unit to measure the number of cycles the core was stalled during execution of the CP-ALS algorithm. We then divided the stall cycles by the number of non-zeros in the data set being processed. This yields a metric, *stall cycles per non-zero*, that is somewhat comparable across tools and

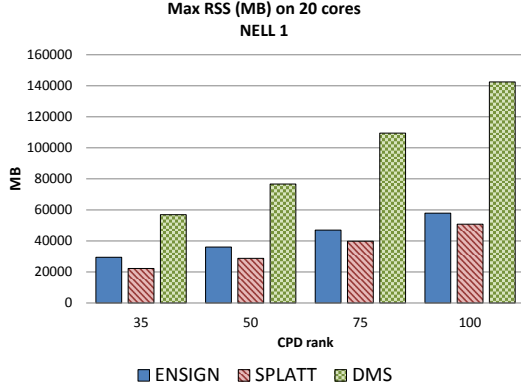


Figure 7: Maximum RSS usage on 20 cores when computing rank 35, 50, 75 and 100 CPDs on the NELL-1 data set. DFacTo has been omitted from these results because it required more memory than a single machine in our system had available when computing a CPD of rank-75 or higher.

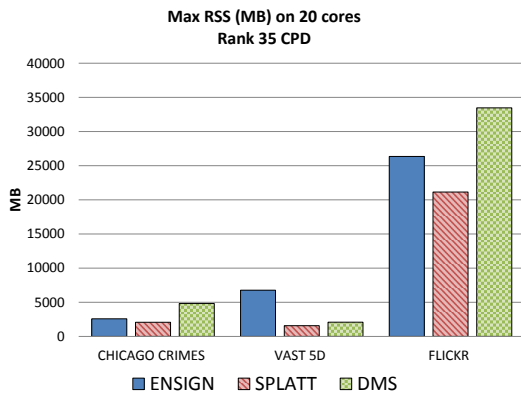


Figure 8: Maximum RSS on 20 cores for the higher dimensional data sets when computing a rank-35 CPD.

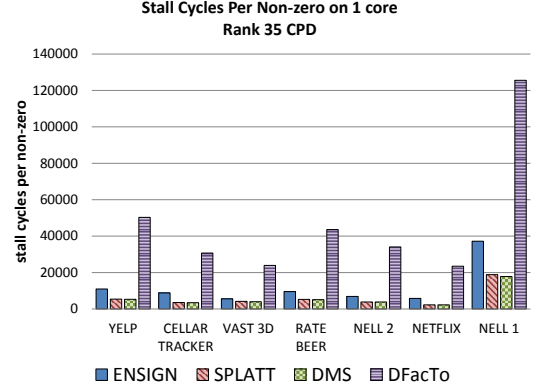


Figure 9: Stall cycles per non-zero on 1 core when computing a rank-35 CPD. Shorter bars represent better performance.

data sets. It indirectly reflects spatial and temporal locality and the amount of latency exposure.

The stall cycle measurements were taken from the same portion of the CP-ALS algorithm as the memory usage measurements presented in Section 4.2 and were measured using PAPI event RES_STL.

4.3.1. Serial Stall Cycles

Figure 9 presents the stall cycles per non-zero across the 3-dimensional data sets when using a single core and computing a rank-35 CPD. SPLATT and DMS consistently exhibit the fewest number of stalls, with DMS performing marginally better than SPLATT on average. ENSIGN stalls 1.4 – 2.56 times more than DMS. DFacTo has by far the highest number of stall cycles per non-zero, stalling 6 – 10.32 times more than DMS.

The stall cycles per non-zero are loosely correlated with the total memory usage seen in Figure 1. This reflects that higher memory usage leads to more cache misses and higher memory bandwidth requirements. Indeed, we observe that ENSIGN consumes more memory than SPLATT and DMS and it consistently exhibits more stall cycles per non-zero than both of the other tools. However, DFacTo does not follow this trend. While its memory usage is lower than ENSIGN for every data set besides NELL-1, it exhibits an average of 4.18 times more stall cycles per non-zero than that of ENSIGN. This is largely due to the fact that DFacTo computes the MTKRP in a column-oriented fashion, rather than the row-oriented pattern used by the other tools. The benefits of row versus column data traversals are discussed in

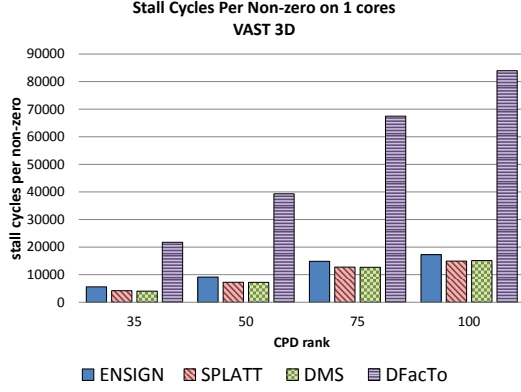


Figure 10: Stall cycles per non-zero on 1 core when computing rank 35, 50, 75 and 100 CPDs on the VAST 3D data set.

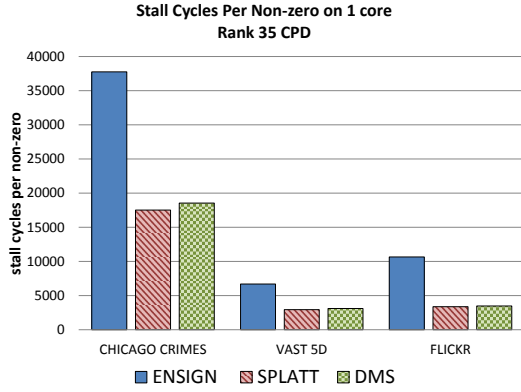


Figure 11: Stall cycles per non-zero on 1 core when computing a rank-35 CPD on the higher-dimensional data sets.

Section 5.1.

Figure 10 shows the stall cycles per non-zero on the VAST 3D data set as the CPD rank increases from 35 to 100. We see an identical trend for ENSIGN, SPLATT and DMS as in Figure 9, namely that SPLATT and DMS have the fewest number of stalls and ENSIGN consistently stalls more than both. Indeed, this trend is also observed on the other data sets when varying the rank of the decomposition. However, DFacTo’s stall cycles per non-zero clearly increase more as the CPD rank becomes larger. DFacTo experiences an average of 5.5 times more additional stall cycles than the other tools as the CPD rank increases. DFacTo’s poor scalability is due to its column-oriented computation of the MTTKRP.

We also investigated the stall cycles per non-zero on the higher dimensional data sets: CHICAGO CRIMES, VAST 5D and FLICKR. Figure 11 shows the stall cycles per non-zero for ENSIGN, SPLATT

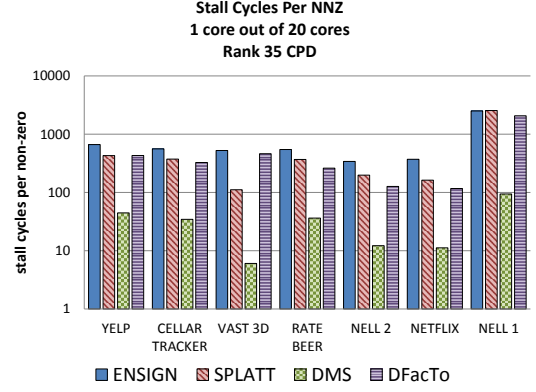


Figure 12: Stall cycles per non-zero on 20 cores when computing a rank-35 CPD. These results are for 1 core out of the 20 that were used. The vertical axis is logarithmic.

and DMS. We observe a similar trend on these higher dimensional data sets as we saw on the 3-dimensional data sets in Figure 9. It is interesting to note that the number of stall cycles per non-zero for the CHICAGO CRIMES data set is significantly higher than the other data sets that have many more non-zeros. We believe this difference is related to the sparsity distribution of the tensors, but a more thorough investigation will be left for future work.

