Legate NumPy: Accelerated and Distributed Array Computing

Michael Bauer
NVIDIA
mbauer@nvidia.com

Michael Garland
NVIDIA
mgarland@nvidia.com

ABSTRACT

NumPy is a popular Python library used for performing array-based numerical computations. The canonical implementation of NumPy used by most programmers runs on a single CPU core and only a few operations are parallelized across cores. This restriction to single-node CPU-only execution limits both the size of data that can be processed and the speed with which problems can be solved. In this paper we introduce Legate, a programming system that transparently accelerates and distributes NumPy programs to machines of any scale and capability typically by changing a single module import statement. Legate achieves this by translating the NumPy application interface into the Legion programming model and leveraging the performance and scalability of the Legion runtime. We demonstrate that Legate can achieve state-of-the-art scalability when running NumPy programs on machines with up to 1280 CPU cores and 256 GPUs, allowing users to prototype on their desktop and immediately scale up to significantly larger machines. Furthermore, we demonstrate that Legate can achieve between one and two orders of magnitude better performance than the popular Python library Dask when running comparable programs at scale.

CCS CONCEPTS

• **Software and its engineering → Runtime environments**; • **Computing methodologies →** *Parallel programming languages*; *Distributed programming languages.*

KEYWORDS

Legate, NumPy, Legion, Python, HPC, Distributed Execution, GPU, Control Replication, Logical Regions, Task-Based Runtimes

ACM Reference Format:
Michael Bauer and Michael Garland. 2019. Legate NumPy: Accelerated and Distributed Array Computing. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356175

1 INTRODUCTION

Python has become one of the most widely used languages for data science, machine learning, and productive numerical computing. NumPy is its *de facto* standard library for array-based computation, providing a simple and easy to use programming model whose interface corresponds closely to the mathematical needs of applications. NumPy array objects also act as a common data storage format that can be used to share data between other Python libraries, such as Pandas [13], SciPy [9], or h5py [4]. Thus, NumPy has become the foundation upon which many of the most widely used data science and machine learning programming environments are constructed.

While its interface is powerful, NumPy's implementation is currently limited to a single-node, occasionally multi-threaded, CPU-only execution model. As datasets continue to grow in size and programs continue to increase in complexity, there is an ever increasing need to solve these problems by harnessing computational resources far beyond what a single CPU-only node can provide. A user can address this need today by combining explicitly parallel and distributed tasking systems [23] with facilities that support GPU acceleration [11, 19]. However, such solutions require rewrites of application code and additional programming expertise, while often suffering from limited scalability.

To address these problems, we have developed Legate, a drop-in replacement for NumPy that, to our knowledge, is the first programming system that can transparently execute NumPy-based programs with GPU acceleration across machines of any scale. Legate is implemented on top of the Legion task-based runtime system, which from its inception was designed to achieve high performance and scalability on a wide range of supercomputers [2]. Legate empowers users to harness as many computing resources as they desire, while remaining easy to use by providing an interface identical to that of NumPy. Using Legate simply requires replacing uses of the `numpy` module with uses of the `legate.numpy` module, which typically requires changing only a single line of code in a program. As a result, users can write and debug programs with limited datasets on their desktop and then immediately scale execution to whatever class of machine is needed to process their full dataset.

To create Legate, we have developed a novel methodology for dynamically translating from the NumPy application interface to the programming model of the Legion runtime system (Section 3). This involves both a mapping of n-dimensional array objects onto the Legion data model and a mapping of array-based operators onto the Legion task model. To achieve efficient execution of this program, we have developed a set of heuristics that leverage domain-specific characteristics of the NumPy interface to make appropriate decisions when mapping work and data to physical locations in the machine (Section 4). We have also developed an approach for leveraging Legion's dynamic control replication mechanism to avoid the sequential bottleneck that would otherwise be imposed by having a single-threaded interpreter (Section 5). Finally, we demonstrate in Section 6 that Legate can weak-scale interesting NumPy programs out to machines with 1280 CPU cores and 256 GPUs across 32 nodes and deliver between one and two orders of magnitude

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356175

```
1 try:    import legate.numpy as np
2 except: import numpy  as np
3
4 # Generate a random n×n linear system
5 # for illustration purposes
6 A = np.random.rand(n,n)
7 b = np.random.rand(n)
8
9 x = np.zeros(b.shape) # Initialize solution
10 d = np.diag(A)        # Extract diagonal
11 R = A - np.diag(d)    # Non-diagonal elements
12
13 # Jacobi iteration x^{i+1} ← (b − Rx^i) D^{-1}
14 for i in range(n):
15     x = (b - np.dot(R,x)) / d
```

Figure 1: Program fragment for Jacobi iteration that uses Legate if available and alternatively falls back to NumPy.

better performance than the popular Python library Dask when executing comparable programs at scale.

2 BACKGROUND

NumPy is a Python library that provides an n-dimensional array data type (`numpy.ndarray`); a variety of operators for constructing, accessing, and transforming arrays; and a collection of algorithms for performing common mathematical operations on vectors, matrices, and tensors stored in arrays (e.g., the dot product `numpy.dot`). The NumPy interface is therefore convenient for writing a variety of numerical algorithms because its notation is broadly similar to the mathematical operations a programmer might wish to express. Figure 1 provides an illustrative example implementing the Jacobi method for iterative numerical solution of linear systems.

2.1 Elements of NumPy

Every n-dimensional array in NumPy [16] is an instance of the class `ndarray`, or a subclass thereof, and has an associated n-tuple `shape` describing the array's extent in each of its n dimensions. The elements of an array are all objects of a single specified type, and in practice elements are most commonly integer or floating point types of a processor-supported size (e.g., 16-, 32-, and 64-bit values) since these are the most efficient.

While it is possible to implement algorithms as explicit Python loops over individual elements of `ndarray` objects, the more idiomatic approach is to use operations that implicitly act on entire arrays. For example, line 11 of our Jacobi example (Figure 1) subtracts one matrix from another in a single operation. The canonical NumPy implementation in turn uses efficient implementations of such operations (e.g., in compiled C or Fortran) to achieve reasonable levels of performance. For some operations, NumPy implementations can even call out to high-performance BLAS libraries [6]; for example, the expression `np.dot(R,x)` on line 15 of Figure 1 can be mapped to a BLAS `GEMV` operation. We will similarly focus on mapping operations on entire arrays to parallel tasks in Legate.

NumPy supports a feature called *broadcasting* that enables operators to work over arrays of different shape. For example, on line 11 of Figure 2, NumPy will broadcast the scalar constant 0.2

```
1 # Given grid, an n×n array with n > 2,
2 # create multiple offset stencil views
3 center = grid[1:-1, 1:-1]
4 north  = grid[0:-2, 1:-1]
5 east   = grid[1:-1, 2:  ]
6 west   = grid[1:-1, 0:-2]
7 south  = grid[2:  , 1:-1]
8
9 for i in range(iters):
10     total = center+north+east+west+south
11     center[:] = 0.2*total
```

Figure 2: Program fragment for a 2-D stencil computation using multiple views of the same underlying array data.

out to the shape of the `total` array. While scalar broadcasting is straightforward, the full semantics of broadcasting generalize to a wider range of mismatched array shapes and dimensionality.

NumPy supports creating sub-arrays via *indexing*. NumPy supports both *basic* and *advanced* modes of sub-array indexing. Basic indexing extends Python's standard slicing notation to multi-dimensional arrays: the sub-array is specified by (optional) start, stop, and/or stride arguments along each dimension. The resulting sub-array object is a *view* onto the original array, meaning that it references the same physical storage as the original array, and a change to one array must be reflected in the other. For example, Figure 2 computes five sub-array views onto the original `grid` array on lines 3-7. These views are all read on line 10 as part of a stencil computation and then the `center` view is written on line 11. All changes written to the `center` sub-array must be reflected in all the other views. Note that sub-arrays can themselves be indexed to describe even more refined views of data and there is no limit to the nesting depth of sub-arrays. In contrast to basic indexing which accepts only slices, advanced indexing accepts indices in the form of an arbitrary data structure (e.g., a Python list). The resulting sub-array object is not a view but instead a separate *copy* of the data from the original array. An important property of our approach for translating NumPy's interface to Legion's data model is that it allows us to preserve the semantics of both forms of indexing.

