Title
NumS: Scalable Array Programming for the Cloud

Author
Elibol, Huseyin Melih

Publication Date
2022

Peer reviewed|Thesis/dissertation

NumS: Scalable Array Programming for the Cloud

by

Huseyin Melih Elibol

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Michael I. Jordan, Co-chair
Professor Ion Stoica, Co-chair
Professor Kenneth Goldberg

Spring 2022

NumS: Scalable Array Programming for the Cloud

Abstract

NumS: Scalable Array Programming for the Cloud

by

Huseyin Melih Elibol

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael I. Jordan, Co-chair

Professor Ion Stoica, Co-chair

Scientists increasingly rely on Python tools to perform scalable distributed memory array operations using rich, NumPy-like expressions. Existing solutions achieve sub-optimal performance on numerical operations and training of machine learning models by relying on dynamic scheduling provided by task-based distributed systems. This can lead to performance problems which are difficult to address without in-depth knowledge of the underlying distributed system. In particular, generalized linear models are difficult to scale given their reliance on element-wise array and basic linear algebra operations.

In this thesis, I present these problems in terms of scalable linear algebra and automatic parallelization of Python. The solutions presented seamlessly scale the NumPy API and generalized linear models on task-based distributed systems. Our overall solution is presented in three primary parts: (1) An approach to parallelizing generalized linear models (GLMs) using blocked matrix operations. (2) The open source library NumS, an implementation of these ideas for the NumPy API optimized for the distributed system Ray. (3) Formal syntax and semantics for automatic parallelization of basic Python and linear algebra operations.

Our primary contribution is NumS, a modular Python-based distributed numerical array library optimized for Ray. Load Simulated Hierarchical Scheduling (LSHS), the scheduler developed for NumS, is capable of attaining communication lower bounds on some common numerical operations. Our empirical study shows that LSHS enhances performance on Ray by decreasing network load by a factor of 2×, requiring 4× less memory, and reducing execution time by 10× on the logistic regression problem. In a comparison to related solutions, LSHS achieves up to 2× speedup on logistic regression compared to Dask ML and Spark's MLlib on a terabyte of data.

To Isla

Toward technologies applied to improving your future.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, thank you to my advisors, Ion and Mike, for providing me the space and autonomy to succeed, and guidance whenever I needed it. Thank you to my dissertation committee, and Alvin Cheung, for thoughtfully and critically evaluating my thesis. NumS would not be as successful as it is today without the help of Vinamra Benara, Samyu Yagati, Lianmin Zheng, Daniel Zou, Brian Park, Priyans Desai, Balaji Veeramani, Aditya Bose, Tynan Sigg, and Mohamed Elgharbawy.

Thank you to the original Ray team, including Stephanie Wang, Alexey Tumanov, Robert Nishihara, Philipp Moritz, Eric Liang, Richard Liaw, and Zongheng Zang for the exciting collaborations and myriad of opportunities early on in my PhD. Thanks to Amazon Core AI for supporting my research, and thank you to Intel researchers Todd Anderson and Padmanabhan Pillai for critically evaluating NumS, and providing paths to improving its performance.

Thank you to my old friend, Michael Matthews, for your unconditional support. Finally, and most importantly, thank you to my wife, Noelle Fogg Elibol, for supporting me and our family for the past five years, and my daughter, Isla Nevim Elibol, for the joy and meaning you bring to my life.

Part I

Scalable Blocked Linear Algebra Operations.

Chapter 1

Introduction

Within the past decade, the proliferation of data science and machine learning applications has rapidly increased, and the demand to scale these methods continues to grow. These applications typically consist of executing several independent analyses on multi-dimensional numerical arrays, such as administrative health data [14, 26], genomics [20], climate [39], and physics data [3]. These applications are written using numerical array operations on multi-dimensional numerical arrays. Python [40] has emerged as the language of choice for these types of applications. Within this domain, Python programmers use a combination of NumPy [25] and scikit-learn [28] for numerical array, statistical, and machine learning operations. NumPy popularized conventions around expressing multi-dimensional numerical array operations in Python, and scikit-learn popularized a framework for machine learning model training and inference on NumPy data.

1.1 Shared Memory Solutions

However, NumPy primarily provides serially executing operations, except for linear algebra operations. Linear algebra operations are exposed via BLAS [7] and LAPACK [4] implementations installed on the operating system. BLAS provides an interface for parallel execution of linear algebra operations, but assumes a *shared-memory architecture*. These architectures typically consist of one or more multi-core processing unit(s), all of which have access to the same random access memory. Within the context of cloud-based distributed systems, a single instance of this kind of architecture is referred to as a *node*. Parallel algorithms on shared memory provide concurrent execution of code, but are unable to *efficiently* scale beyond a single *node*. In theory, *virtual shared memory* could be used to scale any parallel shared memory algorithm beyond a single node. This method exposes a shared memory abstraction on top of a distributed memory system. High performance BLAS libraries are designed for specific architectures. Typically, these implementations optimize performance by implementing algorithms which exploit architecture-specific cache hierarchies [2], which makes them either incompatible or useless on virtual shared memory.

Scikit-learn models are written against the NumPy API, resulting in similar limitations to NumPy. Ensemble methods, such as the random forest, which fit many small independent models, can be trivially scaled to multiple nodes, but these are only a small subset of the greater set of models made available by the library. In particular, scaling generalized linear models (GLMs) requires parallelization of *NumPy operations themselves*. We present our solution to this problem in Chapters 2 and 3.

1.2 Distributed Memory Solutions

Parallel algorithms written to operate on *distributed memory* typically consist of multiple processes which are distributed across multiple nodes. Distributed memory parallel algorithms are able to exceed the processor and memory capacity of a single node by networking multiple instances of such nodes into a *multi-node cluster*. *Cloud services*, such as AWS and Azure, provide multi-node clusters for distributed memory computing.

We now turn our attention to existing solutions in this space. We observe that high-performance solutions require expertise beyond the programming knowledge required to write numerical algorithms in NumPy, and solutions which expose a NumPy-like API achieve sub-optimal performance.

Solutions such as ScaLAPACK [8] and SLATE [16] are very similar to one another, but SLATE is currently viewed as the successor of ScaLAPACK. We view SLATE as state-of-the-art in terms of distributed memory linear algebra. SLATE is built on the message passing interface (MPI) [12] and implements algorithms that are optimized for linear algebra operations. These specialized libraries are state-of-the-art for scalable linear algebra, but they do not support the NumPy API, making them inaccessible to an increasing number of scientists who are adopting Python. They also do not handle out-of-memory issues and fault tolerance.

Dask Arrays [32] implements parallel numerical array operations (Chapter 3, Section 3.3) by constructing discrete task graphs which represent the desired computation and schedules these tasks dynamically. While this decoupling of algorithm from scheduling has desirable software design properties, the loss of information to the scheduler leads to sub-optimal data and operator placement. In particular, when data placement is not optimized for numerical array operations, unnecessary communication among processes is often required in order to carry out basic operations, such as element-wise addition and vector dot products. In general, any scheduling algorithm which dynamically schedules distributed numerical array operations as discrete task graphs is susceptible to sub-optimal performance.

The Dask ML [32] library provides several machine learning models. The optimization algorithms written for these models frequently execute code on the driver process. The library is written using Dask's array abstraction, which achieves sub-optimal performance on a variety of linear algebra operations previously mentioned.

Spark's MLlib [22] is a library for scalable machine learning. MLlib depends on Breeze [10], a Scala library that wraps optimized BLAS [7] and LAPACK [4] implementations for numer-

ical processing. Breeze provides high-quality implementations for many common machine learning algorithms that have good performance, but because it relies on Spark primitives, it introduces a learning curve for NumPy users.

In this thesis, we provide solutions to all of these problems. The remaining sections in this chapter summarize the structure of this thesis.

1.3 Scalable Linear Algebra for Second Order Optimization of Unconstrained Convex Problems

Chapter 2 develops an intuition to our approach to distributed memory linear algebra. It provides a path to parallelizing basic linear algebra operations, and their application to second order methods for unconstrained convex problems. This directly solves the scalability problem of generalized linear models using fast second-order optimizers. We provide robustness through elastic net regularization and show that, analytically, the non-smoothness of L_1 regularization is addressed with an appropriate choice for its sub-differential at 0. The use of second order methods provides asymptotically faster rates of convergence than first-order methods [21, 33], and provides greater opportunities to exploit resources at scale for expensive computations, such as direct computation of the Hessian matrix involved in Newton's method.

1.4 Scaling the NumPy API

With an intuition for blocked matrix operations for basic linear algebra operations on distributed memory systems, we turn our attention to scaling NumPy and generalized linear models. An ideal solution provides the same familiar API to existing users of these libraries, while enabling high performance execution on distributed systems which are competitive with the state-of-the-art.

Chapter 3 presents our solution, NumS. NumS is a scalable numerical array programming library which enables programmers to write code using the NumPy API. NumS scales the NumPy API by partitioning multi-dimensional arrays automatically according to the softmax distribution of the array's *shape*. Array partitions are physically mapped to nodes according to a *node grid*, which generalizes the concept of a *block-cyclic* data layout, and optimizes layouts for hierarchical network topologies.

Beyond data layout, numerical operations are represented *lazily* by an array-of-graphs data structure, called a GraphArray. GraphArrays are executed by our scheduler, Load Simulated Hierarchical Scheduling (LSHS), which is tailored to the architecture and primitives provided by task-based distributed systems. As placement decisions are simulated or dispatched, our scheduler models memory load, network load, and object locality on each node and worker process. A simple greedy operator placement algorithm is used to execute GraphArrays. See Section 3.5 for details.

The empirical results of our work are presented in Chapter 3. An ablation study on our method shows that our solution to scheduling enhances NumS' performance on both Ray and Dask in almost every benchmark we perform. In particular, LSHS combined with Ray's low overhead achieves peak performance, outperforming Dask ML on the logistic regression problem by 2×, and Spark MLlib by 3× on a terabyte of data. We provide similar gains in performance for QR decomposition and core operations involved in tensor factorization.

When tuned, NumS achieves competitive performance to ScaLAPACK and SLATE on square dense matrix-matrix multiplication within cloud-based environments. We support these results with a comprehensive communication analysis in Chapter 4.

1.5 Automatic Parallelization of Basic Python Operations

Seamlessly scaling basic Python operations, as well as the NumPy API, are described in Chapters 5 and 6. These approaches to parallelization form the basis of our approach to seamlessly scaling the NumPy API.

In Chapter 5, we provide the syntax and semantics of a subset of the Python language, and a language which supports concurrently executing futures. We provide a procedure for translating Python programs comprised of the syntax we specify to our language of concurrently executing futures, and provide a rigorous proof of correctness. In Chapter 6, we extend our solution to incorporate syntax and semantics for 2-dimensional numerical arrays.

Chapter 2

Scalable Second-Order Convex Optimization

In this chapter, we'll describe a straightforward approach to parallelizing and scaling unconstrained convex optimization problems. In particular, we develop a technique optimized for tall-skinny matrices, matrices which frequently occur when modeling data using generalized linear models. We develop a second-order solution to elastic net regularization for these models. We show that the speedups achieved using second order methods for our approach to parallelism are substantial, showing significant speedups over optimizers provided by the scikit-learn library [28] for GLMs.

Our approach is based on the block decomposition of vectors and matrices, which is a powerful approach to parallelizing linear algebra operations, and performing such operations on terabytes of data on distributed memory supercomputers and distributed systems in the cloud.

Any algorithm that uses the linear algebra operations we formulate can be parallelized with this approach. Due to the abundance of computational resources made available by block partitioning these basic operations, we are able to employ second order methods on terabytes of data. We'll go over how Newton's method is parallelized for logistic regression, and how the same principles apply to generalized linear models with elastic net regularization.

2.1 Blocked Representations and Operations

The *block* decomposition of vectors and matrices can be described as follows. For a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$, let g_1 and g_2 be the dimensions of the *grid* into which $\mathbf{X}$ is decomposed into blocks. For $i \in \{0, \ldots, g_1 - 1\}$ and $j \in \{0, \ldots, g_2 - 1\}$, let $\mathbf{X}_{i,j}$ denote the i, j block of $\mathbf{X}$, as depicted in the following equation.

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_{0,0} & \cdots & \mathbf{X}_{0,g_2-1} \\ \vdots & \ddots & \vdots \\ \mathbf{X}_{g_1-1,0} & \cdots & \mathbf{X}_{g_1-1,g_2-1} \end{bmatrix} \tag{2.1}$$

Note that each block $\mathbf{X}_{i,j}$ *is itself a matrix with dimensions* $(m/g_1) \times (n/g_2)$. We will assume the dimension along any axis of the matrix is divisible by the size of its grid along the same axis. For example, $b_1 = m/g_1$ and $b_2 = n/g_2$ are integer values. We call b_1 and b_2 the dimensions of the blocks of $\mathbf{X}$. With that said, an extension to arbitrary matrix and grid dimensions is relatively straightforward.

For any matrix multiplication $\mathbf{C} = \mathbf{AB}$, we have $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$ so that $\mathbf{C} \in \mathbb{R}^{m \times n}$. We decompose $\mathbf{C}$ into an $r \times c$ grid. We decompose $\mathbf{A}$ into an $r \times q$ grid and $\mathbf{B}$ into a $q \times r$ grid. Thus, block $\mathbf{C}_{i,j} = \sum_{h=0}^{q-1} \mathbf{A}_{i,h}\mathbf{B}h, j$.

We treat vectors as a special case of the above formulation, with the second axis set to 1. For example, $\mathbf{y} = \mathbf{Ax}$ is computed by decomposing $\mathbf{y}$ into an $r \times 1$ (column) grid and $\mathbf{x}$ into a $q \times 1$ grid so that $\mathbf{y}_i = \sum_{h=0}^{q-1} \mathbf{A}_{i,h}\mathbf{x}_h$.

For element-wise operations, such as $\mathbf{X} + \mathbf{Y}$, we assume that the dimensions of the matrices are equivalent, and the grid into which they're decomposed are also equivalent so that the blocks within the respective grids have compatible dimensions.

Example: Matrix Multiplication

Consider a 6×4 matrix $\mathbf{A}$ decomposed into blocks of size 2×2. The grid dimensions of matrix $\mathbf{A}$ are 3×2. Let us also define $\mathbf{B}$ as 4×10 with blocks of size 2×5 and grid dimensions 2×2. Note that, in both cases, the element-wise multiplication of the grid and block dimensions yield the dimensions of the entire matrix. To compute the matrix multiplication of $\mathbf{A}$ and $\mathbf{B}$, we proceed as follows.

$$\mathbf{C} = \left[\begin{array}{c|c} \sum_{h=0}^{1} \mathbf{A}_{0,h}\mathbf{B}_{h,0} & \sum_{h=0}^{1} \mathbf{A}_{0,h}\mathbf{B}_{h,1} \\ \hline \sum_{h=0}^{1} \mathbf{A}_{1,h}\mathbf{B}_{h,0} & \sum_{h=0}^{1} \mathbf{A}_{1,h}\mathbf{B}_{h,1} \\ \hline \sum_{h=0}^{1} \mathbf{A}_{2,h}\mathbf{B}_{h,0} & \sum_{h=0}^{1} \mathbf{A}_{2,h}\mathbf{B}_{h,1} \end{array}\right]$$

Thus, we represent the 6×10 matrix $\mathbf{C}$ above as a 3×2 grid of 3×5 blocks.

2.2 Parallelizing Blocked Linear Algebra Operations

Parallelizing block partitioned vectors and matrices can be achieved in a number of different ways. In this section, we describe a simple approach that computes the output blocks of an

operation on a grid of *nodes*. In this context, nodes may be distinct processes, GPU devices connected on a single computer, or computers that are networked in the cloud.

Let us begin with a grid of $r \times c$ nodes. We have $p = rc$ nodes total. We represent the memory contents of node i, j as the set $\mathbf{N}_{i,j}$. Each node is unable to access the memory of any other node, and nodes are assumed to have sufficient memory to carry out matrix multiplication on pairs of blocks. How do we decide on which node to store each block that makes up each matrix? How do we decide on which nodes to execute computations?

We decide where each block is stored by assuming a *block cyclic data layout*. If $\mathbf{X}$ is decomposed into a $g_1 \times g_2$ grid, then we store $\mathbf{X}_{i,j}$ on node $(i \mod r), (j \mod c)$. Note that, when $g_1 = r$ and $g_2 = c$, each block is stored on a distinct node. As for where to execute operations, we will adopt a simple convention: If the output block is in $\mathbf{N}_{i,j}$, then all operations will be performed on node i, j.

Matrix Multiplication

Recall our general formulation of matrix multiplication of the two matrices $\mathbf{A}$ and $\mathbf{B}$. We have that $\mathbf{C}_{i,j} = \sum_{h=0}^{q-1} \mathbf{A}_{i,h}\mathbf{B}_{h,j}$. Recall that $b_1 = m/g_1$ and $b_2 = n/g_2$ are the dimensions of each block. We will rewrite the basic procedure for matrix multiplication so that the memory requirements per node are at most $O(b_1 b_2)$. By doing so, we will have a general procedure for matrix multiplication that scales.

In the following algorithms, we write $\forall_{i=0,j=0}^{m-1,n-1}\{expression\}$ to denote the concurrent execution of *expression* for all instances of the indices $i \in \{0, \ldots, m-1\}$ and $j \in \{0, \ldots, n-1\}$. Consider the approach to matrix multiplication given in Algorithm 1. Notice that the

Algorithm 1: Serial Matrix Multiplication.

```
C ← 0;
for i ← 0 to m do
    for j ← 0 to n do
        for h ← 0 to k do
            C_{i,j} ← C_{i,j} + A_{i,h}B_{h,j}
        end
    end
end
```

operations within the outer two loops (over i and j) are all independent of one another. If we reorder the loops and rewrite our matrix multiplication procedure as Algorithm 2, we can bound the memory required by each node to a constant factor of the block sizes. Lets assume we have an $m \times n$ grid of nodes on which to execute the above computation. We can see that, for each iteration of the outer loop, we need 1 block to store the result, and 2 blocks to perform the matrix multiplication. Thus, Algorithm 2 provides a way to arbitrarily scale matrix operations.

