2024 IEEE International Conference on Big Data (BigData) | 979-8-3503-6248-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/BigData62323.2024.10825779

A Planner for Scalable Tensor Programs

Tanvir Ahmed Khan
Univ. of Texas at Arlington, Arlington, TX 76019, USA
tanvirahmed.khan@uta.edu

Leonidas Fegaras
Univ. of Texas at Arlington, Arlington, TX 76019, USA
fegaras@cse.uta.edu

***Abstract*—Current machine learning systems, such as TensorFlow and PyTorch, rely on high-performance linear algebra libraries for efficient tensor computations. Although they provide numerous fine-tuned array algorithms based on well-studied data placement and communication patterns, these libraries are hard to customize to capture irregular array programs and unconventional array storages. We present a framework for constructing distributed task workflows from ad-hoc tensor programs by partially evaluating these programs against the block coordinates of the tensors. In addition, we present a novel task scheduler based on pattern matching that assigns processes to tasks by recognizing certain patterns inside the task workflow. Although each such pattern applies to a small fixed number of tasks, when applied collectively, these patterns generate communication schemes that resemble optimal block-based algorithms, such as SUMMA. The tiling of the task workflow is based on pattern-matching and is done bottom-up, guided by cost.**

I. Introduction

Tensors and large-scale array computing are ubiquitous in scientific computing and machine learning. Current general-purpose big data analysis systems, such as Spark [27], are not suitable for large-scale array processing and cannot compete with distributed linear algebra libraries, such as ScaLAPACK [5], in terms of performance. These libraries on the other hand are too rigid since they support a limited number of array storages and fine-tuned algorithms based on well-studied data placement and communication patterns that are hard to customize. Irregular array programs are forced to be coded in terms of the supported array operators, which may result in suboptimal code when these programs do not completely match any of these operators. More importantly, the introduction of a new array storage would require adding many more operator implementations to handle operations that may involve multiple dissimilar array storages. Moreover, an array created by one operator may not match the input of subsequent operators, thus requiring unnecessary and expensive restructuring.

Consider for example the matrix-matrix multiplication between two distributed block matrices, each consisting of $N \times N$ blocks. A typical implementation of this operation in a synchronous distributed system, such as Spark, is with a join (to bring together and multiply related blocks from the two matrices), followed by a reduce-by-key (to group and add the blocks from the join). This join will generate N^3 intermediate blocks which can only be reclaimed after they have been used by reduce-by-key, that is, after they have been shuffled through the network. Furthermore, these operations are synchronous

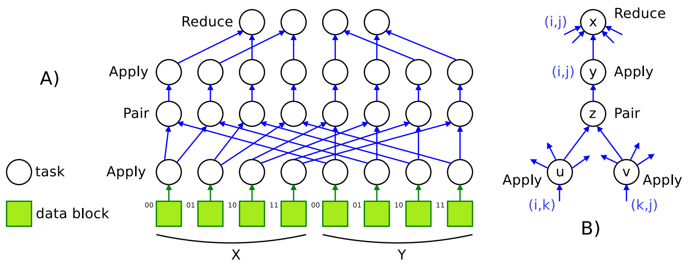


Fig. 1: A) The Task Workflow for 2×2 Block Matrix Multiplication $X \times Y$ and B) the Scheduling Pattern for GroupBy-Join

and need to wait for the slowest process to finish. This wait time can be long when data is skewed causing uneven data loading during shuffling. These problems can be addressed effectively by scheduling and executing tasks asynchronously, where each task processes one block at a time.

Our system, called *TensorPlanner*, builds on our earlier work on tensor comprehensions [6]–[8]. TensorPlanner is a customizable framework for large-scale distributed array programs, based on task scheduling and asynchronous execution. Unlike other systems, such as NumPy, that are implemented as light-weight wrappers around high-performance array libraries, our framework is flexible and customizable as it is capable of translating high-level programs on abstract arrays into efficient low-level computations on compact and efficient array storages, guided by user-defined storage mappings [8]. This separation between high-level and low-level specifications makes our system independent of the limited array storage structures and routines provided by these libraries and obviates the restructuring of array storage to fit the underlying library kernels, without a significant impact on performance.

In our framework, distributed arrays are implemented as tensors and array computations as higher-order operations on tensors, expressed in a language called the *tensor algebra*. A tensor is a multi-dimensional array that can have any number of sparse and dense dimensions, and is implemented as a distributed collection of array blocks of the same size. Our approach is to first construct a task workflow from array computations by partially evaluating these computations on the block coordinates - without looking at the block data. A task workflow is a directed acyclic graph where nodes are tasks and edges are dependencies between tasks, thus forming a partial order of task execution. When the tasks of a task workflow are assigned to processes, it becomes a task schedule. The task

979-8-3503-6248-0/24/$31.00 ©2024 IEEE

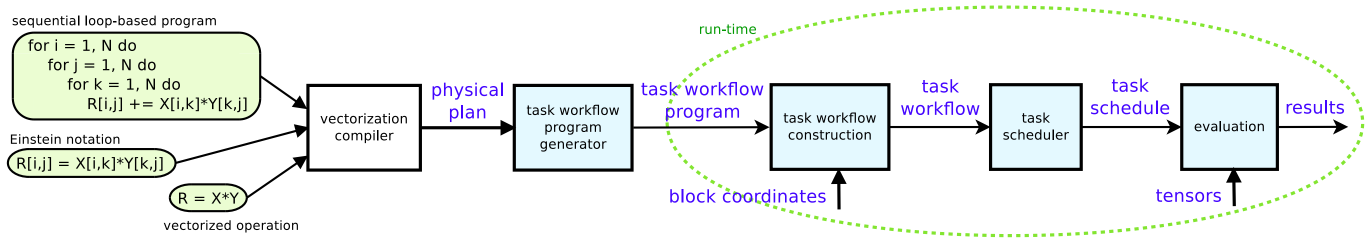


Fig. 2: System Architecture

schedule is then evaluated over the array block contents using MPI and OpenMP. Each task in the task workflow is a simple operation on a small number of array blocks and is executed at a single MPI process using shared-memory parallelism with OpenMP. Since each task works on few blocks at a time, the block size must be large enough to justify spending extra time for task scheduling before execution, given the performance improvement we gain in return.

For example, consider the matrix-matrix multiplication $R = X \times Y$, such that $R_{ij} = \sum_k X_{ik} \times Y_{kj}$. For two block matrices of size 2×2 blocks, the task workflow is shown in Fig. 1A. It corresponds to the physical plan of a join between X and Y on their block coordinates so that the column block coordinate of X is equal to the row one of Y, followed by a map operation that multiplies each join match (two blocks) using matrix multiplication, followed by a reduce operation that performs the block summation. The task workflow is constructed by partially evaluating the physical plan over the input tensors using only the block coordinates and ignoring the block data. In this workflow, a bottom Apply task is a no-op. A Pair task pairs a block from X with a block from Y (by construction, the column block coordinate of the former is equal to the row block coordinate of the latter), while the Apply task after Pair multiplies the two paired blocks using in-memory matrix multiplication. Finally, a Reduce task aggregates the input blocks using block summation.