2.2 Legion Programming Model

Legion is a data-driven task-based runtime system [2] designed to support scalable parallel execution of programs while retaining their apparent sequential semantics. All long-lived data is organized in *logical regions*, a tabular abstraction with rows named by multi-dimensional coordinate indices and *fields* of arbitrary types along the columns (see Figure 3). Logical regions organize data independently from the physical layout of that data in memory, and Legion supports a rich set of operations for dynamically and hierarchically partitioning logical regions [27, 28]. The Legion data model is thus ideally suited for programming models such as NumPy which must manipulate collections of multi-dimensional arrays and support creation of arbitrary views onto those arrays at runtime.

Legion programs decompose computations into units called *tasks*. A program begins with a single top-level task, and every task is permitted to launch any number of sub-tasks. Every task must specify all of the logical regions it will access when it runs and what

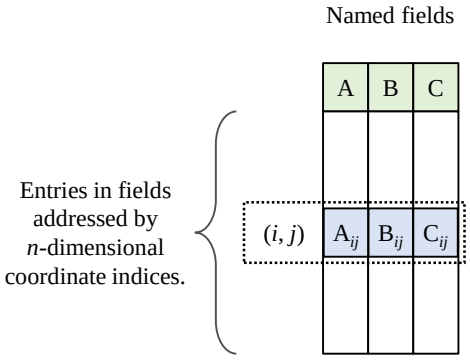

Figure 3: Logical regions are a tabular data model with multiple fields (columns) where elements (rows) are accessed via *n*-dimensional indices.

access privileges (e.g., read-only, read-write, or reduce) it requires for each. Legion also supports bulk *index space task launches* where the specified task is run at every point in a given index space.

Legion uses a deferred execution model where task launches return immediately to the caller, while the launched tasks execute asynchronously. Task return values are wrapped in futures which can be examined or passed directly to later tasks. The Legion runtime performs dynamic dependence analysis based on its knowledge of the regions used by each task and their relationships through partitions to maintain the sequential semantics of the program; tasks determined to be independent can be re-ordered or executed in parallel. This greatly simplifies the design of Legate, since the translation from NumPy to Legion can focus purely on domain-specific semantics rather than low-level details of distributed execution.

A Legion application specifies neither where tasks will run nor where data will be placed, and is therefore a machine-independent specification of a computation. All machine-specific *mapping* decisions, including both where tasks should execute as well as in which memories *physical instances* of their required logical regions should be placed, are made by Legion mapper objects separated from the application logic [2]. Mappers may also choose between any number of functionally equivalent task variants, which may be dynamically registered with the runtime. Regardless of the mapping decisions, Legion maintains the original program's machine-independent semantics by automatically inserting any data movement and/or synchronization required. This allows Legate to keep the details of its mapping logic entirely separate from the user's application logic while dynamically choosing the kind and location of processors on which to run tasks.

3 TRANSLATING NUMPY TO LEGION

Legate translates the NumPy interface to Legion by associating NumPy arrays with logical regions and converting each NumPy operation into one or more task launches over the logical regions that hold its array arguments. Legate adopts Legion's deferred execution model, allowing tasks to run asynchronously with the caller and blocking only when the Python program explicitly accesses an array value (e.g., to evaluate a conditional). Legate launches tasks in program order and relies exclusively on Legion's runtime dependence analysis to correctly preserve the semantics of the program while finding opportunities for parallel execution. This ability to rely on dynamic dependence analysis to preserve correctness is particularly valuable in dynamic languages, like Python, where comprehensive static analysis is not feasible.

Deferred execution makes two key contributions to the performance of Legate. First, it allows the application to run ahead of the execution of NumPy operations, assuming that it does not need to inspect corresponding array values. This exposes additional task parallelism that would not be available if the application had to wait for each NumPy operation to complete before proceeding. Additional parallelism, in turn, gives Legate the opportunity to run computations in parallel across many processors, or allow it to re-order the execution of tasks to help hide any long latency data movement or synchronization operations that occur in large distributed machines. Second, deferred execution helps hide the cost of the dynamic dependence analysis performed by Legion. While essential for correctness, the cost of dependence analysis can be non-trivial. By overlapping the dependence analysis with application execution, the additional latency can be hidden.

Legate also leverages Legion's support for partitioning of logical regions to provide the mechanism for implementing NumPy's view semantics, where different arrays are actually backed by the same data. Views in NumPy are handled naturally by mapping each view to a sub-region in a partition as sub-regions are always aliased with their ancestor regions in Legion. Legate further uses Legion to partition logical regions in multiple ways to adapt the distribution of arrays to the computation being performed. Legate simply determines the best partitioning of each array for performing a given operation and then launches its tasks without consideration for the way these arguments have been partitioned previously. The Legion runtime automatically manages the coherence of data between all partitions of each array. This is in stark contrast to other distributed NumPy implementations (see Section 7), which support only a single physical partitioning of arrays and are thus forced to provide custom dependence and communication analyses.

3.1 Legate Core

While the majority of Legate is currently centered around providing support for the NumPy interface, there is a `core` module that provides the basic functionality needed by any Python library that intends to use Legion for accelerated and/or distributed execution. This core module provides two primary services: an initialization script that will launch any Python program on a distributed machine with Legion, and a set of Python bindings for performing calls into the Legion runtime.

The execution of any Legate program is begun by the start-up script which initializes the Legion runtime on all nodes allocated to the application and runs a top-level task with no NumPy-specific functionality in an unmodified Python interpreter[1]. The custom top-level task configures the environment, including the Legion runtime, and uses Python's `exec` statement to invoke the Python application file. The application can use any Python modules, including NumPy, without restriction. The `legate.numpy` module provides its own implementation for the NumPy API when it is imported. When

[1]The Legate start-up script can also be used to start an interactive interpreter, even for multi-node machines, if the job scheduler supports such jobs.

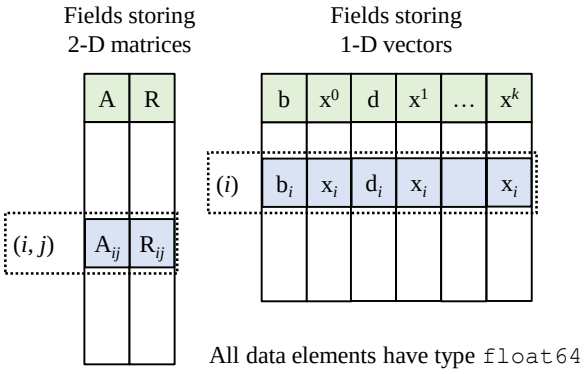

Figure 4: Allocation of logical regions and fields for the Jacobi solver example. By sharing a common index space, arrays can re-use partitions.

the Python program is done executing, Legate will free any Legion resources and then exit the top-level task, allowing the Legion runtime to shut down.

The core Legate module also provides a custom set of Python bindings for the Legion runtime interface. The bindings call out through the Python CFFI module to Legion's C runtime interface to perform operations such as creating logical regions and launching tasks. Legate's bindings differ from the canonical set of Python bindings for Legion which are designed for embedding the Legion programming model in generic Python. Instead, Legate's bindings are specialized for the construction of Python libraries that use Legion as an execution runtime.

3.2 Organizing Arrays in Logical Regions

The first step in translating the NumPy interface to the Legion programming model is to organize NumPy `ndarray` objects into logical regions. The most straightforward approach would be to create a unique region for each `ndarray`. However, we observe a frequent need to share partition arrangements across arrays involved in an operation. For example, when adding two vectors $x + y$ we would naturally prefer that both x and y be partitioned in exactly the same way. Therefore, we have developed an approach that packs arrays into separate columns of common regions. Since Legion directly supports launching tasks with arbitrary subsets of a region's columns, Legate can amortize the cost of dynamic partitioning by sharing a common partitioning scheme across a set of arrays organized in a single region.