Algorithm 2: Concurrent Matrix Multiplication.

$\mathbf{C} \leftarrow \mathbf{0}$;
for $h \leftarrow 0$ **to** k **do**
  $\forall_{i=0,j=0}^{m-1,n-1} \{\mathbf{C}_{i,j} \leftarrow \mathbf{C}_{i,j} + \mathbf{A}_{i,h}\mathbf{B}_{h,j}\}$

While we've shown that this approach achieves good scaling, we have not discussed the matter of transmitting blocks between nodes in order to perform the required operations. Recall our simple assumption: If the output block $\mathbf{C}_{i,j}$ is in $\mathbf{N}_{s,t}$, then all operations are performed on $\mathbf{N}_{s,t}$. We'll simply assume that any operation performed on $\mathbf{N}_{s,t}$ transmits the required blocks implicitly from the nodes on which they reside under the block cyclic data layout. It's worth noting that sophisticated data communication algorithms are typically employed to efficiently interleave communication and computation for distributed memory computations such as the one we've described. In many cases, simply transmitting objects to wherever they are needed is sufficient to achieving good scalable performance.

Element-wise Unary and Binary Operations

Element-wise unary operations on a matrix $\mathbf{X}$ decomposed into a grid $g_1 \times g_2$ of blocks distributed over a grid of nodes $r \times c$ can be done by simply applying the operations in place. For example, to perform $e^{\mathbf{X}}$, if $\mathbf{X}_{i,j}$ is in $\mathbf{N}_{s,t}$, we simply apply $e^{\mathbf{X}_{i,j}}$ on node $\mathbf{N}_{s,t}$.

Since the unary operation transpose drastically rearranges the way blocks are distributed over a grid of nodes, we treat it differently. If a transpose is performed, we *fuse* it with the next operation without actually changing the nodes on which the transposed matrix's blocks reside. The transpose is then performed before the operation with which it is fused. For example, if we must perform $\mathbf{Y} = \mathbf{X}^T\mathbf{X}$, instead of updating the data layout of $\mathbf{X}^T$, we keep all the blocks in place and simply perform $\mathbf{Y}_{i,j} = \sum_{h=0}^{k}(\mathbf{X}^T)_{i,h}\mathbf{X}_{h,j}$ and transmit blocks between nodes as needed for the matrix multiply.

For element-wise binary operations between two matrices $\mathbf{A}$ and $\mathbf{B}$, we assume $\mathbf{A}$ has the same dimension and decomposition as $\mathbf{B}$. By the block-cyclic data layout, if $\mathbf{A}_{i,j}$ is in $\mathbf{N}_{s,t}$, so is $\mathbf{B}_{i,j}$, and the operation is performed on node $\mathbf{N}_{s,t}$ without requiring any communication between nodes. Thus, we also have that each output block $\mathbf{C}_{i,j}$ also resides on $\mathbf{N}_{s,t}$. For example, the expression $\mathbf{C} = \mathbf{A} + \mathbf{B}$ induces the computation $\mathbf{C}_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$ on $\mathbf{N}_{s,t}$.

2.3 Scalable Logistic Regression with Newton's Method

Given a dataset $\mathbf{X} \in \mathbb{R}^{m \times n}$ decomposed into a grid $g_1 \times 1$ of blocks, $\mathbf{y} \in \mathbb{R}^{m \times 1}$ decomposed into a grid $g_1 \times 1$, a logistic regression model[1] m with corresponding twice differentiable convex

[1]This formulation works for any generalized linear model.

Algorithm 3: Newton's method.

$\beta \leftarrow \mathbf{0}$;
while *true* **do**
  $\mu \leftarrow m(\mathbf{X}, \beta)$;
  $\mathbf{g} \leftarrow \nabla f(\mathbf{X}, \mathbf{y}, \mu, \beta)$;
  $\mathbf{H} \leftarrow \nabla^2 f(\mathbf{X}, \mathbf{y}, \mu, \beta)$
  $\beta \leftarrow \beta - \mathbf{H}^{-1}\mathbf{g}$;
  if $\|\boldsymbol{g}\|_2 \leq \epsilon$ **then**
    return β;
  end
end

objective f, and minimum gradient norm ϵ, Algorithm 3 computes the global minimum β of f using Newton's method. Upon initialization, β is decomposed into a 1×1 grid.

As-is, Algorithm 3 appears no different than a serial implementation of Newton's method, which is precisely why this approach to parallelization is so appealing. In what follows, we work through the execution of Algorithm 3 on an $r \times 1$ grid of nodes. We assume the usual block cyclic data layout, so that $\mathbf{X}$ and $\mathbf{y}$ are distributed row-wise over r nodes, and the single block $\beta_{0,0}$ of β is in node $\mathbf{N}_{0,0}$.

1. For logistic regression, we have that $m(\mathbf{X}, \beta) = \frac{1}{1+e^{-\mathbf{X}\beta}}$. We see first that the linear operation $\mathbf{X}\beta$ will yield an intermediate value $\mathbf{C}$ comprised of a grid of blocks $\mathbf{C}_{i,0} = \mathbf{X}_{i,0} \times \beta_{0,0}$. According to the rules we've defined thus far, this is computed by broadcasting $\beta_{0,0}$ to all the nodes in which the $\mathbf{X}_{i,0}$ reside. The remaining unary and binary operations are applied in place to $\mathbf{C}$ to yield the output μ, which is decomposed into a $g_1 \times 1$ matrix.

2. The gradient of f is given by the expression $\mathbf{X}^T(\mu - \mathbf{y})$. We see that μ has the required grid decomposition to perform the element-wise subtraction operation with $\mathbf{y}$, yielding the vector $\mathbf{c}$. Finally, the transpose operator is fused with the matrix-vector multiply, resulting in the final set of operations $\mathbf{g}_{0,0} = \sum_{h=0}^{g_1-1}(\mathbf{X}^T)_{0,h}\mathbf{c}_{h,0}$. Like β, $\mathbf{g}$ is comprised of a single block.

3. The Hessian of f is given by the expression $\mathbf{X}^T(\mu \times (1-\mu) \times \mathbf{X})$, where $\times$ denotes element-wise multiplication. The expression $\mu \times (1-\mu)$ yields an intermediate vector $\mathbf{c}$ that has the same decomposition as μ, and the vector-matrix element-wise operation $\mathbf{c} \times \mathbf{X}$ broadcasts $\mathbf{c}$ so that $\mathbf{c}$ is multiplied element-wise with every column of $\mathbf{X}$, yielding an intermediate matrix $\mathbf{C}$ with the same decomposition as $\mathbf{X}$. Finally, the operation $\mathbf{X}^T\mathbf{C}$ results in the computation $\mathbf{H}_{0,0} = \sum_{h=0}^{k-1}(\mathbf{X}^T)_{0,h}\mathbf{X}_{h,0}$, where $\mathbf{H}$ is square with dimension n.

4. Finally, we update β. At this point, β, $\mathbf{g}$ and $\mathbf{H}$ are all comprised of single blocks, so the update to beta is executed serially on node $\mathbf{N}_{0,0}$.

5. Similar to the update to β, the gradient norm is computed serially on node $\mathbf{N}_{0,0}$.

In this example, it is crucial that n is much smaller than m, which is usually the case when $\mathbf{X}$ is a design matrix consisting of m samples with n features. This allows us to decompose $\mathbf{X}$ row-wise only, which significantly improves the performance of this algorithm. In particular, while distributed algorithms based on L U decomposition exist for matrix inversion, the approach we've taken allows us to execute matrix inversion serially.

2.4 Unconstrained Minimization with L-BFGS

As with Newton's method, let us assume the objective function $f(x)$ is convex and twice differentiable, only now we allow $x \in \mathbb{R}^n$. For Newton's method, the computational complexity of the Hessian H is on the order of $O(qn^2)$, where q is the number of operations required to compute each entry of the Hessian. For large n, computing the Hessian is a bottleneck.

A popular solution to overcome this issue is the BFGS algorithm, a quasi-Newton method that approximates H with an $n \times n$ symmetric positive definite matrix B_t that is updated at every iteration. At iteration t, the quadratic model of the objective function is

$$m_t(p) = f(x_t) + \nabla f(x_t)^T p + \frac{1}{2} p^T B_t p.$$

Similar to Newton's method, this model is convex with a minimum at iteration t given by

$$p_t = -B_t^{-1} \nabla f(x_t).$$

Since B_t is positive definite, we have $p_t^T \nabla f(x_t) = -\nabla f(x_t)^T B_t^{-1} \nabla f(x_t) < 0$, and therefore p_t is a descent direction.

We learned that Newton's method may diverge for some f if the starting point x_0 is too far from the global minimum x^*. To avoid divergence for any convex f, both Newton's method and BFGS can be rewritten to require a step length parameter α_t. The iterate for BFGS with a step length parameter is defined as

$$x_{t+1} = x_t + \alpha_t p_t.$$

Notice that, when α_t is near 0, x_{t+1} is near x_t. In most cases, the exact Hessian captures the curvature of the local approximation, and α_t can be set to 1.

BFGS

In BFGS, since the inverse Hessian is required to update x, we approximate the inverse Hessian directly. When computing the inverse Hessian approximation at step t, we apply

a simple modification to the current approximation based on the most recent information obtained from the objective function. Let

$$s_t = x_{t+1} - x_t$$
$$y_t = \nabla f(x_{t+1}) - \nabla f(x_t).$$

The inverse Hessian approximation at step $t+1$ is defined as $B_{t+1}^{-1} = min_H||H - B_t^{-1}||$ subject to $H = H^T, Hy_t = s_t$. This choice determines B_{t+1}^{-1} uniquely by choosing the closest solution to B_t^{-1} that satisfies the *secant equation* $Bs_t = y_t$, which is enforced for the inverse Hessian by the constraint $Hy_t = s_t$. The secant equation captures the requirement that the gradient at m_{t+1} matches the gradient of f at x_t and x_{t+1}. Thus, B_{t+1}^{-1} is unique, and maps y_t into s_t so long as s_t and y_t satisfy the *curvature condition* $s_t^T y_t > 0$, which is true for any two points x_t and x_{t+1} if f is strongly convex [2].

Using a weighted Frobenius norm, the solution to the optimization problem given above is

$$B_{t+1}^{-1} = V_t^T B_t^{-1} V_t + \rho_t s_t s_t^T,$$

where $\rho_t = \frac{1}{y_t^T s_t}$ and $V_t = I - \rho_t y_t s_t^T$. A common choice for the initial approximation B_0^{-1} is the identity matrix.

L-BFGS

For high-dimensional x, A simple extension to BFGS can provide a significant decrease in the memory required to compute the Hessian approximation at each iteration. Instead of storing the Hessian approximation at iteration t, which requires $O(n^2)$ memory, limited-memory BFGS, or L-BFGS, provides a solution that reduces the memory requirement to $O(n)$. The basic idea is to store the last m vectors of s_i and y_i, and to compute the search direction p_t via a sequence of vector operations. The computation of p_t is given by Algorithm 4. A practical approach to computing H_t^0 is to simply set it to $\left(\frac{s_{t-1}^T y_{t-1}}{y_{t-1}^T y_{t-1}}\right) I$, which can be stored as a vector and used to compute the initial value of r by element-wise multiplication. With p_t, we compute x_{t+1} as is done for BFGS. At the end of each iteration, if $t > m$, then we delete s_{t-m} and y_{t-m} from memory. In most cases, values as small as $m = 10$ work well.

Line Search

For BFGS and L-BFGS, a *line search* method is used to compute α_t, the step size used in the computation $x_{t+1} = x_t + \alpha_t p_t$. Line search methods begin with an initial guess for α_t, such as 1, and update the guess iteratively to achieve a *sufficient decrease* in f. *Backtracking* line search (see Figure 2.1) is one such line search method and is defined by Algorithm 5 [3].

[2] When f is not strongly convex, the curvature condition can be explicitly enforced by requiring Wolfe or strong Wolfe conditions on the line search (see Section 2.4).

[3] While backtracking works for strongly convex functions, a line search satisfying strong Wolfe conditions is recommended when using L-BFGS for non-convex functions.

Algorithm 4: Computation of Search Direction.

$q \leftarrow \nabla f(x_t)$;
for $i \leftarrow t$ **down to** $t - m$ **do**
  $\alpha_i \leftarrow \rho_i s_i^T q$;
  $q \leftarrow q - \alpha_i y_i$;
$r \leftarrow H_t^0 q$;
for $i \leftarrow t - m$ **up to** t **do**
  $\beta = \rho_i y_i^T r$;
  $r = r + s_i(\alpha_i - \beta)$;
$p_t \leftarrow -r$;

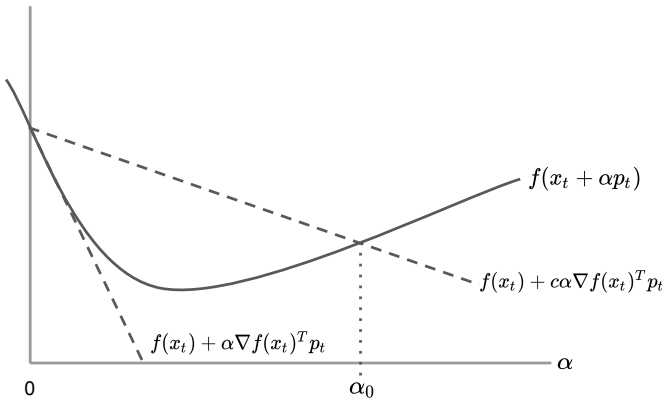


Figure 2.1: *Backtracking line search.* The curve shows f, restricted to the line over which we search (e.g. $\alpha \in (0, 1]$). The lower dashed line shows the linear extrapolation of f, and the upper dashed line has a slope a factor of c smaller. Note that $f(x_t + \alpha_0 p_t) = f(x_t) + c\alpha_0 \nabla f(x_t)^T p_t$. The backtracking condition is that f lies below the upper dashed line, which in this diagram is $0 \leq \alpha \leq \alpha_0$.

The parameter c is usually chosen to be small, such as 10^{-4}.

2.5 Elastic Net Regularization

The procedure outlined in Section 2.3 is the same for any generalized linear model. Inference is reduced to the process of computing $\eta = \mathbf{X}\beta$, and applying the inverse link function associated with the model [6]. We use μ to refer to the result of the inverse link function. For instance, the inverse link for linear regression is the identity, $\mu = \eta$. For logistic regression, we have $\mu = \frac{1}{1+e^{-\eta}}$. For Poisson regression, $\mu = e^{\eta}$. Elastic net regularization is a popular

Algorithm 5: Backtracking Line Search.

$\alpha > 0$; $\rho \in (0, 1)$; $c \in (0, 1)$;
while $f(x_t + \alpha p_t) > f(x_t) + c\alpha \nabla f(x_t)^T p_t$ **do**
  $\alpha = \rho\alpha$;
$\alpha_t = \alpha$

regularization technique for generalized linear models [44]. Elastic net interpolates L_1 and L_2 regularization for a given loss function $Loss(\mathbf{X}, \mathbf{y}, \beta)$ as follows:

$$Loss(\mathbf{X}, \mathbf{y}, \beta) + \lambda \left(\alpha \|\beta\|_1 + (1 - \alpha)\|\beta\|_2\right). \tag{2.2}$$

Given the L_1 term, coordinate descent is recommended to optimize the objective. Given the level of parallelism and convergence properties provided by Newton's method, we apply Newton's method to optimization of elastic net regularized generalized linear models. The gradient of the L_1 norm is -1 when $\beta_i < 0$ and 1 when $\beta_i > 0$. While the gradient is not defined for $\beta_i = 0$, the *sub-differential* is, which provides the set $(-1, 1)$ as valid choices at 0. We therefore choose 0 as the derivative of $\beta_i = 0$.

Part II

Scalable Multi-dimensional Array Operations.

Chapter 3

A System for Scalable N-Dimensional Arrays

3.1 Introduction

Many popular Python programming tools [32] exist in the space of solutions which address scalable linear algebra and generalized linear models. These tools primarily provide scalable distributed numerical operations using rich, Numpy-like expressions. However, many of these tools rely on dynamic schedulers [23, 32] optimized for abstract task graphs, which do not exploit the structure of distributed numerical arrays, such as apriori knowledge of input and output sizes, or properties shared among element-wise, reduction, and tensor algebra operations, such as the locality inherent in their parallel decompositions. This can lead to performance problems which are difficult to address without in-depth knowledge of the underlying distributed system.

Dask Arrays [32] implements parallel numerical array operations (Section 3.3) by constructing discrete tasks graphs which represent the desired computation, and schedules these tasks dynamically. While this decoupling of algorithm from scheduling has desirable software design properties, the loss of information to the

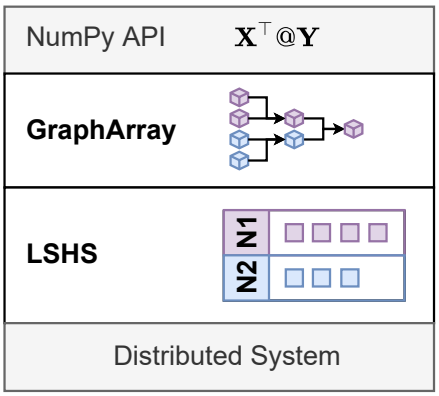


Figure 3.1: Design diagram of NumS. Users express numerical operations using the NumPy API. These operations are implemented by the GraphArray type. LSHS dispatches these operations to the underlying distributed system, specifying data and operator placement requirements. The underlying distributed system moves arrays between nodes and processes to satisfy data dependencies for task execution.

Syntax	Description
$-\mathbf{X}$	Unary operation.
$\mathbf{X} + \mathbf{Y}$	Binary element-wise.
$\mathbf{sum}(\mathbf{X}, \text{axis} = 0)$	Reduction.
$\mathbf{X@Y}$	Basic linear algebra.
$\mathbf{tensordot}(\mathbf{X}, \mathbf{Y}, \text{axes} = 2)$	Tensor contraction.
$\mathbf{einsum}(ik, kj \rightarrow ij, \mathbf{X}, \mathbf{Y})$	Einstein summation.