4.3.2. Parallel Stall Cycles

To further evaluate the efficiency of each tool’s data accesses, we observed the stall cycles per non-zero across the data sets when using 20 cores on a single machine. We isolated a single thread/MPI rank out of the 20 and collected its RES_STL to calculate the stall cycles per non-zero.

The results for the 3-dimensional data sets are presented in Figure 12. Note that the vertical axis is logarithmic due to the significant differences between the tools. It is clear that DMS exhibits the fewest stall cycles per non-zero on all of the data sets. Indeed, it stalls an average of 31.46 times less than ENSIGN and an average of 20.75 times less than DFacTo. We note that the MPI tools generally exhibit fewer stall cycles per non-zero than the OpenMP tools. Because both DMS and DFacTo are distributed-memory based tools, each of the MPI ranks has its own memory space. Therefore, there are no ownership coherency conflicts between the ranks when writing to their data. On the other hand, ENSIGN and SPLATT are shared-memory based tools

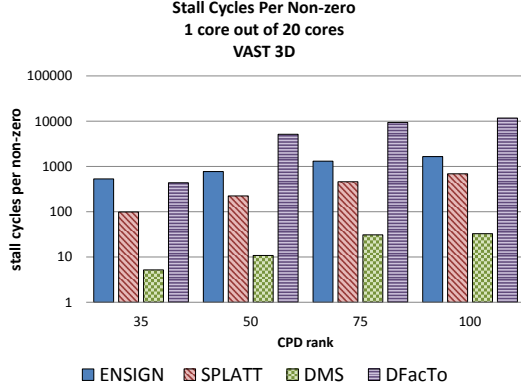


Figure 13: Stall cycles per non-zero on 20 threads/processes when computing rank 35, 50, 75 and 100 CPDs on the VAST 3D data set. These results are for 1 core out of the 20 that were used. The vertical axis is logarithmic.

and have 20 cores, each with its own caches, potentially conflicting when writing to the same memory addresses. While both ENSIGN and SPLATT suffer from these issues, SPLATT stalls an average of 2.03 times less than ENSIGN. This is due to SPLATT’s utilization of thread-local data structures that minimize the memory access contention between threads.

We also measured the multi-core stall cycles of the tools on the data sets when varying the CPD rank. Figure 13 shows the stall cycles per non-zero for the tools on the VAST 3D data set when computing CPDs of rank 35, 50, 75 and 100. Note that the difference in stall cycles between each tool is so great that the vertical axis is logarithmic. The difference between these results and those of the single core case shown in Figure 10 is that the tools exhibit a much smaller increase in the amount of stall cycles per non-zero as the CPD rank increases. However, despite having the advantage of being an MPI-based approach, DFacTo still exhibits a significant amount of additional stall cycles as the CPD rank increases.

For the higher dimensional data sets, the stall cycles per non-zero when running on 20 threads/processes follow the same trend as the single core case in Figure 11, with the exception of the CHICAGO CRIMES data set where SPLATT stalls slightly more than ENSIGN. We present these results in Figure 14.

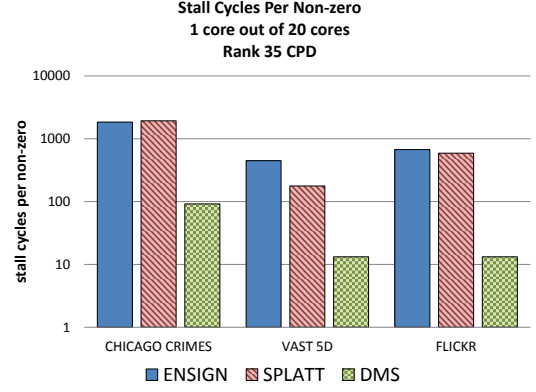


Figure 14: Stall cycles per non-zero on 20 cores when computing a rank-35 CPD on the higher-dimensional data sets. These results are for 1 core out of the 20 that were used. The vertical axis is logarithmic.

4.4. Runtime and Scalability

To focus on the performance of the CP-ALS implementations, we measured the total CPD execution runtime for each of the tools across the different data sets. These runtime measurements were taken from the same portion of the CP-ALS algorithm as the memory usage and stall cycles per non-zero measurements from Sections 4.2 and 4.3. By only measuring the CPD execution time, we remove any pre- and post processing or initialization overhead of the tools. We initially present results for single core runs in order to ignore parallelization and focus on the quality of the data structures and algorithmic implementations without accounting for the overhead of multiple threads/processes. We evaluate the CPD runtime scalability of the tools as more threads/processes are used and present those results in Section 4.4.2.

4.4.1. Serial Runtime

Figure 15 shows the runtime for each tool to compute a rank-35 CPD on the 3-dimensional data sets when using 1 core. Note that the difference in runtimes is so great that the vertical axis is logarithmic. We observe that the runtime results are correlated with the stall cycles per non-zero results shown in Figure 9. This suggests that the CP-ALS algorithm execution time is dominated by memory access latency. Furthermore, as noted in Section 4.3, the stall cycles per non-zero are loosely correlated with the memory usage results presented in Section

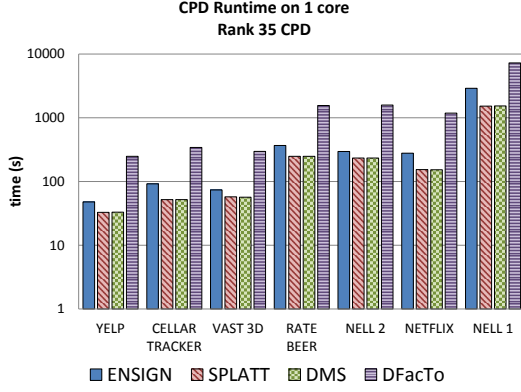


Figure 15: CPD runtime in seconds when performing a rank-35 CPD on 1 core. The vertical axis is logarithmic. Shorter bars represent better performance.

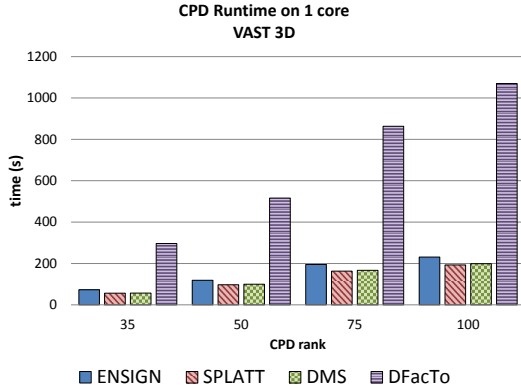


Figure 16: CPD runtime in seconds on 1 core when performing a rank 35, 50, 75 and 100 CPD on the VAST 3D data set.

4.2. Therefore, we can indirectly relate the CPD runtimes in Figure 15 with the memory usage results in Figure 1.

We observe that SPLATT and DMS consistently have the fastest runtimes, with DMS performing slightly better on average. ENSIGN runs an average of 1.55 times slower than DMS. This is reflected in the fact that ENSIGN exhibits an average of 2.04 times more stalls than DMS and uses an average of 2.14 times more memory. DFacTo has the slowest runtime by far, performing 4.74 – 7.5 times slower than DMS. While DFacTo does consume an average of 1.34 times more memory than DMS, the large difference in runtime is due to DFacTo’s column-oriented approach to the MTTKRP, which leads to substantially more stall cycles per non-zero, as shown in Figure 9.