Legate creates one or more logical regions for each array shape encountered during execution depending on the number of arrays of a given shape that are live at a time in a program. Each n-D array needed by the program is associated with a field of a logical region matching its shape. When an application calls an operator that creates a new array (e.g., `np.empty`), Legate selects a logical region with the given shape, or creates one if none exists. In the selected region, Legion either selects an unused field with the same element type (e.g., `float64`) or adds a new field as required. Legate then returns an `ndarray` object to the calling program that stores the necessary information about the associated region and field. This object implements all the standard NumPy interfaces and handles releasing the associated field when it is itself deallocated

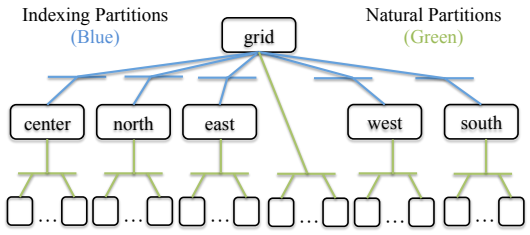

Figure 5: Region tree for the stencil example. Indexing partitions are made to represent view arrays created by the application. Natural partitions are created by Legate for distributing computations on different arrays across the machine.

by Python's garbage collector. Arrays of rank 0 (i.e., scalars) are a special case, and are represented directly as Legion futures rather than being allocated as a field in a region. Note that none of these operations allocate memory for the array; only the metadata is materialized in memory at this point. Associating multiple arrays with the fields of a single region allows them all to share the same set of partitions, thus helping Legate avoid performing redundant partitioning operations.

Figure 4 illustrates the logical regions and fields that Legate creates for the Jacobi solver example from Figure 1. Two logical regions are created for the two different array shapes encountered when executing the program. One logical region holds fields for the $n \times n$ matrices A and R, while a second region holds fields for the vectors of length n, including several versions of x, indicated by superscripts. The precise number of occurrences of x, indicated by k, is determined by the number of loop iterations Legion's deferred execution runs ahead as well as how often the Python garbage collector is invoked, but in practice it is far smaller than n. Recall also that the structure of logical regions does not have any implications on data layout, so adjacent fields in a logical region do not imply that those fields have any locality in memory.

Legate fully supports both basic and advanced indexing on arrays as defined by NumPy. Basic indexing is directly supported using the Legion partitioning API. Sub-regions in Legion are always aliased with their parent (and other ancestor) logical regions, and this matches the view semantics of basic indexing. When the program creates a sub-array via basic indexing, Legate consults the logical region associated with the source array to see if any existing partition of that region matches the requested sub-array. If such a partition exists, the name of the sub-region is extracted and used to create a resulting `ndarray` object. If no such partition exists, Legate will invoke the appropriate Legion partitioning operator [28] to create the sub-region. If possible, Legate attempts to infer potential tilings of the array from the first basic indexing call so that multiple sub-regions can be tracked with the same partition. Figure 5 shows the region tree created by the stencil code example in Figure 2. Legate makes a separate partition, each with a single sub-region, to describe the sub-arrays created by the indexing operations performed on lines 3-7. We describe the creation of the other partitions in this region tree in Section 3.3.

When advanced indexing is performed, Legate first converts the indexing data structure into a Legate array. It then performs either a Legion gather or scatter copy between the source and destination

arrays using the indexing array as the indirection field. This also matches the semantics of NumPy advanced indexing which creates a new copy of the data. In cases where advanced indexing is used in conjunction with a Python in-place operator (e.g., +=), Legate leverages Legion support for gather and scatter reductions with atomic reduction operators to apply updates in-place. In cases where advanced indexing is used in expressions on both the left and right hand sides of a statement, Legate can fuse those into a single indirect copy operation because Legion supports indirection arguments on both source and destination regions.

3.3 Launching Tasks for NumPy Operations

The next step in translation from NumPy to Legion is to decompose individual NumPy operations into one or more asynchronous tasks that can be submitted to, and distributed by, the Legion runtime. The precise number and kind of these tasks is informed by guidance from our mapper, whose design is described in Section 4.

We have designed Legate operations to accept any Python value that is convertible to a NumPy array, including scalar values which can be identified with 0-dimensional arrays. This maximizes programmer flexibility and enables Legate to compose with other Python libraries with minimal overhead. When a program calls one of the NumPy interfaces supported by Legate (e.g., `np.dot`), Legate converts every argument that is not already a Legate array into a NumPy array and then subsequently a Legate array. Conversions from NumPy arrays to Legate arrays have minimal cost, as they are implemented using Legion *attach* operations [8] that inform the runtime of the existence of an external memory allocation that should be treated as a physical instance (see Section 2.2) of a logical region. In this case, the external memory allocation is the buffer of the NumPy array. Using this feature, Legate can ingest a NumPy array without needing to eagerly copy its data.

Legate must next determine whether to perform individual or index space task launches for any computations. While Legate could launch individual tasks onto all target processors, it is more efficient to perform index space task launches that create tasks in bulk. In general, Legate will only use single task launches that directly operate on the logical regions of array arguments if it is targeting a single processor for an operation. For parallel and distributed execution of an operation, Legate will perform index space task launches over partitions of the logical regions for array arguments. The choice of whether to perform single or index space task launch for each operation is controlled directly by the Legate mapper's decisions over whether and how arrays are to be partitioned.

Partitions of logical regions are created both to reflect application level sub-arrays and to perform index space task launches. For example, consider the stencil computation from Figure 2. Execution of the stencil code will create several different partitions of the 2-D array representing the computation domain. The top-level logical region for the array and all of its partitions are shown as a *region tree* in Figure 5. *Indexing partitions* are created to represent the sub-array views created by the application on lines 3-7, while *natural partitions* are created to perform data parallel index space task launches over the different sub-arrays. In addition to indexing and natural partitions, Legate may also need to make *dependent partitions* [28] for computing partitions that are a function of another partition when performing NumPy operations that are not inherently data parallel, such as the `np.dot` from the Jacobi example in Figure 1. Legate computes natural and dependent partitions for performing index space task launches along with input from the Legate mapper for making machine specific decisions.

Natural partitions are the common partitions used by Legate for performing data parallel operations with index space task launches. Natural partitions are computed to balance the tile extents in each dimension to minimize the surface-to-volume ratio for any cases in which the boundaries of the array are needed for communication. In addition to minimizing surface-to-volume ratio, Legate also computes natural partitions for arrays with two additional constraints: (1) a lower bound on the minimum volume of a tile, and (2) an upper bound on the number of tiles. Appropriate choices for these constraints depend on machine-specific characteristics; therefore, we expose both constraints as tunable values which are set by the Legate mapper. Furthermore, our design guarantees that all arrays with the same shape will have the same natural partitioning. Any array smaller than the minimum tile size is not partitioned.

To determine how to launch tasks based on array arguments, Legate identifies the largest input or output array for a computation. We refer to this argument as the *key array*. Since it is the largest argument, the key array is also the one that is likely to be the most expensive to move if it is not used with its natural partitioning. If the key array is too small to have a natural partitioning, then Legate will simply issue a single task launch to process all the data. However, if a natural partition exists for the key array, then Legate uses the index space that describes all the sub-regions in the natural partition[2] of the key array as the index space for any task launches that need to be performed. The point tasks in an index space launch will then have a natural identity mapping onto the key array, which is a property that will be leveraged by the Legate mapper to determine task distribution.

In many cases, such as data parallel operations, all the array arguments will have the same shape as the key array, and their natural partitions will therefore all align. However, array arguments may have different shapes and sizes than the key array in cases like NumPy broadcasting (see Section 2.1), reduction operations, and other NumPy operations such as `np.dot`. For single task launches, this will not matter as the names of the logical regions can be directly used and the task implementation will handle any broadcast or reduction operations. However, if index space task launches are to be performed, Legate must also determine the dependent partitions and *projection functions* needed to be used for the other array arguments which are of different shape than the key array. Projection functions allow index space task launches to compute the names of sub-regions to be used for point tasks based on the point of the task in an index space launch [2].

As an example, consider the NumPy expression in Figure 6 drawn from a k-means clustering implementation which broadcasts two 1-D arrays x and y against each other to produce a new 2-D array z. Assuming we have four processors, each of the arrays will have a natural partition containing four sub-regions (shown in green). The key array will be z as it is the largest and therefore the task launch

[2] This index space is called the *color space* of the partition in Legion terminology [2].

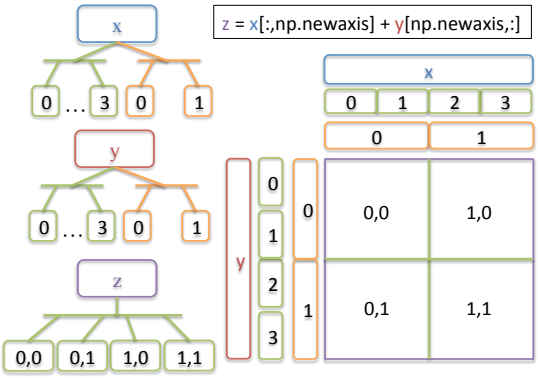

Figure 6: An example broadcast expression that requires the creation of dependent partitions (orange) in addition to the natural partitions (green) of the x and y arrays. Custom projection functions are required to index the dependent partitions for each of the point tasks in the index space task launch over the key array z.

will be performed over the inclusive 2-D space $(0, 0)$-$(1, 1)$. Legate then computes dependent partitions (shown in orange) of the x and y arrays with two sub-regions to align with the needed input data for each point task in the index space launch. After computing these partitions, Legate chooses projection functions that use the corresponding coordinate of the point task when indexing into the dependent partitions to select a sub-region as input.