Our task workflow generation algorithm is holistic as it constructs a single task workflow from the entire program by partially evaluating this program. While-loops and non-vectorizable for-loops are unrolled during partial evaluation (at workflow construction time) as long as their conditions do not depend on array values. If they do, the dependencies needed to evaluate a condition are collected from the task workflow, are scheduled, and executed repeatedly until this condition becomes false, completing the task workflow at the same time. Thus, task workflow construction, scheduling, and evaluation may be interleaved at run-time.

An important contribution of this paper is a novel scheduling algorithm that is custom-tailored for scheduling block-based tensor operations. It is a bottom-up algorithm that recognizes certain patterns inside the task workflow, assigns processes to the tasks in these patterns, and proceeds to the adjacent neighbors until the workflow is covered. Our goal is to generate quality schedules for frequent patterns of tensor computations that generate communication schemes that resemble optimal block-based algorithms, such as SUMMA [10]. The most important pattern used by our scheduler is shown in Fig. 1B and corresponds to the derivation of one result block from a groupBy-join operation. A groupBy-join generalizes many algorithms that correlate two data sources using an equi-join followed by a group-by with aggregation. The two binary operations of the groupBy-join, that is, the aggregation function after grouping and the operation that combines the matched values from the equi-join, must form a semiring, such as $(+, \times)$ and (min,+). Examples include matrix-matrix and matrix-vector multiplications, and one step of finding all-pairs shortest path in a graph. When our scheduler finds the pattern shown in Fig. 1B in the task workflow, it assigns the same process to the tasks x, y, and z, derived from the block coordinates (i, j) of y. More specifically, if there are p processes, the scheduler forms an implicit grid of size $n \times n$, where $n = \lfloor \sqrt{p} \rfloor$ for 2D block coordinates. Then, (i, j) is mapped to the $(i \% n, j \% n)$ process in the grid. When applied to the multiplication of the matrix X of size $M \times K$ blocks with the matrix Y of size $K \times N$ blocks, u will have N Pair parents, each one participating in the construction of an output block with coordinates (i, w), for $0 \leq w < N$. Since each one of these Pair parents is assigned to the process $(i \% n, w \% n)$ in the grid, these parents will be assigned to n different processes (one column in the grid). This scheme is equivalent to broadcasting the u block to n processes. A similar scheme occurs for the task v: it has M parents which will be assigned to one row in the grid. The task x on the other hand reduces the output of K tasks which are all assigned to the same process (since they all have coordinates (i, j)). This implicit grid of processes resembles the SUMMA process grid that handles full rows from X and full columns from Y at each process. Unlike SUMMA though, the TensorPlanner scheduling is done one block at a time and is fully asynchronous.

Another contribution of this work is memory management. It is important that the task results be deleted as soon as they are deemed unnecessary to prevent memory leaks and minimize memory use. This is crucial for ML programs that are translated into deep schedules derived from program iterations. Our implementation uses reference counting to delete blocks as soon as they are found to be obsolete.

The TensorPlanner system architecture is shown in Fig. 2. The vectorization compiler is a module that has already been implemented in our previous work [6]–[8], while the light blue boxes represent the contributions in this paper. Currently, our vectorization compiler translates programs in a Java-like language that can contain sequential loop-based programs as well as vectorized operations (but not Einstein notation yet). Our first new module, the task workflow program generator, synthesizes code from a physical plan that generates a task workflow. The code is generated at compile-time and is independent of the tensor sizes and content. At run-time, the generated task workflow program is evaluated over the block coordinates of the input tensors and generates a task workflow. Then, the task scheduler assigns processes to tasks and the resulting task schedule is evaluated over the tensor blocks. The evaluation engine is a hybrid system based on MPI for node-to-node communication and on OpenMP for shared memory parallelism to process tensor blocks within each MPI node.

We summarize the contributions of our work as follows:

- We present a framework for constructing task workflows from tensor programs by partially evaluating these programs against the block coordinates of the tensors.
- We present a novel task scheduler based on pattern matching that assigns processes to tasks in the task workflow. We provide patterns that produce schedules equivalent to well-known block-based distributed algorithms.
- We have implemented a high-performance asynchronous execution engine using MPI and OpenMP that uses a memory management system to delete unneeded blocks.
- We have evaluated the performance of our system relative to PyTorch [18], Ray [15], MLlib [14], Dask [20], and ScaLAPACK on a variety of linear algebra and ML programs. Based on these results, our system has a competitive performance in relation to these systems.

II. Background: The Tensor Algebra

The work reported in this paper extends our previous work on array and tensor comprehensions [6]–[8]. An abstract array in our framework is a mapping from array indices to values, represented as a dataset of key-value pairs in which the key consists of the array indices (a tuple of integers) and the value is the array value. Abstract arrays are implemented using efficient storage structures based on customized storage mapping functions. Our main storage structure for multi-dimensional arrays is a *tensor*, which is a distributed collection of array blocks and can have any number of sparse and dense dimensions. Operations on both abstract arrays and tensors are expressed in a general algebra, called the *tensor algebra* [7], which is equivalent to the nested relational algebra. It consists of higher-order operations that resemble the Spark RDD core operations [27]. The semantics of the tensor algebra is specified by the semantic function $\mathcal{E}$:

$$\mathcal{E}[\![\, v \,]\!] \;=\; v \qquad \text{(for a variable } v\text{)} \tag{1}$$

$$\mathcal{E}[\![\, \text{flatMap}(f,x) \,]\!] \;=\; \{\, w \mid v \in \mathcal{E}[\![\, x \,]\!] \wedge w \in f(v) \,\} \tag{2}$$

$$\begin{aligned}&\mathcal{E}[\![\, \text{join}(x,y) \,]\!]\\ &\quad = \{\, (k,(v,w)) \mid (k,v) \in \mathcal{E}[\![\, x \,]\!] \wedge (k,w) \in \mathcal{E}[\![\, y \,]\!] \,\}\end{aligned} \tag{3}$$

$$\begin{aligned}&\mathcal{E}[\![\, \text{reduceByKey}(\otimes,x) \,]\!]\\ &\quad = \{\, (k, \otimes/\{\, v \mid (k,v) \in \mathcal{E}[\![\, x \,]\!] \,\}) \mid k \in \pi_1(\mathcal{E}[\![\, x \,]\!]) \,\},\end{aligned} \tag{4}$$

where $\pi_1(s)$ on a set of key-value pairs s returns the set of (distinct) keys and $\otimes/\{v_1, v_2, \ldots, v_n\} = v_1 \otimes v_2 \otimes \cdots \otimes v_n$. The higher-order operation flatMap(f,x) applies the function f to every element of the dataset x and collects the results. The most common case of flatMap is map(f,x) = flatMap$(\lambda v.\, \{f(v)\}, x)$. Given two sets of key-value pairs x and y, join(x,y) joins these sets on the key. Finally, reduceByKey$(\otimes,x)$ groups the set of key-value pairs x by the key and reduces each group of values using the commutative and associative binary function $\otimes$.

