Coupling Parallel and Distributed Programs for Sparse Data

S. Isaac Geronimo Anderson
University of Oregon
igeroni3@uoregon.edu

Abstract—**The rise in big data analytics has impelled the popularity of sparse tensors in High-Performance Computing (HPC). Meanwhile, I/O limitations in HPC have motivated efforts towards in situ and in transit software composition, particularly for visualization, but also for sparse tensors. The wide variety of sparse tensor memory formats and their associated high-performance sparse tensor algorithms lend a basis for considering software composition systems. This work surveys the diverse approaches to HPC software composition with respect to a novel set of four categories, or realms: workflow managers, middleware, discrete processing services, and distributed data services. Each realm is motivated and represented by several software composition systems. Each system is compared with its rivals regarding fitness for a particular purpose. Finally, this paper highlights opportunities for future work, based on the unique challenges and the traditional challenges of software composition for sparse tensor decomposition in HPC.**

1. Introduction

Software composition in high-performance computing (HPC) is a crucial component of modern scientific workflows [1], [2]. When two or more interconnected computational tasks are used together for a science campaign, the tasks (as software programs) are effectively composed together for a scientific workflow. Workflows involve coordination and information exchange between tasks, and in the HPC context these tasks can run on a supercomputing system and on geographically distributed clusters. The tasks can share information through files or via system memory or networking technologies, and the latter is referred to as *in situ* operation. Each task generally carries out its operation arbitrarily according to its design and needs, and also records information in an arbitrary format. It can be understood that the wide variety of software programs, each with their own mode of operation and information format, poses a formidable challenge for software composition in HPC.

Meanwhile, sparse tensors and their decompositions have been an active research area over the past five decades [3]. The rise in big data analytics has also impelled recent popularity of sparse tensors in the HPC community [4]. Sparse tensors refer to multi-dimensional arrays of data where most data are zero-valued, and are higher-order generalizations of sparse matrices. Sparse tensor decomposition can reveal latent information within the sparse data, and has wide applicability across many fields of research. There are many sparse tensor storage formats focused on improving the performance of sparse tensor operations and decompositions. Each sparse tensor format implies a corresponding algorithm designed to exploit the performance advantages of the format. The sparse tensor decompositions, formats, and algorithms are designed to run on different kinds of data (e.g. Gaussian- or Poisson-distributed), on shared-memory and distributed-memory systems, both in-core and out-of-core [5], and on multi-core processors and many-core accelerators (e.g. GPUs). It is clear that the various combinations of decompositions, formats, and algorithms seek to reveal higher-quality latent information, on ever-larger sets of data, on a wider variety of high-performance and emerging technologies.

This work surveys the diverse approaches to HPC software composition with respect to sparse tensors and their decomposition. In particular, limitations in HPC have motivated efforts towards in situ and in transit software composition for visualization [6] and for sparse tensors [7]. From high-level scientific workflow creation and management, to frameworks for incorporating program coupling capability, to discrete computational elements and distributed virtual data stores, there exist systems for nearly every software composition scenario. A comparison between the systems and their suitability for various situations is presented. Distinctions are drawn between included systems and other related, but out-of-scope, context-providing systems [8]. Finally, a motivational discussion is provided which summarizes the overall focus of this work, which lays the foundation for considering whether there is a solution in HPC for coupling programs across the most general data, software, and hardware.

2. Tensors and Sparse Tensors

This section introduces tensors (§2.1) and tensor rank decomposition (§2.3) using the notation listed in Table 1.

2.1. Tensors and Sparse Tensors

A one-dimensional array (perhaps representing a vector) or two-dimensional array (representing a matrix) can be generalized as a multi-dimensional array, or tensorial array, commonly referred to as a *tensor* [3]. Note that a tensor refers here to a multi-dimensional (or multi-*way*) array data

TABLE 1. TENSOR NOTATION USED IN THIS PAPER.

Notation	Definition
i, i_k	Scalars (italic letters)
$\mathbf{a}, \mathbf{i}$	Vectors and tuples (bold lower-case letters)
$\mathbf{A}, \mathbf{\Phi}$	Matrices (bold upper-case letters)
$\mathcal{X}, \mathcal{M}$	Tensors and tensor models (script letters)
nnz	Number of non-zero values in a sparse tensor

structure [9], not to an element of a tensor product of vector spaces [10]. As an example, consider a three-way tensor $\mathcal{X}$ representing a knowledge base whose dimensions are *entity*, *relationship*, and *category*, and whose values represent a level of *belief* [11]; that is, value $\mathcal{X}(i, j, k) = v$ is the level of belief, v, that entity i has relationship j with category k, in the knowledge base $\mathcal{X}$. Generally, tensors are N-way arrays where each element has a corresponding N-tuple index $\mathbf{i} = (i_1, i_2, \dots, i_N)$. Each index coordinate i_k fixes an element's location along the k^{th} dimension, with $k \in \{1, 2, \dots, N\}$ and $i_k \in \{1, 2, \dots, I_k\}$. A length I_1 one-dimensional tensor (or vector) has I_1 indexed elements, and an $I_1 \times I_2$ two-dimensional tensor (matrix) has I_1 rows and I_2 columns for its $I_1 I_2$ indexed elements. Hence, an N-dimensional (or N-*mode*) tensor has $\prod_{k=1}^{N} I_k$ indexed elements.

Sparse tensors are tensors where most elements are zero, and are analogous to sparse matrices [12], [13]. Many real-world tensors are sparse tensors [14]; Consider the previous example of the tensor $\mathcal{X}$, whose modes are entity, relationship, and category. It is intuitive that not every every entity is related to every category, meaning that many entries in $\mathcal{X}$ would have belief level zero. Explicitly storing zero-valued entries in a sparse tensor data structure is undesirable due to the high memory requirement for storing numerous unimportant values. It is also unnecessary because non-zero elements can be stored explicitly, and any unstored elements can be assumed to have the value zero.

2.2. Sparse Tensor Storage Format Overview

This section presents an overview of sparse tensor storage formats, which fall into two broad categories: *List-based*, meaning that non-zero coordinates and values are stored in a given listed order, and *non–list-based*, meaning that non-zero coordinates and values are stored in a hierarchical (tree-based) or hash map structure.

2.2.1. Coordinate (COO). COO (list-based) is the canonical sparse tensor storage format [15], [16], and is similar to the sparse matrix storage format of the same name [13]. For an N-mode sparse tensor, COO lists each non-zero element by its corresponding N-dimensional coordinate, typically sorted by coordinates in lexicographical order. Performing a calculation on a sparse tensor stored in COO format typically involves iterating over each non-zero entry and using its coordinates to perform a desired operation on its value. For example, consider a matrix-vector product using a 2-mode sparse tensor (matrix) $\mathbf{A}$ and a dense (non-sparse) vector $\mathbf{x}$ and storing the result in dense vector $\mathbf{y}$: For each non-zero element v with coordinates (i, j) in $\mathbf{A}$, the mode-0 (row) coordinate i determines the participating row in $\mathbf{y}$, and the mode-1 (column) coordinate j determines the participating row in $\mathbf{x}$. Thus, a simple approach iterates over each non-zero element in $\mathbf{A}$, identifies its coordinates (i, j) and value v, multiplies v by $\mathbf{x}_j$, and adds the result to $\mathbf{y}_i$.

The COO format for an N-mode sparse tensor with nnz non-zero elements is straightforwardly represented using a one-dimensional length-nnz value array and a two-dimensional $\mathrm{nnz} \times N$ coordinate array, such that index i (or row i) in each array identifies the value and corresponding coordinates for the i^{th} non-zero element. This approach uses $(N+1) \cdot \mathrm{nnz}$ storage and allows straightforwardly iterating over the non-zero element list by iterating over the nnz array indices. Additionally, the representation allows parallel operation by dividing the nnz array indices between two or more processors.

One drawback to using COO is that its memory footprint scales with the dimensionality and the number of non-zero elements (cardinality), rather than the cardinality alone [12]. Consider a 3-mode sparse tensor $\mathcal{X}$ with nnz non-zero elements and a 15-mode sparse tensor $\mathcal{Y}$ also with nnz non-zero elements. The storage for $\mathcal{X}$ using COO is roughly $4 \cdot \mathrm{nnz}$, because we must store three coordinates and one value for each of the nnz non-zero elements, while the storage for $\mathcal{Y}$ is $16 \cdot \mathrm{nnz}$ for the same number of elements. (Clearly the storage increases also with nnz.)

A more important drawback to COO is that the memory order of the non-zero elements implies a single optimal iteration order for best cache utilization and performance on state-of-the-art memory systems. This is problematic because sparse tensor decomposition often requires iterating over each of the N modes of an N-mode tensor [17], [18]. The problem is due to synchronization overhead in resolving write conflicts between threads, which arise whenever non-zero elements processed by different threads share identical coordinates [19]. One approach for minimizing the synchronization overhead is to store additional permutation arrays which store array indices for non-zero elements in a write conflict–minimizing order [20]. The permutation array approach increases the COO storage requirement by $N \cdot \mathrm{nnz}$ per additional ordering, for a sparse tensor with N modes and nnz non-zero elements. Overall, the COO format is an intuitive storage format with opportunities for memory-footprint and memory-performance improvements.

2.2.2. Adaptive Linearized Tensor Order (ALTO). ALTO (list-based) is a state-of-the-art space-filling curve sparse tensor storage format which maps N-dimensional coordinates to linear indices. The main idea is that non-zero elements near to each other in tensor coordinates are near to each other in linear indices (i.e., in memory) [21]. For an N-mode sparse tensor $\mathcal{X}$, ALTO maps the set of coordinates of non-zero elements to the set of natural numbers $\mathbb{N}$ by recursively dividing the multi-dimensional space occupied by the tensor into halves until locating a given element.

Once the element is located, the recursive division and subsequent choice of halves forms a bit sequence of choices which can be interpreted as a natural number. This natural number is the ALTO index for the given element. ALTO then stores the non-zero elements sorted by their ALTO indices in memory, with similar layout and storage requirements of a one-dimensional COO sparse tensor, for example.

Unlike COO, under certain conditions the storage requirement for ALTO scales only with the non-zero cardinality, not with the sparse tensor dimensionality (*typically*, under condition that the ALTO index fits into a single memory word). This means that ALTO has lower memory usage due to storing a single linear index value for each non-zero entry, instead of N coordinate values; ALTO requires $2 \cdot \mathrm{nnz}$ storage (typically) for a sparse tensor with nnz elements. The lower memory usage leads to higher memory bandwidth utilization by improving sparse tensor storage and memory transfer economy, which is useful because sparse tensor operations are memory bandwidth–bound [22]. ALTO's higher memory bandwidth utilization is also due to the improved mode-agnostic non-zero element access locality afforded by ALTO's space-filling curve approach [21].

One disadvantage to ALTO is that mapping N-dimensional coordinates to linear indices, or vice-versa, incurs computational overhead. ALTO amortizes the overhead as effectively masked by lower memory usage and higher practical memory bandwidth utilization. Another disadvantage is that the memory order of the non-zero elements implies a preferred iteration order (like COO); The memory order depends on the ALTO index, which depends on which tensor dimension is chosen at each step when recursively dividing the multi-dimensional space in half. (ALTO uses a greedy strategy, viz. the largest undivided dimension is chosen at each step.) Permutation arrays could be used for storing alternate orderings for non-zero elements, and would increase the storage requirements by nnz (typically) for each alternate ordering.

2.2.3. Other List-Based Formats. Linearized coordinate (LCO) is a list-based format that stores a single linearized index for each non-zero element value [23]. For example, given a three-dimensional $I \times J \times K$ tensor, an element with coordinates (i, j, k) is assigned the linearized index $\ell_{i,j,k}$ as follows:

$$\ell_{i,j,k} = i + I(j + Jk) \tag{1}$$

Any permutation of the indices (along with their respective dimensions) is valid, and the scheme is trivially extended to higher dimensions. LCO differs from ALTO because LCO is not based on a space-filling curve. LCO is similar to ALTO because both use a linearization method that performs ideally in the case where for a given sparse tensor the linearized coordinate fits into a single memory word.

The block linearized coordinate (BLCO) format is based on ALTO [21] with a re-encoding of the ALTO indices to allow GPU-friendly decoding of ALTO indices, and with adaptive blocking to allow out-of-memory computation for constrained memory GPUs [5].

Flagged COO (F-COO) is based on COO but stores sparse tensors specifically to be used in sparse tensor-times-matrix or sparse matricized tensor times Khatri-Rao product (MTTKRP) operations [19]. The F-COO format stores only the participating coordinate(s) for the desired operation, plus two flag bit arrays for the *bit-flag* and the *start-flag*. The bit arrays are used to implement the segmented scan algorithm [24], [25] to parallelize and reduce product results in the sparse tensor operation.

Hierarchical COOrdinate (HiCOO) stores a sparse tensor partitioned into blocks using a pre-specified block size B (where B is the length of a partition in each sparse tensor dimension) [17]. Each block then has its own coordinates, and HiCOO stores for each non-zero element within a block its coordinates with respect to the sub-tensor represented by the block. This approach allows using fewer bits to store the coordinates for each non-zero element, and the original coordinates can be recovered based on the block coordinates and the element coordinates. HiCOO uses Z-Morton ordering [26] for blocks and for non-zero elements within blocks, due to HiCOO being based on the Compressed Sparse Blocks (CSB) format for sparse matrices [27].

2.2.4. Other Non–List-Based Formats. Compressed Sparse Fiber (CSF) is a tree-based sparse tensor storage format which generalizes the compressed sparse row (CSR), or Yale, sparse matrix storage format [28] to higher-order sparse tensors [29]. For background, the compressed sparse row (CSR) sparse matrix storage format stores row-sorted non-zero entries and their corresponding column coordinates in separate arrays respectively V and C, and uses one additional array P for row offsets. If a sparse matrix is $M \times N$ with nnz non-zero elements, then V and C are length nnz and P is length M. The non-zero value array V stores non-zero entries in increasing row order, C stores the non-zero entries' column indices, and P stores at index i the row starting index i_{start} in V (equivalently C) for the non-zero entries belonging to matrix row i. That is, if a matrix has a non-zero entry with value x at row i and column j, then the entry's value is $V[P[i]] = x$, its column index is $C[P[i]] = j$, and its row index is i. Iterating over the non-zero entries by row can be accomplished by the following procedure: Starting at index zero in P, obtain the starting index for matrix row zero, i_{start}. Index one in P stores the ending index for matrix row zero, i_{end}. For i from i_{start} to i_{end}, index into the non-zero entry array V and the column coordinate array C to obtain respectively the first non-zero entry x in row zero and its corresponding column coordinate j. After completing the loop, increment to the next index in P and repeat. The compressed sparse column (CSC) format is analogous to CSR sparse matrix format, with the distinguished roles of rows and columns interchanged.

The CSF sparse tensor storage format extends the concept of CSR to higher dimensions by adding an additional coordinate and starting index array for each additional dimension [29]. (Note that the name arises from the definition that a *fiber* is analogous to a matrix row or column, except that instead of fixing a row index or column index to select respectively a row or column vector, all of a tensor's dimension indices are fixed except one [3].)

Two advantages to CSF are that sparse tensor operations can be parallelized by partitioning the root array in the CSF tree, and that storage requirements can be reduced by choosing to store fibers along the longer mode (which minimizes the number stored fibers an increases the average fiber length) [29].

One disadvantage to CSF is that its construction implies a preferred iteration order: Observe that CSR allows, at the outer loop level, iterating over sparse matrix rows and only rows, while CSC allows iterating over columns and only columns; Similarly, the CSF format similarly allows iterating over the sparse tensor dimension that is used for the top-level offset array, and only that dimension. This means iterating over the non-zero entries in a sparse tensor stored in CSF format requires starting with a distinguished *first* mode's coordinate array, then using that mode's pointer array to index into the distinguished *next* mode's coordinate array, and so on, until reaching the non-zero entry array and each non-zero entry's corresponding coordinate in the distinguished *last* mode's coordinate array. In other words, iterating efficiently over sparse tensor elements must be done in the dimension ordering as was used during construction, and it is not possible to iterate over the non-zero entries efficiently in any other order. Efficient iteration over non-zero entries in any mode ordering requires essentially storing multiple versions of the sparse tensor in different CSF orderings, which increases the memory storage requirements. As was mentioned previously for COO, sparse tensor operations often require iterating over each dimension of a sparse tensor [17], [18].

Variations on the CSF tree-based format include Balanced Compressed Sparse Fiber (B-CSF) [30] and Mixed Mode Compressed Sparse Fiber (MM-CSF) [18].