Legate registers a suite of around a hundred projection functions with Legion. As in the example in Figure 6, nearly all projection functions either add, remove, or permute the dimensionality of the index space points in order to specify which sub-region of a partition each point task should use. For each array argument and associated projection function, Legate will also search for the appropriate partitioning on the array for use with any region arguments. If it cannot find the proper dependent partitioning, Legate will use a Legion dependent partitioning operation [28] to compute a new dependent partition of the other array argument based on the key array partitioning. With partitions and projection functions, Legate uses Legion's existing features to handle non-trivial NumPy semantics such as broadcasting and reductions.

While most NumPy operations store their results directly into output arrays, some operations (e.g., computing the minimum element in an array with `np.amin`) have scalar return values. Legion returns futures for any such return values, and Legate in turn wraps these futures in 0-D arrays without blocking on the result. In the case of index space task launches, Legate also passes in a reduction operator to Legion so that Legion can perform a tree reduction of future values across all point tasks efficiently.

3.4 NumPy Task Implementation

While Legate tasks are launched from Python through the Legion C API, all task implementations are written in C++ and CUDA. This highlights an important feature of Legion that Legate leverages: different tasks in a Legion program can be written in different languages and seamlessly interoperate because all data is passed through logical regions. Currently all leaf tasks are hand-written, but we believe in the future they could potentially be generated by one of the many single-node fusion-JIT NumPy systems we discuss in Section 7. For some leaf tasks such as those for dense linear algebra, Legate will call out to libraries like OpenBLAS [29] or cuBLAS [17]. Use of libraries such as cuBLAS guarantee that users of Legate get transparent acceleration of their code using new hardware features such as tensor cores on GPUs [18].

For every kind of task that Legate launches, there are three different task variants that Legate registers with Legion: one for sequential execution on a single core of a CPU, one for parallel execution on a multi-core CPU with OpenMP, and one for parallel execution on a GPU. For many operators we are able to implement all three variants with a single templated C++ function that can be instantiated using different *executors*, objects that decouple the specification of how code is executed from what it is computing. This use of executors eliminates duplicated code for variants and simplifies code readability and maintainability. We use the existing executors for CPUs, OpenMP, and CUDA in the Agency library [1] to instantiate operator templates and create each of the different task variants. Extending Legate to support additional processor variants would simply require the creation of new executors with which to instantiate any operator template.

Regardless of how variants are created, all variants registered for a given NumPy operator must maintain the same ABI-like interface for passing arguments such as regions and futures. This property gives the Legate mapper complete flexibility to choose between any of the available variants at runtime when making its mapping decisions. Having at least three kinds of variants available presents the mapper with numerous options for how best to accelerate computations. For example, the mapper can decide whether to exploit thread-level parallelism on the CPU with an OpenMP variant or leverage task parallelism between different operations by running the single-core CPU variant for tasks on different cores. Mappers can even experiment with different variant combinations at runtime and select the best choice based on profiling data.

In some cases, tasks must perform non-trivial indexing on their array arguments to support NumPy broadcasting semantics. We encapsulate this complexity inside the *accessors* that Legion uses to mediate access to data in physical instances. To handle broadcasting, we construct an affine transformation that describes how to remove broadcast dimensions from the index space of the computation. These are then folded into the strides stored internally by the accessor for each dimension. This eliminates the need to perform the transform explicitly when accessing data within the task body, and thus greatly simplifies the task variant implementation.

3.5 Limitations

While the design of Legate provides comprehensive support for NumPy semantics, including all indexing modes and broadcasting, our current implementation has certain limitations. Our prototype only supports a subset of NumPy types: specifically boolean, 16- , 32- , and 64- bit signed and unsigned integers, and 16- , 32- , and 64- bit floating point types. While these cover the bulk of important use cases, new types (including complex and compound types) can be added without fundamental changes to Legate's architecture.

We currently support only a subset of the full NumPy API (about 150 methods) which is nevertheless sufficient to handle many NumPy programs. Most of the unimplemented parts of the API are in more complex and less frequently used methods such as `np.einsum`, but no fundamental challenges to implementing any of these methods is apparent. New method implementations can create new partitions of arrays and launch one or more tasks to perform their computation without needing to be aware of how existing or future operations are implemented. For now, when an unimplemented method is encountered, Legate will fall back to the canonical NumPy implementation with a performance warning.

Finally, all file I/O in our implementation is done through the normal NumPy interface, with arrays then being passed in directly to Legate. This works well for single node cases, but leaves room for considerable performance improvements from parallel file I/O on distributed machines. We could address this with a drop-in replacement for the Python library `h5py` [4] to take advantage of Legion's support for parallel I/O with HDF5 files [8].

4 NUMPY-SPECIFIC MAPPING HEURISTICS

The Legate translation of NumPy programs to the Legion programming model results in a machine-independent program. As with all machine-independent Legion programs, Legate programs must be *mapped* onto the target machine. The Legion runtime contains no scheduling or mapping heuristics and instead delegates all decisions that impact the performance of a program through a call-back mapping interface [2]. To perform this mapping, Legate comes with a custom mapper that is a complete implementation of the Legion mapping interface. To maintain the ease-of-use of Legate, the existence of the Legate mapper is not made visible to end-users of Legate; the mapper leverages its domain specific knowledge of the NumPy programming model to customize mapping decisions for any target machine without requiring any input from Legate users. The heuristics employed by the Legate mapper achieve reasonable performance for most NumPy programs on many different kind of machines, thereby maintaining the ease-of-use property of NumPy for the vast majority of Legate users. As with all Legion mappers, the option to modify or replace the Legate mapper is still available for expert users that wish to tailor mapping decisions for a specific application and/or machine. However, we expect this to be an exceedingly rare use case in practice.

The Legate mapper currently employs a greedy algorithm for making mapping decisions for a NumPy program. The mapper makes decisions for the mapping of each operation in isolation without considering the mapping of other operations in the stream of tasks launched. There is nothing in the Legion mapping interface that precludes the Legate mapper from deploying a more holistic mapping algorithm in the future, however, for all the NumPy programs we have encountered so far, making mapping decisions for each operation individually has been sufficient to achieve good performance. As a result of this architecture, the remainder of this section will focus on how the Legate mapper makes mapping decisions for individual operations.

In order to make the discussion of the Legate mapper more concrete, we will consider the simple example of how Legate maps the operations from the inner loop (line 15) of the Jacobi solver example in Figure 1 onto a single NVIDIA DGX-1 machine with 8 GPUs. We will consider the case of an input matrix with 40K elements on a side. Please note that the dimensions of other arrays in the program are a direct function of the input matrix size.

When mapping an operation such as `np.dot` from the Jacobi example, there are many important mapping decisions for the Legate mapper to make. The mapper must first determine whether it wants to distribute this operation across many processors, or run it on a single processor. Next, if it decides to distribute the operation, it must choose how to partition the data for each of the input and output arrays for the operation. Finally, it must decide which processors to run different tasks on and where to place the data for each task. All of these decisions are intertwined because the choice over whether to distribute an operation will depend on the choice of how arrays are partitioned and the target processors for a given operation. The Legate mapper employs a general framework for making these decisions cooperatively based on the properties of the array arguments to operations.

Legate decides how to map an operation based on the shapes and sizes of the arrays being processed by the operation with the goal of optimizing achieved memory bandwidth utilization. The choice of optimizing for memory bandwidth utilization derives from an important property of NumPy: most of its operations are memory-bandwidth limited due to their low arithmetic intensities. Consequently, it is important that we optimize our utilization of this critical resource. An important outcome of this decision is that the Legate mapper will choose to assign tasks to processors with higher memory bandwidths, such as GPUs, when they are available. Preferences for GPUs and other accelerators are not inherent in the Legate mapper, but instead derive from analyzing the output of a simple query of the Legion machine model [2] for the processors with the highest available memory bandwidth.