For example, matrix-matrix multiplication $R_{ij} = \sum_k X_{ik} \times Y_{kj}$ on the abstract arrays X and Y can be expressed as follows in the tensor algebra:

$$\begin{aligned}&\text{reduceByKey}(\, +, \text{map}(\, f, \text{join}(\, \text{map}(\, f_a, \text{X}\,)\\ &\qquad\qquad\qquad\qquad\qquad\quad \text{map}(\, f_b, \text{Y}\,)\,)\,)\,),\end{aligned} \tag{5}$$

where the map functions are:

$$\begin{aligned}f_a &= \lambda((i,j),a).\,(j,(i,a)), \qquad f_b = \lambda((i,j),b).\,(i,(j,b))\\ f &= \lambda(k,((i,a),(j,b))).\,((i,j),a \times b).\end{aligned}$$

The tensor algebra has been designed to serve as an intermediate language for high-level array-based languages, such as vector languages and loop-based sequential languages that access and update one array element at a time. It can capture many linear algebra operations, such as inner and outer products of vectors, matrix addition and multiplication, matrix rotation and transpose, and array slicing and padding. Our tensor algebra though must be translated to high-performance distributed code guided by user-defined storage mappings, thus gaining expressiveness and extensibility without a significant impact on performance when compared to array libraries. This can be accomplished if abstract array operations, storage mappings, and low-level algorithms on customized array storages are expressed in the same algebra so that the abstract array operations can be fused with the mappings using algebraic laws to derive the concrete algorithms without materializing any abstract arrays. Such algebraic translation schemes have already been introduced in SAC [6] and STOREL [21]. Our framework is implemented on top of SAC [6] which stores abstract arrays into tensors, which are distributed collections of multi-dimensional blocks that may have any number of dense and sparse dimensions. Unlike SAC, which uses Spark as a back-end to implement tensor operations, our system translates tensor programs into asynchronous tasks, schedules them, and executes them using MPI and OpenMP.

Based on our storage mappings, the algebraic term (5) that specifies matrix-matrix multiplication is translated to a similar term over these tensors, but instead of value multiplication in f, it uses in-memory matrix-matrix multiplication on dense matrices (the tensor blocks), and instead of value addition in ReduceByKey, it uses in-memory matrix addition.

III. Tensor Task Schedules

In our framework, each task in the workflow has a unique task number and is assigned to a single process. Formally, we have a set of task numbers $\mathcal{T} \subseteq \mathbb{N}$ and a set of processes

$$\mathcal{P}[\![\, v \,]\!] \;=\; \text{map}(\lambda(k,x).\,(k,\text{Load}(x)),\ v) \tag{6}$$

$$\mathcal{P}[\![\, \text{join}(x,y) \,]\!] \;=\; \text{join}(\mathcal{P}[\![\, x \,]\!],\mathcal{P}[\![\, y \,]\!]) \tag{7}$$

$$\begin{aligned}&\mathcal{P}[\![\, \text{reduceByKey}(\otimes,x) \,]\!]\\ &\quad = \text{reduceByKey}(\oplus,\mathcal{P}[\![\, x \,]\!])\\ &\qquad \text{where } \oplus/s = \text{Reduce}(\otimes,s)\end{aligned} \tag{8}$$

$$\begin{aligned}&\mathcal{P}[\![\, \text{map}(\lambda(k,p).\,(e_1,e_2),\ x) \,]\!]\\ &\quad = \text{map}(\lambda(k,p).\,(e_1,\text{Apply}(f,\mathcal{S}[\![\, p \,]\!])),\ \mathcal{P}[\![\, x \,]\!])\\ &\qquad \text{where } f = \lambda(k,p).\,(e_1,e_2)\\ &\qquad\quad \text{and } \mathcal{S}[\![\, ((i,p),(i',q)) \,]\!] \;=\; \text{Pair}(\mathcal{S}[\![\, p \,]\!],\mathcal{S}[\![\, q \,]\!])\\ &\qquad\quad \text{and } \mathcal{S}[\![\, v \,]\!] \;=\; v\end{aligned} \tag{9}$$

Fig. 3: Construction of the Task Workflow

$\mathcal{W} \subseteq \mathbb{N}$, which are MPI process ranks. A task $i \in \mathcal{T}$ has the following properties:

- $\text{in}[i] \subseteq \mathcal{T}$: the tasks that produce the input to i
- $\text{out}[i] \subseteq \mathcal{T}$: the tasks that consume the output of i
- $\text{opr}[i]$: the operation code to execute
- $\text{process}[i] \in \mathcal{W}$: the process that evaluates i
- $\text{status}[i]$: the status of the task execution
- $\text{data}[i]$: the result of the operation (when cached)
- $\text{coord}[i]$: the block coordinates of the task i.

The sets in and out define the topology of the task workflow. An operation $\text{opr}[k], k \in \mathcal{T}$ is one of the following:

$\text{Load}(x)$	direct load of the tensor block x
$\text{Pair}(i,j)$	pair the results of $i \in \mathcal{T}$ and $j \in \mathcal{T}$
$\text{Apply}(f,i)$	apply f to the result of task $i \in \mathcal{T}$
$\text{Reduce}(\otimes,[i_1,\ldots,i_n])$	reduce the results of $i_1,\ldots,i_n \in \mathcal{T}$

The producers $\text{in}[k]$ of a task k and the result of evaluating the operation $\text{opr}[k]$ are defined as follows:

$\text{opr}[k]$	$\text{in}[k]$	$\text{eval}(k)$
$\text{Load}(x)$	$\emptyset$	x
$\text{Pair}(i,j)$	$\{i,j\}$	$(\text{data}[i],\text{data}[j])$
$\text{Apply}(f,i)$	$\{i\}$	$f(\text{data}[i])$
$\text{Reduce}(\otimes,[i_1,\ldots,i_n])$	$\{i_1,\ldots,i_n\}$	$\text{data}[i_1] \otimes \cdots \otimes \text{data}[i_n]$

The consumers of a task i are derived from the task producers:

$$\text{out}[i] = \{\, j \mid j \in \mathcal{T}:\ i \in \text{in}[j] \,\}.$$

Finally, the task coordinates $\text{coord}[i]$ are the block coordinates of its result and is equal to $\text{data}[i]$.first. The status property is explained in Section VI.

IV. Construction of the Task Workflow

Our first goal is to translate tensor computations into a task workflow that will form the basis for the task schedule. More specifically, we want to translate an algebraic expression that specifies tensor computations into a program that constructs a task workflow at run time. This task workflow will in turn be scheduled to run across the available processes by assigning processes to the tasks in the workflow (Section V). The program that constructs the task workflow is derived from an algebraic term using compositional rewrite rules that do not depend on the array data. During execution, this program looks at the coordinates of the tensor blocks but not at the actual block content. That is, our task workflow construction is a partial interpretation of the array computation based on the block coordinates exclusively. Since block coordinates are minuscule compared to array blocks, the task workflow construction can be done in orders of magnitude faster than the actual array computations ($\times 10^6$ for dense block matrices with block size 1000×1000), which justifies spending extra time on planning and scheduling before evaluation, given the performance gain we get in return.