We then evaluated how each tool’s CPD runtime is

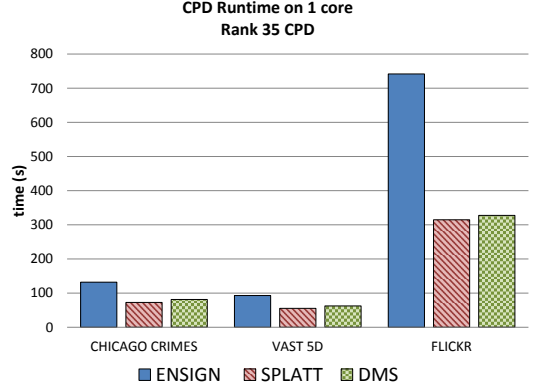


Figure 17: CPD runtime in seconds on the higher-dimensional data sets when performing a rank-35 CPD on 1 core.

affected by the rank of the CPD. Figure 16 shows the results for the VAST 3D data set as the CPD rank is increased from 35 to 100. Each tool exhibits similar relative scaling as the CPD rank grows larger. However, DFacTo requires much more additional runtime than the other tools as the CPD rank increases. This is due to DFacTo’s approach to computing the MTTKRP which requires an extra traversal of the tensor for each additional rank of the CPD. This is discussed further in Section 5.1.

We also investigated the tools’ runtime performance on the NELL-1 data set when increasing the CPD rank. We notice a similar trend as in the VAST 3D results, with DFacTo requiring significantly more runtime than the other tools. To compute a rank-100 CPD, DFacTo requires 16.6 hours of runtime while SPLATT and DMS only require 3 hours of runtime and ENSIGN 8.5 hours.

We also measured the CPD runtime for the higher dimensional data sets, as shown in Figure 17, and observed that they follow the same trend as the 3-dimensional data set results in Figure 15.

4.4.2. Parallel Runtime

In order to evaluate the effectiveness of each tool’s approach to parallelizing CP-ALS, we ran the tools across the data sets and increased the number of cores used on a single machine from 1 to 20. Figure 18 shows the CPD runtime strong scaling achieved by each tool on the data sets when using 20 threads/processes. We also present the numeric results in Table 2, where the bolded values in each row represent the best performance. We observe that EN-

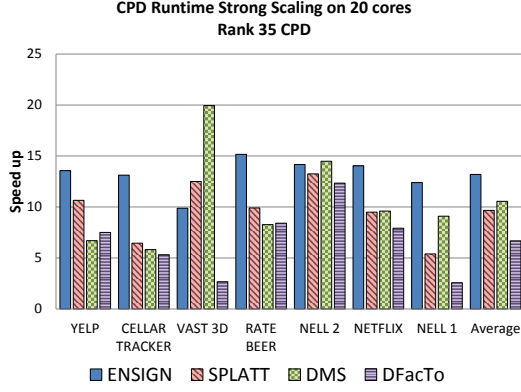


Figure 18: CPD runtime strong scaling achieved by each tool when using 20 threads (SPLATT and ENSIGN) or 20 processes on a single machine (DMS and DFacTo). The average speed-up of each tool is shown on the far right. Larger bars represent better performance.

SIGN and DMS exhibit the largest average speed-up, with ENSIGN out-performing DMS overall. DFacTo shows the worst scaling on average, which is a result of a combination of the characteristics that we have outlined in previous sections regarding DFacTo, namely high memory usage and a significant amount of stall cycles per non-zero.

However, scalability alone is not enough to effectively gauge the performance of these tools. Table 3 shows the CPD runtime for the tools when using 20 threads/processes and computing a rank-35 CPD. Bolded values in each row represent the best performance. While DFacTo does exhibit better scalability than the other tools on some data sets, its runtime is consistently an order of magnitude or more slower than the other tools. For example, DFacTo exhibits a 7.5x speed up on the YELP data set while DMS only achieves a speed up of 6.69x. However, DFacTo runs 6.7 times slower than DMS. Furthermore, DFacTo is 39.17 times slower than DMS on the VAST 3D data set. Even ENSIGN, which shows the best average strong scaling, is noticeably slower than DMS and SPLATT on several data sets. ENSIGN achieves a speed-up of 12.4x on NELL-1 and requires 234.06 seconds to run while DMS only achieves a speed-up of 9.1x but runs in 168.14 seconds, 1.4 times faster than ENSIGN. We attribute this difference to the choice of data structures employed by the tools and discuss it further in Section 5.4. It is also interesting to note that DMS runs 114.04 seconds faster than

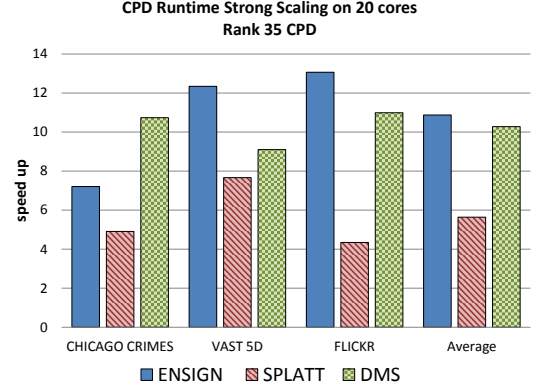


Figure 19: CPD strong scaling on the higher dimensional data sets when using 20 cores to compute a rank-35 CPD. The average speed up achieved by each tool is shown on the far right.

SPLATT on the NELL-1 data set, which is a substantial difference when compared to the other data sets. This can be explained by noting the difference in stall cycles per non-zero between SPLATT and DMS on NELL-1, as shown in Figure 12. Because of NUMA effects and memory coherency snooping requests, SPLATT incurs a significant cost with respect to stall cycles when compared to DMS and thus requires more overall runtime. This is made worse on NELL-1 due to its significant size and the amount of data accesses required.

When increasing the rank of the CPD and using 20 threads/processes, we observe the same trend on the VAST 3D data set as presented in Figure 16. ENSIGN, SPLATT and DMS all require a comparable amount of additional runtime as the CPD rank grows. However, DFacTo requires significantly more additional time to compute a rank-100 CPD when compared to a rank-35 CPD.

In regard to the higher dimensional data sets, we observe a similar trend for CPD runtime and scalability as the 3-dimensional data sets. These results are presented in Figures 20 and 19.

4.5. Multi-Machine Results

We limited our performance study in Sections 4.2 – 4.4 to single machine runs for DMS and DFacTo. In this section, we present the memory usage, stall cycles and CPD runtime results when running DMS and DFacTo across 20 machines, with 1 MPI rank per machine and 1 core per rank. We ran these experiments on the same 24 node cluster mentioned in

Table 2: Rank-35 CPD speed-ups on 20 threads/processes

	ENSIGN	SPLATT	DMS	DFacTo
YELP	13.56	10.66	6.69	7.5
CELLAR TRACKER	13.13	6.46	5.82	5.32
VAST 3D	9.88	12.5	19.95	2.67
RATE BEER	15.16	9.9	8.27	8.41
NELL-2	14.17	13.24	14.49	12.34
NETFLIX	14.04	9.49	9.59	7.92
NELL-1	12.4	5.4	9.1	2.57
Average	13.19	9.66	10.56	6.68

Table 3: Rank-35 CPD runtime in seconds on 20 threads/processes

	ENSIGN	SPLATT	DMS	DFacTo
YELP	3.52	3.08	4.94	33.09
CELLAR TRACKER	7	8.06	8.95	64.31
VAST 3D	7.45	4.6	2.84	111.23
RATE BEER	24.24	25.05	30.06	185.22
NELL-2	20.88	17.65	16.1	128.26
NETFLIX	19.78	16.22	15.9	149.35
NELL-1	234.06	282.18	168.14	2816.16
Geometric Mean	17.11	15.66	14.89	157.13

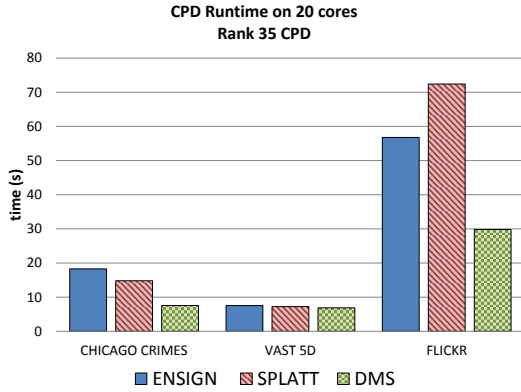


Figure 20: CPD runtime in seconds on the higher dimensional data sets when using 20 cores to compute a rank-35 CPD.