Based on these metrics, the Legate mapper will decide to map the operations in the Jacobi example onto the GPUs of our target DGX boxes. However, the mapper does not yet know whether or how to distribute the computations over the available GPUs. Recall from Section 3.3, that the kind (single or index space) and size of task launch to perform an operation is derived from the natural partitioning (or lack thereof) of the key array argument. The choice of distribution of an operation to specific processors derived directly from natural partitioning of the key array argument as it is the largest array argument for an operation and therefore the one that we most strongly prefer to use with its natural partition to avoid unnecessary data movement. In the case of `np.dot`, the key region is the matrix `R` as it is considerably larger than the input vector array or the output vector array. Whatever the natural partitioning of `R` is then the mapper will assign tasks to processor(s) with direct affinity to the memory or memories containing the data for `R`. For operations such as `np.sub` and `np.div` where the arguments all have the same size, the output array is the key region.

In a naïve implementation of Legate, natural partitions of arrays would evenly divide each array among the target processors of a machine. There are two potential problems with this approach: first, over-decomposition of small arrays across too many processors, and second, over-decomposition of large arrays into too many tiles. In the first case, the overheads of distribution and execution on throughput-optimized processors (e.g. GPUs) can accumulate and

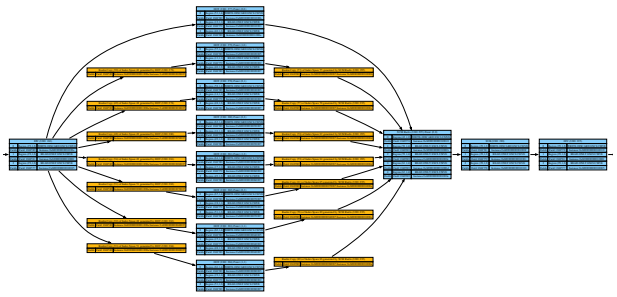

Figure 7: Dataflow graph for one iteration of the Jacobi solver on a DGX-1. Blue boxes are tasks corresponding to NumPy operations and yellow boxes are copies of data between GPU memories.

result in slower execution than running on a few or even a single processor. In the second case, overheads of meta-data management and distribution costs can result in poor performance. While the Legate mapper does not directly control the computation of natural partitions for arrays, it can prevent these cases from occurring by judiciously selecting the values for two critical tunable parameters that place bounds on how natural partitions are computed. Recall from Section 3.3 that the mapper must choose a minimum tile size for an array, and it must also choose a maximum number of tiles to create. By ensuring that the minimum tile size is not excessively small, the mapper guarantees that small arrays are not distributed across too many processors. Furthermore, by specifying the maximum number of tiles that can be made, the mapper ensures that large arrays are not over partitioned.

To select a minimum tile size, the mapper makes educated guesses about what granularity of data is needed to achieve peak bandwidth on different kinds of processors. Currently the mapper makes this guess by requiring that at least every *thread* on the target machine process at least one kilobyte of data from an array. On a CPU, a thread is either one physical core or one hyperthread if hyperthreading is enabled. For the kind of Xeon CPUs with 20 cores used in Section 6, this corresponds to needing 40 KB of data per CPU socket per tile with hyperthreading. On a GPU, we approximate a thread with one warp which is effectively a vectorized thread on a streaming multiprocessor; the total number of threads for a GPU is then the number of streaming multiprocessors (SMs) multiplied by half the maximum number of warps that can be resident on an SM. Note that we use half the maximum warp count because it is unusual to fill a streaming multiprocessor with warps due to register pressure. For the GPUs used in Section 6, this corresponds to needing approximately 2 MB of data per GPU for a tile. While the amounts of memory needed to saturate memory bandwidth for current processors are baked into the mapper at the moment, it would not be difficult for a more sophisticated mapper to profile these constants at start-up time for any past or future machine.

The effects of these parameters can clearly be seen in Figure 7 which shows the dataflow graph for one iteration of the Jacobi solver on a DGX-1 machine for a $40K \times 40K$ matrix. The `np.dot` operations have been distributed across all 8 GPUs (parallel blue tasks) because it is easy to create tiles for the matrix that are at least 2 MB in size. However, the point-wise operations for the vector are all done on a single GPU because the 40K vectors only require 160 KB which is not enough to saturate the memory bandwidth of a single GPU, let alone eight of them. Consequently, the input vector is copied (yellow boxes) to each of the GPUs and the output partial sums are copied back to one GPU over NVLink to perform the sum reduction and later point-wise operations `np.sub` and `np.div`.

The Legate mapper selects the maximum-number-of-tiles tunable value by examining the kind of target memories and how many of them there are in the machine. In most NumPy programs, arrays are dense, and either large enough that they need to be distributed across many nodes or small enough that they can easily fit in a single node. Therefore very little dynamic load balancing is necessary and arrays can be partitioned into large chunks that are evenly distributed. Consequently, Legate usually selects this value to be a small integer multiple of the number of GPU memories or CPU NUMA domains in the machine. This can also be seen in the Jacobi example from Figure 7. The $40K \times 40K$ matrix is 6.4 GB and could be made into 3200 tiles of 2 MB each. However, the Legate mapper only makes 8 tiles, one for each of the 8 GPUs when distributing the `np.dot` operation.

When a Legate mapper is later asked to map tasks for a specific operation, it uses the key array to guide the distribution of tasks. For each task that must be mapped, the mapper assigns that task to the processor with the best affinity (highest bandwidth) to the memory where the most recent physical instance of the (sub-) region for the key array exists. This works identically for both single tasks and the point tasks for index space task launches. After determining which instance to use for the key array, and therefore target processor, the mapper then either finds or creates physical instances for the other array arguments. In general, the Legate mapper prefers to also have these instances be located in the same memory as the key array, with the highest bandwidth to the processor where the task will run. However, under tight memory constraints where memories are mostly full, the Legate mapper will settle for using an instance in a memory that at least has the ability to handle loads and stores directly from the chosen processor. In these cases, it's not uncommon for the Legate mapper to use instances stored in multiple different GPU memories for a task and rely on loads and stores traveling over PCI-E or NVLink interconnects.

In general, the Legate mapper achieves very good out of the box performance on a wide range of NumPy programs. It does especially well on programs with large working sets that need to be distributed across large machines. It is currently less optimized for the case where NumPy programs generate many small arrays that need to be distributed around the machine without any partitioning. However, if a user is dissatisfied with the performance of Legate, they can directly modify the Legate mapper to improve their performance without needing to change any code from the original NumPy program. Most importantly, this can be done without needing to be concerned with breaking the correctness of the translation from NumPy to Legion performed by Legate, or by the mapping onto the hardware performed by Legion. While we do not expect this to be a common case, for expert users that demand peak performance, it is a novel means of recourse for addressing the shortcomings of scheduling and mapping heuristics not afforded to them by any other NumPy programming system of which we are aware.

5 CONTROL REPLICATION

The most important part of Legate that enables it to scale well is its ability to leverage a feature in the Legion runtime system called *control replication*. All Legion programs begin with a single top-level task. Without control replication, whichever node initially begins running this task can quickly become a sequential bottleneck as it attempts to distribute sub-tasks to other nodes in the machine. No matter how small the runtime overhead is for analyzing and launching tasks, at some node count it is guaranteed to stop scaling. This sequential bottleneck is not specific to Legion, but inherent in any system with a centralized controller node [23, 30].

As its name suggests, control replication directly addresses this problem by running multiple copies of a task (usually the top-level task) on different processors, with all the copies of the task functioning collectively with the same behavior as the original logical task. Traditionally, control replication in Legion has referred to a static program transformation performed by compilers that target the Legion runtime[25]. More recently, the Legion runtime has introduced support for a dynamic version of control replication.

Dynamic control replication in Legion parallelizes Legion's dependence analysis so that each node participating in the execution of a control-replicated task becomes responsible for analyzing a sub-set of tasks launched as dictated by the mapper, and figures out how to hook up dependences, communication, and synchronization with any tasks owned by other nodes. Most importantly, the runtime does all this transparently without requiring any changes to the original top-level task. In the case of Legate, this ensures that users do not need to make any changes to their NumPy program, and no changes need to be made to the Legate translation from NumPy to Legion. Consequently, with control replication, Legate ensures that the scalability of a NumPy program becomes dictated by its inherent communication patterns and not limited by any aspects of the underlying programming system.