The translation rules are given in Fig. 3. Given an algebraic expression e, $\mathcal{P}[\![\, e \,]\!]$ constructs a program (expressed in the same algebra) that constructs a task workflow. Basically, a join generates a set of Pair tasks (one Pair task for each pair of joined blocks), a map generates a set of Apply tasks (where each task transforms one or more blocks), and a reduceByKey generates a set of Reduce tasks (where each task generates one block by aggregating its input blocks). The Pair tasks are generated by Rule 9 instead of Rule 7 in order to remove the unneeded indices of the input parameters using $\mathcal{S}$. Rule 8 groups x by its key and generates one Reduce task for each different key. Rule 9 is the only rule that requires the functional parameter be matched to a certain pattern. This rule can be extended to capture more cases, such as conditionals and flatMaps inside the function body.

For example, the block matrix multiplication given in Section II is translated to the following task workflow program:

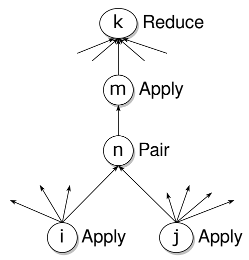
$$\begin{aligned}&\text{reduceByKey}(\oplus, \qquad\qquad\qquad \textit{// where } \oplus/s = \text{Reduce}(\textit{sum},s)\\ &\quad \text{map}(\lambda(k,((i,a),(j,b))).\,((i,j),\text{Apply}(f,\text{Pair}(a,b))),\\ &\qquad \text{join}(\text{map}(\lambda((i,j),a).\,(j,\text{Apply}(f_a,a)),\ \mathcal{P}[\![\, X \,]\!]),\\ &\qquad\quad \text{map}(\lambda((i,j),b).\,(i,\text{Apply}(f_b,b)),\ \mathcal{P}[\![\, Y \,]\!])))),\end{aligned}$$

where the block multiplication is done in f while the block summation is done in $\oplus$. When applied to two block matrices X and Y of size 2×2 blocks each, it will generate the task workflow shown in Fig. 1A.

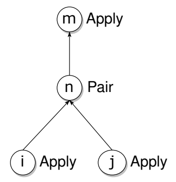
V. Static Scheduling

The goal of scheduling is to allocate each task i to a $\text{process}[i]$ in a way that evenly distributes the execution workload across processors. This allocation must also consider task dependencies and minimize the completion time of the parallel task execution. Specifically, each task should be scheduled in a process that evaluates most of its producers so that communication is minimized. The total cost of executing a set of tasks in parallel depends on two factors: the CPU cost of executing the tasks and the communication cost of sending data between processes. Minimizing communication cost is crucial as it is often the main bottleneck in distributed systems. However, achieving a balanced distribution of workload among processes is also necessary to maximize performance.

In our framework, we adopt a bottom-up solution that identifies common patterns of tasks that occur frequently in



(A) Reduce-Apply-Pair pattern



(B) Apply-Pair pattern

Fig. 4: Patterns used for Scheduling

linear algebra, assign processes to the tasks in the pattern, and proceed to their consumers bottom-up to complete the task assignments. One such pattern, when applied to matrix multiplication, produces schedules that resemble the optimal distributed algorithm SUMMA [10].

We have identified two important patterns that can occur frequently in a plan derived from linear algebra operations. Both patterns contain a Pair task, which is derived from a dataset join. They are as follows:

1) A Pair task followed by an Apply and a Reduce task, as shown in Fig. 4A.
2) A Pair task followed by an Apply with no Reduce task, as shown in Fig. 4B.

The first pattern can be found in computations involving a join followed by a groupBy with aggregation. Such a computation is known as a groupBy-join (GBJ). A groupBy-join generalizes many algorithms that correlate two data sources using an equi-join followed by a group-by with aggregation. Each matching pair from a join is often mapped to a single value before it is reduced with the other values. The join often generates a very large amount of intermediate results (far larger than the join inputs) just to be immediately reduced by the groupBy with aggregation. An example of a GBJ is block matrix multiplication, described in the Introduction. Our framework generates Pair tasks for joins and Reduce tasks for groupBy with aggregation. Therefore, the workflow of a GBJ will look like the example in Fig. 1A. One slice of this workflow that generates a single matrix block is shown in Fig. 1B and is repeated in Fig. 4A. To minimize communication, we want to compute all the Apply tasks m on the same process as the Reduce task k. However, the producers of the Pair n, which are the Apply tasks i and j, can have many different Pair tasks as consumers in other slices of GBJ and if those Pair tasks are assigned randomly, there will be a high communication overhead to send the outputs of i and j to the processes of their consumers. To minimize communication, we aim at minimizing the number of processes to which the Apply tasks i and j need to send their output. To achieve this, we use the coordinates of the Apply task m, coord$[m]$, to choose the process to assign to the tasks k, m, and n. This mapping from coordinates to processes yields a communication pattern that is similar to creating an implicit grid of processes in SUMMA [10] for matrix multiplication. More specifically, we form an implicit grid of size $D \times D$, where $D = \lfloor\sqrt{p}\rfloor$ for p processes. For a non-square p, the grid size will be $D\times(p/D)$. Let coord$[m] = (c_1, c_2)$ be the block coordinates of task m (for other array dimensions, we use a different number of coordinates). Then the tasks k, m, and n are assigned to the same process $(c_1\%D) \times D + c_2\%D$.

This pattern is particularly important because it can be found in matrix-matrix multiplication, matrix-vector multiplication, tensor-matrix multiplication, tensor-vector multiplication and other computations that resemble a groupBy-join. For matrix-matrix multiplication, our scheduling algorithm resembles the optimal distributed block matrix multiplication algorithm SUMMA [10]. SUMMA creates an implicit grid of cells of size $D \times D$, where each cell is handled by one processor. Hence, if there are p processors available, then $D = \lfloor\sqrt{p}\rfloor$. Essentially, each block of the two join inputs is replicated D times and shuffled to D cells in the grid. The blocks of the resulting matrix in each cell are derived from the shuffled blocks without the need for any further shuffling. Our scheduling algorithm works similarly as there is no communication needed to compute the Reduce tasks, while the results of the Apply tasks i and j need to be sent to at most D processes. This is an optimal communication pattern, because each matrix block is multiplied by a row or a column of blocks of the other matrix and each Apply child of a Pair task will be sent only to a single row or a single column of the grid of processes. Therefore, each matrix block is sent to $\sqrt{p}$ number of processes for the join and there is no shuffling during reduce. When applied to matrix-vector multiplication, the task i produces one matrix block while the task j produces one vector block. In that case, j will have one consumer only and coord$[m] = (c_1)$ will be the block coordinate of the task m. Our algorithm will assign the task m to the process $c_1\%p$. This scheme is then equivalent to broadcasting the vector block to all the p processes.