Section 4. All nodes were connected by a 56 Gbit/s FDR Infiniband network.

Figure 21 shows the maximum RSS for DMS and DFacTo when computing a rank-35 CPD. These results are from one of the 20 machines/ranks used. Note that the difference between DMS and DFacTo on NELL-1 is so great that the vertical axis needs to be logarithmic to effectively show the other results. The trend we observe on multiple machines is the same as when using a single machine, namely that DMS consistently uses less memory than DFacTo. There are two main reasons why this is the case. First, DMS represents the tensor with a single data structure, CSF, while DFacTo uses a CSR structure for each dimension of the tensor. Second, DMS' data distribution scheme is more effective at minimizing memory usage per MPI rank than DFacTo's approach, which is discussed more in Section 5.2.

In regards to the stall cycles per non-zero, DMS exhibits significantly fewer stalls than DFacTo when using multiple machines. This can be seen in Figure 22, which shows the stall cycles per non-zero for

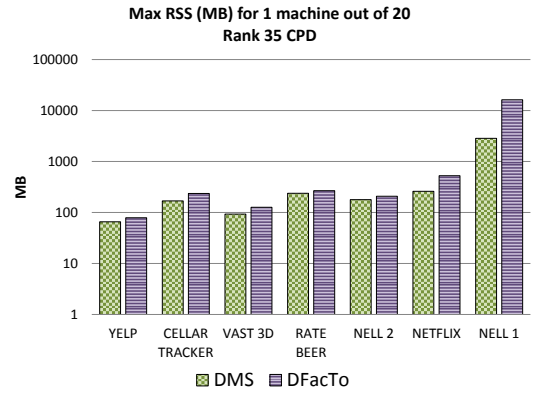


Figure 21: Maximum RSS from 1 machine out of 20 (1 MPI rank per machine) when computing a rank-35 CPD. The vertical axis is shown in MB and is logarithmic.

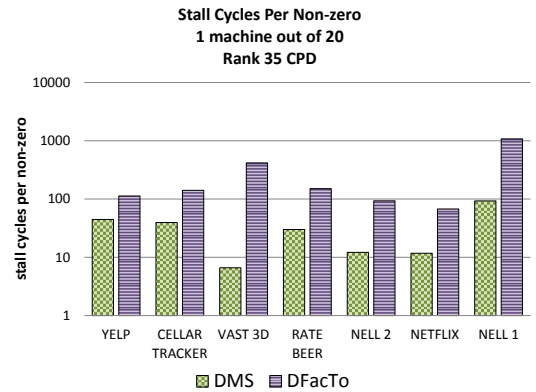


Figure 22: Stall cycles per non-zero from 1 machine out of 20 (1 MPI rank per machine) when computing a rank-35 CPD. The vertical axis is logarithmic.

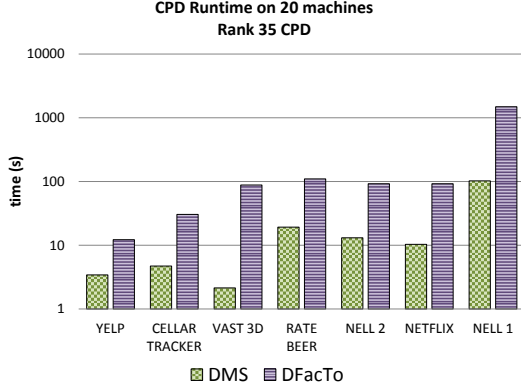


Figure 23: CPD runtime in seconds on 20 machines (1 MPI rank per machine) when computing a rank-35 CPD. The vertical axis is logarithmic.

one of the ranks/machines out of the 20 that were utilized. We can attribute DFacTo’s poor performance to its higher memory usage and column-oriented approach to the MTTKRP, with the latter contributing the most.

Figure 23 shows the CPD runtime for DMS and DFacTo across 20 machines. It is evident that DMS is orders of magnitude faster than DFacTo. DFacTo requires significantly more data communication than DMS, especially for large tensors such as NELL-1, negatively impacting its runtime performance and scalability. DMS and DFacTo’s communication costs are discussed more thoroughly in Section 5.3.

While DFacTo does achieve significant improvements when using multiple machines versus a single machine with multiple MPI ranks, its data distribution and communication scheme and column-oriented MTTKRP computation hinder its scalability with respect to memory usage and execution time, especially for large tensors. On the other hand, DMS’ approach to data distribution and communication allow it to consume less memory per machine/rank and communicate less data. Furthermore, its row-oriented approach to the MTTKRP greatly reduces the stall cycles per non-zero when compared to DFacTo.

5. Analysis of Observed Performance

The results from Section 4 reveal behavior that is a consequence of key implementation choices made

by each tool. In this section, we discuss the decisions that lead to the largest impacts on performance that we observed across the metrics.

5.1. Column Vs Row Approach to MTTKRP

As noted in Section 3.1, DFacTo computes the MTTKRP one column at a time. This is in contrast to the other tools that compute entire rows at a time. DFacTo’s column-oriented approach leads to inefficient cache utilization, resulting in a significant amount of off-chip memory accesses. Because of this, DFacTo exhibits orders of magnitude more stall cycles per non-zero than the other tools when using a single core, as shown in Figure 9. Furthermore, because of DFacTo’s approach to the MTTKRP, it requires an extra traversal of the tensor for each additional CPD rank computed. These extra traversals lead to a sharp increase in the number of memory accesses required by DFacTo, which results in a high number of additional stall cycles per non-zero that occur as the CPD rank increases.

In the multiple processes case, DFacTo does achieve a significant reduction in the number of stall cycles per non-zero when compared to a single process. However, even with these reductions, DFacTo still exhibits an average of 20.75 times more stall cycles per non-zero than DMS, as seen in Figure 12. Due to DFacTo’s column-oriented computation of the MTTKRP, the benefit of each MPI rank running in its own memory space is largely lost with respect to stall cycles.

Because of the correlation between the stall cycles per non-zero and the CPD runtime, as discussed in Section 4.4.1, we can attribute DFacTo’s slow runtime to its large amount of stall cycles per non-zero introduced by its column-oriented approach to the MTTKRP.