Dynamic control replication works by executing multiple copies of a task, called *shards*, on different processors across the system. The number and placement of shards are chosen by the mapper; the Legate mapper will generally choose one shard per physical node. Shards all run the same code as though they were each the only copy of the task. However, when shards call into the Legion runtime to perform operations such as creating logical regions or launching tasks, the runtime is aware that the shards are all operating collectively as a single logical task and behaves accordingly. For example, when creating logical regions, each shard receives back the same name for a newly created region. Similarly, when launching a single sub-task, Legion ensures that only a single shard is ultimately responsible for executing the sub-task. Figure 8 provides an example depicting how this works. Logically, the top-level task launches sub-tasks that Legion analyzes for dependences and constructs a dependence graph. With control replication, both shards execute the same top-level task and construct the same dependence graph. The decision of which sub-tasks are executed by which shards (i.e. not blacked-out) is made by the mapper via its choice of *sharding functions*. Sharding functions determine which shard will be responsible for executing a task in the case of single task launches, or for executing specific points in an index space task launch. Legion then automatically hooks up cross-shard dependences.

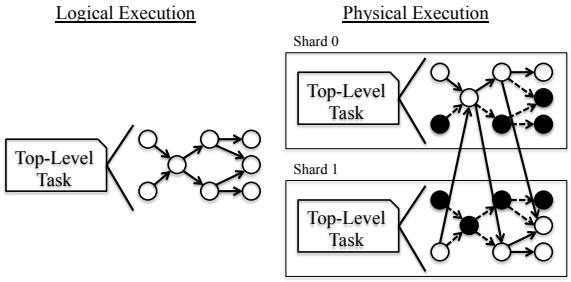

Figure 8: A depiction of control replication in Legion. On the left is the programmer's view of a task that launches off sub-tasks which Legion analyzes to construct a dynamic dependence graph. On the right is how this is executed in practice, with multiple shard copies of the control replicated task launching their own subtasks and Legion hooking up dependences where necessary across the shards.

While the details of how control replication is implemented in Legion are beyond the scope of this paper, it is important to understand how Legate leverages control replication to allow NumPy programs to scale to large machines. For all runs of a NumPy program on multiple nodes, the Legate mapper directs Legion to control replicate the top-level task so that there is a copy running on each node in the system. As mentioned in Section 3.1, each of these copies of the top-level task then executes the normal Python program and Legate translation from NumPy to Legion proceeds as normal. When calls are made into the Legion runtime to launch tasks, the runtime understands that these calls are being performed collectively and executes them as such. The runtime also asks the Legate mapper to select sharding functions for controlling which tasks are handled by different shards. The Legate mapper chooses sharding functions based on locality. The functions that the mapper chooses assign tasks to shards where the mapper knows at least some part of the data needed for those tasks resides.

For control replication to work correctly, the Legion runtime must see the same order of calls into the runtime from each shard. Since the Legate translation to Legion is deterministic, this requires the original Python program to make the same order of NumPy API calls with the same arguments. In general, most Python programs trivially satisfy this requirement, but there are a few ways it can be broken. For example, any control flow that depends on values generated or derived from a non-NumPy random number generator can lead to a different ordering of NumPy API calls on different shards. Similarly, iterating over an un-ordered data structure that depends upon Python hashing functions such as a `set` or `dict` and performing NumPy API calls inside the loop could result in a permuted set of NumPy API calls on different shards as the hashes for different objects could be different across shards. To detect such cases, Legate provides a special execution mode which validates that a program's execution is consistent with the requirements for control replication and reports when it is not. This validation is performed by confirming that all NumPy API calls occur in the same order and with the same arguments across all shards. In practice, we have yet to encounter a NumPy program that required any modifications to be used with control replication.

6 PERFORMANCE RESULTS

In order to evaluate Legate, we tested it on a set of example NumPy programs written by ourselves and others from both machine learning and more traditional HPC workloads. While these programs are fairly small in size (none are more than 100 lines of NumPy), they exercise a surprising number of communication and computation patterns that stress systems in different ways. In part, this illustrates the power of NumPy: many sophisticated algorithms can be expressed easily and succinctly.

For our performance analysis, we measure Legate's performance when running in both GPU-centric and CPU-only modes; CPU-only mode provides a useful comparison against systems that can only run on CPUs. We compare Legate against several different systems with NumPy or NumPy-like interfaces. On a single node, we compare Legate against both the canonical NumPy implementation shipped with most versions of Python, as well as a special version of NumPy developed by Intel, which has MKL acceleration for some operations [6]. We also compare against CuPy [19] on a single GPU because CuPy provides a drop-in replacement interface for NumPy on just one GPU; CuPy also supports multi-GPU execution, but, unlike Legate, only does so with extensive code changes.

For a multi-node comparison, we evaluate Legate against equivalent programs written to target the Dask array library [23]. Dask is a popular task-based runtime system written entirely in Python that supports both parallel and distributed execution of programs. Additionally, Dask supports several different libraries on top of the underlying tasking system, such as its array and dataframe libraries. In particular, the array library presents an interface almost identical to NumPy, making it an obvious choice for developers looking to port their programs to a system that can scale beyond a single node.

The one extension to the NumPy interface that Dask makes is the need to select a *chunk* tuple when creating arrays. The chunk tuple describes how the array should be partitioned along each dimension of the array. Users can pick chunks explicitly, or they can pass an `"auto"` string and Dask will use internal heuristics to select the chunk tunable for them. For all of our experiments, the only difference between the code for the NumPy/Legate version of a program and the Dask version is the specification of chunks. For each experiment, we will have two different Dask comparisons: one running the code with `"auto"` chunks to demonstrate out of the box performance, and a second with hand tuned chunk sizes to demonstrate the best possible Dask performance.

All our experiments are run on a cluster of 32 NVIDIA DGX-1V nodes. Each node has 8 Tesla V100 GPUs with 16 GB of HBM2 memory per GPU. GPUs are connected by a hybrid mesh cube topology with a combination of NVLink and PCI-E connections. Each node also has two 20-core Intel Xeon E5-2698 CPUs with hyperthreading enabled and 256 GB of DDR4 system memory. Nodes are connected by an Infiniband EDR switch with each node having 4 Infiniband 100 Gbps NICs.

For each experiment, we perform a weak-scaling test to emulate the kind of usage that would be expected of a scalable NumPy system: users would progressively require more hardware to handle increasingly large data sizes. To compare performance between CPUs and GPUs, we keep the per-socket problem size constant. This means that GPU-centric curves will extend out four times

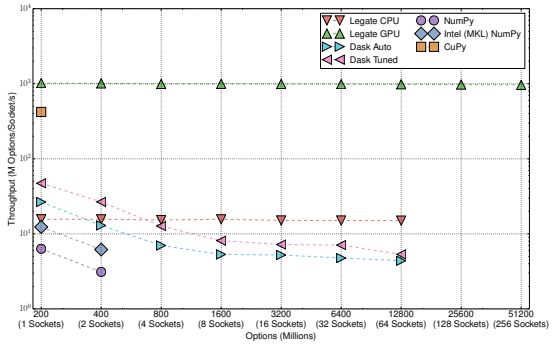

Figure 9: Weak-scaling throughput of Black-Scholes.

farther than CPU-only curves because there are 8 GPUs per node versus only 2 CPU sockets per node. Each data point is the result of performing 12 runs, dropping the fastest and slowest runs, and then averaging the remaining 10 runs. We always plot throughput on a log-log axis due to the extreme differences in performance between the various systems.

Our first experiment is with Black-Scholes option pricing. This is a trivially parallel application where every option may be priced independently. Therefore, we expect to see perfect weak scaling. With both modes of Legate we see exactly this result in Figure 9: the GPU version of Legate maintains a throughput of 1 billion options per GPU per second out to 256 GPUs, while the CPU version maintains a throughput of 11 million options per CPU socket per second out to 64 sockets. Using control replication, Legate does not have a centralized scheduler node that can limit scalability. On the contrary, Dask initially starts off better than the CPU version of Legate on small socket counts because of its ability to perform operator fusion to increase computational intensity. However, it has to coarsen some of its tasks using larger chunk factors to reduce the overheads of distributing tasks to multiple nodes from its single controller node, limiting its scalability. There are no BLAS operations that can leverage both sockets in Black-Scholes so having an extra socket does not aid the performance of canonical NumPy or Intel Python. Note that Legate's GPU-centric version is faster than CuPy on a single GPU because Legion's dependence analysis finds task parallelism between operators which improves GPU utilization by running concurrent kernels.