The second pattern, shown in Fig. 4B, resembles a join computation. It represents the slice of computation for one block of cell-wise array operations, such as matrix addition, $C_{ij} = A_{ij} + B_{ij}$ and cell-wise matrix multiplication, $C_{ij} = A_{ij} \times B_{ij}$. To minimize communication and balance workload of the processes, we assign the task n to either process$[i]$ or process$[j]$, depending on which process has less workload.

The workload of a process is the cost of all the tasks that have been scheduled on the process up to this point. We calculate the total cost of performing a task based on the CPU cost and communication cost. The CPU cost of a task is proportional to the total number of blocks of the inputs tasks, except for Load and Pair operations, which is 0. Based on the translation of tensor comprehensions to tasks, the size of the task output in number of blocks is estimated to be:

$$\text{size}(k) \quad = \quad \begin{cases} \text{size}(i) + \text{size}(j) & \text{if opr}[k] = \text{Pair}(i, j) \\ 1 & \text{otherwise.} \end{cases}$$

We calculate the CPU cost of a task i using,

$$\text{cpuCost}(i) \quad = \quad \begin{cases} 0 & \text{if opr}[i] \text{ is Load or Pair} \\ \sum_{j\in\text{in}[i]} \text{size}(j) & \text{otherwise.} \end{cases}$$

```
1. For a task $i$ at process$[i]$:
   if status$[i]$ = NotReady
         $\wedge\, \forall j \in$ in$[i]$: status$[j]$ = Computed | Completed
     then status$[i] \leftarrow$ Ready
2. For a task $i$ at process$[i]$:
   if status$[i]$ = Ready
     then data$[i] \leftarrow$ eval$(i)$
          status$[i] \leftarrow$ Computed
          $s \leftarrow \{\,$process$[j] \mid j \in$ out$[i]$: process[j] $\neq$ process$[i]\,\}$
          send (“copy”, $i$, data$[i]$) to each process in $s$
          status$[i] \leftarrow$ Completed
3. At any process $w$:
   if process $w$ receives a message (“copy”, $i$, $d$)
     then data$[i] \leftarrow d$
          status$[i] \leftarrow$ Completed
```

Fig. 5: Evaluation Rules

The communication cost of the task i, executed by the process w, is proportional to the number of blocks of the input tasks that are available in different processes since these blocks must be sent to process w:

$$\text{commCost}(i,w) \quad = \sum_{\substack{j\in \text{in}[i]\\ \text{process}[j]\neq w}} \text{size}(j).$$

Then, for some weighting factor α, the total cost is:

$$\text{cost}(i,w) \quad = \quad \text{cpuCost}(i) + \alpha \times \text{commCost}(i,w).$$

For any other case that does not match these two patterns, our algorithm schedules a task to the process from the set of processes of its producers that has the least workload so far. For example, if task i has a set of producers in$[i]$, our algorithm inspects the workload of the processes of each of the tasks in in$[i]$ and finds the process with the least workload.

Our scheduling algorithm initially finds all the entry points, which are Load tasks, and places them in a queue. The workload is updated every time a task is scheduled to a process. Then, it follows a bottom-up approach using the queue to schedule processes to the tasks only after their producers have been scheduled.

VI. Evaluation of the Task Schedule

Each task in a task workflow returns a block that is computed from the blocks generated by its input tasks. Tasks are evaluated based on the rules in Fig. 5. Each task i goes through the following stages, indicated by status$[i]$:

$$\text{NotReady} \longrightarrow \text{Ready} \longrightarrow \text{Computed} \longrightarrow \text{Completed} \longrightarrow \text{Removed}.$$

A task is Ready for execution if all its producers have been completed. A task is Computed if it has finished execution and has cached its result, and is Completed if it has sent its result to its remote consumers (that is, its output tasks). We use two completion stages, computed and completed, instead of one, because there may be a communication failure after computation and before completion, which requires recovery. Finally, a task is Removed if its cache has been deleted. Initially, status$[i]$ = NotReady. Rule 1 indicates that if a task is not ready and all the tasks that produce its input have been completed, then it becomes ready. Rule 2 dequeues one ready task i from the queue Q, evaluates it, and caches the evaluation result into data$[i]$. Then, it sends this result to its remote consumers through a ‘copy’ message and changes its status to Completed. This message, when received by the target process, is cached at the specified task, which is marked Completed (Rule 3). Given the exit tasks of a task workflow, the queue Q is initialized with the entry tasks, which are all tasks with in$[i] = \emptyset$ (that is, Load tasks) that are descendants of the exit tasks. The evaluation is completed when the queue Q becomes empty.

VII. Deleting Unneeded Blocks

Our system eagerly deletes blocks as soon as they are deemed to be obsolete for computing the rest of the workflow and for recovering from failures. Deleting blocks as soon as possible is highly desirable for deep task schedules that consist of millions of operations, such as ML programs on big data, which may generate a large number of intermediate blocks. For example, the matrix multiplication between two matrices of size $N \times N$ blocks may generate N^3 intermediate blocks, when evaluated naively. Each such intermediate block should be partially reduced in the result as soon as it is generated and then immediately removed. On the other hand, eager block deletion hinders recovery from failures because it removes data that may be later needed in reconstructing the state of a failed process. We relax this approach by keeping the cached data long enough to facilitate fault tolerance but not too long to cause memory leaks (details about fault tolerance will be given in an extended version of this paper).

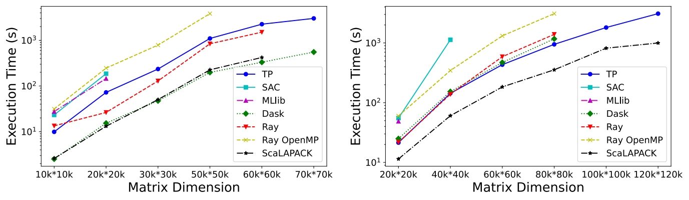
There are two kinds of tasks in our framework: those that propagate some or all their input blocks to the task result and those that construct a new block from the input blocks. Furthermore, tasks that receive blocks through ‘copy’ messages (Rule 3 in Fig. 5) also construct new blocks. We call the tasks that create blocks, *block constructors*. Reduce tasks are always block constructors while Load and Pair tasks are not. We can tell if an Apply task is a block constructor by inspecting the code of its operation.

The *dependents* of a task i at a process w, ds(i), is a set of task numbers, equal to $\emptyset$ if process$[i] \neq w$ (that is, i is not local to w), otherwise, it is equal to:

$$\{\, k \mid j \in \text{in}[i] \,\wedge\, k \in (\textbf{if } i \text{ is a block constructor} \\ \textbf{then } \{j\} \textbf{ else } \text{ds}(j))\,\}.$$

That is, the dependents of i at a process w are the closest descendants of i that are block constructors at w. Note that, by definition, all tasks between the task i and its dependents are local to w. In addition, a task i is called a *sink* if none of its consumers are local, that is, $\forall j \in$ out$[i]$: process$[j] \neq$ process$[i]$. Sinks are treated specially because no other task can claim them as dependents.