Table 3.1: While NumS provides greater coverage of the NumPy API, we list only the set of operations we consider in this work, along with syntax examples.

scheduler leads to sub-optimal data and oper-
ator placement. In particular, when data placement is not optimized for numerical array operations, unnecessary communication among processes is often required in order to carry out basic operations, such as element-wise addition and vector dot products. In general, any scheduling algorithm which dynamically schedules distributed numerical array operations as discrete task graphs is susceptible to sub-optimal performance.

High-performance computing (HPC) tools built on the message passing interface (MPI) [12], such as ScaLAPACK [8] and SLATE [16], implement algorithms that are optimized for linear algebra operations. These specialized libraries are state-of-the-art for scalable linear algebra, but they do not support the NumPy API, making them inaccessible to an increasing number of scientists who are adopting Python.

To enhance the performance of NumPy-based numerical array programming libraries, we present a scheduling framework tailored to the architecture and primitives provided by task-based distributed systems. Our framework is limited to data and operator placement decisions over a collection of compute nodes comprised of worker processes. As placement decisions are simulated or dispatched, our framework models memory load, network load, and object locality on each node and worker process. Within this framework, we implement a simple greedy operator placement algorithm guided by a cost function of our model of the underlying system's state, called Load Simulated Hierarchical Scheduling (LSHS) (see Section 3.5).

To support our empirical analysis, we analyze the communication time between node and worker processes, as well as the latency associated with dispatching placement decisions. We present communication lower bounds for a collection of common operations, and show that LSHS attains these bounds for some operations. We also show, both analytically and empirically, limitations of these distributed systems as they apply to the scalability of distributed numerical array operations.

Figure 3.1 depicts the design of NumS. While NumS operates on Dask, it is optimized for and achieves peak performance on Ray. When we refer to NumS without explicitly stating the underlying distributed system, we assume it is using LSHS and running on Ray.

Overall, our evaluation shows that NumS can achieve competitive performance to SLATE on square matrix multiplication on a terabyte of data, and outperforms Dask ML and Spark MLlib on a number of end-to-end applications. We show that, in every benchmark, LSHS is required for NumS's algorithms to achieve peak performance on Ray.

In comparison to SLATE, NumS is limited by the rate at which operations can be dispatched to the underlying distributed system, and the overhead introduced by remote function invocation. Our empirical and theoretical analysis on these limitations show that NumS performs best on a smaller number of coarse-grained array partitions. Fewer partitions mitigate control overhead introduced by having a centralized control process, and larger partitions amortize the overhead associated with executing remote functions in Ray. Section 3.7 measures these overheads and provides a theoretical framework to model them in our analyses. On the other hand, SLATE runs on MPI, which can operate on a greater number of partitions due to MPI's programming model, which enables decentralized and partitioned application control. For matrix multiplication, SLATE uses SUMMA, which allocates a memory buffer for output partitions, and accumulates intermediate results to achieve better memory efficiency than NumS's matrix multiplication algorithm. However, SUMMA achieves sub-optimal communication time for a variety of matrix multiplication operations, such as matrix-multiplication among row-partitioned tall-skinny matrices. SLATE users must therefore be knowledgeable about the performance trade-offs of the variety of different operations provided by the library in order to take full advantage of its capabilities. We therefore recommend NumS for high-performance, medium-scale (terabytes) projects which may benefit from the flexibility of the NumPy API, and SLATE for projects which require specialized, large-scale distributed memory linear algebra operations.

In this chapter, we outline our research contributions to the following areas.

1. LSHS, a scheduling algorithm for numerical array operations optimized for cloud-based distributed systems.

2. Theoretical lower bounds on the number of bytes required to perform a variety of common operations, including square matrix multiplication and tensor contractions. We show that LSHS attains some of these bounds. We also show theoretical limitations of distributed numerical array computing on Ray and Dask, and how these limitations manifest in practice.

3. An evaluation of our approach, which includes an ablation study comparing LSHS to Ray's and Dask's schedulers on a variety of common basic operations. Our results show that LSHS consistently enhances performance on Ray and Dask.

4. A comparison of NumS to ScaLAPACK and SLATE for square matrix multiplication, and Dask ML and Spark MLlib for QR decomposition and logistic regression. Our results show that our solution achieves competitive performance to ScaLAPACK and SLATE. Our results show that, on the logistic regression problem, LSHS enhances performance on Ray by decreasing network load by a factor of 2×, using 4× less

memory, and decreasing execution time by a factor of 10×. NumS also achieves a speedup of up to 6× on logistic regression compared to Dask ML and Spark's MLlib on a terabyte of data.

3.2 Related Work

NumPy NumPy [25] provides a collection of serial operations, which include element-wise operations, summations, and sampling. When available, NumPy uses the system's BLAS [7] implementation for vector and matrix operations. BLAS implementations use shared-memory parallelism. For large datasets that fit on a single node, NumS outperforms NumPy on creation and element-wise operations. NumPy does not provide a block partitioned representation of arrays on distributed memory.

Dask Dask [32] provides parallelism on a single machine via futures, and is able to scale to multiple nodes via the Dask distributed system, a distributed system framework similar to Spark [43] and Ray [23]. Dask constructs a static task graph as operations are invoked on its futures API. When a task graph is executed by the user, Dask dynamically schedules tasks that have its dependencies computed. Independent operations are scheduled round-robin over workers, and dependent tasks benefit from a variety of dynamic scheduling heuristics.

Dask provides a distributed array abstraction, partitioning arrays along n-dimensions, and providing an API which constructs task graphs of array operations. In particular, when a graph of array operations is executed, array partitions and their operations are dynamically scheduled as tasks in an task graph. Round-robin scheduling of independent tasks results in sub-optimal data layouts for common array operations, such as the element-wise and linear algebra operations. NumS mitigates this issue by statically mapping array partitions to physical nodes in a cyclic layout optimized for hierarchical networks (see Figure 3.3). Furthermore, at schedule time, the output sizes of tasks are unknown, which can result in OOM errors for tasks with large outputs. NumS mitigates this issue by maintaining the output size of all operations, and *simulating* the network and memory load imposed on a given node before scheduling an operation on that node.

Distributed Machine Learning. The Dask ML [32] library provides several machine learning models. The optimization algorithms written for these models frequently execute code on the driver process. The library is written using Dask's array abstraction, resulting in sub-optimal performance on many linear algebra operations.

Spark's MLlib [22] is a library for scalable machine learning. MLlib depends on Breeze [10], a Scala library that wraps optimized BLAS [7] and LAPACK [4] implementations for numerical processing. Breeze provides high-quality implementations for many common machine learning algorithms that have good performance, but because it relies on Spark primitives, it introduces a learning curve for NumPy users.

High performance computing. HPC libraries such as ScaLAPACK (Scalable Linear Algebra PACKage) [8] and SLATE [16] provide tools for distributed memory linear algebra. They implement highly optimized communication-avoiding operations using MPI [12]. These libraries are state-of-the-art in terms of performance. The high performance provided by these libraries comes at the cost of several specialized implementations of various linear algebra operations. ScaLAPACK exposes 14 different routines for distributed matrix multiplication, each optimized for specialized matrices with various properties [8]. Unlike NumS which enables programming against the NumPy API, which includes support for tensor algebra operations, while these libraries provide a C++ API limited to linear algebra operations.

DNN libraries. Deep learning libraries such as Tensorflow [1], PyTorch [27], and MXNet [11] provide tensor abstractions, and JAX [9] provides a NumPy array abstraction which enhances usability. Mesh Tensorflow [36] provides tensor partitioning on top of Tensorflow, but requires specifying layouts for tensors, and targets Google TPUs. These libraries are specialized for DNN training on accelerators. In contrast, NumS is designed for general purpose array programming on CPUs.

Ray. Ray [23] is a task-based distributed system which dynamically executes a distributed, dynamic task graph. A Ray cluster is comprised of a head node and worker nodes. Worker nodes are comprised of worker processes, which execute tasks. The output of a task is written to the shared-memory object store on the node on which the task was executed. Any worker can access the output of any other worker within the same node. Python programs connect to a Ray cluster via a centralized driver process, which dispatches remote function calls to worker nodes. Ray implements a *bottom-up* distributed scheduler. Driver and worker processes submit tasks to their local scheduler. Based on available resources and task meta data, a local scheduler may execute a task locally, or forward the task to a centralized process on the head node. The latter occurs only when a local scheduler is unable to execute a task locally, a decision based on a variety of heuristics. Our understanding is that, when a local scheduler is presented with a collection of tasks which have no dependencies, it distributes tasks to reduce overall load on any given node.

3.3 Background

In task-based distributed systems, a *task* is a unit of work with one or more dependencies. This can be thought of as an arbitrary pure function. *Objects* serve as the inputs and outputs of tasks. A completed task can be viewed as the object(s) it outputs. A *task graph* is a representation of the dependencies between tasks. A *remote function call* (RFC) creates a task with zero or more dependencies. In NumS, numerical kernel operations are executed as RFCs, creating a task with the kernel operation's operands as dependencies. Operands may

be tasks that have not yet been executed, or they may be the output object of a completed task.

A *worker* is an independent process which executes tasks. We use the term *node* to refer to machines comprised of one or more worker processes. *Task placement* refers to the process of deciding on which node or worker a particular task should execute. All task placement decisions are executed on a centralized *driver process*.

Figure 3.2 depicts the network topology for which NumS is optimized. The cluster is comprised of 4 nodes, with 4 workers per node. Node N_3 is expanded to expose the intra-node network topology of worker processes. Thicker edges correspond to greater bandwidth. In Ray, we make placement decisions at the granularity of nodes, leaving worker-level scheduling to each node's local scheduler. Ray implements a shared memory object store, enabling any local worker to access the output of any other local worker without worker-to-worker communication. In Dask, we make placement decisions at the granularity of worker processes. worker-to-worker communication within the same node can be expensive, which we address with our hierarchical data and operator placement design. Our design is given in Section 3.4.

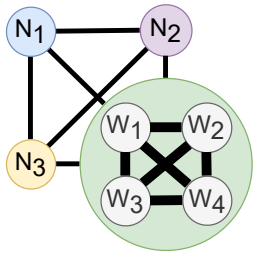


Figure 3.2

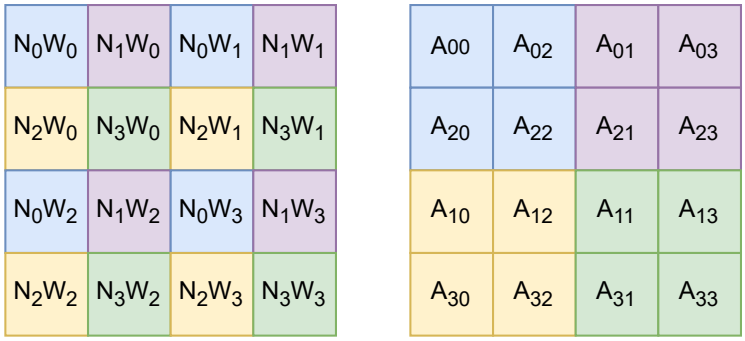
3.4 GraphArray Type

The **GraphArray** type implements distributed array creation, manipulation, and numerical operations. Creation and manipulation operations *execute immediately*, whereas numerical operations are deferred.

A GraphArray is created via read operations, invocation of operations like **zeros**(shape, grid) to create a dense array of zeros or ones, or operations like **random**(shape, grid) to randomly sample a dense array from some distribution. The *shape* parameter specifies the dimensions of the array, and the *grid* parameter specifies the *logical partitioning* of the multidimensional array along each axis specified by the shape. The logical partitioning of an array is called its *array grid*. For example, $\mathbf{A} = \mathbf{random}((256, 256), (4, 4))$ will randomly sample a block-partitioned array partitioned into 4 block along the first axis, 4 blocks along the second axis, and 2 blocks along the third axis for a total of 16 blocks. Mathematically, we use the notation $\mathbf{A}_{i,j}$ to denote the i, j block of $\mathbf{A}$, as depicted in the following equation.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \mathbf{A}_{0,2} & \mathbf{A}_{0,3} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} \\ \mathbf{A}_{2,0} & \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} \\ \mathbf{A}_{3,0} & \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} \end{bmatrix} \tag{3.1}$$

Each block $\mathbf{A}_{i,j}$ *is itself a matrix with dimensions* 64×64.



(a) Worker Mapping. (b) Array Mapping.

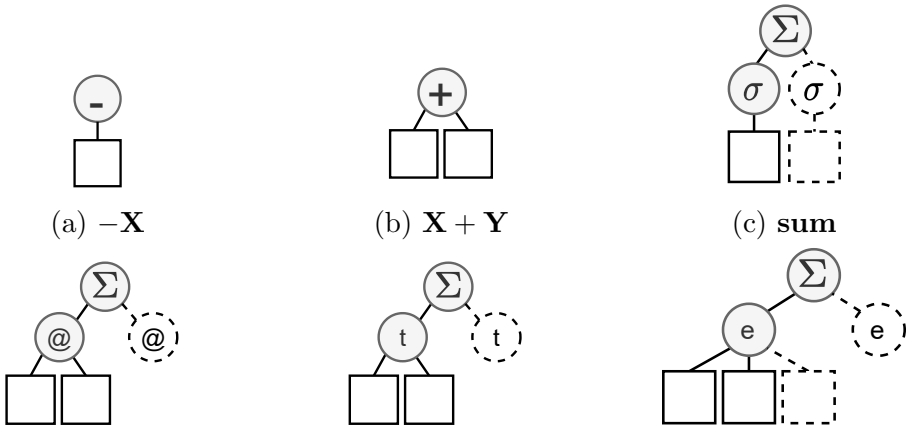
Figure 3.3: Hierarchical mapping of logical array partitions to physical nodes and workers.

When a creation operation is invoked, the logical partitions of an array are mapped *hierarchically* to physical nodes. To carry out this mapping within our framework, a user-defined *node grid*, a multi-dimensional coordinate space for nodes within a cluster, is required. For a cluster consisting of 4 nodes with 4 workers each, figure 3.3a depicts the mapping of the previously defined array $\mathbf{A}$ to nodes and workers from a user-defined 2×2 node grid. $N_i W_j$ corresponds to worker j on node i. The node grid is fixed throughout the execution of a NumS application. Without loss of generality, for a node grid with dimensions $g_1 \times g_2$, $\mathbf{A}_{i,j}$ is placed on node $N_{(i\%g_1)g_2+j\%g_2}$. Within each node, blocks of $\mathbf{A}$ are placed round-robin over available workers. In Ray worker-level mapping is ignored since worker processes on the same node use a shared-memory object store. In our example, $\mathbf{A}_{2,3}$ is placed on node N_1 and worker W_3: The node placement is straightforward, whereas the worker-level placement is due to 3 other partitions which are placed on node N_1. Figure 3.3b depicts the grouping of partitions within each node.

While creation operations are immediately executed, arithmetic operations in NumS are *lazily* executed. Figure 3.4 provides the subgraphs induced when a particular operation is performed among GraphArray's consisting of leaf vertices. When an operation is performed on one or more GraphArrays, an array of subgraphs is generated, depicting the sub-operations required to compute the operation. Operations performed on GraphArrays are referred to as *array-level* operations, whereas the sub-operations performed among leafs and vertices are referred to as *block-level* operations. The operands involved in *block-level* operations are referred to as *blocks*. Blocks are either materialized subarrays, or *future* subarrays which have not yet been computed.

To perform the $-\mathbf{X}$, a new GraphArray is constructed, and each leaf vertex is replaced by the subgraph given in Subfigure 3.4a. For $\mathbf{X} + \mathbf{Y}$, the dimensions of $\mathbf{X}$ and $\mathbf{Y}$ and their grid decompositions are required to be equivalent. The execution of this expression generates a new GraphArray, comprised of an array of subgraphs given by Subfigure 3.4b.

The $\mathbf{sum}(\mathbf{X}, axis)$ operation depends on the $Reduce(add, \dots)$ and $ReduceAxis(add, \mathbf{X}, axis)$



(a) $-\mathbf{X}$ (b) $\mathbf{X} + \mathbf{Y}$ (c) **sum**

(d) **X@Y** (e) **tensordot** (f) **einsum**

Figure 3.4: The subgraphs induced by operations listed in 3.1 within a GraphArray. Rectangular vertices correspond to leaf vertices, and circular vertices correspond to operations. Dashed edges and borders are used to denote arbitrary repetition. Σ corresponds to $Reduce(add, \dots)$, σ corresponds to $ReduceAxis(add, \mathbf{X}, axis)$, and @, t, e correspond to matrix multiplication, tensor dot, and Einstein summation, respectively.

vertex types. The *ReduceAxis* vertex type takes a single block and reduces it using the given operation and axis. The *Reduce* vertex type takes any number of blocks with equivalent dimension, and reduces them using a given operation. The **sum** operation is implemented by first summing each individual block along the given axis, and then summing output blocks along the same axis. For example, if we performed $\mathbf{A}' = \mathbf{sum}(\mathbf{A}, 0)$ on the array we previously defined, the output blocks of $\mathbf{A}'$ would be vectors with dimension 64, and its array grid would be single-dimensional, consisting of 4 blocks.

Figure 3.5 provides a simple example of matrix multiplication. Matrix multiply is broken down into independent sub-matrix-multiplication operations. The output of each sub operation is summed used the $Reduce(add, \dots)$ vertex type.

The rest of the operations given in Figure 3.4 are structurally similar to **sum** and matrix multiplication. The operations are broken down into sub-operations of the same kind, and a $Reduce(add, \dots)$ vertex is used to sum the intermediate outputs. In this sense, these operations can be viewed as *recursive*.