Our next experiment is with the Jacobi solver from Figure 1. Performance results can be seen in Figure 10. The primary operation for the Jacobi solver is a square matrix GEMV computation. Legate has nearly perfect weak scaling with its CPU-only implementation. The GPU-centric version of Legate does not scale as well due to some distributed communication overheads for reducing and broadcasting data, but its throughput still remains higher than the CPU-only version even out to 256 GPUs. In the future, Legate could eliminate these overheads by recognizing the repetitive nature of the solver and leveraging Legion tracing [12]. Dask does not scale nearly as well as Legate. Dask loses significant performance going from 2 sockets (1 node) to 4 sockets (2 nodes). We surmise that this is due to the cost of data movement showing up. Dask then continues to slow down with increasing node count as the centralized controller node becomes a bottleneck. Note that Intel Python starts out faster than normal NumPy on a single socket because GEMV is

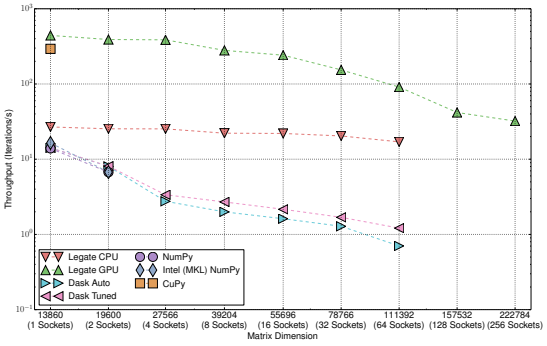

Figure 10: Weak-scaling throughput of a Jacobi solver.

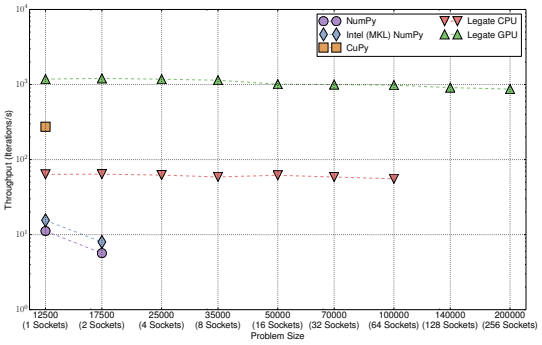

Figure 11: Weak-scaling throughput of a 2D stencil.

an operation that can be accelerated by MKL. However, it falls back to the same performance as NumPy on two sockets due to the cost of moving data between NUMA domains. The Legate mapper on the other hand is highly NUMA-aware and does a better job placing data in NUMA domains from the start, which results in less data movement between NUMA domains and therefore better scaling. Legate's CPU-only version is also better than Intel Python on a single socket because the Legate mapper chooses to use the OpenMP task variants for all operations, whereas Intel Python parallelizes only the GEMV operation.

Figure 11 contains results from running an extended version of the 2-D stencil computation from Figure 2. Legate's creation of detailed region trees, like those in Figure 5 for sub-views, allows Legion to infer the nearest-neighbors communication pattern that weak scales for both CPU and GPU operations. Note that there are no Dask lines on this graph because Dask does not permit assignments into sub-array views like those on line 11 of Figure 2 or during boundary condition assignments in the full program.

In order to test a few machine learning workloads, we implemented both linear and logistic regression in NumPy. Since their performance curves are very similar we only show the results for logistic regression in Figure 12. Both regressions implement batch normal gradient descent which requires keeping all the data points in memory at all times. It also requires a full reduction of weight gradients across all nodes followed by a broadcast of the update to each node. Legate achieves nearly perfect weak scaling for both linear and logistic regression on CPUs. Running with GPUs, Legate weak scales out to 32 GPUs, but then begins losing performance as the additional overheads of the very small reduction tree tasks

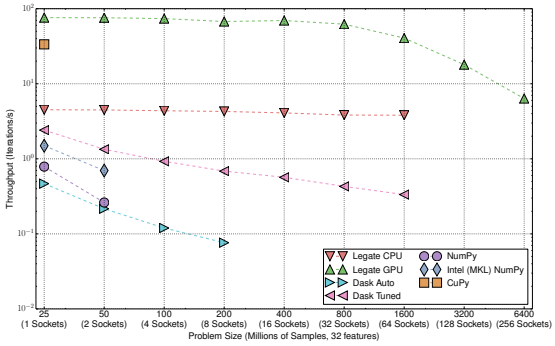

Figure 12: Weak-scaling of logistic regression.

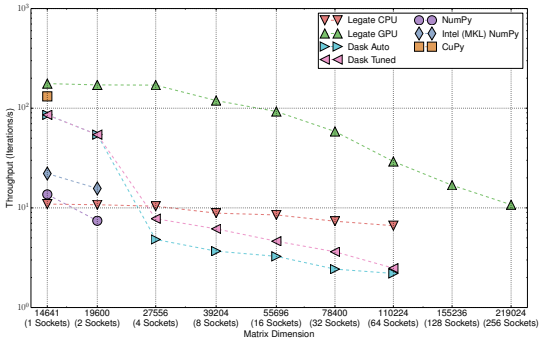

Figure 13: Weak-scaling of a preconditioned CG solver.

begin to accumulate. Dask has difficultly scaling and performance continues to decline as it reaches 32 nodes. Note that in these cases there is more than an order of magnitude in performance difference between the `auto` and `tuned` Dask performance curves. This highlights the importance of manual tuning on systems like Dask. Legate, in contrast, insulates the user from such details and automatically determines the best partitioning to be used for each operation. Legate also outperforms CuPy for logistic regression on a single GPU by discovering task parallelism across operators that enables multiple kernels to execute on the device simultaneously.

Lastly, we implemented a full conjugate gradient solver both with and without preconditioning in NumPy. Adding support for a preconditioner is usually a complex change for a solver, but in NumPy it only requires an additional eight lines of code. The performance results for these examples are also very similar so we only show the results from our experiments with the preconditioned conjugate gradient solver in Figure 13. CPU-only Legate weak scales very well, while the GPU-centric version has kernels that run fast enough to expose several Legion analysis overheads that could be fixed in the future by tracing [12]. Dask loses significant performance when moving from single node (2 socket) to multi-node (4 socket) execution. Its performance continues to decline as the node count increases because of its centralized scheduler. Note that in these examples, Intel Python outperforms Legate's CPU-only execution on both single and dual sockets. MKL is faster than the OpenBLAS GEMV routines that Legate uses and allows Intel Python to overcome its initially poor placement of data in NUMA domains.

7 RELATED WORK

Providing a drop-in replacement for NumPy is a very common approach to improving the performance of NumPy programs. Traditionally, these approaches have fallen into one of three categories. First, are systems that accelerate parts of NumPy on a single node, such as the previously mentioned Intel Python with MKL support [6]. Second, are systems that lazily evaluate NumPy programs to construct an intermediate representation of the program prior to performing operator fusion and specialization for particular kinds of hardware like GPUs. Third are systems that attempt to distribute NumPy or NumPy-like programs across distributed compute clusters. Most drop-in NumPy replacements fall into the second category of systems, while Legate and several others fall into the third category. We discuss each in turn.

One of the most popular ways of accelerating NumPy code today is with CuPy [19]. CuPy is a (mostly) drop-in replacement for NumPy that will off-load operations onto a single GPU. As was mentioned earlier, CuPy supports multi-GPU execution, but not with a drop-in NumPy interface; instead users must manually manage multi-GPU data movement and synchronization.

Another popular drop-in replacement for NumPy that accelerates NumPy programs on a single GPU is Bohrium [10]. Bohrium performs lazy evaluation of NumPy programs before attempting to fuse operators and generate optimized code for both CPUs or a GPU. The original Bohrium implementation also had support for distributed CPU clusters, but that option no longer appears to be supported and, unlike Legate, it could not target a cluster of GPUs.

Grumpy is a similar system to Bohrium that lazily evaluates NumPy programs and then performs fusion of NumPy operators into optimized multi-core CPU kernels using LLVM and single GPU kernels targeting the NVIDIA NVVM compiler [22].

JAX is a drop-in replacement for NumPy with additional functionality for performing auto-differentiation, vectorization, and JIT-ing of fused operations [7]. JAX uses the XLA compiler which can target both CPUs or a single GPU, but supports a restricted IR which limits support for general NumPy programs. JAX can run in a multi-node setting in a SPMD style when it is coupled with a collective library, such as NCCL, but it does not attempt to handle more sophisticated partitions of data and distributions of tasks the way that Legate does.