Our goal is to delete the caches of block constructors, since the other tasks simply propagate their inputs to the output by sharing their references. More specifically, if the task i is a block constructor or a sink, then we set deps$[i] \leftarrow$ ds(i),



(A) On 1 Node (B) On 5 Nodes

Fig. 6: Dense Matrix-Matrix Multiplication

otherwise deps$[i] \leftarrow \emptyset$. For a block constructor i, if every task j with $i \in$ deps$[j]$ has been completed, then the cache of i can be safely deleted. It is safe because the block constructors j have already constructed their output, which directly depends on the output created by the block constructor i. If all these block constructors j have been completed, the result of i is not needed any more. Note that, in addition to block constructors, sinks have dependents too because when a sink is completed, then regardless of being a block constructor or not, it must trigger the deletion of its dependents, Otherwise, these dependents may never be deleted. To efficiently check the previous condition about dependents, we use reference counting based on a dependent counter:

$$\text{dc}[i] \quad = \quad \|\, \{\, j \mid j \in \mathcal{T} \,\wedge\, i \in \text{deps}[i] \,\}\, \| \,.$$

Using dependents and dependent counters, the block deletion rule is expressed as follows:

```
4. For a task $i$ at any process $w$:
  **if** status$[i]$ = COMPLETED
    **for each** $j \in$ deps$[i]$: erase$(j)$,
```

where erase(j) uses reference counting to check whether the cache of task j can be deleted:

```
**if** $j$ is a block constructor {
  dc$[j] \leftarrow$ dc$[j] - 1$
  **if** dc$[j] = 0$ { status$[j] \leftarrow$ REMOVED; delete data$[j]$ } }.
```

VIII. Performance Evaluation

The source code of TensorPlanner (TP) is available at https://github.com/fegaras/TensorPlanner. We have evaluated the performance of our system relative to several popular frameworks. Our analysis covers a wide range of matrix operations and machine learning tasks, considering both weak and strong scaling scenarios on a cluster.

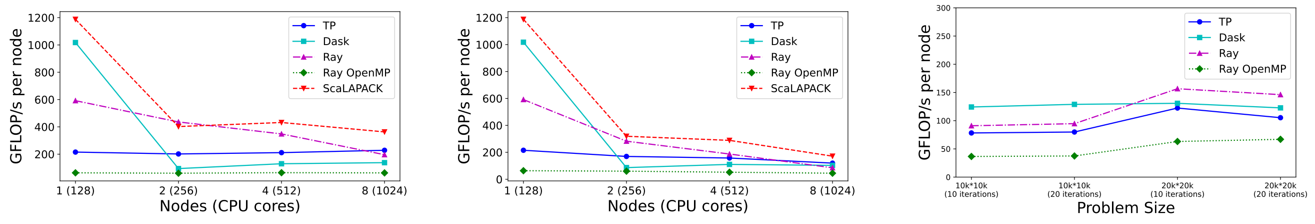
We ran our experiments on the Expanse cloud computing infrastructure at SDSC. Each node in the cluster is equipped with two 64-core AMD EPYC 7742 processors with 2.25 GHz clock speed and 256 GB DDR4 memory, connected through 100 Gb/s InfiniBand. The TP compiler is coded in Scala 2.12, the generated code is in C++ (gcc 7.5) with OpenMP annotations, and the task executor uses OpenMPI 4.1.3.

We evaluate the performance of TP by testing its generated plans on a set of real-world tensor operations and ML programs. We use dense and uniformly random sparse synthetic data for our evaluation, because synthetic data allows us to vary computation parameters and data sizes to demonstrate TP's versatility. In all our experiments, we have used blocks with 1000 double floating point numbers for vectors and 1000×1000 double floating point numbers for matrices. Each program was evaluated over multiple datasets and each evaluation was repeated four times from which the slowest run was ignored and the average value of the remaining three was reported. For all the systems, we used 2 executors per node - one executor per CPU socket.

a) Matrix Multiplication: We evaluated TP's generated plans for distributed dense matrix multiplication by comparing its performance with ScaLAPACK [5], Dask [20], Ray [15], Spark MLlib [14], and SAC [8]. As Ray does not have native support for distributed tensors, we hand-coded matrix multiplication in Ray using NumPy block operations. As TP generates C++ code for block multiplication using parallel for-loops in OpenMP, for a fair comparison and to demonstrate our scheduler's performance, we also compared our system with Ray that calls C++ code with OpenMP implemented using Cython for block operations instead of NumPy.

Memory Management. To demonstrate TP's ability to minimize memory consumption, we scale up the matrix dimensions on a single node and on a setup with 5 nodes. The goal of this experimental setup is to compare the maximum size of matrix multiplication each system can handle before they fail. The results are shown in Fig. 6. On a single node, TP can perform dense matrix multiplication up to $70k \times 70k$ matrices, where each matrix has 39.2GB of data and the multiplication generates 2.7TB of intermediate data. MLlib and SAC could only perform multiplication up to $20k \times 20k$ as they do not optimize memory usage. While Ray and ScaLAPACK could multiply up to $60k \times 60k$, Dask was the only other system that could perform $70k \times 70k$ multiplication. TP's execution time was slower than Ray, Dask and ScaLAPACK as these systems use optimal library calls. However, TP is significantly faster compared to Ray with OpenMP demonstrating TP's superior memory management and scheduler. On 5 nodes, our system can scale to multiplication of size $120k \times 120k$, which takes 115.2GB space for each matrix and creates 13.824TB of intermediate data. MLlib failed at $40k \times 40k$ and SAC failed at $60k \times 60k$. Dask and Ray failed for size $100k \times 100k$. While ScaLAPACK was the best performing system, TP performs better than Ray and Dask as data size increases because of TP's better memory management. Similar to single node execution, TP is much faster than Ray with OpenMP.

Weak scaling. For our weak scaling experiments (where data size per node stays constant), we started with matrices of size $20k \times 20k$ on a single node and increased the size as we added more nodes. Fig. 7A shows the weak scaling benchmarks with number of nodes and CPU cores on x-axis and throughput as GFLOP/s per node on y-axis. For a single node, ScaLAPACK has the highest throughput followed by Dask as these systems use optimal algorithms for matrix multiplication. However, as we scale to multiple nodes, the throughput of both Dask and ScaLAPACK falls drastically. Ray with OpenMP performs much worse than TP, achieving $50 - 70\%$ lower throughput



(A) Weak Scaling (B) Strong Scaling (C) Iterations

Fig. 7: Scalability of Matrix-Matrix Multiplication

than TP. TP is able to maintain similar throughput as we increase the number of nodes showing excellent scalability and on 8 nodes, TP produces similar or better throughput to other systems. This provides compelling evidence that TP is more scalable than the other systems.

Strong scaling. For strong scaling experiments, we multiplied two $20k \times 20k$ matrices and gradually increased the number of nodes. The results are shown in Fig. 7B. Similar to our weak scaling experiments, Dask and ScaLAPACK started with a high throughput on a single node, but showed poor scalability on multiple nodes. TP achieved great scalability here as well, maintaining a similar number of GFLOP/s per node across different experimental setups. TP outperforms Ray with OpenMP in all the cases and outperforms Ray on 8 nodes.