With respect to memory, DFacTo’s column-oriented approach leads to extra data structures being allocated that can significantly increase its maximum memory usage. When measuring the memory usage over the execution of the CPD for each tool, all of the tools increased memory usage during the initial stages of the CPD until reaching a steady-state plateau that was held throughout execution. However, DFacTo contained periodic transient spikes in memory usage during the execution. The memory

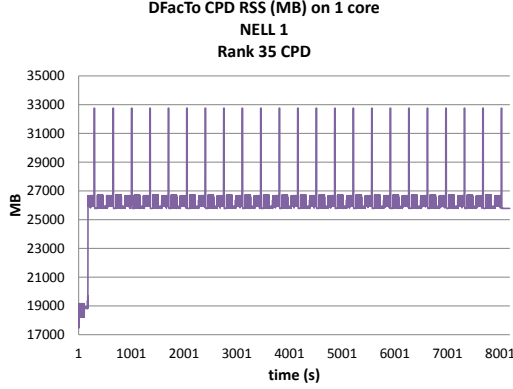


Figure 24: RSS over time for DFacTo when running on NELL-1 with 1 process and computing a rank-35 CPD. The periodic spikes in memory usage are due to the allocation of an auxiliary sparse matrix for the MTTKRP.

spikes are due to DFacTo’s implementation of the MTTKRP, which requires an auxiliary sparse matrix (denoted as \mathbf{M}' in [6]) that contains at most as many non-zeros as there are non-empty fibers in the original tensor. This auxiliary matrix is reallocated each time the MTTKRP is performed. Depending on the original tensor, the reallocation of the auxiliary matrix can result in noticeably higher memory usage for a brief time. Figure 24 shows an example of this memory usage pattern on the NELL-1 data set when computing a rank-35 CPD. The larger spikes correspond to when DFacTo is executing the MTTKRP for the largest dimension of the tensor. Furthermore, because of DFacTo’s implementation of the MTTKRP, it allocates a dense matrix for each dimension of the tensor to store the output of the MTTKRP, as described in Section 3.1. The sizes of these matrices are identical to those of the factor matrices and are stored in full on each MPI rank. This leads to very poor memory scalability as the number of MPI ranks increases, as well as when the size of the tensor and the CPD rank grows.

5.2. MPI Data Distribution Schemes

While DFacTo and DMS are both MPI-based codes, their methods for distributing the original tensor and factor matrices among multiple processes are vastly different. These different approaches lead to significant differences in memory usage as the size of the tensor increases and the number of MPI ranks used increases.

As described in Section 3.1, DFacTo uses a coarse-grained approach to data distribution, partitioning the original tensor among the different MPI ranks and storing full copies of the factor matrices on each process. While this simplifies the MTTKRP and minimizes the amount of communication, it leads to memory scalability issues with respect to the size of the tensor, the number of MPI ranks used and the rank of the CPD performed. Figure 4 shows that for particularly large tensors, such as NELL-1, DFacTo’s maximum memory usage is significantly higher than the other tools. In Figure 5, we can see that DFacTo exhibits a drastic increase in total memory usage on NELL-1 as the number of MPI ranks is increased from 1 to 20. For NELL-1, the factor matrices are close in size to the original tensor, so storing full copies on each MPI rank consumes a substantial amount of memory. Furthermore, for a data set like NELL-1, the full copy of the factor matrices on each MPI rank is no longer a benefit for communication costs due to the significant size of the matrices that have to be communicated. When the rank of the decomposition is increased, DFacTo achieves very poor memory scalability since the factor matrices increase in size and are then replicated on each MPI rank. The increase in CPD rank also increases the size of the dense MTTKRP output matrices allocated by DFacTo on each MPI rank. An example of this poor scalability can be seen in Figure 6

Instead of storing full copies of the factor matrices on each MPI rank, DMS only stores partial copies, as described in Section 3.2. Also, DMS only requires a single dense matrix for the output of the MTTKRP. These characteristics allow DMS to achieve good memory scalability as the size of the tensor increases and the number of MPI ranks used increases, as well as when the rank of the CPD grows. As an example, DFacTo uses 324 GB of memory on the NELL-1 data set when running on 20 MPI ranks, 9.9 times more than it did on a single MPI rank. On the other hand, DMS only uses 1.85 times more memory on NELL-1 when using 20 MPI ranks when compare to a single MPI rank. Furthermore, DMS consumes 109 GB of memory when using 20 MPI ranks on the NELL-1 data set for a rank-75 CPD, a factor of 1.9 increase from the single MPI rank result. In comparison, DFacTo was unable to perform the rank-75 CPD

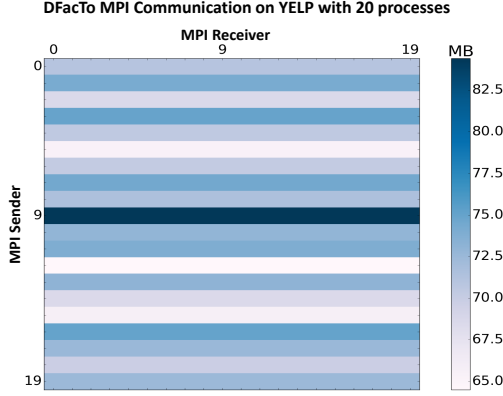


Figure 25: Communication pattern for DFacTo on the YELP data set when computing a rank-35 CPD using 20 processes on a single node. Lighter colors represent less MB sent while darker colors represent more MB sent. The total amount of data sent was 28.67 GB.

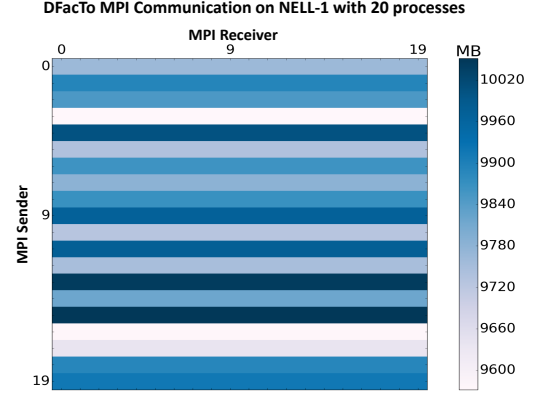


Figure 26: Communication pattern for DFacTo on the NELL-1 data set when computing a rank-35 CPD using 20 processes on a single node. Lighter colors represent less MB sent while darker colors represent more MB sent. The total amount of data sent was 3933 GB.

on NELL-1 with 20 MPI ranks because it used more memory than our system had available, as described in Section 4.2.2.

5.3. MPI Communication Patterns

Due to the differing data distribution schemes used by DMS and DFacTo, they both have different approaches for communicating data among the MPI ranks. These differences can lead to performance results that are orders of magnitude apart. A thorough evaluation of DMS and DFacTo’s communication topology is beyond the scope of this paper. However, we did gather results regarding the communication patterns between MPI ranks during the CP-ALS execution, as well as the amount of data being sent among the ranks. This information provides further insight into the performance of the tools.

DFacTo uses the `MPI_Allgatherv` collective to exchange updated rows of the factor matrices between all of the MPI ranks. At the end of each MTTKRP computation, every process will send every other process its portion of the result via `MPI_Allgatherv`. Examples of DFacTo’s communication pattern are presented in Figures 25 and 26 for the YELP and NELL-1 data sets, respectively. These figures show the amount of data sent between the processes during the execution of CP-ALS when using 20 MPI ranks and computing a rank-35 CPD. From these figures, it is clear that DFacTo is performing a coarse-grained approach to data distribu-

tion, where an MPI rank sends the same amount of data to every other rank. The variation in the amount of data sent between ranks is fairly small, which indicates that the original tensor is partitioned evenly among the ranks.

DMS’ approach to communication is vastly different from DFacTo. This is clear when observing its communication pattern, as shown in Figures 27 and 28 for the YELP and NELL-1 data sets, respectively. DMS decomposes each mode of the tensor into layers, where each layer is assigned a group of MPI ranks. In the case of the YELP and NELL-1 data sets, the size of each group is 5 ranks and 10 ranks, respectively. DMS uses the `MPI_Alltoallv` collective to communicate updated rows of the factor matrices between all of the MPI ranks within a given layer. The main diagonal in both figures shows that there is no communication between a MPI rank and itself. This is not the case with DFacTo, where each rank will send itself updated rows of the factor matrices that it already has. DMS also exhibits heavy communication outside of each layer, as shown by the other diagonals above and below the main diagonal. We believe these to be the result of some dimensions of the tensor requiring more communication than others. In such cases, DMS’ medium-grained decomposition will attempt to localize the communication.