Weld is a drop-in replacement for NumPy and Pandas that performs lazy evaluation and fusion of common operators in order to improve computational efficiency [20]. Weld can currently target both CPUs and single GPUs, but cannot target clusters of GPUs.

As we discussed in Section 6, Dask is a popular Python library for parallel and distributed computing. Dask has both a low-level tasking API as well as several higher-level APIs built on top for supporting arrays and dataframes [23]. The high-level array interface is very similar to the NumPy interface with the exception of needing to specify chunk factors. While Dask can support GPU computations inside of tasks in its low-level API, it is difficult for programmers to keep data in GPU memories across task invocations. Dask's high-level array-based API is also not GPU accelerated.

NumPywren is a high-level interface for performing dense linear algebra operations at scale in the cloud [24]. Due to the limitations of the server-less lambda functions that are employed, NumPywren is limited in the kinds of optimizations that it supports for distributed computing and it is not able to leverage GPUs.

Arkouda is a NumPy-like system built on top of the Chapel programming language and is capable of executing programs interactively at scale [14]. Unlike Legate, Arkouda is not a drop-in replacement for NumPy as it needs additional type information to translate code to Chapel. To the best of our knowledge Arkouda does not provide any support for GPU acceleration of programs.

GAiN is a distributed implementation of the NumPy interface on top of the Global Arrays programming model [3]. Using Global Arrays GAiN was able to achieve good scalability on large CPU clusters but did not support GPUs.

Spartan is a drop-in replacement for NumPy that also lazily evaluates NumPy programs similar to Grumpy and Bohrium but also distributes them around a cluster [5]. The IR of Spartan is limited to just a few common operators which reduces the generality of the kinds of NumPy operators that they support. Spartan also does not target clusters containing GPUs.

Phylanx is a developing project to implement NumPy on top of the HPX runtime [21]. HPX is known to run at scale on large clusters, but it is unclear whether Phylanx will support GPUs.

Ray is a new distributed task-based runtime system for Python [15]. While the design of the schedulers are very different, both Legate and Ray have distributed schedulers and have native support for GPUs. Ray's scheduler is only a two-level distributed scheduler, however, while the Legion scheduler used by Legate works to arbitrary nesting depths of tasks. Ray also does not support a NumPy-like interface for array computing.

Finally, the implementation of Legate relies heavily upon the Legion programming model and runtime system [2, 8, 12, 26–28]. The fully dynamic ability of Legion to create and partition data, launch tasks, and analyze tasks for dependences at runtime is essential to supporting arbitrary NumPy programs with data dependent behavior. Furthermore, Legion's support for multiple and arbitrary partitions of data in conjunction with the sequential semantics of the programming model is crucial to reducing the complexity of Legate's implementation. The Legion mapper interface makes Legate portable across different machines by decoupling performance decisions from the functional specification of a program.

8 CONCLUSION

We have described the design of Legate, an accelerated and scalable implementation of NumPy, that allows users to harness the computational throughput of large-scale machines with unmodified NumPy programs. Legate implements a novel methodology for translating NumPy-based programs to the Legion runtime system and a separate set of domain-specific heuristics for intelligently mapping such programs to target machines. To avoid the sequential scaling bottlenecks of having a single control node, typical of distributed systems with an interpreted front-end, Legate leverages support for dynamic control replication in the Legion runtime. As a result of efficient translation to Legion, effective mapping strategies, and control replication, our Legate implementation enables developers to weak-scale problems out to hundreds of GPUs without code rewrites. Going forward, we think Legate serves as the

foundation for porting a large class of Python libraries on top of Legion, all of which will be able to naturally compose because they all will share a common data model and runtime system capable of analyzing dependences and data movement requirements across library abstraction boundaries.

ACKNOWLEDGMENTS

Work on control replication in the Legion runtime that made Legate possible was funded by Los Alamos National Laboratory through the Department of Energy under Award Number DENA0002373-1 as part of the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Jared Hoberock ported many of the existing Legate task variants to use Agency, both significantly simplifying the code and making it easier to maintain. Special thanks goes to Josh Patterson, Peter Entschev, Felix Abecassis, Keith Kraus, and especially Matthew Rocklin of the NVIDIA RAPIDS team for their help in setting up and running Dask correctly. Alex Aiken and Sean Treichler both provided helpful comments and feedback on earlier drafts of this paper. We also thank Peter Hawkins for providing detailed information on JAX.

REFERENCES

[1] Agency 2019. Agency: Execution Primitives for C++. https://github.com/agency-library/agency.
[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Supercomputing (SC)*.
[3] Jeff Daily and Robert R Lewis. 2010. Using the Global Arrays Toolkit to Reimplement NumPy for Distributed Computation. *PROC. OF THE 9th PYTHON IN SCIENCE CONF* (01 2010).
[4] h5py 2019. h5py. https://www.h5py.org/.
[5] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. 2015. Spartan: A Distributed Array Framework with Smart Tiling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 1–15. https://www.usenix.org/conference/atc15/technical-session/presentation/huang-chien-chin
[6] Intel 2019. IntelPy. https://software.intel.com/en-us/distribution-for-python.
[7] JAX 2019. JAX: Autograd and XLA. https://github.com/google/jax.
[8] Z. Jia, S. Treichler, G. Shipman, M. Bauer, N. Watkins, C. Maltzahn, P. McCormick, and A. Aiken. 2017. Integrating External Resources with a Task-Based Programming Model. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 307–316. https://doi.org/10.1109/HiPC.2017.00043
[9] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. http://www.scipy.org/
[10] Mads RB Kristensen, Simon AF Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. 2013. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. (2013).
[11] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162
[12] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. 2018. Dynamic Tracing: Memoization of Task Graphs for Dynamic Task-based Runtimes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 34, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291702
[13] Wes Mckinney. 2011. pandas: a Foundational Python Library for Data Analysis and Statistics. *Python for High Performance and Scientific Computing* (01 2011).
[14] Michael Merrill, William Reus, and Timothy Neumann. 2019. Arkouda: Interactive Data Exploration Backed by Chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop (CHIUW 2019)*. ACM, New York, NY, USA, 28–28. https://doi.org/10.1145/3329722.3330148
[15] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *CoRR* abs/1712.05889 (2017). arXiv:1712.05889 http://arxiv.org/abs/1712.05889
[16] NumPy 2019. NumPy v1.16 Manual. https://docs.scipy.org/doc/numpy/.
[17] NVIDIA 2019. cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html.
[18] NVIDIA 2019. Tensor Cores. https://www.nvidia.com/en-us/data-center/tensorcore/.
[19] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. http://learningsys.org/nips17/assets/papers/paper_16.pdf
[20] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2017. Weld: Rethinking the Interface Between Data-Intensive Applications. *CoRR* abs/1709.06416 (2017). arXiv:1709.06416 http://arxiv.org/abs/1709.06416
[21] Phylanx 2019. Phylanx: A Distributed Array Toolkit. http://phylanx.stellar-group.org/.
[22] Mahesh Ravishankar and Vinod Grover. 2019. Automatic acceleration of Numpy applications on GPUs and multicore CPUs. *CoRR* abs/1901.03771 (2019). arXiv:1901.03771 http://arxiv.org/abs/1901.03771
[23] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 130 – 136.
[24] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *CoRR* abs/1810.09679 (2018). arXiv:1810.09679 http://arxiv.org/abs/1810.09679
[25] Elliott Slaughter, Wonchan Lee, Sean Treichler, Wen Zhang, Michael Bauer, Galen Shipman, Patrick McCormick, and Alex Aiken. 2017. Control Replication: Compiling Implicit Parallelism to Efficient SPMD with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/3126908.3126949
[26] S. Treichler, M. Bauer, and Aiken A. 2014. Realm: An Event-Based Low-Level Runtime for Distributed Memory Architectures. In *Parallel Architectures and Compilation Techniques (PACT)*.
[27] S. Treichler, M. Bauer, and A. Aiken. 2013. Language Support for Dynamic, Hierarchical Data Partitioning. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
[28] S. Treichler, M. Bauer, Sharma R., Slaughter E., and A. Aiken. 2016. Dependent Partitioning. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
[29] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 25, 12 pages. https://doi.org/10.1145/2503210.2503219
[30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301