Hashed Coordinate (HaCOO) is a hash table–based format that uses Morton Z-ordering for encoding tensor coordinates like HiCOO [17], and stores a single index with each sparse tensor value like ALTO [21], but in a hash table structure unlike either of these [31]. Using a hash table structure allows amortized constant time non-zero element insertion, retrieval, and deletion, which is not possible with state-of-the-art list-based or tree-based storage formats.

2.2.5. Synthesis. There are many sparse tensor storage formats, each with their own specific focus. These foci typically regard improving the performance of sparse tensor calculations Table 2 organizes approximate storage requirements for the sparse tensor storage formats discussed here.

2.3. Sparse Tensor Decomposition

Tensor rank decomposition is a technique that can reveal latent information in data [32] for applications in machine learning [33], [34], [35], function approximation [36], [37], [38], [39], cyber security [40], knowledge-based systems [41], signal processing [42], health analytics [43], and phenotyping [44].

TABLE 2. Sparse Tensor Storage Formats

This table provides approximate sparse tensor storage requirements for an N-mode sparse tensor with nnz non-zero elements and with J_k unique coordinates in mode k for $1 \leqslant k \leqslant N$. For ALTO and LCO, "typically" refers to the case where a linearized coordinate fits into a memory word.

Format	Storage Requirement
COO	$(N+1) \cdot \mathrm{nnz}$
ALTO (typically)	$2 \cdot \mathrm{nnz}$
LCO (typically)	$2 \cdot \mathrm{nnz}$
HiCOO (best case)	$N \cdot \mathrm{nnz}$
HiCOO (worst case)	$(2N+1) \cdot \mathrm{nnz}$
CSF (one mode)	$2 \cdot \mathrm{nnz} + (N-1) + 2\sum_{k=1}^{N-1} \prod_{\ell=1}^{k} J_\ell$
HaCOO	$2 \cdot \mathrm{nnz}$

The most commonly referenced tensor rank decomposition is the canonical polyadic decomposition (CPD) or CANDECOMP/PARAFAC (CP) model [45], [46], which is a higher-order generalization of the matrix singular value decomposition (SVD). Recall that the SVD involves decomposing an $m \times n$ complex matrix $\mathbf{M}$, commonly written as $\mathbf{M} = \mathbf{U\Sigma V}^*$, into a sum $\sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^*$ where $r = \min(m, n)$, the σ_i are weights, and the $\mathbf{u}_i$, $\mathbf{v}_i^*$ are orthonormal bases of M. The CP model generalizes the SVD concept to decompose an N-mode tensor $\mathcal{X}$ into the sum of R rank-one outer products of N vectors, where R is a parameter and where each outer product can reveal latent information about a property within the data. (Note that determining the rank of a tensor is NP-complete [3], [47], hency the rank of a CP decomposition is typically chosen arbitrarily as a parameter.) For notational convenience, we consider a CP model to be represented by $\mathcal{M} = \{\lambda; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}\}$, called a *Kruskal tensor*, where λ is a weight vector, and $\mathbf{A}^{(i)}$ is an R-column *factor matrix* containing R vectors (to be used in outer products) for mode i, for $i = 1, 2, \ldots, N$ [48].

Examples of CP algorithms include an alternating least squares formulation for Gaussian-distributed data (CP-ALS) [45], [46] and alternating Poisson regression for count data (CP-APR) [32].

2.3.1. CP-ALS. The primary algorithm for computing CP on a tensor representing Gaussian-distributed data is the well-known alternating least squares (ALS) method [45], [46]. After fixing the desired rank R, we can consider the algorithm as shown in Algorithm 1.

During each iteration of the algorithm, an individual factor matrix is selected (in an alternating fashion) and updated to yield the best approximation of the sparse tensor while keeping all other factor matrices fixed. Note that an initial guess for the factor matrices $\mathbf{A}^{(i)}, i = 1, 2, \ldots, N$ can be provided in any desired way (e.g. random or otherwise). The algorithm concludes when the desired fit is achieved, which is typically within some error tolerance. A maximum number of iterations is also typically specified as a stopping criterion.

The most computationally intensive calculation of CP-ALS is the sparse matricized tensor times Khatri-Rao product (MTTKRP) [29]. This calculation involves first flattening

Algorithm 1 CP-ALS algorithm.

Given an N-mode tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, and an initial guess for a model $\mathcal{M} = \{\lambda; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}\}$, where:

- $*_{i=1}^{N} \mathbf{A}^{(i)}$ is the element-wise product over $\mathbf{A}^{(i)}$.
- $\mathbf{X}_{(n)}$ is the mode-n flattening of $\mathcal{X}$.
- $\bigodot_{i=1}^{N} \mathbf{A}^{(i)}$ is the column-wise Kronecker product over $\mathbf{A}^{(i)}$.
- $\mathbf{V}^{\dagger}$ is the Moore-Penrose pseudoinverse of $\mathbf{V}$.

```
for (maximum iterations or until desired fit) do
    for $n \leftarrow 1, 2, \ldots, N$ do
        $\mathbf{V} \leftarrow *_{i=1, i \neq n}^{N} \mathbf{A}^{(i)\top} \mathbf{A}^{(i)}$
        $\mathbf{A}^{(n)} \leftarrow \mathbf{X}_{(n)} \left( \bigodot_{i=N, i \neq n}^{1} \mathbf{A}^{(i)} \right) \mathbf{V}^{\dagger}$
        $\lambda \leftarrow \big(\text{norms of normalized columns of } \mathbf{A}^{(n)}\big)$
    end for
end for
return $\lambda, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$
```

the given N-mode tensor $\mathcal{X}$ into a matrix $\mathbf{X}_{(n)}$, where the mode-n fibers of $\mathcal{X}$ are used as the columns for $\mathbf{X}_{(n)}$. Note that the matrix $\mathbf{X}_{(n)}$ (commonly referred to as a *matricized* sparse tensor) is typically formed on-the-fly. This on-the-fly formation of $\mathbf{X}_{(n)}$ can be achieved by iterating appropriately over the elements of $\mathcal{X}$, and this formulation is desirable for large sparse tensors because it avoids explicitly duplicating $\mathcal{X}$ in memory. The second part of the calculation involves multiplying $\mathbf{X}_{(n)}$ by the Khatri-Rao product of the factor matrices while excluding the n^{th} factor matrix. Note that the Khatri-Rao product in this context is the column-wise Kronecker product of matrices. For example, given two matrices $\mathbf{B} \in \mathbb{R}^{J \times R}$ and $\mathbf{C} \in \mathbb{R}^{K \times R}$, the Khatri-Rao product of $\mathbf{B}$ and $\mathbf{C}$ is as defined as $\mathbf{A} \in \mathbb{R}^{JK \times R}$ in Equation 2:

$$\mathbf{A} = \mathbf{B} \odot \mathbf{C} = [\mathbf{b}_1 \otimes \mathbf{c}_1, \mathbf{b}_2 \otimes \mathbf{c}_2, \ldots, \mathbf{b}_R \otimes \mathbf{c}_R] \qquad (2)$$

The remaining calculations used in CP-ALS are calculating an element-wise product $\mathbf{V}$ and its Moore-Penrose inverse, and calculating norms of the normalized columns of the current factor matrix in order to update the weights vector λ.

2.3.2. CP-APR. The primary algorithm for computing CP on a tensor containing sparse count data is CANDECOMP/PARAFAC via Alternating Poisson Regression (CP-APR), due to how a Poisson distribution describes the random variation in such data [32]. There are three methods for computing CP-APR:

(i) Multiplicative update (MU)
(ii) Projected damped Newton for row-based sub-problems (PDN-R)
(iii) Projected quasi-Newton for row-based sub-problems (PQN-R)

We focus our discussion on MU due to its popularity, which it possesses despite the fact that it suffers from slower convergence than PDN-R and PQN-R [49]. The convergence rates are different because MU uses a form of scaled steepest-descent with bound constraints over all rows during each iteration, while PDN-R and PQN-R use second-order information to solve independent row sub-problems. For a detailed discussion on the CP-APR MU algorithm, see [32].

2.3.3. CP-APR MU Algorithm Overview. The CP-APR MU algorithm is shown in Algorithm 2, using the notation from Table 1.

Algorithm 2 CP-APR MU algorithm.

Given an N-mode tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, and an initial guess for a model $\mathcal{M} = \{\lambda; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}\}$, where:

- R is the desired number of model components.
- $\kappa_{\min}$ is the inadmissible zero minimum.
- κ_{adj} is the inadmissible zero adjustment value.
- $\mathbf{B}$ is an intermediate representation of $\mathbf{A}^{(n)}$.
- λ is the weight vector.
- $\mathbf{\Lambda}$ is the diagonal weight matrix $(\text{diag}(\lambda))$.
- $\mathbf{\Pi}$ and $\mathbf{\Phi}$ are intermediate calculation matrices.
- $\bigodot_{i=1}^{N} \mathbf{A}^{(i)}$ is the column-wise Kronecker product over $\mathbf{A}^{(i)}$.
- $\mathbf{X}_{(n)}$ is the mode-n flattening of $\mathcal{X}$.
- $\oslash$ is element-wise division.
- $\max^{\circ}$ is the element-wise maximum.
- ϵ is a minimum divisor to prevent divide-by-zero.
- $\mathbf{1}$ is the all-ones vector.
- τ is the convergence error tolerance.
- $*$ is element-wise multiplication (Hadamard product).

```
for (maximum outer iterations) do
    is_converged $\leftarrow$ true
    for $n \leftarrow 1, 2, \ldots, N$ do
        $\mathbf{S} \leftarrow \mathbf{0}$
        if (completed at least one outer iteration) then
            for $i \leftarrow 1, 2, \ldots, I_n$ and $r \leftarrow 1, 2, \ldots, R$ do
                if $\mathbf{A}_{i,r}^{(n)} < \kappa_{\min}$ and $\mathbf{\Phi}_{i,r}^{(n)} > 1$ then
                    $\mathbf{S}_{i,r} \leftarrow \kappa_{\text{adj}}$
                end if
            end for
        end if
        $\mathbf{B} \leftarrow \big(\mathbf{A}^{(n)} + \mathbf{S}\big)\mathbf{\Lambda}$
        $\mathbf{\Pi}^{\top} \leftarrow \bigodot_{k \neq n} \mathbf{A}^{(k)}$
        for (maximum inner iterations) do
            $\mathbf{\Phi} \leftarrow \big(\mathbf{X}_{(n)} \oslash \max^{\circ}(\mathbf{B\Pi}, \epsilon)\big)\, \mathbf{\Pi}^{\top}$
            if $\max_{i,r} \big|\min\big(\mathbf{B}_{i,r},\, 1 - \mathbf{\Phi}_{i,r}^{(n)}\big)\big| < \tau$ then
                break
            end if
            is_converged $\leftarrow$ false
            $\mathbf{B} \leftarrow \mathbf{B} * \mathbf{\Phi}$
        end for
        $\lambda \leftarrow \mathbf{1}^{\top}\mathbf{B}$
        $\mathbf{A}^{(n)} \leftarrow \mathbf{B}\mathbf{\Lambda}^{-1}$
    end for
    if is_converged = true then
        break
    end if
end for
```

The overall calculation takes an N-mode input tensor $\mathcal{X}$ and iterates towards an approximate model tensor $\mathcal{M}$ in two nested stages called *outer* and *inner*, where the outer iteration updates each of the factor matrices and the inner iteration calculates successive updates to the current factor matrix. In each outer iteration, CP-APR MU updates the N factor matrices, $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$, as follows: If at least one outer iteration has completed, then the algorithm searches the current model factor matrix $\mathbf{A}^{(n)}$, $1 \leqslant n \leqslant N$, for values smaller than a minimum tolerance $\kappa_{\min}$. These values are called *inadmissible zeros* because they may interfere with solution convergence; if an inadmissible zero is detected, it is shifted by an adjustment value parameter, κ_{adj}. After shifting inadmissible zeros, $\mathbf{A}^{(n)}$ is scaled by the weight vector λ, yielding the current working model factor matrix $\mathbf{B}$. Before entering the inner iterative stage, the intermediate $\Pi^{\top}$ matrix is calculated as a chained column-wise Kronecker product over all model factor matrices except $\mathbf{A}^{(n)}$.

In each iteration of the inner stage, the current model factor matrix $\mathbf{A}^{(n)}$, $1 \leqslant n \leqslant N$, undergoes a series of multiplicative updates. Each multiplicative update begins by calculating an element-wise division between the mode-n flattening of $\mathcal{X}$ (i.e., $\mathbf{X}_{(n)}$) and the product of $\mathbf{B}$ and $\mathbf{\Pi}$, all followed by a matrix multiplication by $\mathbf{\Pi}^{\top}$, to produce the intermediate $\mathbf{\Phi}$ matrix. The $\mathbf{\Phi}$ matrix calculation just described is the most time-intensive component of the CP-APR MU algorithm [22].

After calculating $\mathbf{\Phi}$, it is compared element-wise with $\mathbf{B}$ against the convergence tolerance τ: if the convergence tolerance is satisfied, then the inner stage concludes. Otherwise, $\mathbf{B}$ is multiplied element-wise by $\mathbf{\Phi}$ and the inner stage repeats. After satisfying the convergence tolerance, or otherwise completing the maximum desired iterations in the inner stage, the outer stage updates the weights in λ from $\mathbf{B}$ and normalizes $\mathbf{A}^{(n)}$ using the updated λ. The algorithm concludes when all N tensor modes have converged simultaneously, or otherwise after completing the maximum desired iterations in the outer stage.

2.3.4. CP-APR MU Sparse Tensor Implementation.

Most real-world tensors contain a massive amount of multidimensional data, which would requre storage on the order of exabytes (e.g., 3.2×10^{20} bytes for *LBNL-Network* [14]) if stored as a dense multi-dimensional array. This storage requirement is infeasible on most computing systems. In addition, these data are sparse, meaning that the majority of the elements in the multidimensional space are zero-valued. Thus it is practical to store only the non-zero values of a tensor (e.g., 13.5×10^{6} bytes for *LBNL-Network*) with their corresponding coordinates in a sparse tensor storage format. This presents a challenge for sparse tensor algorithm designers because each format will require a different implementation (e.g. [12], [17], [50]).

The sparse tensor storage formats discussed previously (with the exception of HaCOO) do not allow a programmer to access an arbitrary non-zero sparse tensor element in constant time. Instead, each format stores the non-zero values and provides the means to retrieve the coordinates corresponding with each non-zero value: For COO, the coordinate N-tuples are stored corresponding with each non-zero value. For CSF, the array indices and fiber indices allow calculating the coordinate N-tuple for each element. For HiCOO, the hierarchical block indices representing compressed versions of the coordinate N-tuples corresponding with each non-zero value. For ALTO, the linear indices representing encoded versions of the coordinate N-tuples corresponding with each non-zero value. This means that while constant-time arbitrary element access is not possible, the coordinate for a given non-zero value can be retrieved. For this reason, state-of-the-art implementations of sparse CP-APR MU iterate over the non-zero values of a given input sparse tensor and retrieve the corresponding coordinates for performing any necessary calculations [50]. The participating rows and columns in the current working factor matrix $\mathbf{B}$ can be accessed because it is stored densely (e.g. as a common two-dimensional array). Only the rows of $\mathbf{\Pi}$ which correspond with coordinates for non-zero values in the sparse tensor are required for calculating CP-APR MU, and the rows can be calculated explicitly or each element can be calculated on-demand [32].

Program listings for sparse CP-APR MU $\mathbf{\Pi}$ and $\mathbf{\Phi}$ calculations are shown respectively in Algorithms 3 and 4.

Algorithm 3 Sparse CP-APR MU mode-n $\mathbf{\Pi}$ calculation.

Given an N-mode tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, and an initial guess for a model $\mathcal{M} = \{\lambda; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}\}$, where:

- $\texttt{coord}_{\mathcal{X},n,i}$ is the n^{th} coordinate for $\mathcal{X}_i$.
- R is the desired number of model components.
- $*$ is element-wise multiplication (Hadamard product).

```
for $k \leftarrow 1, 2, \ldots, N$ do
    if $k \neq n$ then
        for $i \leftarrow 1, 2, \ldots, \text{nnz}$ do
            $\texttt{row} \leftarrow \texttt{coord}_{\mathcal{X},n,i}$
            for $r \leftarrow 1, 2, \ldots, R$ do
                $\mathbf{\Pi}_{i,r} \leftarrow \mathbf{\Pi}_{i,r} * \mathbf{A}^{(k)}_{\texttt{row},r}$
            end for
        end for
    end if
end for
```

2.3.5. $\mathbf{\Phi}^{(n)}$ Calculation.