When two or more sub-operations generated by these induced subgraphs require operands which are located on different nodes, the operation may be executed, or *placed*, on multiple nodes. The decision process for placing operations is left to our scheduling algorithm, which is described in Section 3.5, but LSHS requires a set of *placement options* to be provided by every vertex in a GraphArray. For unary, and *ReduceAxis* operations, there is only a single operand. For element-wise binary operations, the data is already located on the same node and workers by our hierarchical mapping procedure, so only one potential option is provided

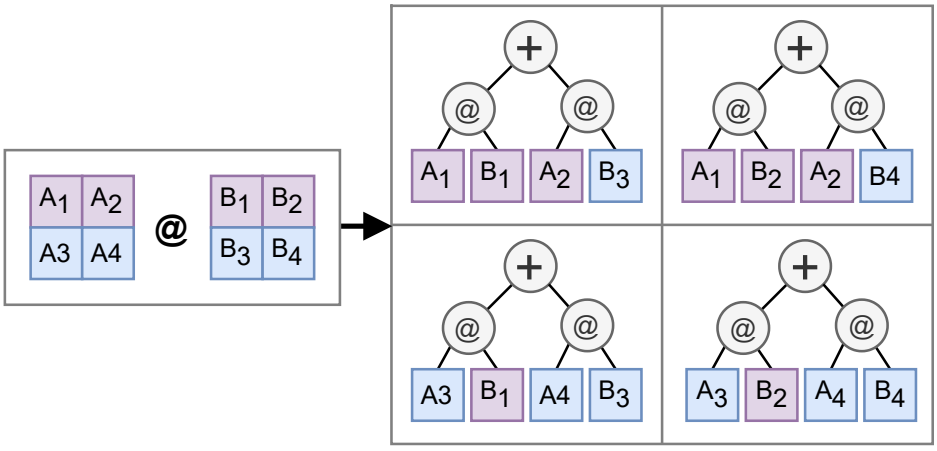


Figure 3.5: Matrix multiplication **A**@**B** of two arrays **A** and **B** partitioned into 2×2 array grids. The operation is invoked on a cluster with node grid 2×1. Blocks on node $0, 0$ are colored purple, and blocks on node $1, 0$ are colored blue.

to the scheduler for these vertex types. For matrix multiplication, tensor dot, and einsum, the set of placement options is the union of all the nodes on which all the operands reside.

The *Reduce* vertex may have n operands. Our scheduler must place $n-1$ binary operations in order to complete the execution of the *Reduce* vertex. For each of the $n-1$ binary operations, the set of placement options is the union of all the nodes on which each of the two operands reside. The *Reduce* vertex is responsible for deciding which operands to pair for each of the $n-1$ binary operations. We pair operands according to their locality within the hierarchical network. We first pair operands on the same workers, then we pair operands on the same node. Our scheduler must make placement decisions for operands which are not on the same workers. We describe our solution to this scheduling problem in the next section.

3.5 Load Simulated Hierarchical Scheduling

LSHS approximates an optimization-based formulation of operator scheduling on a distributed system. There are three primary components to LSHS: A GraphArray; a cluster state object used to simulate load imposed on the cluster for a particular placement decision; and an objective function which operates on an instance of the cluster state. LSHS is a discrete local tree search algorithm [34]. Pseudo-code for the algorithm is given in Figure 6. The LSHS algorithm executes the GraphArray s by sequentially scheduling *frontier* vertices. An operation vertex is on the frontier when all of its children are leaf vertices. A vertex is sampled from the frontier, and the placement option which minimizes the cost function is selected. The GraphArray is then *transitioned* to a new GraphArray by either updating a

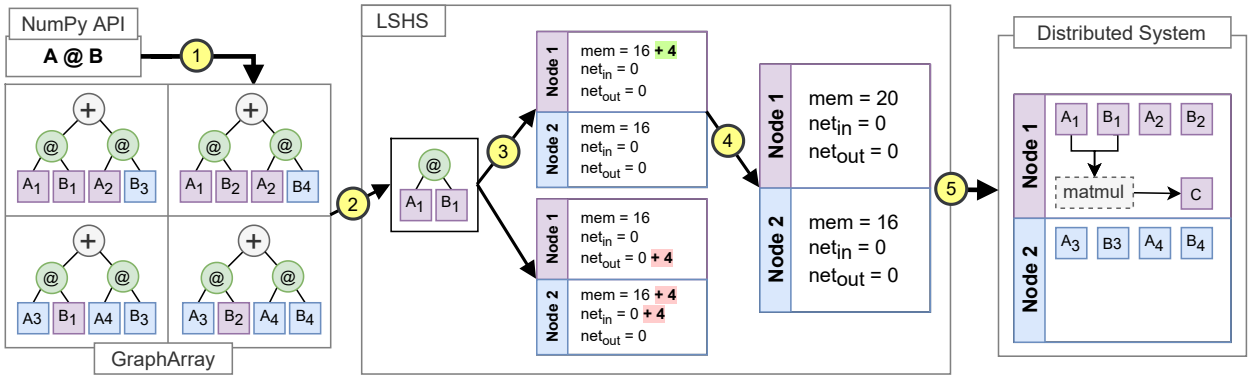


Figure 3.6: Partitioning and scheduling of the expression **AB** on a 2 node cluster, where **A** and **B** are both 4×4, and both have a block shape of 2×2. The example assumes **A** and **B** are already in memory. Objects are color-coded to the nodes on which they reside.

Algorithm 6: LSHS.

Function `lshs(`s`)`**:**
  while `frontier(`s`)` **do**
    $N_{min} \leftarrow null$;
    $C_{min} \leftarrow \infty$;
    $v =$ `sample(frontier(`s`))` ;
    for $i \leftarrow 0$ **to** k **do**
      if `cost(`v, N_i`)` $< C$ **then**
        $C_{min} =$ `cost(`s, N_i`)` ;
        $N_{min} = N_i$;
    $s \leftarrow$ `transition(`s, N_{min}`)` ;
  return s ;

Reduce vertex to reflect its remaining child operands, or converting an operation vertex to a leaf vertex. The algorithm terminates when s consists of all leaf vertices.

Figure 3.6 provides a step-by-step depiction of the execution of the matrix multiplication operation given in 3.12. Each step in the algorithm is depicted by a numerically labeled arrow. Step 1 generates the GraphArray as described in Section 3.4. The vertices which are highlighted green are frontier nodes. Step 2 randomly samples a frontier node. In step 3, each placement option is simulated, and its costs is computed. In step 4, the option that minimizes the cost function is chosen, and the GraphArray is transitioned to its new state. In Step 5, we show how the transition procedure performs the actual remote function call to the underlying distributed system, which carries out the required operation on Node 1.

Cluster State and Optimization Problem

The cluster state is used to monitor the memory and network load imposed on all nodes within a Ray cluster. To simplify exposition, we use the number of elements in an array to signify both memory and network load [1]. For a given node, we compute the network load as two integers: The total number of incoming and outgoing array elements. The memory load is the total number of array elements on a node resulting from the transmission of arrays to that node, and the output of any operation executed on that node. Let $\mathbf{M}$ denote a data structure that maintains a mapping from all objects to their corresponding nodes, and $\mathbf{S}$ denote a $k \times 3$ matrix maintaining the memory, network in, and network out of a k node cluster. Let $m = 0, i = 1, o = 2$ so that $\mathbf{S}_{j,m}$ corresponds to the memory load on node j, and likewise i corresponds to input load, and o corresponds to output load. Let $\mathbf{A}$ correspond to the set of scheduling actions (i.e. nodes on which to schedule an operation) available for vertex $\mathbf{v}$. An action $\mathbf{a} \in \mathbf{A}$ is a tuple $(j, size)$, where j corresponds to the jth node in $\mathbf{S}$, and $size$ corresponds to the size of the output of vertex $\mathbf{v}$. Let $\mathbf{S}', \mathbf{M}' = \mathbf{T}(\mathbf{S}, \mathbf{M}, \mathbf{a})$ be a transition function that takes $\mathbf{M}, \mathbf{S}, \mathbf{a}$ and returns $\mathbf{S}', \mathbf{M}'$ such that the operation $\mathbf{v}$ is simulated on $\mathbf{S}$ via the action $\mathbf{a}$. With $\mathbf{M}$, the transition function $\mathbf{T}$ has enough information to simulate object transfers between nodes.

At a given cluster state $\mathbf{S}, \mathbf{M}$, the objective function which obtains the best action $\mathbf{a}$ from the set of actions $\mathbf{A}$ available to vertex $\mathbf{v}$ is formulated as follows.

$$\begin{aligned} \min_{\mathbf{a} \in \mathbf{A}} \ & \left(\max_{j=1}^{k} \mathbf{S}'_{j,m} + \max_{j=1}^{k} \mathbf{S}'_{j,i} + \max_{j=1}^{k} \mathbf{S}'_{j,o} \right) \\ \text{subject to } \ & \mathbf{S}', \mathbf{M}' = \mathbf{T}(\mathbf{S}, \mathbf{M}, \mathbf{a}). \end{aligned} \tag{3.2}$$

See Figure 6 for pseudo-code, where Equation 3.2 is computed on lines 9-12.

Optimization Problem is NP-Hard

If we modify Equation 3.2 to jointly minimize the maximum memory and network loads over all nodes for a collection of operations, the problem is very similar to load balancing: Given a collection of task execution times, load balancing minimizes maximum execution time over a collection of nodes.

To construct a reduction from load balancing, we need an optimization problem that minimizes the maximum load over all possible scheduling choices. We introduce the superscript t to identify the state of the variables at a given step in the sequence of actions required to compute the optimal solution. The superscript t is used to identify the vertex $\mathbf{v}^t$ being computed at step t. In addition to containing the node on which to compute vertex $\mathbf{v}^t$ and its output size, the set of actions $\mathbf{A}^t$ are expanded to include the next vertex $\mathbf{v}^{t+1}$ on which to operate. We incorporate these new requirements into the transition function $\mathbf{T}$

[1] An additional coefficient we use to discount worker-to-worker communication within the same node on Dask. Ray does not require such a coefficient since workers operate on a shared-memory object store within each node.

by returning the set of next actions $\mathbf{A}^{t+1}$ as well. We represent the optimal solution as a sequence of actions π, where $\pi^t \in \mathbf{A}^t$. For a computation tree consisting of n operations, the optimal sequence of actions of length n is given as follows.

$$\begin{aligned} \min_{\pi \in (\mathbf{A}^0, \ldots, \mathbf{A}^{n-1})} & \left(\max_{j=1}^{k} \mathbf{S}^n_{j,m} + \max_{j=1}^{k} \mathbf{S}^n_{j,i} + \max_{j=1}^{k} \mathbf{S}^n_{j,o} \right) \\ \text{subject to} \quad & \mathbf{S}^{t+1}, \mathbf{M}^{t+1}, \mathbf{A}^{t+1} = \mathbf{T}(\mathbf{S}^t, \mathbf{M}^t, \pi^t). \end{aligned} \tag{3.3}$$

The optimization problem given in Equation 3.3 is NP-hard by a straightforward reduction from load balancing. Tasks in load balancing are independent. In our formulation, independent tasks require no object transfers between nodes. Thus, a load balancing problem instance can be converted to an instance of the problem given by Equation 3.3, and all solutions to these problem instances will have zero network load. Instead of maximum memory load, the remaining term $\max_{j=1}^{k} \mathbf{S}^n_{j,m}$ is used to compute the maximum time for all tasks to execute on any given node. □

3.6 Generalized Linear Models

Listing 3.1: Newton's Method.

```
for _ in range(max_iter):
    mu = model.forward(X, beta)
    g = model.gradient(X, y, mu, beta=beta)
    H = model.hessian(X, y, mu)
    beta += -linalg.inv(app, H) @ g
    if app.max(app.abs(g)) <= tol:
        break
```

To illustrate NumS's usage, we provide a detailed explanation of the implementation and execution of Newton's method on generalized linear models (GLMs) [24, 6]. This is in fact a concrete implementation, with scheduling, of the distributed memory GLM algorithm outlined in Chapter 2. Given a tall-skinny dataset $\mathbf{X} \in \mathbb{R}^{n \times d}$ decomposed into a grid $q \times 1$ of blocks, $\mathbf{y} \in \mathbb{R}^{n \times 1}$ decomposed into a grid $q \times 1$, a GLM m with corresponding twice differentiable convex objective f, and minimum gradient norm ϵ, Algorithm 3.1 computes the global minimum $\beta \in \mathbb{R}^d$ of f using Newton's method. Upon initialization, β is decomposed into a 1×1 grid.

We work through the execution of Algorithm 3.1 on an $r \times 1$ grid of nodes. We use the notation $\mathbf{N}_{i,0}$ to refer to the ith node in the $r \times 1$ grid. We assume the usual block cyclic data layout, so that $\mathbf{X}$ and $\mathbf{y}$ are distributed row-wise over r nodes, and the single block $\beta_{0,0}$ of β is created on node $\mathbf{N}_{0,0}$. In this example, we assume that expressions are computed upon assignment. For example, LSHS is applied to the computation tree induced by the expression $\mu = m(\mathbf{X}, \beta)$, which schedules the operations which comprise the computation

tree and places the output μ using the block-cyclic data layout. Similarly, the expressions assigned to $\mathbf{g}$, $\mathbf{H}$, and β are represented as computation trees which are submitted to LSHS for scheduling.

Let $m = model.forward$. For logistic regression, we have that $m(\mathbf{X}, \beta) = \frac{1}{1+e^{-\mathbf{X}\beta}}$. We see first that the linear operation $\mathbf{X}\beta$ will yield an intermediate value $\mathbf{C}$ comprised of a grid of blocks $\mathbf{C}_{i,0} = \mathbf{X}_{i,0} \times \beta_{0,0}$. It is usually the case that $\mathbf{X}_{i,0}$ has more elements than β, so LSHS will broadcast $\beta_{0,0}$ to all the nodes on which the $\mathbf{X}_{i,0}$ reside. LSHS will schedule the remaining unary and binary operations to the blocks of $\mathbf{C}$ in-place: Since all data is local, the node placement options are reduced to the node on which all of the data already resides. The output μ is decomposed into a $g_1 \times 1$ matrix and distributed according to the block-cyclic data layout.

The gradient of f is given by the expression $\mathbf{X}^T(\mu - \mathbf{y})$. We see that μ has the required grid decomposition to perform the element-wise subtraction operation with $\mathbf{y}$, yielding an intermediate vector $\mathbf{c}$. Because μ and $\mathbf{y}$ have the same size, partitioning, and layout, the data is local, and LSHS schedules the operations without data movement. In NumS, transpose is executed lazily by fusing with the next operation. Thus, the transpose operator is fused with the matrix-vector multiply, resulting in the final set of operations $\mathbf{g}_{0,0} = \sum_{h=0}^{g_1-1}(\mathbf{X}^T)_{0,h}\mathbf{c}_{h,0}$. Like the element-wise operations, because blocks of $\mathbf{X}$ and $\mathbf{c}$ have the same size and partitioning along axis 0, the product operation executes locally. The sum of the products is executed as a reduction: A reduction tree is formed, and any local sums are executed first. The remaining sums are scheduled according to the cost function defined for LSHS. Like β, $\mathbf{g}$ is comprised of a single block, therefore the final sum is scheduled on node $\mathbf{N}_{0,0}$ to satisfy the block-cyclic data layout required by the outputs of LSHS.

The Hessian of f is given by the expression $\mathbf{X}^T(\mu \times (1-\mu) \times \mathbf{X})$, where $\times$ denotes element-wise multiplication. The expression $\mu \times (1-\mu)$ is scheduled locally by LSHS because the input data for these sub trees all reside on the same node. The intermediate vector $\mathbf{c}$ resulting from the previous operation has the same block decomposition as μ, and μ has the same size and partitioning as the first dimension of $\mathbf{X}$, therefore the blocks of $\mathbf{c}$ and $\mathbf{X}$ are distributed the same way over the r nodes. Thus, the vector-matrix element-wise operation $\mathbf{c} \times \mathbf{X}$ is executed without data movement: NumPy executes this kind of expression by multiplying $\mathbf{c}$ with every column of $\mathbf{X}$, yielding an intermediate matrix $\mathbf{C}$ with the same decomposition as $\mathbf{X}$. Finally, the operation $\mathbf{X}^T\mathbf{C}$ results in the computation $\mathbf{H}_{0,0} = \sum_{h=0}^{k-1}(\mathbf{X}^T)_{0,h}\mathbf{C}_{h,0}$, where $\mathbf{H}$ is square with dimension d. The matrix multiplications are executed without data movement, and the sum operation is executed as a reduction like the previous reduction. Since $\mathbf{H}$ is square with dimension d, it is single partitioned, and the final operation to compute $\mathbf{H}$ is scheduled to occur on node $\mathbf{N}_{0,0}$.

Finally, we update β. At this point, β, $\mathbf{g}$ and $\mathbf{H}$ are all comprised of single blocks, and all reside on node $\mathbf{N}_{0,0}$, so the update to beta is executed locally on node $\mathbf{N}_{0,0}$. Similar to the update of β, the gradient norm is computed locally on node $\mathbf{N}_{0,0}$.

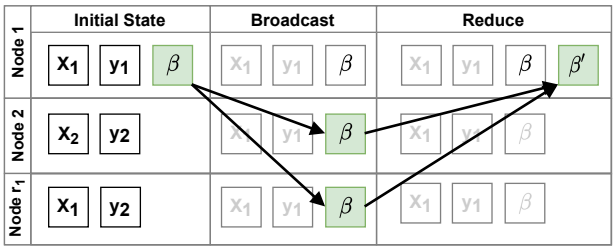


Figure 3.7: Optimal layout and scheduling of data-parallel training.

Data Parallel Machine Learning

In figure 3.7, we show a parameter server design for data-parallel training of machine learning algorithms: The model parameter β is broadcasted to a collection of worker nodes, the worker nodes perform some local computations, and the results of their computations are aggregated on the parameter server to update β for the next iteration of training. The scheduling behavior of CRTS as described in section 3.6 uses node $\mathbf{N}_{0,0}$ as a parameter server, where the aggregation procedure is done more efficiently as a tree reduction of the local outputs of the worker processes. Thus, CRTS achieves similar communication patterns to state-of-the-art implementations of synchronous distributed data parallel machine learning.