Iterative Matrix Multiplication. In this experiment, we used iterations to create a long matrix multiplication chain to demonstrate our scheduler's ability to handle complex iterative algorithms where a lot of data needs to be shuffled across multiple nodes between operations. We executed the matrix chain multiplication on 5 nodes with different matrix sizes and different number of iterations. As can be seen from Fig. 7C, TP performs slightly worse than Ray and Dask, but performs much better than Ray with OpenMP. This substantiates the capability of TP's scheduler as it can perform similarly to Ray and Dask without using optimal BLAS library and it can thoroughly outperform Ray with OpenMP.

b) Machine Learning: We evaluate TP's ability to perform distributed machine learning workloads (Linear Regression, Neural Networks) by comparing its execution time against similar algorithms in PyTorch [18]. We used synthetic datasets for the experiment and ran each algorithm for 10 epochs on both systems as we are only interested in execution time, not accuracy or convergence. We did not include Tensorflow [1] in this comparison as it does not perform well in distributed CPU-only workloads.

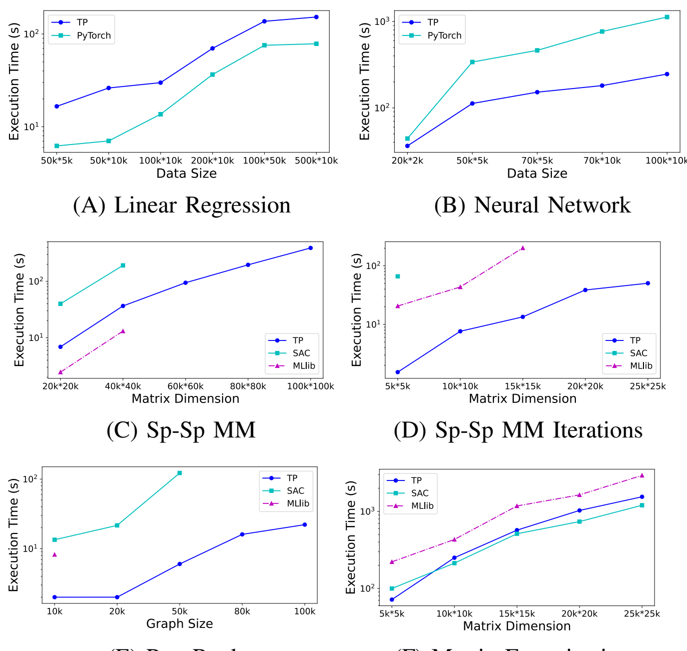
Linear Regression. For this experiment, we implemented Linear Regression using gradient descent and compared TP execution time with a similar algorithm from PyTorch's Neural Network module. Linear Regression has been selected for this experiment as it is a simple algorithm that still combines matrix-vector multiplication, addition and transformation operations, making it a suitable benchmark to evaluate TP's performance. We used 6 different data sizes and feature sizes and ran these on 5 nodes. The experiment results are shown in Fig. 8A. As PyTorch calls C++ BLAS functions internally, it performs better than TP. However, TP is versatile and can support any user-defined optimizations and activation functions which is not possible in PyTorch.

Neural Network. For this experiment, we used a neural network to perform a binary classification task. Our network architecture consisted of an input layer, a single hidden layer, and an output layer. For classification, the output layer employed the relu activation function and the hidden layer utilized the sigmoid activation function. We selected this experimental setup of binary classification using a small neural network as a baseline to determine how TP might perform in deep learning tasks involving larger datasets and deeper neural networks. We varied data sizes and feature sizes keeping the hidden layer size the same and conducted these experiments on 5 nodes. As demonstrated by the results in Fig. 8B, TP consistently performs noticeably better than PyTorch. This shows that our scheduler can outperform specialized ML systems on complex ML workloads even without using optimized BLAS library.

c) Sparse Linear Algebra: We demonstrate TP's ability to perform distributed sparse linear algebra operations by comparing against different benchmarks using SAC sparse tensors and MLlib sparse BlockMatrix. Both systems support optimized sparse linear algebra block operations.

Sparse Matrix Multiplication. For this experiment, we perform distributed sparse-sparse matrix multiplication. We used 1% non-zero elements and ran the experiments on 5 nodes for different matrix dimensions. This simple experimental setup allows us to showcase TP's performance in sparse linear algebra computations. TP achieves better performance compared to SAC but worse than MLlib. This is because MLlib uses the optimal BLAS library for matrix multiplication. However, both SAC and MLlib can handle matrix multiplication only up to $40k \times 40k$ while TP can handle $100k \times 100k$ as can be seen from Fig. 8C. We also compared sparse chain multiplications with 10 iterations and different matrix dimensions. From the results in Fig. 8D, we can see that TP is significantly faster and handles larger matrices than the other two systems.

PageRank. For this experiment, we used synthetic graphs with different sizes and ran the PageRank algorithm which involves iterative sparse matrix operations. This simple algorithm effectively tests how well TP handles iterative sparse algorithms. As shown in Fig. 8E, TP significantly outperforms both SAC and MLlib, highlighting the superiority of its



(A) Linear Regression (B) Neural Network

(C) Sp-Sp MM (D) Sp-Sp MM Iterations

(E) PageRank (F) Matrix Factorization

Fig. 8: Dense ML and Sparse Linear Algebra Benchmarks

memory management and scheduler.

Matrix Factorization. Sparse matrix factorization is an important algorithm for many recommendation systems. We used the SVD-like sparse matrix factorization proposed by Simon Funk [9]. This algorithm incorporates matrix multiplication, addition and transformation, providing a reasonable measure of how well TP handles complex sparse algorithms. For this experiment, we randomly generated a sparse matrix with 1% non-zeros and $m \times n$ dimension and derived two dense factors with dimensions $m \times d$ and $d \times n$. We used different matrix dimensions keeping $d = 1000$ and ran it on 5 nodes. The results are shown in Fig. 8F. Though TP performs very similar to SAC, it thoroughly outperforms MLlib, largely due to TP's superior scheduler.

IX. Related Work

Distributed Tensor Algebra Systems. There have been many recent systems built to run distributed tensor operations. The Tensor Relational Algebra [26] introduces an abstraction for distributed tensor operations based on relational algebra operations. Koutsoukos et al. [13] show how graph processing and relational operator algorithms can be implemented as tensor computations using tensor abstraction and tensor computation runtimes. DISTAL [25] is a compiler for dense tensor algebra that allows users to independently describe how tensors and computations map onto their target machines. By combining a data distribution language with a set of scheduling commands, DISTAL compiles a domain-specific language for tensor algebra into a distributed task-based runtime system. DiMage [12] finds optimal mappings of tensor and operators of a given DAG of tensor operations onto a multi-dimensional grid of processors using the Z3 SMT solver and automatically generates MPI code. ArrayLoop [23] enhances iterative array processing by using optimizations such as incremental iterative processing, overlap iterative processing and multi-resolution iterative processing. TensorPlanner provides a broader framework for large-scale array programs with efficient execution using MPI and OpenMP.