This section takes a closer look at the $\mathbf{\Phi}^{(n)}$ calculation, which is the most computationally intensive part of CP-APR MU [22]. Calculating $\mathbf{\Phi}^{(n)}$ for each tensor mode n uses the formula shown in Equation 3:

$$\mathbf{\Phi}^{(n)} \leftarrow \left(\mathbf{X}_{(n)} \oslash \left(\max\left(\mathbf{B\Pi}, \epsilon\right)\right)\right) \mathbf{\Pi}^{\top} \tag{3}$$

The matrix $\mathbf{\Pi}^{\top}$ is the result of a chained column-wise Kronecker product of factor matrices, and hence is size $\prod_{k \neq n} I_k \times R$, where R is a parameter. Note that the matrices $\mathbf{X}_{(n)}$ and $\mathbf{\Pi}$, if stored densely, would require as much storage as a densely-stored tensor $\mathcal{X}$. If $\mathcal{X}$ is too large to store densely, then $\mathbf{X}_{(n)}$ and $\mathbf{\Pi}$ cannot be stored

Algorithm 4 Sparse CP-APR MU mode-n $\mathbf{\Phi}$ calculation.

Given an N-mode tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, a Khatri-Rao product $\mathbf{\Pi}^\top = \bigodot_{k \neq n} \mathbf{A}^{(k)}$ from a model $\mathcal{M} = \{\lambda; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}\}$, and an arbitrary factor matrix $\mathbf{B}$, where:

- $\mathcal{X}_i$ is the i^{th} non-zero value in $\mathcal{X}$.
- $\texttt{coord}_{\mathcal{X},n,i}$ is the n^{th} coordinate for $\mathcal{X}_i$.
- R is the desired number of model components.
- $*$ is element-wise multiplication (Hadamard product).

```
 1: for $i \leftarrow 1, 2, \ldots, \text{nnz}$ do
 2:     $\texttt{row} \leftarrow \texttt{coord}_{\mathcal{X},n,i}$
 3:     $\texttt{temp} \leftarrow 0$
 4:     for $r \leftarrow 1, 2, \ldots, R$ do
 5:         $\texttt{temp} \leftarrow \texttt{temp} + \mathbf{B}_{\texttt{row},r} * \mathbf{\Pi}_{i,r}$
 6:     end for
 7:     $\texttt{temp} \leftarrow \mathcal{X}_i / \max(\texttt{temp}, \epsilon)$
 8:     for $r \leftarrow 1, 2, \ldots, R$ do
 9:         $\mathbf{\Phi}_{\texttt{row},r} \leftarrow \mathbf{\Phi}_{\texttt{row},r} + \texttt{temp} * \mathbf{\Pi}_{i,r}$
10:     end for
11: end for
```

densely. For example, consider a four-mode tensor $\mathcal{S}$ of size $1000 \times 1000 \times 1000 \times 1000$, whose values require four bytes for storage. The $\mathbf{\Pi}$ matrix for $\mathcal{S}$ will have size $10^9 \times R$, which requires $10^9 \times 10 \times 4$ bytes $= 40\,\text{GiB}$ storage when using $R = 10$ and storing single-precision floating-point values. That said, $\mathbf{X}_{(n)}$ need not be stored explicitly because it is simply a flattening of $\mathcal{X}$ into matrix form, meaning we can access all of its elements by translating their 2-D indices to N-D indices and accessing the elements in $\mathcal{X}$. Furthermore, computing CP-APR MU on a sparse tensor $\mathcal{X}$ does not require the entire $\mathbf{\Pi}$ matrix. It requires only the rows of $\mathbf{\Pi}$ which correspond with nonzero entries in $\mathcal{X}$. For this reason, high-performance CP-APR MU implementations calculate only the rows of $\mathbf{\Pi}$ required by each non-zero element, reducing the size requirement greatly [50]. As an example, suppose that $\mathcal{S}$ has 1 M non-zero elements, whose values require four bytes each. The participating rows of $\mathbf{\Pi}$ require $1 \times 10^6 \times 10 \times 4$ bytes $= 40\,\text{MiB}$ storage (single-precision), a 1000-fold reduction in memory requirements from forming the entire $\mathbf{\Pi}$ matrix.

Calculating $\mathbf{\Phi}^{(n)}$ in parallel can manifest race conditions, particularly when using sparse tensor storage. Sparse tensor algorithms iterate over the non-zero elements rather than their indices, because random access to sparse tensor storage format is typically not possible. Observe that if two non-zero elements share the same coordinate for mode n, then both elements will correspond with updating the same row in $\mathbf{\Phi}^{(n)}$. This creates a challenge for parallel performance because the updates to the same row by different resources must be serialized (e.g. via atomic operations) to maintain correctness [32], [50], which reduces the benefits of parallel processing. For example, if two threads are assigned many non-zeros which update the same row in $\mathbf{\Phi}^{(n)}$, then processing those non-zeros is effectively sequential. One way to mitigate the performance reduction is to sort the non-zero elements by coordinate so that elements which update the same row in $\mathbf{\Phi}^{(n)}$ are stored contiguously [50]. This increases the likelihood that a processing resource will receive a workload containing elements which typically and exclusively update the same row in $\mathbf{\Phi}^{(n)}$, meaning that atomic operations can be avoided for several of the elements. For example, if a processing resource is assigned non-zero elements which update row $r-1, r, r+1$ in $\mathbf{\Phi}^{(n)}$, then all non-zeros in $\mathcal{X}$ which update row r are solely processed by this thread, and the thread can avoid atomic operations for those non-zero elements. Sorting by coordinate will be required for each mode n when using this approach, because $\mathbf{\Phi}^{(n)}$ is calculated for each mode. All sorting can be performed in advance, storing the permutation information in arrays, one for each mode [50].

2.3.6. Other Decompositions. Other sparse tensor decompositions include the Tucker decomposition [51], [52], [53], [54], which has been shown to provide high-quality, high-compression for dense data (e.g. images [55], volume rendering [56], and scientific simulation [57]) and to reveal useful latent information (e.g. health care [58], network traffic [59]), and generalized CP (GCP) [20], [60], [61], [62], which generalizes the loss function for determining the low-rank model.

2.3.7. Synthesis. As shown, there are many reasons for using the sparse tensors and their decompositions, and respectively there are many storage formats for representing them and algorithm implementations for decomposing them.

3. Decisions for Software Composition

This section describes the decisions one must make in order to identify a composition system appropriate for a particular HPC application composition. Systems for coupling software together can be grouped into four main categories, or realms, where each system in a realm is similarly suited for addressing challenges that arise in that realm.

The first decision to make is whether you have the means to instrument the software. If so, then this brings us to our first realm, Workflow Managers (§4.1), where the following statements are true:

1) I don't have the means to instrument the software. Integrating that software into a single software stack is prohibitive from a developer perspective. I want the software to be integrated and work together. I am less concerned about efficiency.

Note that this is the only available realm if you do not have the means to instrument the software.

Otherwise, there are more decisions to make. The remaining realms all exist due to the decision to instrument the software, and include Middleware (§4.2), Discrete Processing Services (§4.3), and Distributed Data Services (§4.4):

2) I want my software to be integrated and work together. I don't want to instrument the software again if I decide to integrate it with future software.

3) My software constitutes a producer-consumer relationship. The producer is highly sensitive to performance variation.
4) The software all interact with data in a many-to-many relationship. The software all run at distinct scales and times. The data formats require adaptation between the software.

The next sections describe systems fit for the challenges represented by these realms.

4. Systems for Software Composition

This section provides an overview of composition systems for HPC applications. The high-level types of systems are Workflow Managers (§4.1), Middleware (§4.2), Discrete Processing Services (§4.3), and Distributed Data Services (§4.4). Each high-level type of system focuses on a particular realm of applicability.

4.1. Workflow Managers

A workflow manager (WM) is a software package that orchestrates the execution of disparate software and connects their outputs and inputs. The most compelling feature of WMs in the context of this paper is that none of them require modifying source code. In other words, if one wishes to compose HPC software but the source code cannot be modified, then WMs are the only option. Each WM in this section is nevertheless well-suited to scenarios where compute resources and data formats are heterogeneous, and where tool usage across these resources would benefit from an orchestration system and from data exchange coordination [63], [64], [65].

The remainder of this section surveys three noteworthy examples of WMs, where each has a unique focal point compared to the others: Swift enables general-purpose workflow scripting for scientists (§4.1.1), Pegasus maps and runs workflows represented as DAGs on high-performance resources (§4.1.2), and Kepler streamlines workflow creation and execution by via abstraction, execution, and data access (§4.1.3).

4.1.1. Swift. Programmers in scientific computing are focused on managing large numbers of datasets and tasks, rather than on optimizing interprocessor communication [63]. Swift is a parallel programming system allowing users to express operations on data sets, execute tasks locally or remotely, and capture provenance information. As an alternative to conventional shell scripting, Swift uses XML Dataset Typing and Mapping (XDTM) to allow users to map logical data structure (such as studies, groups, subjects, and runs) to physical data structure (directories and files). Further, the SwiftScript scripting language enables users to define typed logical procedures which map to binary executables for performing operations on those logical data structures, concisely and in parallel. Swift uses Karajan [66] as its lightweight threading-based execution engine and supports load balancing, fault tolerance, computation restart, and large-scale distributed execution.

4.1.2. Pegasus. Scientists require reliable and scalable access to the capability of local and remote distributed computing resources for processing and analyzing vast quantities of data [64]. The Pegasus Workflow Management System maps resource-independent (abstract) scientific workflows to executable workflows on distributed data and compute resources. Abstract workflows are represented as directed acyclic graphs (DAGs) where nodes in a workflow DAG represent computational tasks and edges represent data- or control-flow dependencies between tasks. Executable workflows are HTCondor [67] DAGs where nodes represent executable jobs. Data are computed on demand or transferred via files as necessary between executable jobs, and can be made available via automatically generated data stage-in and stage-out jobs. Pegasus enables abstract workflows to be portable across execution environments, and allows improving executable workflow reliability and performance via optimizations that can be employed at compile time or run time.

4.1.3. Kepler. Scientists conduct experiments using a variety of computational tools and hardware systems (a scientific workflow) while mentally coordinating data export and import between the tools and between the systems [65]. Scientific workflows are computationally intensive and dataflow-oriented, and they operate on large, complex, heterogeneous data, producing derived data products which may be archived or reused in subsequent workflows. Kepler is a system for streamlining the creation and execution of scientific workflows by combining high-level workflow design, execution and runtime capabilities, and local and remote data access. Kepler is based on Ptolemy II [68], which identifies workflow steps with *actors* representing data sources, sinks, or transformers, and analytical or arbitrary computational steps. Each actor can be parameterized for specific behavior, and can have zero or more input and output ports for dataflow, with type checking. Actors can be coalesced into composite actors for achieving the desired level of design abstraction [69].

Kepler allows scientists to prototype actors and workflows prior to implementing code, use web and grid distributed services, interact with databases and web browsers, and transform data that is semantically compatible yet syntactically incompatible (such as XSLT, XQuery, Perl), all via specialized actors. In particular, Kepler provides actors supporting ProxyInit, GlobusJob, DataAccessWizard, GridFTP, Storage Resource Broker (SRB), and Ecological Metadata Language (EML). Finally, Kepler (via its underlying system Ptolemy II) supports reusing existing analytic and computational tools and foreign language interfaces via the Java Native Interface, and includes actors for Python and MATLAB.

4.1.4. Synthesis. All of these Workflow Manager systems are sensible choices when source code cannot be modified.

Each system has a unique focus that may be useful when deciding which one to use:

Swift is the only system focused on allowing programs to be run as simply as invoking functions in a script, while using the return value of one program as an argument to another program, for example.

Pegasus uniquely focuses on *on-demand* data generation via a dataflow-oriented directed acyclic graph approach, which also supports data and control to pass between composed programs.

Kepler, which extends Ptolemy II, is the only system directly supporting a design-centric, actor-based modeling approach. This approach allows actors to perform nearly any task, share data and control information, operate concurrently, and be rearranged easily into a new workflow configuration as desired.

None of the Workflow Managers shown here are able to avoid using file I/O for sharing data between composed programs. This is an intuitive caveat, as most programs use file I/O for inputting and outputting data, and if they cannot be modified then this is their only means of sharing data.

4.2. Middleware

Middleware is a library that provides an interface between software and the OS. Middleware enables *the software that uses it* to 1) share data and 2) control other software. The primary motivator for using a Middleware system is to reduce the need to re-instrument software each time that a new software composition is desired. Beyond that, the Middleware systems discussed in this section have their own motivations:

- There is a need to compose workflow managers, which themselves compose individual software [70].
- Resilience is reduced when composed programs share memory space [71].
- Ease of Middleware integration is as important as scalability [72].
- Middleware should reduce, not increase, the need to re-instrument software [73].
- Performance drops when composed programs share memory due to serialized access [74].
- Modularizing programs eases Middleware integration [75].

Six noteworthy examples of Middleware are shown in this section, each with a unique perspective compared with the others: Decaf handles communication and data exchanges to couple in situ tasks to form hierarchical workflows (§4.2.1). ADIOS provides an intra-/inter-node in-transit I/O bus for composition (§4.2.2). Freeprocessing uses dynamic binary translation to lower the in situ entry barrier (§4.2.3). SENSEI eliminates multiple in situ implementations via an intermediate-representation data model and instrumentation API (§4.2.4). TCASM supports one-to-many intra-node copy-on-write-demand in-transit processing (§4.2.5). Henson enables loosely-coupled producer-observers using coroutines and process-independent executables (§4.2.6).

4.2.1. Decaf. Computational science workflows can be modeled as a directed graph, where nodes represent computational tasks and edges represent data transfer, with cycles so that the result of a task can be used to modify another task [70]. Decaf is a dataflow library for in situ workflows, where dataflow is the data transferred between tasks and where in situ workflows feature dataflow via memory or supercomputer interconnect to avoid the storage I/O bottleneck. The focus of Decaf is on the data transformation and redistribution between tasks (the dataflow) and hence Decaf is designed for composing different workflow systems. That said, it can also be used as a stand-alone workflow system. Data structures between tasks are often different and require transformation or remapping, and Decaf addresses this need by using an intermediate parallel program serving as a data staging area (a link) between tasks. Tasks may run asynchronously and at different rates, so Decaf uses a flow control library to mitigate potential dataflow issues arising from tasks exchanging data asynchronously (such as stalling or overflow) by buffering messages between tasks. Decaf uses the data model and redistribution components from Bredala [76] and the flow control library from Manala [77]. Finally, Decaf provides a means for describing data contracts between tasks (as producers and consumers) so that a producer sends only the data required by each particular consumer and no superfluous transfers.

Workflow graphs in Decaf are described by the user in a Python script describing individual tasks (MPI programs) and a set of resources (MPI ranks), and then describing dataflows between tasks including a set of resources and a choice of redistribution strategy. The modifications required for participating tasks is to replace their `MPI_COMM_WORLD` with Decaf's communicator, and to use Decaf's get/put API (available in C/C++) to transfer data within the workflow.

4.2.2. ADIOS. Composing programs often requires modifying sources, and shared process images between composed programs often leads to reduced resilience [71]. ADIOS provides an intra-node or inter-node, and in-line or in-transit, I/O bus for program composition.

Applications almost universally read and write files from storage. ADIOS is a middleware layer that manages data movement by interposing between a scientific simulation and the underlying computer system. ADIOS supports runtime-determined data targets for simulations, including file systems, in situ methods, and memory buffers (local and networked). Applications can read from in situ methods or file systems as data sources. Data sharing and movement operations are decoupled from producers and consumers, and the operations can be modified at runtime.

Workflow construction is easier when producers and consumers can be connected together without modifying their source codes. Evolving software complexity is reduced because data access issues are handled by the middleware. Producers and consumers are more resilient because they need not share the same process image.

An application must use (or be modified to use) the ADIOS file-like API for file I/O. Producers and consumers can run on the same or different computing resources. The consumer can directly access the producer's memory or a local or remote buffer of the producer's memory. Producers and consumers can run synchronously or asynchronously, with or without human-in-the-loop controls.