3.7 Communication Analysis

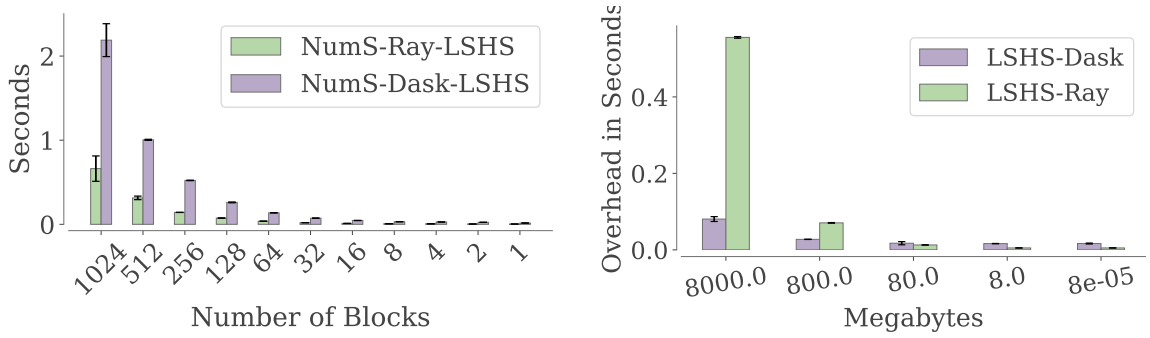
We extend the $\alpha-\beta$ model of communication to analyze the communication time of element-wise, reduction, and basic linear and tensor algebra operations. In our model, for a particular channel of communication, α denotes latency and β denotes the inverse bandwidth of the channel. We also model the time to dispatch an operation from the driver process as γ. The time to transmit n bytes between two nodes is given by $C(n) = \alpha + \beta n$. We also model the implicit cost of communication between workers within a single node of Ray as $R(n) = \alpha' + \beta' n$. For a dense array of size N, let p the number of workers, $N/p = n$ the block size, or number of elements, and $r = p/k$ the number of workers-per-node on a k node cluster.

The results of our theoretical analysis are summarized in 3.2. A rigorous analysis is given in Chapter 4. We drop intra-node communication time whenever inter-node communication is required, since inter-node communication will dominate. We also exclude the γ in our table, since this constant is present in every placement decision in both Dask and Ray. For element-wise operations, LSHS on Dask achieves the given lower bound of 0, which is not listed in the above table.

We are not able to provide bounds for square matrix multiplication for LSHS, but we are able to provide evidence that LSHS has the potential to achieve competitive performance to SUMMA. SUMMA has a communication time of $2\sqrt{p}\log(\sqrt{p})C(n)$, which is greater than

Operation	Lower Bound	LSHS
$-\mathbf{x}$	0	$R(n)$
$\mathbf{x}+\mathbf{y}$	0	$R(n)$
$\mathbf{sum}(\mathbf{X})$	$log_2(k)C(n)$	$log_2(k)C(n)$
$\mathbf{X}^T\mathbf{Y}$	$log_2(k)C(n)$	$log_2(k)C(n)$
$\mathbf{X}\mathbf{Y}^T$	$2(\sqrt{k}-1)rC(n)$	$2(\sqrt{k}-1)rC(n)$
$\mathbf{XY}$	$\left(\sqrt{k}+\log(\sqrt{k})\right)rC(n)$	-

Table 3.2: All arrays are partitioned row-wise into p partitions, except for arrays in $\mathbf{X}\mathbf{Y}^T$, which are partitioned row-wise into $\sqrt{p}$ partitions, and arrays in $\mathbf{XY}$, which are square and partitioned into a $\sqrt{p} \times \sqrt{p}$ grid. For **sum**, $\mathbf{X}^T\mathbf{Y}$, and $\mathbf{X}\mathbf{Y}^T$, arrays are $\mathbb{R}^{n,d}$ where $n >> d$.



(a) Control Overhead.

(b) RFC Overhead.

Figure 3.8: Control overhead can be seen by measuring the time to allocate a vector of dimension 1024 on a 16 node cluster, with a total of 1024 workers (one per virtual core). This is captured by the γ term. As we decrease the number of blocks, γ decreases. RFC overhead is measured by executing $-\mathbf{x}$ on a single block vector $\mathbf{x}$, forcing the system to execute the task using a single worker. The overhead is directly measured as the difference between the time it takes to perform this operation using NumPy. Ray writes task outputs to an object store, resulting in greater RFC overhead. This is captured by the $R(n)$ term in our analysis.

the lower bound we provide for matrix multiplication. In Chapter 4, we also provide a deterministic algorithm for matrix multiplication which achieves a bound that is $\sqrt{k}$ faster than SUMMA. We also show that this algorithm is balanced for memory and network load, which proves the existence of this algorithm within the search space of LSHS.

We are also unable to provide bounds for Ray's and Dask's dynamic schedulers. Since data placement is dynamic, it is possible that all data may be placed on any number of nodes between 1 and k with no locality guarantees. We believe this can often lead to scheduling

decisions which require inter-node communication among all nodes, but we are not able to show this formally.

3.8 Evaluation

The primary focus of this paper is LSHS, a scheduler which enhances NumS's performance by balancing data placement within multi-node cloud-based distributed systems, and placing operations in a fashion which maintains load balance while minimizing inter-node communication. While NumS provides automatic block partitioning like Dask and Spark, these partitioning heuristics introduce additional complexity to our evaluation of LSHS, which is the primary focus of this work. We therefore manually tune block partitioning and node/worker/process grid layouts where applicable to evaluate peak scheduling performance directly.

We answer the following questions in our evaluation:

1. How does Dask and Ray perform with and without LSHS?
2. How well does NumS on Ray perform for linear algebra, tensor algebra, and machine learning applications?
3. How does NumS compare to state-of-the-art high-performance computing solutions?
4. How does NumS compare to other systems which provide linear algebra, tensor algebra, and machine learning solutions?

Experimental Setup

NumS is designed for medium-scale (terabytes) problem sizes. We evaluate on problem sizes of this scale, in order to highlight the benefits NumS does provide. We provide limitations to our approach in Section 3.7, which show that NumS's performance degrades as the number of array partitions increase, and when partition sizes are too small.

We run all CPU experiments on a cluster of 16 `r5.16xlarge` instances, each of which have 32 Intel Skylake-SP cores at 3.1Ghz with 512GB RAM connected over a 20Gbps network. Each AWS instance is running Ubuntu 18.04 configured with shared memory set to 512GB. For CPU-based experiments, the Ray cluster uses 312GB for the object store, and 200GB for workers. All NumS experiments run on a single thread for BLAS operations and use only Ray worker nodes for computation, leaving the head node for system operations.

All synthetic classification data are drawn from a bimodal Gaussian with 75% of the data concentrated at mean 10 with variance 2 (negative samples), and the remaining 25% concentrated at mean 30 with variance 4 (positive samples). Each sample is 256-dimensional. We sample from these distributions to satisfy the required dataset size. For example, a 64GB dataset of 64bit floats consists of a design matrix X with $31,250,000$ rows and 256 columns, and a target vector y consisting of $31,250,000$ values $\in \{0, 1\}$.

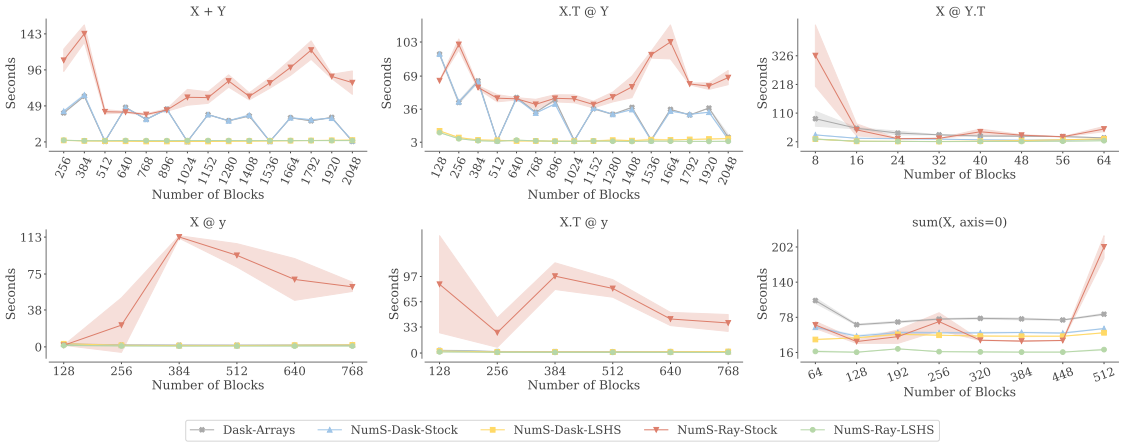


Figure 3.9: An ablation study comparing NumS on Dask and Ray, with and without LSHS, and include Dask Arrays as an additional point of comparison. All experiments are run on 16 node clusters, with 32 workers per node, for a total of 512 workers. In all but **sum**, $\mathbf{X}, \mathbf{Y}$ are 1 terabyte arrays, partitioned row-wise. $\mathbf{y}$ is partitioned to match the partitioning of $\mathbf{X}$ in $\mathbf{X}@\mathbf{y}$ and $\mathbf{X}^T@\mathbf{y}$. **sum** is executed on a multi-dimensional tensor partitioned along its first axis.

Unless otherwise noted, all experiments are executed by sampling data using random number generators, and all experiments are repeated 12 times. The best and worst performing trials are dropped to obtain better average performance. This is done primarily to avoid bias results due to cold starting benchmarks on Dask, Ray, and Spark. Our generated datasets resemble the data our industry collaborators encounter.

Dask We evaluate Dask's logistic regression (Dask ML 1.6.0) and QR decomposition, which implements the direct tall-skinny QR decomposition [5].

Spark We evaluate Spark-MLlib's (v2.4.7) logistic regression and QR decomposition. Logistic regression uses the L-BFGS solver from Breeze, which is an open source Scala library for numerical processing. Mllib's QR decomposition implements the indirect tall-skinny QR decomposition [13] and uses the QR implementation from Breeze, which is internally implemented using LAPACK.

Microbenchmark Ablation

The results of our ablation study are given in Figure 3.9. In every experiment, NumS on Ray is significantly enhanced by LSHS. For $\mathbf{X} + \mathbf{Y}$ and $\mathbf{X}@\mathbf{Y}^T$, NumS on Dask without LSHS and Dask Arrays achieve good performance whenever the number of partitions is divisible by the number of workers, whereas LSHS performs addition with 0 and $\mathbf{X}@\mathbf{Y}^T$ with minimal

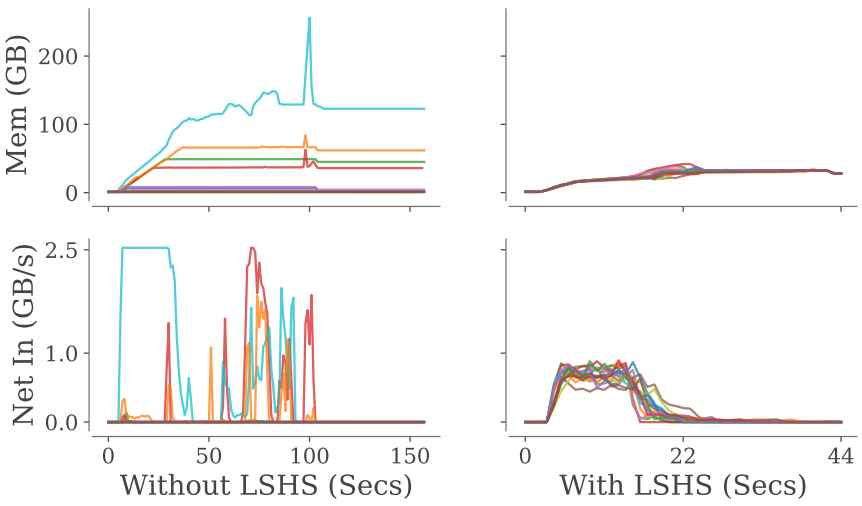


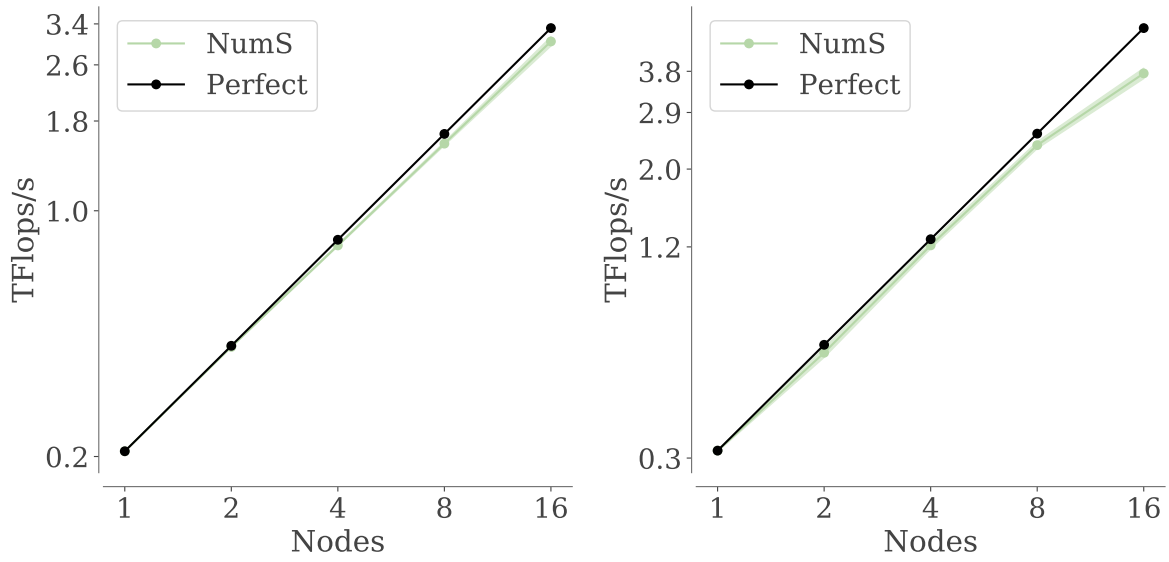
Figure 3.10: Memory and network load for NumS with and without LSHS.

communication. With fewer partitions, we believe $\mathbf{X}@\mathbf{Y}^T$ is sub-optimal on Dask and Ray due to assigning large creation operations on fewer nodes, resulting in under-utilization of available cluster resources. For matrix-vector operations, LSHS does not provide any significant enhancement over Dask scheduling. The optimal scheduling behavior is to move $\mathbf{y}$ to the nodes on which the partitions of $\mathbf{X}$ reside. For NumS on Ray without LSHS, we observe object spilling due to too many large objects being assigned to a few nodes, and large object transmissions between nodes. For **sum**, we measure the performance of reductions over large object transmissions. Ray's high inter-node throughput achieves good results for this experiment. We believe Dask Array's poor performance on this task is due to a sub-optimal tree reduce, pairing partitions which are not co-located. Overall, we see that NumS on Ray is the most robust to partitioning choices, and achieves well-rounded performance. See Section 3.7 for our theoretical analysis, which supports the claims we make in these results.

Memory and Network Load Balancing

We measure memory, network in, and network out of every node at equally spaced intervals for NumS on Ray with and without LSHS. For these experiments, we use 16 nodes with 32 workers per node and measure over a single iteration of Newton's method on a 128GB logistic regression problem. These experiments measure execution time and resource utilization required to load data from S3, execute a single iteration of Newton's method.

Figure 3.10 shows memory usage and network input over one iteration of Newton's method. Each curve tracks the load on one node. Densely clustered curves, where all nodes have similar load over the experiment, indicate good load balance. Lower y-axis values in-



(a) QR Decomposition.

(b) Logistic Regression.

Figure 3.11: QR decomposition achieves near-perfect scaling. Logistic regression exhibits a slowdown at 16 nodes primarily due intermediate reduction operations over a 20Gbps network.

dicate lower resource footprint. LSHS significantly enhances NumS performance on Ray in terms of load balance and resource footprint. Without LSHS, Ray executes the majority of submitted tasks on a single node, while LSHS distributes load without increased network communication. In particular, Ray's bottom-up scheduling does not define a clear strategy for scheduling tasks with no dependencies [23], potentially resulting in a sub-optimal data layout for both element-wise and linear algebra operations. Overall, for this task, LSHS enhances maximum memory load by about 60%, and maximum network load by about 80%.

Scalability

We measure the weak scaling of NumS on dense square matrix multiplication, our implementation of indirect QR decomposition, and logistic regression using Newton's method. We use 32 workers per node, the number of physical cores available on `r5.16x` EC2 instances.

For each doubling of resources, we double the amount of work, starting with 64GB of data on a single node. QR and logistic regression are evaluated on tall-skinny matrices. For QR-decomposition, scaling is near perfect (Figure 3.11a). For logistic regression, scaling is near perfect until we execute on 16 nodes, at which point performance degrades due to inter-node reductions over a 20Gbps network (Figure 3.11b).