General-purpose Distributed Execution Engines. Dryad [11] is a general-purpose, high-performance distributed execution engine, which represents jobs as DAGs, where each node represents a program and the edges represent data channels. The Dryad runtime automatically maps the DAG onto the available physical resources. Dask [20] is a parallel computing library that enables computations on large datasets by managing data efficiently on disk. It minimizes memory usage through techniques like lazy evaluation, distributed caching, blocked algorithms, and dynamic, memory-aware task scheduling. Ray [15] is a distributed computing framework that unifies task-parallel and actor-based computations, enhancing scalability and fault tolerance through a bottom-up distributed scheduling strategy and a Global Content Store (GCS) for task and object metadata. Although Ray's primary focus is reinforcement learning algorithms, it can be used to distribute any machine learning and Python workload.

Machine Learning Systems. Most popular machine learning systems have support for distributed execution on heterogeneous systems. Tensorflow [1] uses a unified dataflow graph to represent computation and state, distributing it across multiple machines and devices. PyTorch [18] provides an imperative programming style, while its fundamental components are written in C++ to improve efficiency, simplify debugging and support widely-used scientific computing libraries. MXNet [4] utilizes symbolic expressions and tensor abstraction for computation across heterogeneous systems. It uses a distributed key-value storage system for data synchronization and optimizes memory allocation using strategies like "inplace" and "co-share". Horovod [22] accelerates distributed training across multiple GPUs on single or multiple nodes using a highly optimized ring-allreduce, though it lacks fault tolerance. These systems mostly focus on deep learning-specific optimizations, whereas TensorPlanner is more general-purpose and handles tensor computations beyond neural networks.

Distributed Fault-tolerant Systems. MLlib [14] is a scalable machine learning library, which provides various distributed linear algebra and ML operators. It is built on top of Apache Spark which leverages Resilient Distributed Datasets (RDDs) to achieve flexible data partitioning and efficient task scheduling, with lineage graphs for fault recovery. However, unlike TensorPlanner, Spark is not built to optimize memory usage. CIEL [16] is a scalable and fault-tolerant execution engine that efficiently manages distributed execution of data-parallel tasks a dynamic data-flow DAG, supporting data-dependent iterative or recursive algorithms. Lineage Stash [24] is a fault recovery system that reduces runtime overhead by asynchronously recording task information and forwarding it with tasks, enabling efficient reconstruction of missing inputs. It manages growing lineage by maintaining a local memory stash on each worker and flushing it asynchronously to a persistent data storage.

MPI-based Distributed Systems. Spark+MPI [3] enables MPI-based programs to be run within Spark by serializing data from RDDs to shared memory for MPI processing. MPI4Spark [2] enhances Spark's communication capabilities using MPI libraries, adding a new transport layer that uses MPI Java bindings to communicate with native MPI libraries, allowing for various network interconnects. Unlike TensorPlanner, these systems are mostly useful for applications that have a lot of communication or complex communication patterns

Distributed Schedulers. Sparrow [17] is a stateless decentralized scheduler that uses a decentralized, randomized sampling approach that provides near-optimal performance while avoiding the throughput and availability limitations of a centralized design. In Canary [19], the controller distributes data to workers, while the workers locally schedule and execute tasks based on their data partitions.

X. Conclusion

We have presented a general framework for translating high-level tensor programs to high-performance distributed code, implemented on MPI and OpenMP. The task workflows are generated from array programs by partially evaluating these programs on block coordinates. Optimal communication schemes are incorporated into the scheduler as patterns that apply to a few tasks at a time but generate optimal communication schemes when applied collectively. Our performance results indicate that, although it supports general tensor computations and does not use any high-performance array library, TensorPlanner is nearly as efficient and scalable as some of these libraries and ML systems, and occasionally outperforms them in complex ML workloads. For future work, we are planning to extend our system to run on hardware accelerators, such as GPUs, and to investigate more workflow patterns that capture well-known distributed tensor algorithms.

Acknowledgments: This work used Expanse at SDSC through allocation CCR200036 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by NSF grants #2138259, #2138286, #2138307, #2137603, and #2138296.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI'16*, 2016.

[2] K. Al-Attar, A. Shafi, M. Abduljabbar, H. Subramoni, and D. K. Panda. Spark Meets MPI: Towards High-Performance Communication Framework for Spark using MPI. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 71–81, 2022.

[3] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke. Bridging the Gap between HPC and Big Data Frameworks. *Proc. VLDB Endow.*, 10(8):901–912, April 2017.

[4] T. Chen, M. Li, Y. Li, and et al. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv*, abs/1512.01274, 2015.

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–121, 1992.

[6] L. Fegaras. Scalable Linear Algebra Programming for Big Data Analysis. In *24th International Conference on Extending Database Technology (EDBT)*, pages 313–324, 2021.

[7] L. Fegaras and Md H. Noor. Translation of Array-Based Loops to Distributed Data-Parallel Programs. *Proc. VLDB Endow.*, 13(8):1248–1260, 2020.

[8] L. Fegaras, T. Khan, Md H. Noor, and T. Sultana. Scalable Tensors for Big Data Analytics. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 107–114, 2022.

[9] Simon Funk. Netflix Update: Try This at Home. http://sifter.org/~simon/journal/20061211.html.

[10] R. A. Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[12] M. Kong, R. Abu Yosef, A. Rountev, and P. Sadayappan. Automatic Generation of Distributed-Memory Mappings for Tensor Computations. SC'23, 2023.

[13] D. Koutsoukos, S. Nakandala, K. Karanasos, K. Saur, G. Alonso, and M. Interlandi. Tensors: An Abstraction for General Data Processing. *Proc. VLDB Endow.*, 14(10):1797–1804, June 2021.

[14] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, DB Tsai, M. Amde, S. Owen, and et al. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[15] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI'18*, pages 561–577, Carlsbad, CA, October 2018.

[16] D. G. Murray, . Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. NSDI'11, page 113–126, 2011.

[17] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, page 69–84. ACM, 2013.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8026–8037, 2019.

[19] H. Qu, O. Mashayekhi, D. Terei, and P. Levis. Canary: A Scheduling Architecture for High Performance Cloud Computing. *ArXiv*, abs/1602.01412, 2016.

[20] M. Rocklin. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, pages 126 – 132, 2015.

[21] M. Schleich, A. Shaikhha, and D. Suciu. Optimizing Tensor Programs on Flexible Storage. In *SIGMOD'23*, May 2023.

[22] A. Sergeev and M. Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv:1802.05799*, 2018.

[23] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient Iterative Processing in the SciDB Parallel Array Engine. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. ACM, 2015.

[24] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica. Lineage Stash: Fault Tolerance off the Critical Path. SOSP'19. ACM, 2019.

[25] R. Yadav, A. Aiken, and F. Kjolstad. DISTAL: the Distributed Tensor Algebra Compiler. PLDI 2022, page 286–300. ACM, 2022.

[26] B. Yuan, D. Jankov, J. Zou, Y. Tang, D. Bourgeois, and C. Jermaine. Tensor Relational Algebra for Distributed Machine Learning System Design. *Proc. VLDB Endow.*, 14(8):1338–1350, April 2021.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.