ADIOS seeks to provide an abstraction for I/O operations for parallel and distributed applications to express what data are produced by producers and when the data are ready for output, or what data are consumed by consumers and when the data are desired. ADIOS provides I/O *engines* for use by producers and consumers, where the engine handles actual data movement. The data movement can be to disk, to a consumer sharing the same resources as the producer, to a separate node, or across the network to a remote computing center, for example. Engines can be selected by producers or consumers at runtime, and are delegated the responsibility for any implementation details needed for achieving scalable data movement performance. Each engine supports a unified interface for *putting* and *getting* data, and ADIOS provides several engines including those for parallel file I/O (BPFile and HDF5) and for data staging (via RDMA, TCP, UDP, shared memory, and MPI). This data staging allows producers to share data with one or more consumers via shared memory or network, with the simplicity of reading and writing to files. ADIOS supports performing arbitrary calculations on data within engines via a general purpose callback operator, and provides a file compression implementation as an example. File compression is useful for avoiding the situation where a file system is overloaded by the amount of data generated and written from a high-performance scientific simulation. The VisIt [78] and ParaView [79] visualization tools represent example consumers supporting ADIOS for data reading.

4.2.3. Freeprocessing. Simulation authors integrate visualization and analysis tasks into the simulation itself in order to avoid using disk storage as an intermediary, which is the most expensive bottleneck of visualization and analysis pipelines, and most in situ approaches presuppose integration with particular libraries or require significant effort to integrate simulation and visualization software [72]. Freeprocessing uses binary instrumentation to enable unmodified simulation software to forward output data to another program conditionally and with or without data copying, obviating the potentially complicated in situ requirement of simulation source code instrumentation. The method used by Freeprocessing is to redefine standard I/O functions, such as the POSIX I/O layer C functions open(2) and write(2), dynamically at load time to forward data to a loadable module (called a freeprocessor) that implements a desired in situ computation while continuing to operate normally. The module interface for a freeprocessor (C or Python) uses a stream processing model where data are input to the freeprocessor and may be immediately utilized or ignored, and are otherwise unavailable. Freeprocessing is an in situ approach that requires minimal or no effort on the part of the simulation author and enables novel applications in cases where source code is unavailable, but requires per-simulation effort and supports only unidirectional data transfer from the simulation to the composed program.

4.2.4. SENSEI. In situ runtime and hardware choices have different requirements, necessitating multiple algorithm implementations [73]. SENSEI eliminates the need for multiple in situ implementations by providing a generic in situ and in transit interface comprising an instrumentation API and an intermediate representation for data models.

In situ processing generally requires adding instrumentation code to data producers and consumers. There exist in situ APIs, such as those included in Libsim [80] and Catalyst [79], but they are not compatible with each other. An application would need to be instrumented for both in situ APIs in order to use both of these visualization tools simultaneously, and this issue compounds as more in situ processing tools are desired. SENSEI terms this issue as *tool portability* in the sense that a SENSEI-instrumented data producer can use several external tools interchangeably and cumulatively without requiring further instrumentation.

There are three design considerations for SENSEI: Once an application has been instrumented with SENSEI, it should require no further instrumentation for utilizing any in situ external tool. If an external tool works with SENSEI, it should require only minor changes at the metadata level (e.g. variable names) rather than larger code rewrites. Simplified creation of in situ methods and tools for simulation scientists, data analysts, and visualization experts. SENSEI's two main components for meeting these design considerations are as follows: Solving a data model problem so that producers and consumers are able to exchange data, and defining an API with respect to common design patterns that is suitable for data producer and data consumer instrumentation.

The common data description for SENSEI is an extension to the VTK data model, chosen due to its wide use [78], [79], native support for many common scientific data structures, and community support towards exascale computing. The SENSEI extension includes support for arbitrary layouts for multicomponent arrays, which minimizes effort and memory overhead when mapping memory layouts for data arrays from applications to the VTK data model. The SENSEI interface comprises three parts: An *adaptor* for a data producer which maps producer data to the VTK data model, an adaptor for a data consumer which maps data from the VTK data model to consumer data, and a bridge between the adaptors. SENSEI adaptors support meshes, which include spatially geometric data, and non-spatially oriented arrays, tables, and graphs. Each mesh represents a logical grouping meta-structure comprising blocks which can be distributed for parallel execution via MPI. Finally, SENSEI supports invoking user-written parallel Python-based methods and in situ, at scale on HPC systems.

4.2.5. TCASM. The explosion of data produced by large-scale simulations increases the difficulty of moving the data from the HPC system to permanent storage and back again for additional processing steps, prompting the development of in situ approaches based on absorbing additional processing into the simulation by quickly coupling the two together with a small amount of scripting or glue code in an ad-hoc manner with little system support, promoting work replication, unscalable and fragile frameworks, and moving large amounts of data off-node [81]. Transparent Consistent Asynchronous Shared Memory (TCASM) remedies the lack of system support for application composition by leveraging copy-on-write (COW) and virtual memory mappings to allow applications to share data on demand and without undesired sharing of changes to the data.

The TCASM approach provides a shared memory interface that avoids data duplication and synchronization in a producer-observer setting by utilizing operating system COW to mediate producer-side data changes, ensuring that the observer is guaranteed read access to consistent data even when the producer and observer advance at different rates. TCASM provides modified mmap and msync, which are two existing memory functions in Linux [74], for minimally instrumenting producer and observer codes at locations where i.o. data are ready for observation (produced) and observation is desired (consumed). Note that the producer is responsible for sharing any metadata necessary for the observer to make sense of the data, and that the COW overhead scales as the product of the observer count with the producer-modified page count.

Overall, TCASM preserves the asynchronous aspect of a multi-buffer approach while minimizing memory and synchronization requirements by sharing memory pages that contain unmodified data and allowing the operating system to manage COW data at memory page sized granularity, and supports a one-to-many producer-to-observer relationship, with loose coupling between producer and observer manifested solely by the shared data and allowing the producer to progress continuously without coordinating with the observer. TCASM is being integrated into the Hobbes [82] exascale operating system as part of Hobbes' composite application support.

4.2.6. Henson. Simulations produce snapshots sized on the order of tens of terabytes, and there is not enough space to store them when the simulation produces many of them. Furthermore, saving the data to disk for subsequent analysis is inefficient due to I/O costs. A common approach is to run the simulation and analysis processes in situ by closely integrating them together, thus requiring the effort of modifying the execution regime of one or both processes [75]. Henson implements a reduced effort design for running multiple codes in situ, synchronously or asynchronously, enabling cooperative multitasking between simulation and analysis and allowing processing on-the-fly and post-processing, via using coroutines and position-independent executables.

Three parts compose Henson: A *controller* application (like a puppeteer) for controlling execution flow between the simulation and analysis codes (called *puppets*), a library in C to which the puppets must link, and auxiliary tools for transitioning an in situ analysis approach from in line to in transit. Coroutines generalize subroutines [83] and maintain their state across invocations, which is necessary for switching execution between the puppets. Position-independent executables (PIEs) allow proper execution at any memory address without modification, meaning that a PIE can be loaded within another process, meaning that compiling the puppets as PIEs allows them to be loaded within the puppeteer, meaning that the puppets share the same address space and can exchange zero-copy data simply by passing memory addresses. The advantage of this approach is that simulation and analysis processes need not be aware of what is occurring in each other, and can simply post and retrieve pointers to data in a shared table when desired.

4.2.7. Synthesis. All of the Middleware systems focus on reducing the need to re-instrument software for every new software composition. Each system has specific strengths which may be useful when deciding which system to use.

Decaf is the only dataflow library system, with flow control and data contracts, and is a sensible choice for composing programs that operate in a dataflow-like fashion.

ADIOS uniquely allows for instrumenting software once and composing the software dynamically, meaning that a composed program (producer/consumer) can be replaced during runtime. Unlike Henson, ADIOS is more resilient because composed programs do not run in the same process.

Freeprocessing is the only system that does not require instrumenting one or more participating programs in a software composition, and is useful in the case where the "producer" software cannot easily be modified and rebuilt.

SENSEI is the only system that employs the VTK data model as an intermediate representation for connecting software data together, and is a sensible choice for the case where one or more software components already are using the VTK data model.

TCASM is the only system that allows concurrent read/write access to shared data by one or more composed "consumer" programs. (Freeprocessing allows copy-on-write for Python freeprocessors only, and only one composed "consumer.")

Henson uniquely follows a position-independent executable approach, meaning that instrumented programs can run standalone or as coroutines within a composing program's process. This allows pointer-based zero-copy data sharing (unlike ADIOS) and asynchronous operation between composed programs or program groups.

Finally, for all systems except TCASM and SENSEI, the composing application must be able to work with streaming (sequential) data where no metadata are included, as the data coming from the composed program are simply what would have been written to a file.

4.3. Discrete Processing Service

Discrete processing services (DPS) are software frameworks that provide a dedicated resource for post-processing (e.g. analysis or visualization) operations. In this model, simulations can use the discrete processing service to share data and invoke operations, and do so with minimal impact on the simulation's performance. The high-level motivators for DPSes are as follows:

- State of the art in situ methods encumber simulation code and do not support heterogeneity [84].
- In situ frameworks focus on capability and neglect ease of development, deployment, and maintenance [85].
- Simulations waste processor cycles that could be used for post-processing [86].
- In situ leads to performance variability [87].
- Blocking I/O delays sim execution [88].
- Simulations which use a coprocessing library often must be recompiled for each new coprocessing library version [80].
- In situ wastes transfer and compute resources, and in-transit resources cannot be time-shared [89].
- Asynchronous in situ simulation and post-processing leads to underused resources [90].

Some examples of systems answering these motivations: Ascent is an in situ library focused on minimizing execution time, memory usage, binary size, and integration effort for applications incorporating Ascent, and on providing diverse and powerful capabilities for modern supercomputers (§4.3.1). Catalyst is a simple in situ API allowing runtime switching between in situ software without recompilation (§4.3.2). Damaris reduces in situ performance variability by using dedicated cores for I/O and visualization (§4.3.3). DataStager stages data for I/O and post-processing on dedicated nodes (§4.3.4). GoldRush harvests idle cycles for in situ (§4.3.5). Libsim is a general-purpose coprocessing library with low developer burden and performance impact (§4.3.6). SERVIZ provides a many-to-one in-transit visualization service (§4.3.7). TINS implements a work-stealing task-based dynamic helper core strategy (§4.3.8). These systems are discussed in the next subsections.

4.3.1. Ascent. State of the art in situ methods encumber simulation code, do not provide integrated capability, and do not support heterogeneity [84]. Ascent is an in situ library focused on minimizing execution time, memory usage, binary size, and integration effort for applications incorporating Ascent, and on providing diverse and powerful capabilities for modern supercomputers. Ascent has an API for visualization and analysis that supports batch and interactive modes (via Jupyter notebooks), and it allows zero-copy shared memory data transfer between composed applications.

Ascent minimizes execution time and memory usage by incorporating VTK-m, which provides native support for many-core processors (e.g. multiple GPUs) and multi-core processors, and for zero-copy array layouts [91]. Ascent extends VTK-m's shared-memory parallelism with MPI-based distributed-memory parallelism, and reduces binary size and build complexity by requiring few linking dependencies viz. VTK-m and Conduit [92]. Ascent uses Conduit to simplify describing hierarchical data and in situ activities, and to support zero-copy data between composed programs.

Ascent includes support for producing Cinema and HDF5 files and for interfacing with ADIOS and Catalyst. It provides bindings for control by C, C++, Python, and Fortran applications, by an API including dynamically scanned YAML files, or by Jupyter notebook integration. Most importantly for program composition, Ascent can incorporate arbitrary code such as C/C++ code and Python scripts.

Ascent organizes its capabilities into abstractions for transforming data, rendering images, extracting data, inspecting data, and triggering execution. These abstractions can interact and can perform concurrently in multiples. For example, Ascent can (during an iteration cycle) transform data D to data D', then inspect D' and compare it with D, with a trigger on the comparison result so that if some threshold is met then Ascent will execute an image rendering operation while exporting D to disk. Examples of Ascent transformations (called *filters*) are histogram, gradient, and sampling, and transformations can be composed in series. Image renders (called *scenes*) can be produced from simulation data or transformed data, and can be saved to disk. Note that image rendering is not a transformation, so its output cannot be the input to a transformation. Extracted data can go to files (like HDF5), to ParaView Catalyst [79] for analysis and visualization, or to ADIOS [71] for in-transit processing. Ascent includes extractions for running user-provided code in Python, and for incoming Jupyter notebooks via a web browser. Inspections (called *queries*) can be performed on simulation data, or on the results of one or more transformations. Example inspections include getting the simulation cycle count and getting the maximum value of a data field. Inspection results are stored and can be used for subsequent inspections. Triggers address the problem of choosing a cycle of a simulation for performing analysis or visualization, such as for the purpose of balancing simulation time with analysis or visualization time. Triggers comprise a condition and an action, where conditions can incorporate inspections and actions can execute transformations, renders, extractions, and further inspections. Example conditions are whether the simulation has reached a particular cycle count, or a data field has met a threshold value.

Ascent supports three kinds of interactions between its capabilities: Producer-consumer, parameter update, and action generation. Producer-consumer allows the output of a computation to be used as the input for another computation on a one-to-one, one-to-many, many-to-one, or many-to-many relationship. Parameter update allows inspection results to set parameters for transformations, renders, and extractions. Action generation allows triggers to execute their associated conditional computations.

Ascent incorporates the following technologies for its API: Conduit, a library for describing and sharing hierarchical scientific in-memory data; Mesh Blueprint, an application-level data model interface; and a control interface for instructing Ascent on carrying out its capabilities. Conduit structures data as hierarchical key-value pairs. Mesh Blueprint supports representing coordinate sets, topologies, fields, and domain decomposition information.

Ascent's control interface comprises the calls open (initialize Ascent), publish (pass data to Ascent), execute (invoke Ascent capabilities), info (retrieve data from Ascent), and close (finalize Ascent).

4.3.2. Catalyst. A wide array of libraries and frameworks are available for in situ data anlysis, targeting different use cases or environments, and many require instrumenting simulation code to pass and convert data structures into a format compatible with coprocessing software [79]. Moreover, many frameworks focus on capability and neglect ease of development, deployment, and maintenance [85]. The Catalyst API is an easy-to-use in situ API that allows runtime switching between in situ software without recompilation.

Catalyst originally involved developing an adaptor that converted simulation structures to VTK data model objects, and required the ParaView SDK to build. Developing the adaptor required deep understanding of the VTK data model, whose APIs have varying implications for memory overhead and performance. The VTK data model has also continually evolved, forcing adaptors to be continually updated. The adaptor required a ParaView SDK, which posed the challenge of building ParaView and its dependencies from source. The adaptor's build system also needed to be updated to reflect changes in ParaView's build system, hence the simulation has a transitive dependence on ParaView via the adaptor. It was not easy to switch between different adaptor or ParaView versions, which made regression testing difficult. These challenges apply in varying degrees to in situ frameworks in general [85].

The Catalyst API is a separate project from ParaView with no external dependencies [85]. It is an application binary interface (ABI) compatible, and stable, in situ API design that alleviates maintenance difficulties for production in situ analysis and visualization software. ABI-compatible means that a simulation built using any Catalyst API implementation can swap the implementation dynamically without rebuilding the simulation. Stable means that future API changes can continue to support earlier implementations.

The simulation development community has been the most active in developing adaptors, rather than the in situ development community [85]. Developing an adaptor for the previous version of Catalyst required a deep understanding of the VTK data model to make critical design choices when mapping simulation data structures to the model. This requirement was unreasonable and burdensome, and impeded adoption and ease of efficient performance. The Catalyst API adopts the Ascent [84] approach of employing Conduit [92], where adaptor developers simply must describe the simulation data structures instead of converting those structures to VTK data structures. The API comprises four functions: Initialize, Execute, Finalize, and About. Each function takes a `conduit_node` object that provides configuration parameters for Catalyst and the simulation meshes. Instrumenting a simulation involves populating the `conduit_node` and invoking the functions appropriately. Developers provide the simulation data description, and the ParaView-Catalyst library handles converting simulation data to VTK data objects. Once the simulation is instrumented, Catalyst can be used to dispatch analysis *pipelines* in situ with the simulation [79]. Analysis pipeline customization is supported via Python Scripting, and users can write scripts by hand or use the ParaView GUI to prototype pipelines and export them as Catalyst scripts. Examples of such Catalyst scripts represent pipelines producing images, computing statistical quantities, generating plots, or extracting derived information such as polygonal data or iso-surfaces for further post-processing.

Catalyst is an in situ API specification allowing runtime compatibility between Catalyst API implementations. This means that simulations compiled for one implementation can be executed with a different implementation. The ParaView SDK is no longer required for compiling instrumented simulation code, switching between versions of Catalyst API implementations does not require simulation recompilation, and future API changes can continue to support earlier implementations.