For dense square matrix multiplication, we compare to state-of-the-art baselines ScaLA-

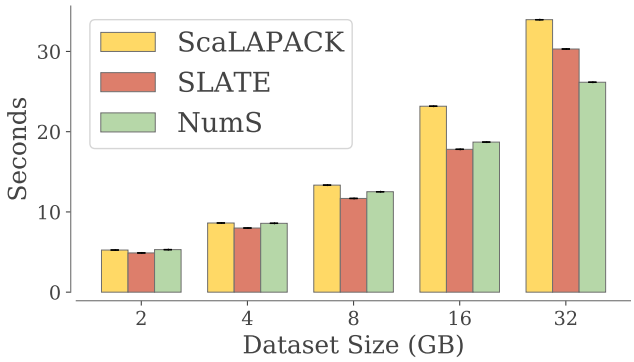


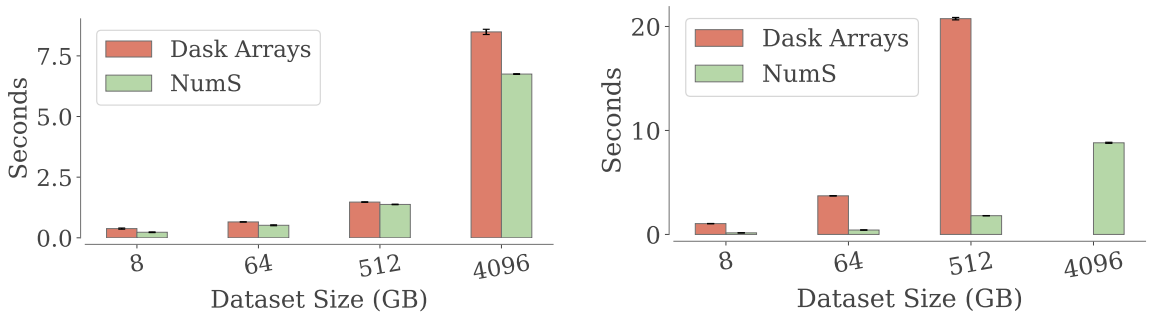
Figure 3.12: Dense square matrix-matrix multiplication.

	2GB	**4GB**	**8GB**	**16GB**	**32GB**
ScaLAPACK	224	480	992	64	992
SLATE	992	928	992	1408	992
NumS	3953	3727	3953	5591	5271

Table 3.3: Square block size settings for ScaLAPACK, SLATE, and NumS on the DGEMM benchmark. DGEMM is distributed over 1 node for 2GB, 2 nodes for 4GB, etc. up to 32GB on 16 nodes.

PACK and SLATE [8, 16]. We start with 2GB matrices on a single node, and double the amount of data as we double the number of nodes. We tune all libraries to their optimally performing block dimension. These settings are given in Table 3.3.

Figure 3.12 shows that NumS is competitive with HPC libraries on this benchmark. Both ScaLAPACK and SLATE implement the Scalable Matrix Multiplication Algorithm (SUMMA) [17] for their dgemm routine. While these results may seem surprising, the theoretical results we present in Section 3.7, as well as our comprehensive analysis in Chapter 4 show that our approach to parallelizing and scheduling distributed arrays can attain asymptotically lower communication time for this operation. We show the existence of a square dense matrix-matrix multiplication algorithm which is faster than SUMMA by a factor of $\sqrt{k}$ on a k node cluster. While the SUMMA algorithm provides good communication bounds on distributed memory, it assumes every process has equivalent communication time. While the difference in communication time between vs. within nodes on supercomputers may not be significant, on multi-node clusters in the cloud, inter-node communication is generally much more expensive than intra-node communication. LSHS places data and operations in a network topology-aware manner. For a particular operation, LSHS approximates the location of data, including data which is cached by Ray's object store from previous object transmissions. This enables LSHS a greater variety of operator placement decisions, enabling it to



(a) Tensor Double Contraction. (b) MTTKRP.

Figure 3.13: Comparison of tensor algebra operations on NumS vs Dask Arrays.

reduce inter-node communication by placing operations on nodes where operands are already co-located. Coupled with a locality-aware reduction operation, the final set of summations invoked in this operation can be performed entirely locally prior to performing an inter-node reduction. Furthermore, the overhead associated with performing intra-node communication among processes is implicit given Ray's shared-memory object store.

Tensor Algebra

We compare NumS to Dask Array's implementation of the **tensordot** and **einsum** operators, primitives which enable the expression of distributed dense tensor algebra operations. We perform these operations on a 16 node cluster with 32 workers per node. For the **einsum** operator, we perform the Matricized Tensor Times Khatri Rao Product (MTTKRP), which we express in Einstein summation notation. This operation is the closed-form solution to the alternating least squares algorithm for tensor factorization [35]. For the **tensordot** operator, we perform the standard tensor double contraction on operands which frequently occur in a variety of other tensor decompositions[31].

For MTTKRP, we sample $\mathbf{X} \in \mathbb{R}^{I \times J \times K}$, $\mathbf{B} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$, and perform $\textbf{einsum}(ijk, if, jf \rightarrow if, \mathbf{X}, \mathbf{B}, \mathbf{C})$. For both Dask and NumS, we partition every array to achieve peak performance. In NumS, we also tune a cubic configuration of the available compute nodes, to further control the mapping of partitions to nodes. We set $F = 100$ and vary the number of elements of $I = J = K$ to set the size of $\mathbf{X}$ from 8GB to 4TB.

For tensor double contraction, we sample $\mathbf{X}$ identically to the MTTKRP benchmark, and sample $\mathbf{Y} \in \mathbb{R}^{J \times K \times F}$ with $F = 100$. We also vary the size of $\mathbf{X}$ identically to the MTTKRP benchmark.

Both results depend heavily on LSHS. Array partitioning and node grid also play a significant role. For NumS, MTTKRP partitioned along dimension J, and a node grid of $16 \times 1 \times 1$ performed best. For the double contraction benchmark, a node grid of $1 \times 16 \times 1$ performed best, with relatively balanced partitioning along dimensions J and K. In both

benchmarks, both NumS and Dask Arrays perform a collection of sum-of-products. Both libraries perform tree-based reductions.

For MTTKRP, Dask performed best with relatively balanced partitioning of blocks, but is unable to achieve good initial data placement to minimize inter-node communication due to its inability to specify a node grid. Furthermore, its reduction tree is constructed before any information about the physical mapping of blocks to nodes is available, resulting summations between vs. within nodes. After tuning, the 4TB benchmark on Dask Arrays took approximately 241.9 seconds, which exclude from Figure 3.13.

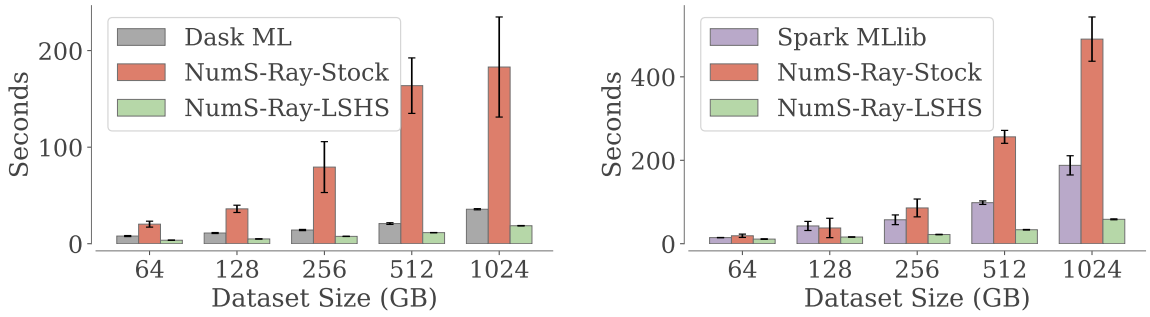
For double contraction, Dask and NumS performance is relatively the same. Unlike MTTKRP, there is no good factoring of 16 nodes into a node grid that LSHS can take advantage of in order to reduce inter-node communication. This is mainly due to the structure of the problem: The tensor contraction sums over the dimensions J and K, and the ordering of dimensions for tensors X and Y align only along dimension J. This is why the $1 \times 16 \times 1$ factoring of the nodes performs best.

Machine Learning

We compare NumS's performance on logistic regression to Dask ML and Spark MLlib. We include results for NumS on Ray without LSHS to highlight the role of scheduling. Dask and Spark implement different versions of these algorithms, so we implement both versions of both algorithms for a fair comparison. In these experiments, we hold the cluster resources fixed at 16 nodes, varying only the dataset size to evaluate the performance of each system. All experiments perform the same number of steps and operations.

For our comparison to Dask, we sample data row-wise in 2GB blocks, which yields peak performance for both Dask and NumS. We use Newton's method for both libraries. Newton's method is optimal for logistic regression's convex objective. Figure 3.14a shows that NumS outperforms Dask at every dataset size. The performance gap is partially explained by differences between LSHS and Dask's dynamic scheduler. We believe the majority of the performance gap is due to Dask ML's implementation logistic regression which, based on our inspection of their source code, aggregates gradient and hessian computations on the driver process to perform updates to model parameters and test for convergence.

Since Spark does not support Newton's method, we compare our implementation of the L-BFGS optimizer to Spark's version. We initialize our logistic regression implementation to execute 10 optimization steps, with no regularizer, and L-BFGS configured to use a history length of 10. Both implementations use identical line search algorithms and are configured identically. Figure 3.14b shows that our implementation of logistic regression with the L-BFGS optimizer outperforms Spark. We have read Spark's logistic regression and Breeze's L-BFGS [10] thoroughly, which shows that logistic regression with the L-BFGS optimizer is essentially a statically scheduled implementation. To our knowledge, the algorithms and scheduling of operations on partitions is identical to NumS's implementation, and the scheduling behavior of LSHS for this problem. We therefore believe that the performance gap is explained by differences between Spark and Ray.



(a) Newton's method.

(b) L-BFGS.

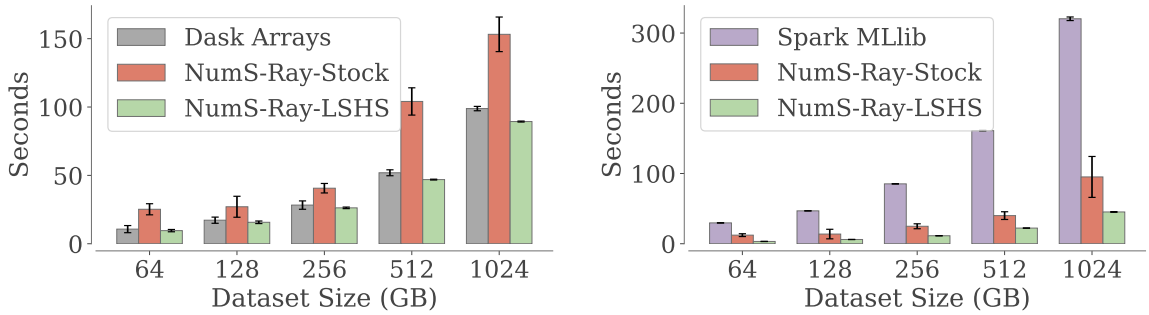
Figure 3.14: Logistic regression runtime on NumS, Dask, and Spark.

Linear Algebra

In this section, we evaluate the performance of NumS, Dask, and Spark on QR decomposition. We include results for NumS on Ray without LSHS to highlight the role of scheduling. All experiments perform the same number of steps and operations. Each experiment is repeated 12 times, and the best and worst performing trials are dropped to obtain better average performance.

We compare Dask and NumS on the direct tall-skinny QR decomposition algorithm [5]. Similar to our logistic regression experiment, we sample data row-wise in 2GB blocks, which achieves peak performance for both libraries. Figure 3.15a shows the results of our direct TSQR benchmarks. NumS performs comparably to Dask on this benchmark. Dask's QR decomposition implementation is highly optimized and does not rely on any array operations. Dask's direct tall-skinny implementation requires a single column partition, leaving only one dimension along which data is partitioned. The partitioning of data for Dask which achieves peak performance implicitly results in data locality for a number of intermediate operations due Dask's round-robin placement of initial tasks. The results of our ablation study in Figure 3.9 measures this phenomena directly.

Since Spark does not implement a direct QR decomposition, we evaluate both NumS and Spark on indirect TSQR decomposition. Similar to logistic regression, Spark's implementation of indirect TSQR decomposition is sensitive to partition tuning. Figure 3.15b compares the results of our indirect TSQR implementation to Spark's. Similar to direct TSQR for Dask, indirect TSQR is a statically scheduled algorithm for Spark and NumS. Since both algorithms are identical, and scheduling is static, we attribute the difference in performance to differences between Spark and Ray, which is further evidenced by the results obtained for NumS without LSHS.



(a) Direct TSQR.

(b) Indirect TSQR.

Figure 3.15: TSQR runtime on NumS, Dask, and Spark.

Tool Stack	Load	Train	Predict	Total
Python Stack	65.55	61	0.43	126.98
NumS	11.79	3.21	0.20	15.2

Table 3.4: NumS vs. a Python stack consisting of Pandas for a data loading, and scikit-learn (which uses NumPy) for training a logistic regression model. All values are reported in seconds.

Comparison to Python Tools

NumS not only provides a speedup in distributed memory settings, but it also provides a significant speedup on a single node for GLMs. All experiments in this section make use of NumS's automatic block partitioning, showing that a speedup can be achieved by simply replacing Python import statements of the libraries used in this section with NumS equivalents. While these experiments run on the same instances we have been using throughout this section, NumS is pip-installable on your laptop and can provide comparable speedups to what we present in this section.

Figure 3.16 shows that NumS achieves a speedup over NumPy that scales linearly with the number of blocks used in the computations. Since NumS uses NumPy's PCG64 RNG to enable parallel sampling, we also use the non-default PCG64 RNG for our NumPy baseline (which is faster than the default RNG implementation). The growing gap between single-block sampling and NumPy is due to Ray's RPC overhead: When an RPC, such as `sample`, is invoked, the output of the RPC is copied into Ray's object store, causing overhead that grows linearly with the size of the data. Our data sampling experiment samples data from the same uniform distribution using the `rand` method, provided by both NumPy and NumS.

Pandas, NumPy, and scikit-learn make up a common stack for data science in Python. NumS provides a parallel `read_csv` method comparable to Pandas', eliminating one layer in the package stack for numerical CSV files. Table 3.4 shows that NumS achieves an 8× speedup over Pandas' [37] serial `read_csv` operation with scikit-learn's training and

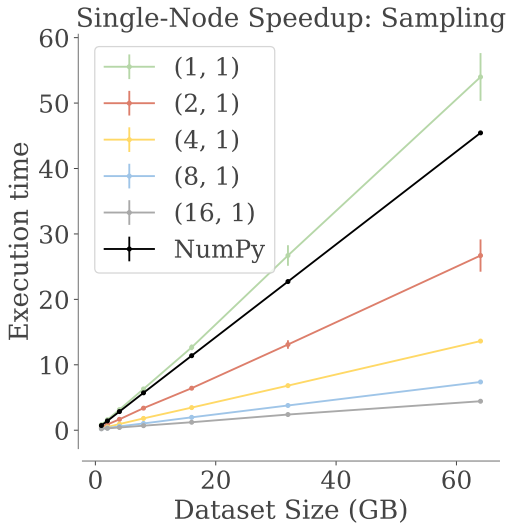


Figure 3.16: Parallelization of sampling. $(k, 1)$ is the node grid used for NumS, effectively scaling the available resources by k.

prediction procedures for a logistic regression model trained on the 7.5GB HIGGS dataset [3]. Both NumS and scikit-learn's logistic regression are configured to use 32 cores.

We tune scikit-learn to use its fastest optimizer, which is l-bfgs. Compared to NumS' Newton optimizer, l-bfgs requires significantly more iterations to converge. Furthermore, l-bfgs requires line search at every iteration, which requires multiple calls to the logistic regression objective function. Newton's method does not require a line search and converges in fewer iterations than l-bfgs. Our Newton's method implementation is optimized for the kinds of tall-skinny matrices which commonly occur in data science, achieving greater utilization of available memory and cores through efficient parallelization of basic linear algebra operations.

To further dig into these differences, we compare NumS to scikit-learn on different fractions of the HIGGS dataset. Figure 3.17 shows that, at smaller scales, NumS is 5× slower than scikit-learn, and at larger scales, it is 20× faster than scikit-learn.

Beyond differences in the optimizer, the primary difference in performance is due to NumS's parallelization of all array operations, not just those parallelized by the underlying system's BLAS implementation. We measure this by implementing Newton's method in pure NumPy, with full parallelization of BLAS operations (32 cores). We measure the amount of time spent in serial operations vs. parallel operations, and we find that *90% of the time for Newton's method using NumPy is spent on serial operations.* In total, our NumPy implementation of Newton's method takes approximately 11 seconds, a 5× speedup over scikit-learn's fastest optimizer. Compared to this implementation of Newton's method, NumS achieves a speedup of 3.5×. Compared to Newton's method, l-bfgs is comprised of less expensive matrix operations (no direct computation of the Hessian), and many more

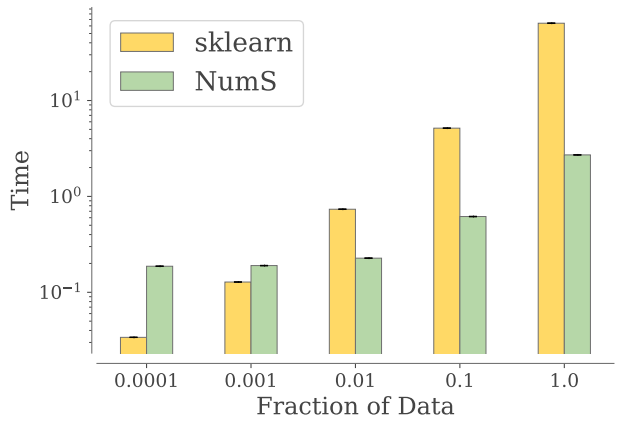


Figure 3.17: Evaluating NumS vs. scikit-learn on fractions of the HIGGS dataset.

serially executing element-wise operations.

3.9 Discussion

This chapter presents LSHS, a scheduler for numerical array operations in NumS which is optimized for task-based distributed systems. Our results show that LSHS achieves competitive performance with state-of-the-art approaches to scheduling, and in many instances outperforms these libraries. Based on our work, we believe that all distributed data structures, not just distributed arrays, that rely on dynamic scheduling require some combination of LSHS with data layouts optimized for operations on those data structures. In particular, Dask's dataframes and project Modin [29], a portable dataframe abstraction that runs on Dask and Ray, could benefit from similar techniques presented in this work. Future directions for NumS include (1) generalizing our findings and providing a framework on which any distributed data structure can benefit from LSHS; (2) improving the usability of NumS's user API by enhancing automatic block-partitioning and eliminating the need for a user-specified node grid; and (3) reducing RFC overhead by introducing operator fusion.