The total amount of data communicated and the time required for communication are also vastly dif-

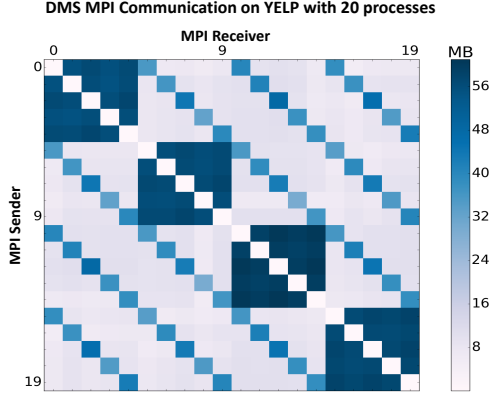


Figure 27: Communication pattern for DMS on the YELP data set when computing a rank-35 CPD using 20 processes on a single node. Lighter colors represent less MB sent while darker colors represent more MB sent. The total amount of data sent was 8.99 GB.

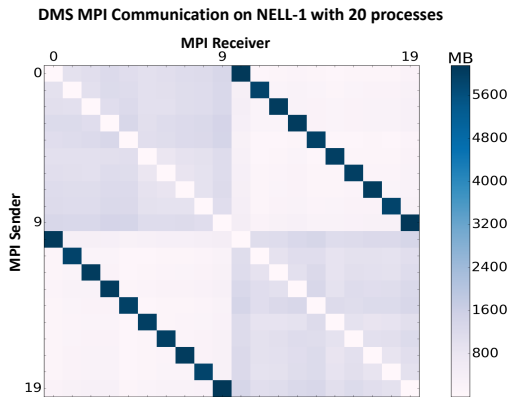


Figure 28: Communication pattern for DMS on the NELL-1 data set when computing a rank-35 CPD using 20 processes on a single node. Lighter colors represent less MB sent while darker colors represent more MB sent. The total amount of data sent was 353 GB.

ferent for DMS and DFacTo. For the YELP data set, DFacTo sends a total of 28.67 GB of data while DMS only sends 8.99 GB. Due to DFacTo sending substantially more data than DMS, it requires 1.74 times more communication time than DMS. For NELL-1, DFacTo sends a total of 3933 GB of data and DMS sends 353 GB, a difference of 11.14x. As far as communication time, DFacTo requires 15.14x more time than DMS. Since DMS does not need to update entire factor matrices, it scales much better than DFacTo as the size of the tensor increases with respect to how much data is communicated and the time required for communication. When increasing the CPD rank from 35 to 100, DFacTo sends an additional 79.9 GB of data on the YELP data set and requires 4.49x more time for communication. DMS requires an additional 25 GB of data communication for a rank-100 CPD and 4.48x more time for communication. While DMS and DFacTo exhibit nearly identical relative scaling in the amount of data sent and the amount of time required for communication as the CPD rank increases, DFacTo still sends 3.19 times more data than DMS and requires 1.74x more time for communication.

5.4. Data Structures for Tensor Storage

The choice of data structure used to represent and store tensors has a large impact on memory consumption and overall code performance. While this is true for both the shared and distributed memory based tools, we can attribute the vast performance difference between DMS and DFacTo to the implementation choices discussed in Sections 5.2 and 5.3. In regards to the shared memory tools, SPLATT and ENSIGN, our results show that ENSIGN uses noticeably more memory than SPLATT and exhibits CPD execution times that are typically slower. We attribute these differences specifically to the choice of data structures employed by the two tools.

As discussed in Section 3.2, SPLATT uses the CSF data structure for the storage of tensors. The benefit of CSF is that only one structure is needed to represent all of the dimensions of a tensor. This results in a small memory footprint with respect to the size of the tensor and the number of dimensions. On the other hand, ENSIGN uses the MSS tensor data structure, as mentioned in Section 3.3, and requires

separate copies to be allocated for each dimension in the tensor. While this replication can be used to exploit parallelism, it results in wasted memory in the serial case. For this reason, ENSIGN can be seen using much more memory than SPLATT in Figure 1. As we observed throughout Section 4, lower memory consumption leads to fewer stall cycles, which results in overall lower execution time. Indeed, Figure 9 shows that ENSIGN stalls noticeably more than SPLATT and Figure 15 shows that ENSIGN’s CPD runtime is consistently slower.

ENSIGN’s MSS tensor data structure and replication technique do allow it to perform well with respect to CPD runtime scalability, as presented in Figure 18 and Table 2. However, the extra memory required to store these replicated data structures generally results in a slower CPD runtime on 20 cores when compared to SPLATT. Furthermore, this replication technique is not suitable for tensors with small dimensions, since little to no parallelism can be exploited for those dimensions. This can be seen by noting the difference in strong scaling and CPD runtime between ENSIGN and SPLATT on the VAST 3D data set as shown in Tables 2 and 3.

We can compare the strong scaling results in Table 2 with those in Table III from our previous work [14] to observe the recent improvements made to SPLATT. It should be noted that the ENSIGN results presented in our previous evaluation were from an entirely different implementation ENSIGN than the version that we evaluate in this paper. Also, the DMS and DFacTo results in Table III from our previous work are from running 20 MPI ranks across 20 machines, with 1 MPI rank per machine. Therefore, we can only directly compare the SPLATT results. SPLATT’s average speed-up has increased by a factor of 1.14 over the previous results, largely because of the improvement on the VAST 3D data set (denoted as VAST 2015 MC 1 in Table III from [14]). The poor performance of SPLATT was due to the short third dimension of the tensor, being only two in size, which negatively affected CSF-based computations [12]. In the older version of SPLATT, coarse-grained parallelism was used when decomposing the computation and the tensor’s dimensions were sorted in ascending order and then decomposed, resulting in the short third dimension of the VAST 3D tensor be-

ing decomposed first. This approach only performed well with tensors that have large dimensions since very little parallelism can be extracted from short dimensions. In response to our results, SPLATT was modified to employ tiling on a subset of the dimensions, placing priority on the largest dimensions, and using atomics for the remaining dimensions [12]. Due to these improvements, SPLATT is able to achieve CPD runtime scalability on VAST 3D that is a factor of 11.45 better than the results in our previous evaluation [14].

ENSIGN could be modified to not employ its replication technique in the serial case, which would drastically reduce its memory usage and perhaps improve its overall code performance. In the parallel case, a heuristic could be developed to determine when to use the replication technique based off of the structure of the tensor.

5.5. Performance on Higher Dimensional Tensors

When executing CP-ALS on the data sets with greater than three dimensions, we observed performance results that followed similar trends as the 3-dimensional tensors. These results are important because they show that the data structures and implementation choices employed by ENSIGN, SPLATT and DMS are robust enough to handle higher dimensional tensors. Prior to this work, very little research has been done in regards to the scalability of these tools on tensors with more than three dimensions.

With regard to memory usage when using a single core, SPLATT uses an average of 2.3 times less memory than ENSIGN on the higher dimensional data sets (see Figure 3). This is in line with what we observed for the 3-dimensional data sets shown in Figure 1. In both cases, ENSIGN’s high memory consumption can be attributed to its allocation of copies of its MSS tensor data structure, as discussed in Section 5.4. As in the 3-dimensional results on a single core, DMS uses slightly more memory than SPLATT on the higher dimensional data sets. This slight increase in memory usage is from the MPI communication structures allocated by DMS, as well as the copy of the factor matrices, which is discussed in Section 4.2.1. This difference is magnified for large tensors, such as NELL-1 and FLICKR, which have very large factor matrices.