4.3.3. Damaris. High performance computing simulation performance varies during I/O-intensive phases of execution, and this variation leads to unpredictable overall application performance [87]. The performance variation (I/O jitter) stems from the simulation's I/O pattern and contention for the network and file system, and is exacerbated by contention between multiple processor cores and concurrent compute jobs. Damaris is a framework which handles all aspects of data management in HPC simulations, with the goal of completely hiding I/O jitter exhibited by HPC simulations by dedicating a core and shared memory buffer per multicore SMP node to perform asynchronous data processing and I/O. Two traditional approaches to large-scale HPC I/O are writing one file per process, which is not scalable and delegates I/O jitter mitigation to the file system, and using coordinated collective I/O to a shared file, which imposes additional process synchronization and to date does not support file compression.

Damaris comprises a set of MPI processes, one per compute node of the simulation, and each process accesses a shared memory buffer for performing post-processing, filtering, indexing, and I/O, all in response to events as notified by the simulation. The shared memory buffer is used for data only, while metadata is stored in an external configuration file on the file system, to avoid overburdening the buffer. Post-processing can be compression, statistical calculations, or any user-provided transformation, by which Damaris can serve as the composer between the simulation and any other desired application. The event queue is used for write notifications, or user-defined events, by the simulation or external tools. As Damaris operates asynchronously, the I/O cost to the simulation appears to be the cost of a single memory copy from the simulation's memory to the shared memory buffer. Damaris also supports trading the shared memory buffer for a zero-copy approach with additional coordination by the simulation, which avails the Damaris approach even to highly memory constrained situations.

4.3.4. DataStager. High performance applications incur substantial I/O costs during checkpointing and due to undesirable performance variations during I/O activities (referred to as I/O jitter in §4.3.3), and these costs become untenable as applications scale to tens of thousands of processing cores [88]. DataStager is a high performance asynchronous data transport layer and staging service which extracts data from compute nodes. Instead of blocking and waiting for data output to complete, DataStager allows the application to overlap data extraction with computation, resulting in lower overhead for data output than POSIX I/O. In general, data staging services leverage RDMA-based network infrastructures for shifting the burden of synchronization, aggregation, collective processing, and data validation to a staging area subset of the compute partition. Data processing in the staging area can be used for formatting data storage on disk, for moving data to secondary or remote machines, or for visualization. DataStager uses the Fast Flexible Serialization (FFS) data format library [93] for casting data output into a self-describing binary form, as opposed to serving as a simple byte transfer layer from compute to disk storage (see [94] for details on the Portable Binary Input/Output format, PBIO, which was a predecessor of FFS). This means that data staging services can be customized to the data type produced by the simulation (e.g. particle, mesh), attained through dynamic code generation for decoding and manipulating data. The main focus of DataStager is improving I/O performance and reducing overheads, using meaningful data object movement (as opposed to raw byte-transfer) and application phase aware movement scheduling to reduce perturbation of application performance.

Asynchronous I/O can outperform synchronous I/O by up to a factor of two [95]. DataStager comprises the DataTap client library, which is co-located with the compute application and minimizes application level code changes by interfacing with the ADIOS API [96], and the DataStager parallel processes (DataStagers), which comprises additional compute nodes providing the data staging service to the application. DataTap ensures low runtime impact by copying data into fixed, limited-size, and minimally-bookkept compute node buffers, and provides flexibility in I/O by using FFS, which is a self-describing, marked-up binary data format allowing in transit inspection, modification, and post-processing. As an asynchronous I/O system, DataStager requires sufficient local memory space to buffer the output data. The DataStagers control data transfer scheduling by operating as a request-read service: An application issues a request which is then enqueued, and the DataStagers use resource aware schedulers to select a request from the queue. After selecting a request, the DataStagers issue an RDMA read request to the originating application node and call the handler for the data for processing (such as writing to disk, forwarding across the network, or further processing). Multiple requests may be overlapped.

4.3.5. GoldRush. Large scale scientific simulations imposing data pressure on I/O and storage subsystems is a severe performance bottleneck for simulations and data post-processing, leading scientific applications to reduce on-machine data movement and disk I/O volume by using in situ data analytics where simulation output data is processed as it is generated, at the cost of substantial unused compute resources during sequential simulation phases which aggregate up to 65 percent of total execution time [86]. GoldRush is a lightweight runtime system which improves the efficiency of running in situ data analytics without perturbing simulations, by co-locating simulation and analytic computations on effectively over-subscribed compute nodes and using low-overhead monitoring to identify and employ simulation-focused idle node resources for running in situ data analytics while continuously mitigating resource contention by the analytics on the simulation. The GoldRush method leverages the FlexIO transport in ADIOS [96] to map suitably-sized portions of the in situ analytics to idle node resources, to analytics-dedicated node resources, or to post-processing tasks after the data have been moved to disk, while negotiating short idle periods in the simulation and contention for compute node resources between the simulation and analytic computations.

GoldRush exhibits fine-granularity operation during simulation execution by identifying, predicting, and selecting sufficiently lengthy idle periods during which to run analytics, and avoiding idle periods which are too short to offset context switching overheads. The analytics are completely suspended while cores are being used by the simulation, and GoldRush can detect contention on shared memory resources between the simulation and analytics and dynamically mitigate the contention by throttling the analytics execution rate. There is negligible runtime overhead (never exceeding 0.3 percent of total runtime with representative applications) for using GoldRush, and its methods are easily integrated into existing techniques. GoldRush complements existing in situ data analytics techniques by effectively co-locating simulation and analytics workloads, obviating dedicated compute node resources for analytics, and improving performance and saving costs at large scales.

The GoldRush runtime is instrumented in the simulation and analytics codes, and uses signaling to operate the analytics on-demand during sequential phases of the simulation's execution when the phases are predicted to be longer than a specified threshold. Each sequential simulation phase is instrumented by the user and is monitored for duration by GoldRush, which maintains duration runtime estimates using a simple heuristic (average duration of longest recorded sequential phase occurrence) for predicting the phase length. A shared memory buffer with performance information is used for assessing interference between the analytics and the simulation, and GoldRush throttles the analytics execution rate to limit interference. GoldRush leverages the placement flexibility offered by ADIOS and FlexIO for defining analytics pipelines to match available compute resources and for efficient shared-memory data movement.

4.3.6. Libsim. Supercomputers provide far more compute capacity than I/O bandwidth and suffer from an inherent I/O bottleneck. Applications unable to reduce their amount of written data must suffer a wait penalty while their data are written to disk. This means that scientists are less able to save data as they are generated, leading to increased risk of losing important scientific data. Post-processing applications also suffer a wait penalty as they load the simulation's output data.

A sensible solution is to reduce I/O by using extra cycles for combining simulations with in situ post-processing activities. Data analysis and visualization are two common post-processing steps, and both significantly reduce the size of the data down to sustainable quantities. For example, simulation data on the order of petabytes can be visualized as image data on the order of megabytes. This means that if the simulation data takes thousands of seconds to write to disk, then the image would take fewer than ten seconds.

Two categories for in-memory simulation and post-processing coupling are as follows: Decoupling I/O from the simulation by staging (e.g. ADIOS [71]), and coupling general coprocessing libraries (e.g. Catalyst [85]). Libsim resembles the latter category because coprocessing libraries have direct access to simulation data and share compute resources, which is the case for Libsim. But Libsim supports read-on-demand for any quantity exposed by the simulation.

Libsim uses message-passing (MPI) and supports distributed memory simulations. The goal is an interactive visualization and analysis system. Libsim's philosophy is to maximize features for general-purpose program composition, minimize burden on simulation developers, minimize performance impact on simulations, and support on-demand in situ capability with unperturbed off-demand simulation performance.

Libsim uses VisIt [78] as the in situ analysis and visualization software for interfacing with simulations. Libsim interfaces with VisIt clients by assuming the role of a parallel VisIt server. The Libsim in situ analysis organization is similar to a normal VisIt analysis organization, except that the simulation and the VisIt server are one and the same process. The simulation polls for VisIt client connections, and upon a connection, the simulation loads the VisIt server for completing the connection. The VisIt server solicits a description of meshes and data from the simulation, and handles any VisIt-specific operations on the data.

The Libsim library differs from VisIt plug-ins which read files from disk. Instead, Libsim provides a plug-in with data access callback functions to read data from a currently running simulation. The plug-in typically passes simulation memory data pointers. For some simulations, coordinate transformations or data gathering may be required.

Libsim comprises two pieces: a small and lightweight *front-end* static library, and a heavyweight *runtime* library. The front-end library allows minimal simulation modifications for supporting Libsim, and allows the runtime library to incur zero performance impact when not in use. It also allows the runtime library to change freely as VisIt evolves, benefiting the simulation without requiring the simulation to be recompiled.

Utilizing Libsim requires initializing the library, writing data access callback functions, and adding simulation-steering functions. During initialization, the simulation creates a small data file with host and port information for VisIt. Simulation process zero (e.g. MPI rank-0) creates a listening socket for inbound VisIt connections, which can use timeout and polling-based event handling. After connection, process zero receives VisIt commands, and broadcasts the commands to other processes. Data access callback functions are written in C or Fortran and registered with the runtime library. The runtime library calls the functions on demand when gathering inputs for a data flow network, which ensures no resources are wasted in exposing data that will not be used. The callbacks themselves invoke library functions for creating data objects with opaque handles to simulation metadata, meshes, and variable objects. The runtime library receives the handles and transforms the data into VTK objects that can be used inside of VisIt. If the data use a layout that is incompatible with the VTK data model, then the callbacks can create VisIt-owned, automatically-disposed temporary storage. Else, the data are simulation-owned and read-only to VisIt. Mesh metadata include partitioning information as used by the simulation for parallelization and load balancing. VisIt's load balancer uses the partition information for restricting work on data to processes that own said data. Steering functions can be used in the case where simulations allow command-line steering as they execute. In this case, Libsim can add corresponding command buttons to the VisIt graphical user interface. The end result is a fully-featured parallel visualization and analysis tool that enables in situ computations within simulations by leveraging the capabilities of VisIt, while minimizing the performance impact on the simulation and the amount of new code that must be written.

4.3.7. SERVIZ. In transit visualization for HPC simulations is a form of in situ visualization that avoids the storage overhead of the simulation writing data to the file system, avoids blocking the simulation while the visualization program processes the data, and obviates running the visualization at the same level of concurrency as the simulation, by allowing the visualization program to run on dedicated resources, but incurs data transfer and resource allocation overheads [89]. SERVIZ is a shared service–based approach which allows multiple simulations to connect simultaneously to an in transit dedicated visualization resource, and supports longer phases between data transfer for a given simulation, overall providing an amortization of the data transfer and resource allocation overheads. This approach is based on the notion that a cost model shows that running an in transit application (visualization) at a lower level of concurrency relative to the simulation can improve the cost efficiency of the visualization [6], and that maximizing utilization of the service has a better chance of achieving cost savings with an in transit implementation over an in line implementation.

SERVIZ is implemented as a Mochi [97] microservice and consists of a client library and a server library. The client library exposes a remote procedure call (RPC) API to be used by the HPC simulation, and the server library RPC API is implemented as member functions within a service provider (an object which can receive RPC calls). The RPC client API is implemented to be similar to the Ascent API [84]. The execution model for SERVIZ comprises one or more MPI processes, one for each desired instance of a service provider, and each service provider's RPC address is registered in a well-known file location. Each simulation is granted access to a specific instance of a service provider, and the simulation creates a SERVIZ client object for access to the service provider's RPC API. When the simulation is ready to invoke the visualization, MPI rank zero gathers data from all other ranks and invokes the SERVIZ API to make the RPC, which occurs asynchronously. SERVIZ also supports traditional in line, in situ operation, and same node, shared memory operation, both of which are useful when shared memory operation is desired or dedicated compute nodes for SERVIZ are not desired. Finally, SERVIZ supports *immediate* and *delayed* modes for processing incoming requests; the immediate mode dispatches requests as soon as they are received, and the delayed mode queues incoming requests before dispatching the requests. The motivation for these modes are that while RPCs are asynchronous between the client and the provider instance, the instance will block other clients while dispatching a request; using the delayed mode allows the instance to batch requests from clients and thus reduce blocking in the case of multiple clients.

4.3.8. TINS.

The exascale-era gap between computational capabilities and I/O bandwidth calls for non-traditional data processing methods [90]. Traditional approaches involve outputting simulation data to the file system and reading the data into a post-processing application.

The in situ paradigm addresses the performance gap between compute and I/O by co-locating simulation and post-processing on the same compute node, operating on the same resident memory. Post-processing synchronously with the simulation is the simplest approach, while asynchronously offers better performance at the expense of underused resources.

TINS is a work-stealing task-based in situ framework implementing a novel dynamic helper core strategy. Simulation and post-processing tasks run concurrently and helper cores are assigned to post-processing when such tasks are available. If no post-processing tasks are available, then the helper cores are assigned to the simulation tasks for better resource usage. The Intel Threading Building Blocks (TBB) library [98] provides the task-based programming model and work-stealing scheduler. Up to 40% performance improvement is possible with TINS over other approaches including post-processing–dedicated helper cores.

In situ processing has an intuitive strategy where the simulation time loop issues a blocking call to the post-processing library; This is called *synchronous in situ* [90]. The post-processing library can directly access to the simulation data in memory (i.e. zero-copy) but in general a copy is required for data format adaptation. Examples of this approach are Catalyst [85] and Libsim [80], which are not compatible with one another, and SENSEI [73], which provides an intermediate representation for either of these (or e.g. ADIOS [71]).

Parallel simulations are not 100% efficient, as processor cores typically become idle during communication phases or inherently sequential phases of a computation [90]. Rather than idling, these cores can be used for post-processing tasks; This is called *asynchronous in situ* [90]. One approach to this strategy is in using the operating system to schedule the simulation and post-processing tasks as separate programs, but it has been shown that post-processing tasks disturb the simulation [86], [99].

Another approach is to dedicate one or more *helper cores* to the local post-processing tasks. This approach has examples in Damaris [87], FlowVR [100], functional partitioning [101], and GePSeA [102]. Performance is improved over the synchronous approach, but the simulation cores nevertheless become idle as usual. This is because the post-processing tasks run exclusively on the dedicated helper cores and are prevented from using idle simulation cores, and vice versa.

A third approach is to detect sequential computation phases in the simulation for the purpose of scheduling post-processing tasks during those phases. This is the approach taken by GoldRush [86]. The simulation resumes the post-processing tasks when the sequential phase starts, and suspends them when the phase stops. Simulation performance was shown to be improved above the operating system approach. But GoldRush does not allow asynchronous scheduling during brief sequential phases or weakly scalable parallel phases [90].

Task-based programming involves denoting potential parallelism through tasks or loops, which allows a runtime to create and distribute work units between runtime-created worker threads. See Cilk [103], Intel TBB [98], and OpenMP [104] for examples. In *work stealing scheduling*, a thread is assigned a share of work units and can attempt to steal work units from another thread after completing its own share [105]. This form of scheduling has exhibited good performance.

TINS relies on the TBB work-stealing scheduler for implementing in situ processing with tasks. Simulation and post-processing tasks are created concurrently and scheduled by a single instance of TBB. By using MPI, an instance of TBB is run on each MPI process. Each instance has a *simulation main thread* and a *post-processing main thread* (analytics in the following), and a number of worker threads spawned by TBB. The two main threads run with different concurrency levels, and both create tasks pertaining to their respective purpose (simulation tasks, and analytics tasks). An *analytics breakpoint frequency* sets a synchronization point where the simulation main thread copies data into a temporary buffer and notifies the analytics main thread when ready. The analytics main thread waits until the ready signal, then creates analytics tasks for the temporary buffer data. Only one buffer is required, but more can be used to reduce synchronization between the simulation and analytics.

Both the standard *static* helper core strategy and the dynamic helper core strategy are available under the described approach. In the static strategy, the available threads are partitioned into simulation and analytics threads with permanent isolation. That is, a simulation or analytics thread will remain idle if no respective tasks are available. With the dynamic strategy, the main difference is the use of temporary isolation for the simulation threads: If no simulation tasks are available for a simulation thread, that thread can migrate to the analytics partition and assist with performing analytics tasks. Similarly for analytics tasks and threads. The number of threads per partition is chosen to reflect the relative workloads of the simulation and analytics, without oversubscribing the processor cores.