Chapter 4

Communication Analysis

This chapter presents the complete communication analysis of LSHS. We extend the $\alpha - \beta$ model of communication to analyze the communication time of element-wise, reduction, and basic linear and tensor algebra operations. In our model, for a particular channel of communication, α denotes latency and β denotes the inverse bandwidth of the channel. We also model the time to dispatch an operation from the driver process as γ, called the *dispatch latency*. The time to transmit n bytes is given by $\alpha + \beta n$.

For a dense array of size N, let p the number of workers, $N/p = n$ the block size, or number of elements, and $r = p/k$ the number of workers-per-node on a k node cluster.

Let $C(n) = \alpha + \beta n$ denote the time to transmit n bytes between two nodes within a multi-node cluster. Let $D(n) = \alpha'' + \beta'' n$. This is the cost of transmitting data between workers within a single node in Dask. Let $R(n) = \alpha' + \beta' n$. This expression captures the implicit cost of communication between workers within a single node of Ray. Ray maintains a shared memory key/value store on each node, enabling every worker to access data written by any other worker without explicit communication.

In general, by our assumptions on hierarchical networks, $\alpha >> \alpha'' > \alpha'$, and $\beta >> \beta'' > \beta'$. We expect $\alpha'' > \alpha'$ and $\alpha'' > \alpha'$ because Dask relies on the TCP protocol for object transmission between workers within the same node, whereas Ray writes data directly to Linux shared memory.

All lower bounds are given in terms of Ray's communication time. Our lower bounds depend on the assumption that a block need only be transmitted to a node once, after which it is cached by Ray's object store. We also assume Ray's object store is large enough to hold all intermediate objects which it caches. We present the γ term once as part of the lower bound, since in all cases, the same number of operations are dispatched from the driver process.

Algorithm 7: Recursive Matrix Multiplication.

Function RMM($\boldsymbol{A}$, $\boldsymbol{B}$):
  if NOTPARTITIONED($\boldsymbol{A}$, $\boldsymbol{B}$) **then**
    return MATMUL($\boldsymbol{A}$, $\boldsymbol{B}$) ;
  $\forall_{i=0,j=0,h=0}^{m-1,n-1,k-1} \{\mathbf{C}'_{i,j,h} \leftarrow$ RMM($\boldsymbol{A}$_i,h, $\boldsymbol{B}$_h,j)$\}$;
  $\forall_{i=0,j=0}^{m-1,n-1} \{ \mathbf{C}_{i,j} \leftarrow$ Reduce($\boldsymbol{C}'_{i,j}$) $\}$;
  return $\mathbf{C}$;

4.1 Recursive Matrix Multiplication

Our results are more easily presented and understood in terms of the recursive matrix multiplication algorithm, which is given by Algorithm 7. Nested blocks of a recursively partitioned matrix $\mathbf{X}$ are obtained by the subscripts $((\mathbf{X}_{i_1,j_1})_{i_2,j_2}) \dots)_{i_d,j_d}$, where d is the depth of the nested blocks of arrays. At depth d, a sub-matrix is detected using the predicate **NOTPARTITIONED**, and the sub-matrices are multiplied. In our analysis, we only consider nested matrices of depth 2. We refer to the first level block partitions as the node-level partitions, and the second-level partitions as the worker-level partitions.

4.2 Elementwise Operations

For unary operations (e.g. $-\mathbf{x}$), we assume $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{x}$ is partitioned into p blocks. The lower bound for this operation is γp. LSHS on Dask and the Dask scheduler will incur 0 communication overhead. LSHS on Ray and the Ray scheduler will incur a communication overhead of approximately $R(n)$.

For binary element-wise operations, we assume $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and both are partitioned into p blocks. The lower bound for this operation is γp. LSHS on Dask achieves 0 communication for element-wise operations: The LSHS data placement procedure ensures blocks in $\mathbf{x}$ and $\mathbf{y}$ are stored on the same workers, and the cost of executing on some other worker is strictly greater than executing on the worker on which the two input blocks already exist: The memory load is the same on whatever worker the operation is executed, but the network load is greater on workers other than the worker on which the input blocks already reside. Similarly, LSHS on Ray ensures blocks are placed on the same nodes, but can guarantee at most $R(n)$ time due to constant overhead associated with writing function outputs to its object store.

In our empirical analysis, Dask often coincidentally achieves locality at the node-level and at the worker-level: The results suggest that data is placed round-robin when workers are idle. For instance, when the number of blocks of each input array are divisible by p, Dask's scheduler achieves the same execution time as LSHS.

4.3 Reduction Operations

We define $reduce(bop, \mathbf{X})$ as the reduction of operation *bop* on blocks of $\mathbf{X}$. The *reduce* operation constructs a binary tree of binary operations *bop* over the blocks of $\mathbf{X}$.

Assume $\mathbf{X} \in \mathbf{R}^{n,d}$ is tall-skinny, $n >> d$, so that $\mathbf{X}$ is block-partitioned along its first axis into p partitions of blocks with dimensions $(n/p) \times d$. For the addition operation, the *reduce* operation sums all blocks in $\mathbf{X}$ and outputs a block of $(n/p) \times d$. As a subroutine of $reduce(bop, \mathbf{X}, axis)$ and $tensordot(X, Y, axes)$, this operation is critical to achieving high performance reduction and tensor algebra operations.

Due to the same argument presented for element-wise operations, LSHS first performs *bop* on operands which already reside on the same nodes. Recall that $r = p/k$. The lower bound for this operation is $\gamma(p-1) + log_2(r)R(n) + log_2(k)C(n)$.

For Dask, LSHS incurs a cost of $log_2(r)D(n)$ for local reductions, plus $log_2(k)C(n)$ for the remaining k blocks. Likewise, LSHS on Ray incurs a cost of $log_2(r)R(n) + log_2(k)C(n)$.

4.4 Block-wise Inner Product

Assume $\mathbf{X}, \mathbf{Y} \in \mathbf{R}^{n,d}$ is tall-skinny, $n >> d$, so that $\mathbf{X}, \mathbf{Y}$ are block-partitioned along their first axes into p partitions of blocks with dimensions $(n/p) \times d$. Under these conditions, we define the block-wise inner product as $X^T Y$. This is the most expensive operation required to compute the Hessian matrix for generalized linear models optimized using Newton's method.

This operation will execute matrix multiplication between X_i and Y_i, where Z_i denotes the ith block of array Z. Let the output of the previous proceduce be denoted by the block-partitioned array W. The final step of this operation is $reduce(add, W)$. Thus, the analysis provided for element-wise and reduce operations also apply to this operation.

The lower bound for this operation is $\gamma(p+p-1) + log_2(k)C(n) + (1+log_2(r))R(n)$. For LSHS on Dask, we have $log_2(k)C(n) + log_2(r)D(n)$, and for Ray we have $log_2(k)C(n) + (1 + log_2(r))R(n)$.

Empirically, we observe that LSHS on Ray is slightly faster than LSHS on Dask for this operation. This suggests that $(1 + log_2(r))R(n) < log_2(r)D(n)$ and $R(n) < log_2(r)(D(n) - R(n))$, which is reasonable given our assumption that $R(n) < D(n)$. As $R(n)$ goes to 0, this inequality goes to $0 < log_2(r)D(n)$, suggesting that Dask's performance is explained by worker-to-worker communication within a single node.

4.5 Block-wise Outer Product

Assume $\mathbf{X}, \mathbf{Y} \in \mathbf{R}^{n,d}$ is tall-skinny, $n >> d$, so that $\mathbf{X}, \mathbf{Y}$ are block-partitioned along their first axes into $\sqrt{p}$ partitions of blocks with dimensions $(n/\sqrt{p}) \times d$. The block-wise outer product is defined as $\mathbf{X}\mathbf{Y}^T$. The output $\mathbf{Z}$ will be a $\sqrt{p} \times \sqrt{p}$ grid of blocks. Every node must transmit $2(\sqrt{k} - 1)rC(n)$ node-level blocks to every row and column in its grid (mi-

nus itself), and every off-diagonal node must receive $2rC(n)$ node-level blocks, resulting in a communication lower bound of $2(\sqrt{k}-1)rC(n)$, which is also the communication time attained by LSHS.

For LSHS on Dask, blocks placed within the diagonal of our $k \times k$ logical grid of nodes will not incur any inter-node communication overhead. This constitutes k of the k^2 logical nodes. Within this diagonal grid of nodes, blocks placed within the diagonal of the $r \times r$ grid of logical workers within each node can be used to compute the output $\mathbf{Z}_{i,i} = \mathbf{X}_i \mathbf{Y}_i^T$ without worker-to-worker communication. This constitutes r such workers per node along the diagonal of our logical grid of nodes. We therefore incur a communication time of $k(r^2 - r)D(n)$ for blocks placed on the diagonal of the logical grid of nodes. Computing output blocks on the rest of the nodes requires an inter-node communication time of $(k^2 - k)C(n)$.

Example

Let's assume we have $k = 4$ nodes and $r = 4$ workers per node. $\mathbf{X}$ and $\mathbf{Y}$ are therefore partitioned into $p = 16$ blocks. The output $\mathbf{Z}$ will be a $16x16$ grid of blocks distributed over a $2x2$ grid of nodes, and within each node we will distribute the blocks over a $2x2$ grid of workers. Each worker will therefore have a factor $\sqrt{p} = 4$ more blocks. $kr = 16$ of the $\mathbf{X}_i, \mathbf{Y}_i$ blocks will be placed on the same workers, allowing for the corresponding $\mathbf{Z}_{i,i}$ to be computed without communication.

4.6 Matrix Multiplication

Let $\mathbf{Z} = \mathbf{XY}$ for $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{n \times n}$. Both are partitioned into $\sqrt{p} \times \sqrt{p}$ grids. $\mathbf{Z}$ will have the same dimension and partitioning as $\mathbf{X}$ and $\mathbf{Y}$. We have $O(\sqrt{p}^3 = p^{3/2})$ block operations: For each of the $\sqrt{p}^2 = p$ output blocks, we have $\sqrt{p}$ matrix multiplies, and $\sqrt{p} - 1$ additions. The same arguments used to derive the communication time for block-wise inner and outer products can be applied to matrix multiplication.

LSHS Lower Bound on Ray

Our implementation of matmul is a special case of our recursive implementation of tensordot. We can therefore view the entire computation as a $\sqrt{k} \times \sqrt{k}$ grid of node-level blocks, each of which are comprised of $\sqrt{r} \times \sqrt{r}$ worker-level blocks. At the node-level, we have $k^{3/2}$ matrix multiplies, $\sqrt{k}$ of which require 0 inter-node communication. The remaining $k^{3/2} - \sqrt{k}$ node-level blocks are parallelized over k nodes, which requires at least $(k^{3/2} - \sqrt{k})/kr = (k-1)/\sqrt{k}r$ of the worker-level blocks to be transmitted between nodes, yielding a lower bound of $(k-1)/\sqrt{k}rC(n) < \sqrt{k}rC(n)$.

On each node, we have approximately $r\sqrt{r}$ addition operations. Over r local workers, we achieve approximately $\log(\sqrt{r})R(n)$ communication time. At the node-level, we now need

to add $k\sqrt{k}$ node-level blocks of size rn. With k nodes, this will require approximately $\log(\sqrt{k})rC(n)$ communication time.

Putting this all together, the communication lower bound for matrix multiplication is approximately $\left(\frac{k-1}{\sqrt{k}} + \log(\sqrt{k})\right) rC(n) + \log(\sqrt{r})R(n)$. We ignore the diagonal terms for the simpler expression of $\left(\sqrt{k} + \log(\sqrt{k})\right) rC(n) + \log(\sqrt{r})R(n)$.

Note that this result depends on the assumption that a block need only be transmitted to a node once in order to be used for other local operations, and that bytes may be sent and received in parallel, which is possible in Ray.

LSHS Lower Bound on Dask

While Dask provides some worker-level caching, the caching mechanism does not lend itself to a clear lower-bound analysis. We are therefore unable to provide a lower bound for Dask on matrix multiplication.

SUMMA

Algorithm 8: SUMMA

$\mathbf{Z}_{i,j} \leftarrow \mathbf{0}$;
for $h \leftarrow 0$ **to** $\sqrt{p}$ **do**
  Broadcast $\mathbf{X}_{i,h}$ to $\sqrt{p}$ workers in worker grid row i;
  Broadcast $\mathbf{Y}_{h,j}$ to $\sqrt{p}$ workers in worker grid column j;
  $\mathbf{Z}_{i,j} \leftarrow \mathbf{Z}_{i,j} + \mathbf{X}_{i,h}\mathbf{Y}_{h,j}$;
end

In the given setting, the blocked Scalable Universal Matrix Multiplication Algorithm (SUMMA) [17] is characterized by Algorithm 8. The blocks are partitioned over workers so that worker with grid coordinates i, j stores blocks $\mathbf{X}_{i,j}$, $\mathbf{Y}_{i,j}$, and the output block $\mathbf{Z}_{i,j}$.

A tree-based broadcast has an inter-node communication cost of $(\log \sqrt{p})C(n)$. The algorithm performs 2 such broadcasts per iteration over $\sqrt{p}$, yielding an inter-node communication complexity of $2\sqrt{p}\log(\sqrt{p})C(n)$.

Deterministic Recursive Algorithm

We provide a deterministic algorithm within our framework which achieves a lower theoretical upper bound than SUMMA. If we place every operation on its output worker according to the LSHS data layout, then each node in the cluster will receive exactly $2(\sqrt{k} - 1)rC(n)$ bytes: An entire row and an entire column of node-level blocks for each node, minus itself. This can be thought of as stacking the $\sqrt{k}r$ blocks required in the summation to compute the

final output block. Thus, the sum step requires only intra-node communication, resulting in an upper bound of $2(\sqrt{k}-1)rC(n) + \log(\sqrt{r})R(n)$.

SUMMA requires

$$2\sqrt{kr}\log(\sqrt{kr})C(n) = 2\sqrt{k}(\log(\sqrt{k})C(n) + 2\sqrt{r}\log(\sqrt{r})C(n).$$

We can see that the terms which depend on k have a difference of

$$\begin{aligned} 2(\sqrt{k}-1)rC(n) - 2\sqrt{k}\log(\sqrt{k})C(n) = \\ 2C(n)((\sqrt{k}-1)r - \sqrt{k}\log(\sqrt{k})) = \\ 2C(n)(\sqrt{k}r - r - \sqrt{k}\log(\sqrt{k})) = \\ 2C(n)(\sqrt{k}(r - \log(\sqrt{k})) - r). \end{aligned}$$

In our analysis, we assume that r is the number of workers per node, so r will tend to be fixed as k grows. As k approaches ∞, the inner factor $2r - \log(\sqrt{k})$ tends to $-\infty$, which suggests that this algorithm has lower asymptotic complexity in k.

The terms which only depend on r have a difference of

$$\begin{aligned} \log(\sqrt{r})R(n) - 2\sqrt{r}\log(\sqrt{r})C(n) = \\ \log(\sqrt{r})(R(n) - 2\sqrt{r}C(n)). \end{aligned}$$

The inner factor $R(n) - 2\sqrt{r}C(n)$ is constant in this algorithm, and by assumption, we have $R(n) << C(n)$. Putting this all together, this proves that this algorithm has lower asymptotic communication complexity than SUMMA.

This result shows that, given a homogenous cluster of k nodes, where each node has r workers, as we increase the number of nodes k, this algorithm requires fewer bytes of communication than SUMMA. We also note that this algorithm is in the *feasible set* of solutions for our objective: Every node sends and receives exactly the same amount of data, resulting in balanced network in, network out, and memory across all nodes. Note, also, that this is not the lower bound we obtained for matrix multiplication.

Part III

Automatic Parallelization of Basic Python Programs.

Chapter 5

Compiling Basic Python Programs on Task-based Distributed Systems

In this chapter, we formalize the foundations of our approach to parallelizing a subset of Python. The system described in Chapter 3 converts all Python and NumPy objects, operations, and expressions in NumS to a language of concurrently executing futures described in this chapter. We use the syntax of a basic subset of the Python programming language, along with a basic set of distributed systems primitives, to define a translation operator that translates Python code to a language of concurrently executing futures. We summarize the syntax of our languages, informally describe our semantics, and provide a proof sketch of correctness in Section 5.1. A complete treatment of the syntax, semantics, translation, and correctness proofs are provided in the remaining sections.

Notation

We adopt much of the syntax and semantics to describe our language from Dijkstra's language of guarded commands, Hoare and Milner's language of communicating processes, and Milner's CCS. [41] At a minimum, a basic understanding of BNF notation, a notation for defining programming languages recursively, is required. While we use the term "programming language" to refer to our source and target languages, much of what we define in this section are key operations required to formulate our translation procedure. The syntax $a ::= \mathit{constant} \mid a_1 \bullet a_2$ can be broken down as follows. The variable to the left of $::=$ is a recursive definition of a piece of syntax, while the right side may contain subscripted instances of the variable being defined, as well as other variables that have been previously defined. $\mid$ can be interpreted as "or." In this example, a can be *constant*, or the binary operation $a_1 \bullet a_2$. In turn, a_1 and a_2 can be any piece of syntax that appears in the definition of a. We use $\ldots$ to denote the operations which appear previously for a particular variable which has already been defined.

Similar to our language definition, the translation solution we present in this work is defined recursively, where $\equiv$ is used to recursively define the translation of every operator

in our source language. This approach to defining our translation procedure allows us to reason about the runtime behavior and correctness of different translation solutions.