When using up to 20 cores on the higher dimensional data sets, we observe nearly identical trends with respect to memory usage, stall cycles and CPD runtime when compared to the single core results. This shows the robustness of each tool’s approach to tensor storage and parallelization for higher dimensional tensors.

It is worth noting that while DFacTo is currently not implemented to support tensors with more than three dimensions, it should be clear from the results in Section 4 and our discussions in Sections 5.1 – 5.3 that DFacTo’s current approach to CP-ALS would not scale for tensors with many dimensions. Each dimension in the tensor requires a factor matrix and a MTTKRP output matrix, which are stored on each MPI rank for DFacTo. This would lead to significant memory consumption. Furthermore, the MTTKRP is computed for each dimension of the tensor per iteration of CP-ALS. Since DFacTo’s approach to the MTTKRP is very inefficient, the CPD runtime would be drastically increased.

In future work, it would be worth evaluating tensors with even more dimensions, as well as those with very large dimensions. Such tensors will provide a more thorough evaluation of each tool’s scalability. We expect that given enough large dimensions, ENSIGN’s memory usage would become significantly higher than SPLATT and DMS. This is because SPLATT and DMS use the CSF data structure, which can represent all of the dimensions of the tensor in a single structure. On the other hand, ENSIGN creates a copy of its MSS tensor data structure for each dimension of the tensor.

6. Future CP-ALS Implementations

In this section, we provide an outline for a CP-ALS implementation that is both distributed and heterogeneous, in that it could execute across multiple nodes/processes and leverage graphics processing units (GPUs) via Compute Unified Device Architecture (CUDA). This approach is designed to address some of the performance considerations presented in Section 5, specifically those related to DFacTo. There are efforts to perform tensor decomposition on GPUs [23, 24], but to the best of our knowledge, such efforts are not distributed or leverage multiple GPUs.

6.1. Distributed and Heterogeneous CP-ALS

DFacTo’s MTTKRP is unique in that it consists mostly of SpMVs. The current implementation of DFacTo strictly uses Eigen for its data structures and BLAS, including SpMV. Eigen can use Intel’s vectorized Math Kernel Library if it is provided during the build process, allowing DFacTo to take advantage of a highly efficient CPU-based SpMV. However, there is a significant body of work focused on efficient implementations of GPU-based SpMV [25, 26, 27]. Because DFacTo relies heavily on Eigen’s classes and functions for both the storage of its data structures and the operations on such structures, significant modifications would need to be made to DFacTo’s implementation to leverage GPUs for SpMV.

Once an implementation has been developed to perform SpMV on the GPU, it becomes possible to perform the MTTKRP entirely on the GPU using dynamic parallelism, a feature of CUDA that allows for GPU kernels to launch and synchronize new kernels. This can reduce the overhead of launching new kernels from the host for each SpMV performed. Furthermore, other routines in CP-ALS could be ported onto the GPU without significant overhead. For example, part of computing line 6 of Algorithm 1 involves a matrix multiplication on the result of the MTTKRP and \mathbf{V}^\dagger , both dense matrices. If the MTTKRP is formed by a series of SpMVs that are executed on a GPU, the following dense matrix multiplication can also be performed on the GPU without transferring control back to the CPU by using dynamic parallelism. As fewer operations are performed on the CPU, the amount of host-to-device (HtoD) and device-to-host (DtoH) copying is reduced since very little of the data is required on the CPU.

Many of today’s clusters have a GPU attached to each node. Using DFacTo’s existing data distribution scheme and communication pattern, it is possible to assign each MPI rank to a single node and have that rank perform its part of the CP-ALS algorithm on the available GPU. If all of the routines in CP-ALS are performed on the GPU, then each MPI rank only needs to transfer control back to the CPU to communicate the updated rows of the factor matrices. This would provide a distributed heterogeneous approach to tensor decomposition. However, the communica-

tion costs for this multi-node GPU-based approach are further increased when considering the time it takes for DtoH copying prior to sending the data over Infiniband to the other ranks and then performing HtoD copying to put the updated data back onto the GPU for the next iteration.

As well as clusters with a GPU attached to each node, it is also common for systems to have two or more GPUs connected together within a single node, such as NVIDIA’s DGX-1. On a system such as the DGX-1, the distributed heterogeneous approach described above could assign each MPI rank to one of the GPUs. Furthermore, communication costs could be improved by leveraging NCCL [28], NVIDIA’s library for multi-GPU collective communication. The advantage of using NCCL over MPI in this case is that NCCL is optimized for communication over NVLink, NVIDIA’s high-speed interconnect for GPU-to-GPU communication that is present in the DGX-1. Using NCCL and NVLink would eliminate the need to perform the HtoD/DtoH operations prior sending data since the actual GPU-memory can be sent amongst the GPUs/ranks. As we have observed in Section 5.3, the amount of communication and its associated cost can become a performance issue for distributed CP-ALS. Reducing this cost can significantly improve the overall performance of the implementation.

This distributed heterogeneous design would still inherit DFacTo’s column-oriented approach to the MTTKRP. For example, DFacTo forms each column of the MTTKRP with two SpMV’s, each of which need to extract a column from a factor matrix to use as input to the SpMV. The result of the second SpMV is stored as a column in the MTTKRP output matrix. In DFacTo’s implementation, these dense matrices are stored in row-major order, making the column reading and writing very inefficient, especially for tensors with large factor matrices. One possible solution would be to use column-major matrices. While this would allow for more efficient memory accesses during the MTTKRP, we predict that the overall performance would degrade because all other routines within CP-ALS operate row-by-row. Instead, we suggest exploiting the fine-grained parallelism of GPUs to perform the column extraction and inserting operations. This would significantly speed up the

routines while still preserving the row-major structure of the matrices.

An issue that arises when relying on GPUs for computation is the significant reduction in the amount of memory available on the GPU when compared to the CPU. For example, a single node in our cluster has 512GB of available memory while the NVIDIA Tesla P100 GPUs within the DGX-1 system each only have 16GB of global memory. Because of this constraint, careful consideration needs to be given with respect to the memory usage of a CP-ALS implementation for GPUs. A naive port of DFacTo’s implementation onto a GPU would only be successful for relatively small tensors due to its significant memory overhead, as presented in Section 4.2. Upon closer inspection of DFacTo’s implementation, we determined that the constant re-allocation of the \mathbf{M}^r auxiliary matrix, as described in Section 5.1, is unnecessary. Furthermore, storing separate MTTKRP output matrices for each dimension of the tensor is also unnecessary, as only the output from the last dimension is ever used in future computations. By eliminating these unnecessary memory overheads, DFacTo’s implementation can be made much more susceptible to porting onto a GPU, especially in a distributed manner where the size of the sub-tensors held by each GPU is decreased as the number of MPI ranks is increased.

7. Conclusions

We conducted an extensive performance analysis of parallel CP-ALS implementations for sparse tensors. The analysis included memory usage, processor stall cycles, execution time and scalability with respect to those metrics. We also explored the data distribution schemes and communication patterns used by the distributed memory tools. Furthermore, we evaluated the performance of the tools when performing high rank decompositions as well as when performing CP-ALS on tensors with more than three dimensions.

The key points highlighted in the analysis are:

1. Column-oriented computation of the MTTKRP does not scale with respect to processor stall cycles and execution time as the rank of the de-

composition increases and/or the size of the tensor increases.

2. Coarse-grained decomposition of a tensor for distributed memory implementations does not scale with respect to memory usage as the size of the tensor grows, the number of processes increases and the rank of the decomposition increases.
3. CSF offers a small memory footprint by representing the tensor with a single structure. This benefits overall code performance and scalability with respect to the size of the tensor and the number of dimensions in the tensor.