Simulation and analytics code are kept separate via a *plugin* system, where a plugin is code compiled as a shared library. A plugin must be developed using MPI and TBB, and must take an MPI communicator as input. The MPI communicator is used for keeping simulation and analytics messages from becoming intermixed. Finally, the temporary buffer described previously must be created as a shared data structure, for use by both the simulation and the analytics.

4.3.9. Synthesis. All of the discrete processing services shown here are capable of invoking computational resources for post-processing tasks or composed programs. The computational resource can be part of the process image (Ascent, Catalyst, Libsim), part of the process schedule (GoldRush), or part of the job hardware allocation (Damaris, DataStager, SERVIZ, TINS). Each system also implies favor towards the performance of a distinguished primary program, which then invokes post-processing tasks or composed programs on lesser compute resources. The choice of which system to use depends on the software being composed and their usage, and the following descriptions may help in identifying which system to choose.

Ascent uniquely provides a holistic approach to discrete processing services, including data filtering, rendering, and inspecting. Ascent is a good choice for when a Workflow Manager–like system is desired but maximal performance and scalability on heterogeneous processor organizations is also required.

Catalyst is the only system claiming to be runtime-compatible between Catalyst versions without recompilation. This is due to Catalyst's primary focus on easing development, maintenance, and deployment of in situ software. Its API uses only four simple functions, and it could nearly be considered Middleware except that its usage centers on instrumenting simulations for invoking post-processing pipelines, similarly to other systems in the Discrete Processing Service realm.

Damaris uniquely leverages a dedicated core per compute node for post-processing tasks or composed programs. This arrangement would be most appropriate when the post-processing tasks are distributed and require a fraction of the main computation time.

DataStager is the only system focused on masking performance-perturbing I/O activity (such as simulation checkpointing) by staging data for file writing or post-processing on one or more dedicated nodes. As an in transit system, DataStager will be most appropriate when composing programs which do not already saturate the interconnect or memory systems, as both will be used in the process of staging.

GoldRush uniquely focuses on leveraging unused simulation processor cycles for composed programs. This makes GoldRush a compelling system in the case where a simulation experiences relatively lengthy sequential phases.

Libsim is the only system claiming to have zero performance impact on a simulation when the Libsim in situ capabilities are not in use. This would be useful in the case where a simulation is sometimes but not always desired to be composed with additional software, and performance is paramount.

SERVIZ is uniquely intended to run on a dedicated node and offering post-processing services to one or more client programs. This is similar to DataStager, except that DataStager is intended to run on one or more dedicated nodes which collectively offer post-processing services to one application.

TINS is the only system that takes a work-stealing task-based approach to dedicating processor cores to composed programs, where dedicated cores can be used by the simulation when no post-processing tasks are currently running. This system is best suited to the case when two composed programs have variable computation phases, and a guaranteed minimum amount of post-processing or composed program time is required.

All of the described systems are focused on prioritizing the simulation and invoking the post-processing tasks on secondary resources. Which system to use depends mostly on what resources are available or desired for the post-processing tasks.

4.4. Distributed Data Services

A Distributed Data Service (DDS) is a software framework that allows data objects to be shared between instrumented programs via files, interconnect, or shared memory. The primary reason for using a DDS in the context of this paper is when the computational coordination between programs is not as important as the data shared between them. Each system in this section focuses on addressing a specific issue that arises when sharing data between composed programs: Data representation diverseness imposes a challenge when connecting data between codes, particularly when the data are in-memory and cannot be duplicated (zero-copy) [92]. Sharing data also presents a challenge when the codes are running on distinct resources [106] or on heterogeneous storage and interconnect technology [97].

The remainder of this section surveys three noteworthy examples of Distributed Data Services, where each has a unique focus compared to the others: Conduit (§4.4.1) simplifies sharing data between software via a well-specified in-memory data representation. DataSpaces (§4.4.2) provides a dynamic distributed shared asynchronous virtual space abstraction. Mochi (§4.4.3) provides a methodology and tools for creating distributed data storage services.

4.4.1. Conduit. Flexible data representation and data coupling are essential for connecting modular HPC simulation tools. Conduit [92] provides an intuitive model for describing hierarchical scientific data, and is ideally suited to fill the need for flexible data representation and data coupling. Conduit is designed to have flexible and human-understandable in-memory representation, and supports zero-copy data sharing. Conduit provides a core data model including array representation and tree-based hierarchy support. Blueprint and Relay help applications share simulation mesh data by extending Conduit's data model support. Blueprint provides rules and transformations that apply to Conduit data for compatibility between composed programs. These transformations include conversions to the in-memory representation, and partitioning for distributed operation. Relay connects Conduit data with I/O libraries, including HDF5 and MPI. Relay HDF5 can be used for checkpoint-restart and file import or export. Relay MPI can be used for distributed-memory algorithms.

4.4.2. DataSpaces. HPC scientific and engineering applications comprise multiple heterogeneous and coupled processes that interact and often run independently on distinct and distributed resources. The interaction and coordination between the processes is dynamic and often asynchronous, and may vary during the overall computation. Modern workflows require support for coupling several such HPC applications together, which involves efficiently transporting, redistributing, and transforming data. Two basic approaches for providing such support are by existing parallel frameworks like MPI, or by using a file storage system as an intermediary between applications. These approaches can be unsatisfactory, as MPI requires careful coordination between sends and receives, and the storage system performance is often insufficient for the large size and low latency performance required by HPC applications. There is a need for a flexible interaction and coordination framework with high-level abstractions supporting the dynamic and asynchronous needs of data-intensive HPC applications. DataSpaces [106] provides a dynamic distributed shared asynchronous virtual space abstraction that can be associatively accessed by application components and services.

4.4.3. Mochi. Networking and storage technology have had recent advances which are useful for HPC applications. Data-intensive HPC applications read and write data at irregular intervals and are increasingly common. The data type and structure in these applications is often irregular with similarly irregular access patterns. Modern workflows incorporate multiple distinct data-intensive HPC applications, each with its own data needs and behaviors. There is a need for a data service foundation which facilitates efficient utilization of networking and storage technology while supporting heterogeneous data structures, rapid development, and ease of porting. Mochi [97] addresses this need by providing a methodology and tools for creating distributed data storage services.

4.4.4. Synthesis. Each Distributed Data Service discussed in this section provides an abstraction for sharing data and control between composed programs, and each supports transferring data via the network interconnect for distributed program composition. Nevertheless, there are unique foci for each respective system:

Conduit is unique amongst the systems discussed because it supports transferring zero-copy data, and because it is designed to support flexible and intuitive in-memory data representations.

DataSpaces focuses on simple, high-level abstractions for data-intensive program couplings. It provides a virtual shared space where distributed clients can connect, exchange data, and disconnect, all on-demand.

Mochi could uniquely be described as a "meta-system" because it is a system that can *serve as* and *create* distributed data services.

4.5. Others/Distinguishing Systems

Other systems are outside the scope of this area exam, and are provided here as for helpful context. These systems include the following:

- UMap [107], a user-level `mmap` (memory mapping) library supporting out-of-core execution and backing stores with varying characteristics,
- Metall [108], a persistent memory allocator,
- Hobbes [82], an HPC operating system focused on application composition,
- CGL-MapReduce [109], a file-avoiding MapReduce runtime,
- Melissa [110], a file-avoiding framework for global sensitivity analysis, and
- Ad-hoc system library–level approaches (e.g. `shm_open`, `mmap`) [7], [111], [112].

The Future of Scientific Workflows [1] and Common Compact Architecture (CCA) [113] may also provide helpful context.

5. Synthesis for Software Composition

See Tables 3, 4, and 5 for qualitative, implementation, and dependency details for the program composition systems. This section uses the shorthand notation of *simulation* (or simply "sim") and in situ program (or "in situ") to refer to distinct programs being composed together. The simulation is typically assumed to be a computationally intensive program which produces large volumes of data. The in situ program is assumed to be any program that is being composed together with the simulation, e.g. for visualization, analysis, or post-processing. Note that because the in situ program is being composed together with a data-producing simulation, it is assumed to be in a dependent relationship with the simulation. For this reason, *in situ* is also used in this section to refer to any software component that connects the simulation and in situ programs together. Discussion on each of the tables follows.

5.1. Qualities of Systems

Table 3 compares all the systems previously discussed qualitatively, with respect to their target audience, ease of use, developer time requirement, and data generality. In this context, *target audience* can refer to domain scientists, HPC simulation developers, in situ software developers, or in situ middleware developers. Each target is assumed to have some knowledge, but not expected to have expertise, in the fields of the other two targets. For example, domain scientists are not expected to have expertise in HPC software development, simulation developers are not expected to have expertise with in situ software development, and in situ software developers are not expected to have domain expertise. It can be seen from the table that few systems are targeted at domain scientists, which is due to the fact that most systems for software composition in HPC require the involvement of simulation or in situ developers. It can also be seen that most systems require involvement of both simulation and in situ developers; This is due to the close coordination generally required between the simulation and in situ programs, whether between computational tasks or information transfer. In the cases where a system targets simulation developers only, this means that the system views a simulation as a data producer and the onus is on the simulation for making its operation and data production easily available to an arbitrary composed program. On the other hand, a system specifically targeting in situ developers views the latter as having all of the responsibility of program composition, which can include modifications to both simulation and in situ program codes. The choice of system is easy for scientists due to the relatively fewer options, and simulation and in situ developers can gauge their relative involvement in order to identify their options.

Ease of use refers to the conceptual and concrete challenges in incorporating the system in question. Creating a script or workflow graph for running programs is assumed (conceptually and concretely) to be easier than refactoring existing software to use a new API. Developer time refers to which parties are most responsible for incorporating a system, where the responsible parties can be the simulation developer team, the in situ developer team, or both. It can be seen that most systems have moderate ease of use and involve refactoring software to use the API of a given system. Nevertheless, there are systems which are assumed to be easier to use. These systems require only reformulating a workflow as a script or a dataflow network, or using dynamic binary translation (DBT) to perform one part of the program composition automatically, for example. This quality is useful for identifying the ease of use, and the high-level means of use, of the various systems discussed.

Developer time refers to the design and implementation effort required for incorporating a system. The developers in question can be the simulation developers, the in situ program developers, or both. The implementation effort for a single developer team can be light, moderate, or heavy, referring to the relative design and implementation effort between systems bearing one of these designations. In the case of both simulation and in situ developers, the implementation effort is given as a respective ratio: 50/50 for equal effort shared between the simulation and in situ developers, or 20/80 representing less effort on the part of the simulation developers relative to the in situ developers. Recall that in the current context, *in situ* refers both to a program being composed with the simulation and to any additional code required for connecting the two programs. It is shown that most systems require equal developer time by both the simulation and in situ developers. This is to be expected, given the close coordination generally required between programs working together on HPC systems. Some systems nevertheless require less time (or no time) on the part of the simulation developers, which can be important in the case where the simulation developer team (or simulation source code) is unavailable. Note that this quality is distinct from *target audience* and *ease of use* because it refers specifically to design and implementation effort, and it is useful for assessing systems on that basis.

Data generality refers to the default information representation used by a system. The generality can be general, meshes, key-value pairs, or a combination of these. In this context, *general* means that the system supports common data types (e.g. arrays), *meshes* means that the system uses a mesh-based data representation, and *key-value pairs* means the system uses a dictionary (abstract data type) representation. Nearly all of the systems support general data types, with some systems also supporting meshes (due to their foundations in scientific computing and visualization) or key-value pairs. This quality is useful when considering the formatting of the information used by composed programs and any transformations that may be required when sharing the information between the programs.

5.2. Details of Systems

Table 4 gives implementation details for each system previously discussed, including notable technologies leveraged by the system, whether or not the system requires file I/O, and the means by which the system performs data exchange. A concise *mnemonic* is also included for distinguishing each of the systems, and summarizes the previous discussion on the system in question. The *technologies leveraged* detail lists common HPC technologies, or in some cases foundational previous work, as used by a system. This detail is useful when considering the technologies currently in use by programs which are to be composed, or when exploring the foundations of a given system. The *requires file I/O* detail refers specifically to whether or not a system requires the use of file I/O for transferring information between composed programs. It does not mean that a system can or cannot use file I/O for transferring information, as file I/O for transferring information is generally ubiquitous. Instead, this detail is useful when considering the amount of data that is to be shared between composed programs, and to what extent file I/O is expected to be a bottleneck. For the case of a substantial file I/O bottleneck, there are systems which do not require file I/O for transferring information,

TABLE 3. PROGRAM COMPOSITION SYSTEM QUALITIES

This table provides an overview of the program composition systems and their qualitative characteristics.

System	Realm	Target audience	Ease of use	Developer time	Data generality
Swift	Workflow manager	Scientists.	easy (scripting)	in situ team, light	general
Pegasus	Workflow manager	Scientists.	easy (DAG for workflow)	in situ team, heavy	general
Kepler	Workflow manager	Scientists.	easy (dataflow)	in situ team, moderate	general
ADIOS	Middleware	Sim and in situ developers.	moderate (refactor for API)	sim/in situ 50/50	general
Decaf	Middleware	Sim and in situ developers.	easy (dataflow)	sim/in situ 50/50	general
Freeprocessing	Middleware	In situ developers.	easy (DBT and class)	in situ team, heavy	general
Henson	Middleware	In situ developers.	moderate (refactor for API)	in situ team, heavy	general
SENSEI	Middleware	In situ developers.	moderate (refactor for API)	sim/in situ 50/50	general, meshes
TCASM	Middleware	Sim and in situ developers.	easy (refactor for API)	sim/in situ 50/50	general
Ascent	Discrete processing service	Sim and in situ developers.	moderate (refactor for API)	sim/in situ 50/50	general, meshes
Catalyst	Discrete processing service	Sim and in situ developers.	moderate (refactor for API)	sim/in situ 50/50	general, meshes
Damaris	Discrete processing service	Sim and in situ developers.	moderate (refactor for API)	sim/in situ 50/50	general
DataStager	Discrete processing service	Sim developers.	moderate (refactor for API)	sim/in situ 20/80	general
GoldRush	Discrete processing service	Sim and/or in situ developers.	moderate (refactor for API)	sim/in situ 50/50	general
Libsim	Discrete processing service	Sim developers.	moderate (refactor for API)	sim/in situ 20/80	general, meshes
SERVIZ	Discrete processing service	Sim and in situ developers.	moderate (refactor for API)	sim/in situ 50/50	general
TINS	Discrete processing service	Sim and in situ developers.	moderate (refactor for API)	sim/in situ 50/50	general
Conduit	Distributed data service	Sim and in situ developers.	moderate (refactor for API)	sim/in situ 50/50	general, meshes
DataSpaces	Distributed data service	Sim and in situ developers.	moderate (refactor for API)	sim/in situ 50/50	key-value pairs
Mochi	Distributed data service	Sim and in situ developers.	moderate (refactor for API)	sim/in situ 50/50	general
System	Realm	Target audience	Ease of use	Developer time	Data generality

which reduce the likelihood of exacerbating the bottleneck and present the opportunity to mitigate the bottleneck.

5.3. Incidentals of Systems

Table 5 gives incidental information for each system previously discussed, including invasiveness and composition awareness. In this context, *invasiveness* refers to how a program (simulation or in situ) must be modified to use a given system. Most systems provide an API to be used within a program, and a library to be linked to the program during compilation. Nevertheless, there are systems which do not require any modification of the composed programs, and this feature may be important in the case where the programs or their compilation process cannot be modified. The *composition awareness* information refers to whether the simulation program, the in situ program, both programs, or neither program is to be modified during composition. In other words, if a program X must be modified in order to compose it with another program Y, then effectively X is composition-aware. Most systems require that both the simulation and in situ programs are composition-aware, but there are systems which require composition awareness for one or neither program. This information is similar to *invasiveness* but distinguishes between the simulation and in situ programs. It is useful when considering whether one, both, or none of the programs to be composed can be modified.

5.4. Summary

This section has provided a concise global comparison of the systems presented in this document and examples of use cases where a particular subset of systems would be more suitable. The comparison was presented in the form of Tables 3, 4, and 5, and the use cases were presented during the discussion of each of the tables. Each table is useful for assessing the features of a system at a glance, and each discussion is useful for more information and context on the features presented in the associated table. Ultimately, the information in this section assists in choosing a system for a desired program composition.

6. Motivational Discussion

Tensors are a popular form of data storage because they readily store values, value labels, and value relationships. This allows complete specification of interrelated (or unrelated) data within an arbitrary coordinate space. Decomposing sparse tensors can reveal latent information in data, hence sparse tensors and their decompositions are used in many scientific fields [3].

Sparse tensor non-zero values are irregularly distributed and not meant to be interpolated. If we wish to relate these values to the concept of meshes, then the values are most similar to vertices on *sparse* unstructured grids of arbitrary dimensionality. An example of a sparse tensor is the Amazon Reviews tensor, whose dimensions represent users, products, and words, and whose values are the number of times a given word appears in a review by a given user on a given product [14]. Intuitively, we can expect that not every user has reviewed every product with every word, hence we expect that the non-zero values are irregularly distributed. We can also expect that we cannot “fill in” missing values with an interpolation scheme, as that would imply that a user reviewed a product using particular words when they did not. By contrast, visual data are commonly represented by a dense (as opposed to sparse) grid of three or fewer

TABLE 4. PROGRAM COMPOSITION SYSTEM DETAILS

This table provides implementation details for the program composition systems.

System	Realm	Mnemonic	Technologies leveraged	Requires file I/O	Data exchange
Swift	Workflow manager	Workflow scripting.	Karajan	yes	files
Pegasus	Workflow manager	Workflow DAG.	HTCondor DAGMan	yes	files
Kepler	Workflow manager	Workflow actors.	Ptolemy II, Java	no	files/network
ADIOS	Middleware	I/O pipes.	HDF5, ZeroMQ, MPI	no	files/shared memory/interconnect
Decaf	Middleware	Workflow graph.	Bredala, MPI	no	interconnect
Freeprocessing	Middleware	DBT I/O pipes.	(self)	no	files (streams)
Henson	Middleware	Same-process PIEs.	MPI, PIC	no	same binary (zero-copy)
SENSEI	Middleware	In transit IR and API.	ADIOS, VTK data model	no	files/same binary/interconnect
TCASM	Middleware	COW shared memory.	mmap, msync	no	shared memory
Ascent	Discrete processing service	Workflow pipeline.	Conduit, VTK-m, Flow	no	same binary (zero-copy)
Catalyst	Discrete processing service	Dynamic in situ API.	VTK data model, ParaView, Conduit	no	same binary/interconnect (ParaView)
Damaris	Discrete processing service	In-transit cores/nodes.	MPI, Boost.Interprocess	no	shared memory/interconnect
DataStager	Discrete processing service	I/O outsourcing.	ADIOS, RDMA	no	interconnect
GoldRush	Discrete processing service	Idle cores for in situ.	ADIOS	no	files/shared memory/interconnect
Libsim	Discrete processing service	VisIt-centric steering.	MPI, VTK data model	no	same binary (zero-copy)
SERVIZ	Discrete processing service	In-transit vis. service.	Mochi, RDMA	no	same binary/shared memory/interconnect
TINS	Discrete processing service	Work-stealing cores.	Intel TBB, MPI	no	same binary (not–zero-copy)
Conduit	Distributed data service	Flexible data coupling.	MPI, HDF5	no	files/shared memory/interconnect
DataSpaces	Distributed data service	DHT PGAS.	DART, RDMA	no	interconnect
Mochi	Distributed data service	Dist. ADT PGAS.	RDMA, RPC	no	interconnect
System	Realm	Mnemonic	Technologies leveraged	Requires file I/O	Data exchange

TABLE 5. PROGRAM COMPOSITION SYSTEM INCIDENTALS

This table provides incidental information about the program composition systems.

System	Realm	Invasiveness	Composition awareness
Swift	Workflow manager	none (scripting)	Neither sim nor in situ.
Pegasus	Workflow manager	none (workflow DAG)	Neither sim nor in situ.
Kepler	Workflow manager	none (actor-based model)	Neither sim nor in situ.
ADIOS	Middleware	use API, link together	Both sim and in situ.
Decaf	Middleware	use API, link together	Both sim and in situ.
Freeprocessing	Middleware	use API, link together (in situ only)	In situ only.
Henson	Middleware	use API, link together	Both sim and in situ.
SENSEI	Middleware	use API, link together	Both sim and in situ.
TCASM	Middleware	use API, link together	Both sim and in situ.
Ascent	Discrete processing service	use API, link together	Both sim and in situ.
Catalyst	Discrete processing service	use API, link together	Both sim and in situ.
Damaris	Discrete processing service	use API, link together	Both sim and in situ.
DataStager	Discrete processing service	use API, link together	Both sim and in situ.
GoldRush	Discrete processing service	use API, link together	Both sim and in situ.
Libsim	Discrete processing service	use API, link together	Both sim and in situ.
SERVIZ	Discrete processing service	use API, link together	Both sim and in situ.
TINS	Discrete processing service	use API, link together	Both sim and in situ.
Conduit	Distributed data service	use API, link together	Both sim and in situ.
DataSpaces	Distributed data service	use API, link together	Both sim and in situ.
Mochi	Distributed data service	use API, link together	Both sim and in situ.
System	Realm	Invasiveness	Composition awareness

dimensions which can be structured or unstructured and amenable to interpolation, and hence visual data affords relatively more leniency in data organization [91], [114].

The rigidity of sparse tensor data representation eliminates from consideration many mesh-focused interoperability features that could otherwise be used for sharing data between applications, so there is a unique opportunity for unique solutions in the intersection of sparse tensors and interoperability in HPC.

Most visualization and analysis algorithms are pleasantly parallel in the sense that data can be partitioned and processed in any order without collective coordination [78]. This allows algorithms to lean on underlying infrastructure for managing partitioning and parallelism, meaning that the partitioning scheme used by a given interoperability system is unlikely to have a significant impact on algorithm performance. Conversely, sparse tensor calculations ranging from the MTTKRP [4], [16], [20], [115] to the CPD-ALS [116], CP-APR [117], GCP-SGD [61], [118], [119], Tucker decomposition [53], [120], and others [121], [122], [123] are highly sensitive to partitioning and require extensive communication [52], [62], [124]. The partitioning sensitivity stems from irregularly-distributed sparse tensor values, which leads to load imbalance; the communication volume is due to the MTTKRP's expand and fold communications, and increases with load imbalance as well as with sparse tensor size and dimensionality. This situation places more emphasis on the ability for algorithms to manage parallelism themselves, hence increasing variability between interoperating programs. While one cannot make blanket statements for either visualization or sparse tensors, this sensitivity and communicativeness contributes to a unique challenge of data organization in the intersection of sparse tensors and interoperability.

High performance sparse tensor libraries have stringent algorithmic, memory layout, and data distribution requirements. SparTen [117] uses the Kokkos performance portability library and yet contains two implementations depending on whether CPU or GPU operation is desired. GenTen-MPI [62] upholds specific sparse tensor distribution requirements that depend on implicit data (zero-valued elements) in addition to explicitly stored data (non-zeros) as a means for limiting communication overhead. SPLATT [29] loads and redistributes sparse tensors for balancing, prior to distributed computation. Many of these applications employ a variety of linear algebra operations as the building blocks for more complicated calculations, meaning that all the challenges associated with efficient linear algebra implementations are alive and well in the world of sparse tensor software.

There are many sparse tensor data formats, such as F-COO [125], LCO [23], COO [16], CSF [29], HiCOO [17], HaCOO [31], and ALTO [21]. Each format arose to meet a specific need, such as minimizing the in-memory footprint, easing the programmability, improving temporal locality, or a combination of these. F-COO supports highly-optimized implementations of sparse tensor computations on GPUs. LCO provides faster and more memory-efficient sorting performance for sparse tensors. COO is the most human-understandable format, and is similar in structure to Matrix Market [126] format. HiCOO improves the spatial and temporal locality of sparse tensor operations. The similarly-named HaCOO reduces the cost required for building an efficient blocked sparse tensor structure. ALTO improves workload balance and reduces synchronization overhead. Many data formats are mode-specific, meaning they exhibit good locality for a specific access pattern but poor locality for other access patterns, which gives rise to data duplication or novel approaches for mitigating the specificity to improve performance [18]. This variability in data formatting impedes interoperability in HPC by decreasing the chances that two applications use the same data format, thus necessitating additional resources for format transformation or duplication.

As mentioned previously, there are many decompositions (and algorithms for computing them): CP-ALS, CP-APR (MU, PDNR, PQNR), Tucker, and GCP-SGD. Each decomposition is suited to a particular assumption about data, such as that the data are normally distributed or Poisson distributed. There are also many system organizations: CPU (x86, ARM, POWER9), GPU (Nvidia, AMD, Intel), and FPGA. Each is suited to a particular kind of calculation, and each typically requires a particular style of programming (e.g. OpenMP [127], MPI [128], CUDA [129]) for effective utilization [130], [131], [132], [133], [134], [135], [136]. For example, applications written for CPU are usually in an arbitrary general purpose programming language, whereas applications for GPU typically require vendor-specific libraries for specific languages.

Finally, there are many programming languages: Python, C/C++, Chapel [137], [138], [139], Julia, and MATLAB are common examples. Each has its strengths, where some are typically used by scientists and some are used by application developers. For example: Python has Numpy, which is used by many scientists. C/C++ has high performance and is widely available. Julia is suited for numerical analysis. Chapel facilitates productivity. MATLAB features mathematical expressiveness.

Each format, decomposition, and language must be reimplemented as needed for a specific processor organization or decomposition algorithm. Implementing sparse tensor decompositions for HPC requires extensive effort towards performance and scalability, which presents a unique opportunity for interoperability due to the high degree of effort wasted in reimplementing a decomposition in a new language or on a new processor organization.

The most basic form of interoperability is to share data via the file system using an intermediate data representation [71]. While this approach is sensible in out-of-core scenarios [140], the file system is often too slow for HPC applications, or wastes memory storage or I/O bandwidth, leading to the in situ (in line and in transit) efforts described. In the case of sparse tensors and their decompositions, the file system overhead scales with dimensionality, the number of formats, and the number of decompositions. Current work shows that straightforward approaches to interoperability do not scale past the complexity of 1-D arrays for sophisticated decomposition algorithms [7]. In order to support interoperability more generally, we have to do something more advanced. Although this work focuses on sparse tensors and their decompositions, the goal is not about showing a result for sparse tensors in particular. It's about showing that interoperability across the most general data, hardware, and parallel applications has a solution.

7. Conclusion and Opportunities

This document seeks to answer the question of how do sparse tensors and sparse tensor decomposition affect program composition in HPC. The answer stems from the combination of two categories of challenges: Traditional challenges, and unique challenges. Traditional challenges arise from the multiple programming models and data layouts used for sparse tensors and sparse tensor decomposition software, and from the programming requirements and performance considerations associated with these. Addressing these traditional challenges is a current research area in the context of systems for software composition. Unique challenges for sparse tensors and sparse tensor decomposition arise from the general case where the distribution of nonzero elements along one dimension is much more dense than along another dimension. This situation leads to difficulty when efficient distributed data parallelism is desired (i.e., always), because state-of-the-art sparse tensor decomposition requires processing across each dimension. Addressing these unique challenges is a current research area in the context of sparse tensor decomposition. These two categories of challenges combine to create new challenges, and a rich area of research focused on whether program composition has a solution across the most general data, software, and hardware combinations.

References

[1] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[2] A. Malony, "Performance understanding and analysis for exascale data management workflows," Univ. of Oregon, Eugene, OR (United States), Tech. Rep., 2019.

[3] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[4] J. Choi, X. Liu, S. Smith, and T. Simon, "Blocking optimization techniques for sparse tensor computation," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 568–577.

[5] A. Nguyen, A. E. Helal, F. Checconi, J. Laukemann, J. J. Tithi, Y. Soh, T. Ranadive, F. Petrini, and J. W. Choi, "Efficient, out-of-memory sparse mttkrp on massively parallel architectures," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–13.

[6] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire, "Opportunities for cost savings with in-transit visualization," in *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds., Springer. Springer International Publishing, 2020, pp. 146–165.

[7] S. I. Geronimo Anderson and D. M. Dunlavy, "Computing sparse tensor decompositions via chapel and c++/mpi interoperability without intermediate i/o," 2023.

[8] S. Ramesh, "Evolution of hpc software development and accompanying changes in performance tools," University of Oregon, Computer and Information Sciences Department, Area Exam AREA-202105-Ramesh, 5 2021, available at https://www.cs.uoregon.edu/Reports/AREA-202105-Ramesh.pdf.

[9] M. Kothari and R. W. Vuduc, "An interface for multidimensional arrays in arkouda," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–2.

[10] V. De Silva and L.-H. Lim, "Tensor rank and the ill-posedness of the best low-rank approximation problem," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 3, pp. 1084–1127, 2008.

[11] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. Hruschka, and T. Mitchell, "Toward an architecture for never-ending language learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 24, no. 1, 2010, pp. 1306–1313.

[12] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 61–70.

[13] R. F. Boisvert, R. Pozo, and K. Remington, "The matrix market exchange formats: Initial design," National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. Technical ReportNISTIR 5935, December 1996.

[14] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[15] B. W. Bader and T. G. Kolda, "Algorithm 862: Matlab tensor classes for fast algorithm prototyping," *ACM Trans. Math. Softw.*, vol. 32, no. 4, p. 635–653, dec 2006. [Online]. Available: https://doi.org/10.1145/1186785.1186794

[16] ——, "Efficient matlab computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, 2008. [Online]. Available: https://doi.org/10.1137/060676489

[17] J. Li, J. Sun, and R. Vuduc, "Hicoo: Hierarchical storage of sparse tensors," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 238–252.

[18] I. Nisa, J. Li, A. Sukumaran-Rajam, P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–25.

[19] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on gpus," in *2017 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 2017, pp. 47–57.

[20] E. T. Phipps and T. G. Kolda, "Software for sparse tensor decomposition on emerging computing architectures," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C269–C290, 2019. [Online]. Available: https://doi.org/10.1137/18M1210691

[21] A. E. Helal, J. Laukemann, F. Checconi, J. J. Tithi, T. Ranadive, F. Petrini, and J. Choi, "Alto: Adaptive linearized storage of sparse tensors," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 404–416. [Online]. Available: https://doi.org/10.1145/3447818.3461703

[22] S. I. Geronimo Anderson, K. Teranishi, D. M. Dunlavy, and J. Choi, "Analyzing the performance portability of tensor decomposition," *arXiv preprint arXiv:2307.03276*, 2023.

[23] A. P. Harrison and D. Joseph, "High performance rearrangement and multiplication routines for sparse tensor arithmetic," *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. C258–C281, 2018.

[24] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for gpus," 2008, available at https://research.nvidia.com/sites/default/files/pubs/2008-12_Efficient-Parallel-Scan/nvr-2008-003.pdf.

[25] S. Yan, G. Long, and Y. Zhang, "Streamscan: fast scan algorithms for gpus without global barrier synchronization," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 229–238.

[26] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," 1966.

[27] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 233–244.

[28] R. E. Bank and C. C. Douglas, "Sparse matrix multiplication package (smmp)." *Adv. Comput. Math.*, vol. 1, no. 1, pp. 127–137, 1993.

[29] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, 2015, pp. 61–70. [Online]. Available: https://doi.org/10.1109/IPDPS.2015.27

[30] I. Nisa, J. Li, A. Sukumaran-Rajam, R. Vuduc, and P. Sadayappan, "Load-balanced sparse mttkrp on gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 123–133.

[31] R. Lowe, M. Charles, and A. Singh, "Hashed coordinate storage of sparse tensors," in *Research Poster, 2021 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'21)*, 2021.

[32] E. C. Chi and T. G. Kolda, "On tensors, sparsity, and nonnegative factorizations," *SIAM Journal on Matrix Analysis and Applications*, vol. 33, no. 4, pp. 1272–1299, 2012.

[33] E. Acar and B. Yener, "Unsupervised multiway data analysis: A literature survey," *IEEE transactions on knowledge and data engineering*, vol. 21, no. 1, pp. 6–20, 2008.

[34] G. Beylkin, J. Garcke, and M. J. Mohlenkamp, "Multivariate regression and machine learning with sums of separable functions," *SIAM Journal on Scientific Computing*, vol. 31, no. 3, pp. 1840–1857, 2009.

[35] S. Rendle, "Factorization machines," in *2010 IEEE International conference on data mining*. IEEE, 2010, pp. 995–1000.

[36] G. Beylkin and M. J. Mohlenkamp, "Numerical operator calculus in higher dimensions," *Proceedings of the National Academy of Sciences*, vol. 99, no. 16, pp. 10 246–10 251, 2002.

[37] ——, "Algorithms for numerical analysis in high dimensions," *SIAM Journal on Scientific Computing*, vol. 26, no. 6, pp. 2133–2159, 2005.