From the experiments and performance evaluations, we have highlighted key characteristics of these tools, offering insight for future high performance implementations for CP-ALS. We observed that computing the MTTKRP in a column-oriented manner results in severely inefficient cache utilization, leading to slow execution time. The way in which the data is distributed and communicated within a MPI-based implementation has far reaching consequences with respect to performance. We observed that the medium-grained decomposition used by DMS offers the best performance, minimizing the amount of total memory required since only partial copies of the factor matrices are stored on each MPI rank. This also minimizes the amount of data that needs to be communicated between MPI ranks, and thus the amount of time required for such communication. The choice of data structures used to store and represent sparse tensors also has a large impact on performance. We have found that the approach taken by SPLATT's CSF data structure offers the smallest memory footprint, which results in better cache utilization and in turn, better execution time. We also observed that the tools that could perform CP-ALS on tensors with more than three dimensions were able to do so without serious performance scalability issues. However, further analysis on tensors with many more dimensions should be conducted to thoroughly evaluate the scalability of the tools.

To address some of these performance considerations, we provided an outline for a distributed heterogeneous CP-ALS implementation. This implementation could run across multiple machines, leveraging

an available GPU on each node, or within a GPU cluster, distributing work to each GPU. Such an approach would be able to utilize the fine grained parallelism of GPUs to speed up computation as well as use high-speed interconnects such as NVLink to improve communication costs.

Acknowledgments

The authors appreciate the help of Albert Scalo of University Technical Services in installing tools and running several of the early experiments that led to this work. We also thank TC Tuan at the Laboratory for Physical Sciences for his guidance and support.

References

- [1] T. G. Kolda, B. W. Bader, Tensor decompositions and applications, *SIAM Review* 51 (2009) 455–500. doi:10.1137/07070111X.
- [2] U. Kang, E. Papalexakis, A. Harpale, C. Faloutsos, Gigatensor: Scaling tensor analysis up by 100 times - algorithms and discoveries, in: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12*, ACM, New York, NY, USA, 2012, pp. 316–324. URL: <http://doi.acm.org/10.1145/2339530.2339583>. doi:10.1145/2339530.2339583.
- [3] T. G. Kolda, J. Sun, Scalable tensor decompositions for multi-aspect data mining, in: *IEEE International Conference on Data Mining*, 2008, pp. 363–372. doi:10.1109/ICDM.2008.89.
- [4] M. Baskaran, B. Meister, N. Vasilache, R. Lethin, Efficient and scalable computations with sparse tensors, in: *High Performance Extreme Computing (HPEC)*, 2012 IEEE Conference on, 2012, pp. 1–6. doi:10.1109/HPEC.2012.6408676.
- [5] S. Smith, N. Ravindran, N. D. Sidiropoulos, G. Karypis, Splatt: Efficient and parallel sparse tensor-matrix multiplication, *29th IEEE International Parallel & Distributed Processing Symposium* (2015).
- [6] J. H. Choi, S. Vishwanathan, Dfacto: Distributed factorization of tensors, in: Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 27*, Curran Associates, Inc., 2014, pp. 1296–1304. URL: <http://papers.nips.cc/paper/5395-dfacto-distributed-factorization-of-tensors.pdf>.
- [7] I. Jeon, E. E. Papalexakis, U. Kang, C. Faloutsos, Haten2: Billion-scale tensor decompositions, in: *Proceedings of the International Conference on Data Engineering*, 2015, pp. 1047–1058. doi:10.1109/ICDE.2015.7113355.
- [8] G. Guennebaud, B. Jacob, et al., *Eigen v3*, <http://eigen.tuxfamily.org>, 2010.

- [9] S. Smith, G. Karypis, Tensor-matrix products with a compressed sparse tensor, *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (2015).
- [10] S. Smith, G. Karypis, A medium-grained algorithm for distributed sparse tensor factorization, *International Parallel & Distributed Processing Symposium (IPDPS)* (2016). doi:10.1109/IPDPS.2016.113.
- [11] S. Smith, J. Park, G. Karypis, An exploration of optimization algorithms for high performance tensor completion, *Proceedings of the 2016 ACM/IEEE conference on Supercomputing* (2016).
- [12] S. Smith, J. Park, G. Karypis, Sparse tensor factorization on many-core processors with high-bandwidth memory, *IEEE International Parallel & Distributed Processing Symposium (IPDPS)* (2017).
- [13] O. Kaya, B. Uçar, Scalable sparse tensor decompositions in distributed memory systems, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15*, ACM, New York, NY, USA, 2015, pp. 77:1–77:11. URL: <http://doi.acm.org/10.1145/2807591.2807624>. doi:10.1145/2807591.2807624.
- [14] T. Rolinger, T. Simon, C. Krieger, Performance evaluation of parallel sparse tensor decomposition implementations, *Proceedings of the 6th Workshop on Irregular Applications: Architectures and Algorithms* (2016).
- [15] M. M. Baskaran, B. Meister, R. Lethin, Parallelizing and optimizing sparse tensor computations, in: *Proceedings of the 28th ACM International Conference on Supercomputing, ICS '14*, ACM, New York, NY, USA, 2014, pp. 179–179. URL: <http://doi.acm.org/10.1145/2597652.2600115>. doi:10.1145/2597652.2600115.
- [16] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. H. Jr., T. M. Mitchell, Toward an architecture for never-ending language learning, in: *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence (AAAI 2010)*, 2010.
- [17] K. A. Cook, G. Grinstein, M. A. Whiting, The vast challenge: History, scope, and outcomes: An introduction to the special issue, in: *Information Visualization*, volume 13, 2014, pp. 301–312. doi:10.1177/1473871613490678.
- [18] J. Bennett, S. Lanning, N. Netflix, The netflix prize, in: *KDD Cup and Workshop in conjunction with KDD*, 2007.
- [19] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, <http://snap.stanford.edu/data>, 2014.
- [20] O. Görlitz, S. Sizov, S. Staab, Pints: peer-to-peer infrastructure for tagging systems., in: *IPTPS*, 2008, p. 19.
- [21] City of Chicago: Crimes - 2001 to present, <https://catalog.data.gov/dataset/crimes-2001-to-present-398a4>, 2015.
- [22] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, G. Karypis, FROSTT: The formidable repository of open sparse tensors and tools, 2017. URL: <http://frostdt.io/>.
- [23] B. Zou, C. Li, L. Tan, H. Chen, Gputensor: Efficient tensor factorization for context-aware recommendations, *Information Sciences* 299 (2015) 159 – 177. URL: <http://www.sciencedirect.com/science/article/pii/S0020025514011414>. doi:<https://doi.org/10.1016/j.ins.2014.12.004>.
- [24] B. Zou, M. Lan, C. Li, L. Tan, H. Chen, Context-Aware Recommendation Using GPU Based Parallel Tensor Decomposition, Springer International Publishing, Cham, 2014, pp. 213–226. URL: http://dx.doi.org/10.1007/978-3-319-14717-8_17. doi:10.1007/978-3-319-14717-8_17.
- [25] W. Yang, K. Li, Z. Mo, K. Li, Performance optimization using partitioned spmv on gpus and multicore cpus, *IEEE Trans. Computers* 64 (2015) 2623–2636.
- [26] S. Filippone, V. Cardellini, D. Barbieri, A. Fanfarillo, Sparse matrix-vector multiplication on gpgpus, *ACM Trans. Math. Softw.* 43 (2017) 30:1–30:49. URL: <http://doi.acm.org/10.1145/3017994>. doi:10.1145/3017994.
- [27] Y. Liu, B. Schmidt, Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus, in: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015, pp. 82–89. doi:10.1109/ASAP.2015.7245713.
- [28] S. Jeaugey, Nccl 2.0, <http://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf>, 2017.