[38] W. Hackbusch and B. N. Khoromskij, "Tensor-product approximation to operators and functions in high dimensions," *Journal of Complexity*, vol. 23, no. 4-6, pp. 697–714, 2007.

[39] M. J. Reynolds, A. Doostan, and G. Beylkin, "Randomized alternating least squares for canonical tensor decompositions: Application to a pde with random data," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. A2634–A2664, 2016.

[40] D. Bruns-Smith, M. M. Baskaran, J. Ezick, T. Henretty, and R. Lethin, "Cyber security through multidimensional data decompositions," in *2016 Cybersecurity Symposium (CYBERSEC)*, 2016, pp. 59–67.

[41] H. Fanaee-T and J. Gama, "Tensor-based anomaly detection: An interdisciplinary survey," *Knowl. Based Syst.*, vol. 98, pp. 130–147, 2016. [Online]. Available: https://doi.org/10.1016/j.knosys.2016.01.027

[42] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[43] H. He, J. Henderson, and J. C. Ho, "Distributed tensor decomposition for large scale health analytics," in *The World Wide Web Conference*, ser. WWW '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 659–669. [Online]. Available: https://doi.org/10.1145/3308558.3313548

[44] J. C. Ho, J. Ghosh, and J. Sun, "Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 115–124. [Online]. Available: https://doi.org/10.1145/2623330.2623658

[45] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[46] R. A. Harshman, "Foundations of the parafac procedure: Models and conditions for an "explanatory" multimodal factor analysis," 1970.

[47] J. Håstad, "Tensor rank is np-complete," *Journal of Algorithms*, vol. 11, no. 4, pp. 644–654, 1990.

[48] T. G. Kolda, "Multilinear operators for higher-order decompositions." Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA ..., Tech. Rep., 2006.

[49] S. Hansen, T. Plantenga, and T. G. Kolda, "Newton-based optimization for Kullback-Leibler nonnegative tensor factorizations," *Optimization Methods and Software*, vol. 30, no. 5, pp. 1002–1029, April 2015.

[50] K. Teranishi, D. M. Dunlavy, J. M. Myers, and R. F. Barrett, "Sparten: Leveraging kokkos for on-node parallelism in a second-order method for fitting canonical polyadic tensor models to poisson data," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7.

[51] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[52] O. Kaya and B. Uçar, "High performance parallel algorithms for the tucker decomposition of sparse tensors," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 103–112.

[53] G. Ballard, A. Klinvex, and T. G. Kolda, "Tuckermpi: A parallel c++/mpi software package for large-scale data compression via the tucker tensor decomposition," *ACM Trans. Math. Softw.*, vol. 46, no. 2, jun 2020. [Online]. Available: https://doi.org/10.1145/3378445

[54] J. Choi, X. Liu, and V. Chakaravarthy, "High-performance dense tucker decomposition on gpu clusters," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 543–553.

[55] A. Karami, M. Yazdi, and G. Mercier, "Compression of hyperspectral images using discerete wavelet transform and tucker decomposition," *IEEE journal of selected topics in applied earth observations and remote sensing*, vol. 5, no. 2, pp. 444–450, 2012.

[56] R. Ballester-Ripoll and R. Pajarola, "Lossy volume compression using tucker truncation and thresholding," *The Visual Computer*, vol. 32, pp. 1433–1446, 2016.

[57] W. Austin, G. Ballard, and T. G. Kolda, "Parallel tensor compression for large-scale scientific data," in *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2016, pp. 912–922.

[58] I. Perros, R. Chen, R. Vuduc, and J. Sun, "Sparse hierarchical tucker factorization and its application to healthcare," in *2015 IEEE International Conference on Data Mining*. IEEE, 2015, pp. 943–948.

[59] J. Sun, D. Tao, and C. Faloutsos, "Beyond streams and graphs: dynamic tensor analysis," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 374–383.

[60] D. Hong, T. G. Kolda, and J. A. Duersch, "Generalized canonical polyadic tensor decomposition," *SIAM Review*, vol. 62, no. 1, pp. 133–163, 2020.

[61] T. G. Kolda and D. Hong, "Stochastic gradients for large-scale tensor decomposition," *SIAM Journal on Mathematics of Data Science*, vol. 2, no. 4, pp. 1066–1095, jan 2020.

[62] K. D. Devine and G. Ballard, "Gentenmpi: Distributed memory sparse tensor decomposition," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2020.

[63] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[64] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X14002015

[65] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.* IEEE, 2004, pp. 423–424.

[66] G. Von Laszewski, M. Hategan, and D. Kodeboyina, "Java cog kit workflow," *Workflows for e-Science: Scientific Workflows for Grids*, pp. 340–356, 2007.

[67] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience," *Concurrency and computation: practice and experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[68] J. Eker, C. Fong, J. W. Janneck, and J. Liu, "Design and simulation of heterogeneous control systems using ptolemy ii," *IFAC Proceedings Volumes*, vol. 34, no. 22, pp. 271–276, 2001.

[69] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and computation: Practice and experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[70] O. Yildiz, M. Dreher, and T. Peterka, "Decaf: Decoupled dataflows for in situ workflows," in *In Situ Visualization for Computational Science*. Springer, 2022, pp. 137–158.

[71] D. Pugmire, N. Podhorszki, S. Klasky, M. Wolf, J. Kress, M. Kim, N. Thompson, J. Logan, R. Wang, K. Mehta *et al.*, "The adaptable io system (adios)," in *In Situ Visualization for Computational Science*. Springer, 2022, pp. 233–254.

[72] T. Fogal, F. Proch, A. Schiewe, O. Hasemann, A. Kempf, and J. Krüger, "Freeprocessing: Transparent in situ visualization via data interception," *Eurographics Symposium on Parallel Graphics and Visualization : EG PGV : [proceedings]. Eurographics Symposium on Parallel Graphics and Visualization*, vol. 2014, p. 49—56, 2014. [Online]. Available: https://europepmc.org/articles/PMC4435933

[73] E. W. Bethel, B. Loring, U. Ayachit, D. Camp, E. P. Duque, N. Ferrier, J. Insley, J. Gu, J. Kress, P. O'Leary *et al.*, "The sensei generic in situ interface: Tool and processing portability at scale," in *In Situ Visualization for Computational Science*. Springer, 2022, pp. 281–306.

[74] H. Akkan, L. Ionkov, and M. Lang, "Transparently consistent asynchronous shared memory," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi-org.libproxy.uoregon.edu/10.1145/2491661.2481431

[75] D. Morozov and Z. Lukic, "Master of puppets: Cooperative multitasking for in situ processing," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 285–288.

[76] M. Dreher and T. Peterka, "Bredala: Semantic data redistribution for in situ applications," in *2016 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, 2016, pp. 279–288.

[77] M. Dreher, K. Sasikumar, S. Sankaranarayanan, and T. Peterka, "Manala: a flexible flow control library for asynchronous task communication," in *2017 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, 2017, pp. 509–519.

[78] H. Childs, "Visit: An end-user tool for visualizing and analyzing very large data," 2012.

[79] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "Paraview catalyst: Enabling in situ data analysis and visualization," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2015, pp. 25–29.

[80] T. Kuhlen, R. Pajarola, and K. Zhou, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*, vol. 10. Eurographics Association Aire-la-Ville, Switzerland, 2011, pp. 101–109.

[81] D. Otstott, N. Evans, L. Ionkov, M. Zhao, and M. Lang, "Enabling composite applications through an asynchronous shared memory interface," in *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 219–224.

[82] R. Brightwell, R. Oldfield, A. B. Maccabe, and D. E. Bernholdt, "Hobbes: Composition and virtualization as the foundations of an extreme-scale os/r," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, 2013, pp. 1–8.

[83] D. E. Knuth, "The art of computer programming. vol. 1: Fundamental algorithms," *Reading*, 1978.

[84] M. Larsen, E. Brugger, H. Childs, and C. Harrison, "Ascent: A flyweight in situ library for exascale simulations," in *In Situ Visualization for Computational Science*. Springer, 2022, pp. 255–279.

[85] U. Ayachit, A. C. Bauer, B. Boeckel, B. Geveci, K. Moreland, P. O'Leary, and T. Osika, "Catalyst revised: rethinking the paraview in situ analysis and visualization api," in *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt am Main, Germany, June 24–July 2, 2021, Revised Selected Papers 36*. Springer, 2021, pp. 484–494.

[86] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2503210.2503279

[87] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o," in *2012 IEEE International Conference on Cluster Computing*. IEEE, 2012, pp. 155–163.

[88] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: scalable data staging services for petascale applications," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, 2009, pp. 39–48.

[89] S. Ramesh, H. Childs, and A. Malony, "Serviz: A shared in situ visualization service," in *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* IEEE Computer Society, 2022, pp. 277–290.

[90] E. Dirand, L. Colombet, and B. Raffin, "Tins: A task-based dynamic helper core strategy for in situ analytics," in *Supercomputing Frontiers: 4th Asian Conference, SCFA 2018, Singapore, March 26-29, 2018, Proceedings 4*. Springer, 2018, pp. 159–178.

[91] K. Moreland, "The vtk-m users' guide, version 2.0," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2023.

[92] C. Harrison, M. Larsen, B. S. Ryujin, A. Kunen, A. Capps, and J. Privitera, "Conduit: A successful strategy for describing and sharing data in situ," in *2022 IEEE/ACM International Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, 2022, pp. 1–6.

[93] G. Eisenhauer, "Ffs users guide and reference," 2011.

[94] F. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener, "Efficient wire formats for high performance computing," in *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 2000, pp. 39–39.

[95] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of leading hpc i/o performance using a scientific-application derived benchmark," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 1–12.

[96] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008, pp. 15–24.

[97] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, B. Robey, D. Robinson, B. Settlemyer, G. Shipman, S. Snyder, J. Soumagne, and Q. Zheng, "Mochi: Composing data services for high-performance computing environments," *J. Comput. Sci. Technol.*, vol. 35, no. 1, p. 121–144, jan 2020. [Online]. Available: https://doi.org/10.1007/s11390-020-9802-0

[98] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[99] T. Harris, M. Maas, and V. J. Marathe, "Callisto: Co-scheduling parallel runtime systems," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[100] M. Dreher and B. Raffin, "A flexible framework for asynchronous in situ and in transit analytics for scientific simulations," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2014, pp. 277–286.

[101] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman, "Functional partitioning to optimize end-to-end performance on many-core architectures," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–12.

[102] A. Singh, P. Balaji, and W.-c. Feng, "Gepsea: a general-purpose software acceleration framework for lightweight task offloading," in *2009 International Conference on Parallel Processing*. IEEE, 2009, pp. 261–268.

[103] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *ACM SigPlan Notices*, vol. 30, no. 8, pp. 207–216, 1995.

[104] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[105] Y. Cho, S. Oh, and B. Egger, "Adaptive space-shared scheduling for shared-memory parallel programs," in *Job Scheduling Strategies for Parallel Processing: 19th and 20th International Workshops, JSSPP 2015, Hyderabad, India, May 26, 2015 and JSSPP 2016, Chicago, IL, USA, May 27, 2016, Revised Selected Papers 19*. Springer, 2017, pp. 158–177.

[106] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework for coupled simulation workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 25–36.

[107] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale, "Umap: Enabling application-driven optimizations for page management," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019, pp. 71–78.

[108] I. B. Peng, M. B. Gokhale, K. Youssef, K. Iwabuchi, and R. Pearce, "Enabling scalable and extensible memory-mapped datastores in userspace," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 866–877, 2021.

[109] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *2008 IEEE Fourth International Conference on eScience*, 2008, pp. 277–284.

[110] T. Terraz, A. Ribes, Y. Fournier, B. Iooss, and B. Raffin, "Melissa: Large scale in transit sensitivity analysis avoiding intermediate files," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126922

[111] T. M. McCrary, K. D. Devine, and A. J. Younge, "Integrating pgas and mpi-based graph analysis," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 10 2021. [Online]. Available: https://www.osti.gov/biblio/1832085

[112] T. McCrary, K. Devine, and A. Younge, "Integrating chapel programs and mpi-based libraries for high-performance graph analysis." 6 2022. [Online]. Available: https://chapel-lang.org/CHIUW/2022/McCrary.pdf

[113] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. Epperly, M. Govindaraju *et al.*, "A component architecture for high-performance scientific computing," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 163–202, 2006.

[114] D. Ströter, A. Stork, and D. W. Fellner, "Massively parallel adaptive collapsing of edges for unstructured tetrahedral meshes," 2023.

[115] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–11.

[116] T. B. Rolinger, T. A. Simon, and C. D. Krieger, "Performance considerations for scalable parallel tensor decomposition," *J. Parallel Distributed Comput.*, vol. 129, pp. 83–98, 2019.

[117] K. Teranishi, D. M. Dunlavy, J. M. Myers, and R. F. Barrett, "Sparten: Leveraging kokkos for on-node parallelism in a second-order method for fitting canonical polyadic tensor models to poisson data," *IEEE High Performance Extreme Computing Conference*, vol. 0, no. 0, p. 0, 2020.

[118] D. Hong, T. G. Kolda, and J. A. Duersch, "Generalized Canonical Polyadic Tensor Decomposition," *SIAM Review*, vol. 62, no. 1, pp. 133–163, Jan. 2020.

[119] E. Phipps, N. Johnson, and T. G. Kolda, "Streaming Generalized Canonical Polyadic Tensor Decompositions," *arXiv:2110.14514 [cs, math]*, Oct. 2021. [Online]. Available: http://arxiv.org/abs/2110.14514

[120] Z. Fang, F. Qi, Y. Dong, Y. Zhang, and S. Feng, "Parallel tensor decomposition with distributed memory based on hierarchical singular value decomposition," *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a, p. e6656, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6656

[121] T. M. Ranadive and M. M. Baskaran, "An all–at–once cp decomposition method for count tensors," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–8.

[122] T. G. Kolda, B. W. Bader *et al.*, "Tensor toolbox for matlab," Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA ..., Tech. Rep., 2023. [Online]. Available: www.tensortoolbox.org

[123] S. Eswar, K. Hayashi, G. Ballard, R. Kannan, M. A. Matheson, and H. Park, "Planc: Parallel low-rank approximation with nonnegativity constraints," *ACM Trans. Math. Softw.*, vol. 47, no. 3, jun 2021. [Online]. Available: https://doi.org/10.1145/3432185

[124] S. Smith and G. Karypis, "A medium-grained algorithm for sparse tensor factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 902–911.

[125] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on gpus," in *2017 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 2017, pp. 47–57.

[126] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, "Matrix market: a web resource for test matrix collections," *Quality of Numerical Software: Assessment and Enhancement*, pp. 125–137, 1997.

[127] J. M. Diaz, S. Pophale, K. Friedline, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran, "Evaluating support for openmp offload features," in *International Conference on Parallel Processing Companion*, 2018.

[128] Message Passing Interface Forum, "Mpi: A message-passing interface standard," University of Tennessee, USA, Tech. Rep., 1994.

[129] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[130] "Portability across DOE Office of Science HPC facilities," https://performanceportability.org/, accessed: 2021-06-10.

[131] T. P. Straatsma, K. B. Antypas, and T. J. Williams, *Exascale Scientific Applications: Scalability and Performance Portability*, 1st ed. Chapman & Hall/CRC, 2017.

[132] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[133] K. Czechowski, "Diagnosing performance bottlenecks in hpc applications," Ph.D. dissertation, Georgia Institute of Technology, 2019.

[134] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 12 1995.

[135] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *Proc. ISC High Performance*, 2016, pp. 489–507.

[136] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *Proc. Extreme Scaling Workshop*, 2013, pp. 18–24.

[137] B. Chamberlain, S. Deitz, D. Iten, and S.-E. Choi, "Authoring user-defined domain maps in Chapel," in *Chapel User's Group 2011*. Cray Inc, 2011, pp. 1–6.

[138] B. Chamberlain, S.-E. Choi, S. Deitz, D. Iten, and V. Litvinov, "User-defined distributions and layouts in Chapel: Philosophy and framework," in *2nd USENIX Workshop on Hot Topics in Parallelism*. Cray Inc, 2010, pp. 1–12.

[139] T. B. Rolinger, T. A. Simon, and C. D. Krieger, "Parallel sparse tensor decomposition in chapel," *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2018.00143

[140] A. Nguyen, A. E. Helal, F. Checconi, J. Laukemann, J. J. Tithi, Y. Soh, T. Ranadive, F. Petrini, and J. W. Choi, "Efficient, out-of-memory sparse mttkrp on massively parallel architectures," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–13.