5.1 Intuition

Our approach to concurrency relies on futures and promises. A promise can be thought of as a function which executes asynchronously, immediately returning a future. The returned future can be thought of as a reference to the object that the promise computes. When we allow promises to operate on futures, a single-threaded process can execute a program comprised of futures and promises without doing any of the computation. The underlying execution model can be anything. In this work, we execute futures concurrently. To understand this, it's helpful to consider a future as having two states: It's either computed and references an object, or it's not computed and references a promise. If a promise $\mathbf{f}(x)$ is called with a future $\mathbf{x}$ that has been computed, the promise is considered *independent*. An independent promise can be executed immediately. A promise called with a future $\mathbf{y}$ that has not yet been computed is considered *dependent*. In the concurrently executing futures model, all independent promises are executed concurrently. The goal of a concurrently executing futures-based system is to translate a program comprised of futures and promises to concurrently executing code. Our goal in this section is to provide the intuition behind translating a subset of serial Python code to futures and promises.

$$
\begin{aligned}
\mathbf{c} &::= \mathbf{skip} \mid \mathbf{c}_1; \mathbf{c}_2 \mid \mathbf{x} = \mathbf{e} \\
&\quad \mid \mathbf{if\ b\ then\ c}_1 \mathbf{\ else\ c}_2 \\
&\quad \mid \mathbf{while\ b\ do\ c} \\
\mathbf{f} &::= \mathbf{f}(\mathbf{x}_1, ..., \mathbf{x}_m)\{\mathbf{e}\}
\end{aligned}
$$

Figure 5.1: Definition of Sub-Python statements in BNF form. Assignment stores the result of Python expressions $\mathbf{e}$ as the variable $\mathbf{x}$. $\mathbf{b}$ is defined inductively over Boolean expressions.

The subset of Python we consider is given in figure 5.1. We call this language **Sub-Python**. Some key operations include loops, conditionals, and functions. Our language of futures, called **Sub-Futures**, is the extension of Sub-Python given in figure 5.2. Sub-Futures enables concurrent execution of arbitrary functions on shared memory. This design is heavily inspired by the distributed system Ray [23]. The operator $\mathbf{R}$ is used to create an RPC from an existing function $\mathbf{f}$. The piece of syntax $\mathbf{o}$ (short for object reference), is used to define the futures of our language: $\mathbf{id}()$ constructs the future associated with a collection of values $\mathbf{v}_i$, $r(\mathbf{o}_1, \ldots, \mathbf{o}_m)$ denotes the future returned by a call to the RPC $\mathbf{r}$, and $\mathbf{put}(\mathbf{e})$ denotes the future returned by submitting the result of an expression to the distributed system store.

$$
\begin{aligned}
\mathbf{r} &::= \mathbf{R}(\mathbf{f}) \\
\mathbf{o} &::= \mathbf{id}(\mathbf{v}_1, ..., \mathbf{v}_m) \mid \mathbf{r}(\mathbf{o}_1, ..., \mathbf{o}_m) \mid \mathbf{put}(\mathbf{e}) \\
\mathbf{e} &::= \ldots \mid \mathbf{o} \mid \mathbf{get}(\mathbf{o})
\end{aligned}
$$

Figure 5.2: Sub-Python extended to support futures. The **R** operator is a higher-order function that takes as input Python functions and outputs remote functions. **o** denotes the space of futures, which is comprised of the $\mathbf{id}(\cdot)$ operator, the output of remote function calls, and the **put** operator. The space of expressions **e** is extended to include futures and the **get** operator. For any value **v**, we have $\mathbf{v} = \mathbf{get}(\mathbf{put}(\mathbf{v}))$.

The set of expressions **e** is extended to include futures, and the $\mathbf{get}(\mathbf{o})$ operation, which retrieves the object associated with the future **o**.

Program Translation

Our translation procedure translates all operations from the Sub-Python language to the Sub-Futures language. The key benefit of our translation-based approach is composability: Any composition of operations from Sub-Python can be translated to a semantically equivalent Sub-Futures program. We describe the translation procedure inductively by way of a translation operator **T** below.

$$
\begin{aligned}
\mathbf{T}(\mathbf{skip}) &\equiv \mathbf{skip} \\
\mathbf{T}(\mathbf{c}_1; \mathbf{c}_2) &\equiv (\mathbf{T}(\mathbf{c}_1); \mathbf{T}(\mathbf{c}_2)) \\
\mathbf{T}(\mathbf{x} = \mathbf{e}) &\equiv \mathbf{x} = \mathbf{T}(\mathbf{e}) \\
\mathbf{T}(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_m)\{\mathbf{e}\}) &\equiv \mathbf{R}(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_m)\{\mathbf{e}\}) \\
\mathbf{T}(\mathbf{R}(\mathbf{f})) &\equiv \mathbf{R}(\mathbf{f}) \\
\mathbf{T}(\mathbf{f}(\mathbf{e}_1, \ldots, \mathbf{e}_m)) &\equiv \mathbf{T}(\mathbf{f})(\mathbf{T}(\mathbf{e}_1), \ldots, \mathbf{T}(\mathbf{e}_m)) \\
\mathbf{T}(\mathbf{a}_1 \bullet_2 \mathbf{a}_2) &\equiv \mathbf{R}(\bullet_2)(\mathbf{T}(\mathbf{a}_1), \mathbf{T}(\mathbf{a}_2)) \\
\mathbf{T}(\mathbf{v}) &\equiv \mathbf{put}(\mathbf{v}) \\
\mathbf{T}(\mathbf{x}) &\equiv \mathbf{x}
\end{aligned}
$$

Here $\bullet_2$ is shorthand for arbitrary binary operations on arithmetic expressions. This translation operator takes syntax from a subset of Python code and translates it to futures syntax which, when executed by the driver process **M**, generates correct programs in our computation model.

Correctness

In this section, we provide a proof sketch which shows that Sub-Futures is semantically equivalent to Sub-Python. Consider key/value stores σ, σ', and μ. In our source language, σ maps keys directly to values. In our target language, σ' maps keys to futures, and μ maps futures to values. The basic idea behind our proof is as follows: If some program from the source language terminates with store content σ, then the target language must also terminate with store contents σ' and μ such that for any key/value pair $(\mathbf{x}, \mathbf{v}) \in \sigma$, there exists $(\mathbf{x}, \mathbf{o}) \in \sigma'$ and $(\mathbf{o}, \mathbf{v}) \in \mu$. For instance, if we evaluate the assignment operation $\mathbf{x} = 1 + 2$ in our source language, we end up with some value $(\mathbf{x}, 3) \in \sigma$. According to our translation operator, we want our source language translated as $\mathbf{x} = \mathbf{R}(+)(\mathbf{o}_1, \mathbf{o}_2)$. The semantics of our RPC calls dictate that the expression $\mathbf{R}(+)(\mathbf{o}_1, \mathbf{o}_2)$ evaluates to $\mathbf{o}_3$ such that $(\mathbf{o}_3, \mathbf{v}) \in \mu$, and the assignment yields $(\mathbf{x}, \mathbf{o}_3) \in \mu$.

5.2 Sub-Python: Syntax and Semantics

Syntax

In the Python-inspired grammar below, we use curly braces to scope functions, and semicolons to delimit statements (commands).

$$
\begin{aligned}
\mathbf{c} &::= \mathbf{skip} \mid \mathbf{c}_1; \mathbf{c}_2 \mid \mathbf{x} = \mathbf{e} \mid \mathbf{if\ b\ then\ c}_1 \mathbf{\ else\ c}_2 \mid \mathbf{while\ b\ do\ c} \\
\mathbf{e} &::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{f} \mid \mathbf{f}(\mathbf{e}_1, ..., \mathbf{e}_m) \mid \mathbf{null} \\
\mathbf{f} &::= \mathbf{f}(\mathbf{x}_1, ..., \mathbf{x}_m)\{\mathbf{e}\} \\
\mathbf{a} &::= \mathbf{n} \mid \mathbf{x} \mid -\mathbf{a} \mid \mathbf{a}_1 + \mathbf{a}_2 \mid \mathbf{a}_1 - \mathbf{a}_2 \mid \mathbf{a}_1 * \mathbf{a}_2 \mid \mathbf{a}_1 + \mathbf{a}_2 \\
\mathbf{b} &::= \mathbf{True} \mid \mathbf{False} \mid \mathbf{x} \mid \mathbf{not\ b} \mid \mathbf{b}_1 \mathbf{\ or\ b}_2 \mid \mathbf{b}_1 \mathbf{\ and\ b}_2 \mid \mathbf{a}_1 == \mathbf{a}_2 \mid \mathbf{a}_1 < \mathbf{a}_2 \\
\mathbf{v} &::= \mathbf{True} \mid \mathbf{False} \mid \mathbf{n} \mid \mathbf{f} \\
\mathbf{d} &::= \mathbf{v} \mid \mathbf{error}
\end{aligned}
$$

Evaluation order is defined as follows.

$$
\begin{aligned}
\mathbf{ARG} &::= \mathbf{e} \mid \mathbf{e}\ ,\ \mathbf{ARG} \\
\mathbf{H} &::= [.] \mid \mathbf{not\ H} \mid -\mathbf{H} \mid \mathbf{H} \bullet \mathbf{e} \mid \mathbf{v} \bullet \mathbf{H} \mid \mathbf{f}(\mathbf{H}) \mid \mathbf{H}\ ,\ \mathbf{ARG} \mid \mathbf{v}\ ,\ \mathbf{H}
\end{aligned}
$$

Above, $\bullet$ ranges over Boolean and arithmetic binary operations. We introduce $\mathbf{d}$ above to deal with cases where expressions evaluate to either a value $\mathbf{v}$ or **error**.

Semantics

The semantics for this grammar are equivalent to small-step operational semantics of IMP [42]. The program state σ is not type-safe. We allow for $\perp \in \Sigma$ to indicate non-terminating

programs. In what follows, we define the semantics for terminating and non-terminating expressions (due to recursion), as well as functions. Let $\to^k$ denote k small steps, and $\mathcal{V}$ denote the set of values. We define $\mathbf{d} \in \mathcal{V} \cup \{\textbf{error}\}$.

$$\frac{\exists k.\, H[\mathbf{e}] \to^k H[\mathbf{v}]}{\langle H[\mathbf{e}], \sigma\rangle \to^* \langle H[\mathbf{v}], \sigma\rangle}\ \textsc{Expr} \qquad \frac{\forall k. H[\mathbf{e}] \to^k H[\mathbf{e}'] \qquad \mathbf{e}' \notin \mathcal{V}}{\langle H[\mathbf{e}], \sigma\rangle \to \langle \textbf{error}, \sigma\rangle}\ \textsc{Expr-}\infty$$

$$\frac{\mathbf{e} \to \textbf{error}}{\langle \mathbf{x} = \mathbf{e}, \sigma\rangle \to \langle \textbf{skip}, \bot\rangle}\ \textsc{Assgn-}\infty$$

$$\textsc{Assgn}\ \frac{\mathbf{e} \to^* \mathbf{v}}{\langle \mathbf{x} = \mathbf{e}, \sigma\rangle \to \langle \textbf{skip}, \sigma[\mathbf{x} = \mathbf{v}]\rangle} \qquad \frac{\mathbf{e} \to^* \mathbf{d}}{\langle \mathbf{x} = \mathbf{e}, \bot\rangle \to \langle \textbf{skip}, \bot\rangle}\ \textsc{Assgn-Dead}$$

$$\textsc{Read}\ \frac{\sigma(\mathbf{x}) = \mathbf{v}}{\langle \mathbf{x}, \sigma\rangle \to \langle \mathbf{v}, \sigma\rangle} \qquad \frac{}{\langle \mathbf{x}, \bot\rangle \to \langle \textbf{error}, \bot\rangle}\ \textsc{Read-Dead}$$

$$\frac{}{\langle \mathbf{c}, \bot\rangle \to \langle \textbf{skip}, \bot\rangle}\ \textsc{Cmd-}\infty$$

$$\frac{\langle \mathbf{e}_i, \sigma\rangle \to^* \langle \mathbf{v}_i, \sigma\rangle \qquad \langle [\mathbf{v}_i/\mathbf{x}_i]_{i=1}^m \mathbf{e}, \sigma\rangle \to^* \langle \mathbf{v}, \sigma\rangle}{\langle \mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_m)\{\mathbf{e}\}(\mathbf{e}_1, \ldots, \mathbf{e}_m), \sigma\rangle \to^* \langle \mathbf{v}, \sigma\rangle}\ \textsc{Func-Eval}$$

Note that in the rule for non-terminating expressions, the entire context transitions to **error**, which covers cases such as $\textbf{error} + \mathbf{e}$, $\mathbf{f}(\textbf{error}, \ldots, \mathbf{e}_m)$, etc. We have reads on $\bot$ evaluate to **error** to be consistent with read behavior in the multi-process setting.

Loop Semantics

We define the semantics of while loops in terms of a bounded while loop which executes at most k times before transitioning to $\bot$, written $\textbf{while}_k\ \textbf{b do c}$. The operational semantics of $\textbf{while}_\textbf{k}$ is defined as follows.

$$\frac{\langle \mathbf{b}, \sigma\rangle \to \langle \textbf{True}, \sigma\rangle}{\langle \textbf{while}_0\ \textbf{b do c}, \sigma\rangle \to \langle \textbf{skip}, \bot\rangle}\ \textsc{While-0}$$

$$\frac{\langle \mathbf{b}, \sigma\rangle \to \langle \textbf{True}, \sigma\rangle \qquad k > 0}{\langle \textbf{while}_k\ \textbf{b do c}, \sigma\rangle \to \langle \mathbf{c}; \textbf{while}_{k-1}\ \textbf{b do c}, \sigma\rangle}\ \textsc{While-k-True}$$

$$\frac{\langle \mathbf{b}, \sigma\rangle \to \langle \textbf{False}, \sigma\rangle}{\langle \textbf{while}_k\ \textbf{b do c}, \sigma\rangle \to \langle \textbf{skip}, \sigma\rangle}\ \textsc{While-k-False}$$

The semantics for $\mathbf{while_k}$ can be interpreted simply as follows: If after k executions of $\mathbf{c}$ the conditional $\mathbf{b}$ still evaluates to **True**, then transition the program state to $\perp$. We use this key property of $\mathbf{while_k}$ below to define the semantics of **while** to handle non-terminating programs.

$$\frac{\langle \mathbf{b}, \sigma\rangle \to \langle \mathbf{True}, \sigma\rangle \qquad \exists k.\ \langle \mathbf{while}_k\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}, \sigma\rangle \to^* \langle \mathbf{skip}, \sigma'\rangle \qquad \sigma' \neq \perp}{\langle \mathbf{while}\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}, \sigma\rangle \to \langle \mathbf{c}; \mathbf{while}\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}, \sigma\rangle}\ \textsc{While-True}$$

$$\frac{\forall k.\ \langle \mathbf{while}_k\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}, \sigma\rangle \to^* \langle \mathbf{skip}, \perp\rangle}{\langle \mathbf{while}\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}, \sigma\rangle \to \langle \mathbf{skip}, \perp\rangle}\ \textsc{While-}\infty$$

$$\frac{\langle \mathbf{while}_k\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}, \sigma\rangle \to \langle \mathbf{skip}, \sigma\rangle}{\langle \mathbf{while}\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}, \sigma\rangle \to \langle \mathbf{skip}, \sigma\rangle}\ \textsc{While-False}$$

The derivation **While-True** can be interpreted as follows: When $\mathbf{b}$ evaluates to **True**, if there is some k for which $\mathbf{while}_k\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}$ transitions to a program state other than $\perp$, then $\mathbf{while}\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}$ is terminating and can make progress. The derivation **While-**∞ simply states that if no k exists such that $\mathbf{while}_k\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}$ terminates in a state other than $\perp$, then the loop $\mathbf{while}\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}$ cannot make progress and yields state $\perp$. The derivation of **While-False** is self-evident.

5.3 Concurrently Executing Futures: Syntax and Semantics

We define a futures-based extension of the Sub-Python language, called Sub-Futures, which provides an API enabling concurrent execution of arbitrary functions on a shared memory store. We adopt much of the syntax from Dijkstra's language of guarded commands, Hoare and Milner's language of communicating processes, and Milner's CCS.

We define the configuration of a futures program with k workers as follows:

$$\langle \mathbf{M} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu)\rangle\,, \tag{5.1}$$

where $\mathbf{M}$ denotes the main futures process (aka the driver), σ denotes the state for $\mathbf{M}$, $\mathbf{S} = \mathbf{S}^r \parallel \mathbf{S}^w = \mathbf{S}^r_0 \parallel \mathbf{S}^r_1 \parallel ... \parallel \mathbf{S}^r_k \parallel \mathbf{S}^w$ denotes a collection of "store" processes which are responsible for writing to and reading from a shared state μ across all processes, called the store, and $\mathbf{W} = \mathbf{W}_1 \parallel ... \parallel \mathbf{W}_k$ denotes k worker processes.

Futures: Communication Syntax

This section and the next can be skipped if the reader is already familiar with these languages. Duplication of some of the operations is necessary to ensure correct semantics when defining transitions between command operations and operations which evaluate to data. **s** is used to label an arbitrary channel of communication, and τ is the null process and is equivalent in functionality to **skip** for commands.

Communication Commands:

$$\begin{aligned}\mathbf{c} \ ::= \ & ... \\ & \mid \mathbf{s}!\mu(\mathbf{o}) \\ & \mid \mathbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{e}_1, ..., \mathbf{e}_m)) \\ & \mid \mathbf{sealed}(\mathbf{o}) \ \Rightarrow \ \mathbf{c} \\ & \mid \mathbf{gc} \\ & \mid \mathbf{do\ gc\ od}\end{aligned}$$

Guarded commands:

$$\begin{aligned}\mathbf{gc} \ ::= \ & \mathbf{s}?(\mathbf{x}_1, ..., \mathbf{x}_m) \ \Rightarrow \ \mathbf{c} \\ & \mid \ \sum(\mathbf{s}_0?(\mathbf{x}_1, ..., \mathbf{x}_m) \ \Rightarrow \ \mathbf{c}_0, ..., \mathbf{s}_k?(\mathbf{x}_1, ..., \mathbf{x}_m) \ \Rightarrow \ \mathbf{c}_k) \ \Rightarrow \ \mathbf{c}_k\end{aligned}$$

Communication Operations:

$$\begin{aligned}\mathbf{p} \ ::= \ & \tau \mid \mathbf{wait}(\mathbf{o}) \mid \mathbf{snd}(\mathbf{o}) \mid \mathbf{rcv}(\mathbf{o}) \mid \mathbf{seal}(\mathbf{o}, \mathbf{v}) \\ & \mid \mathbf{s}!(\mathbf{v}_1, ..., \mathbf{v}_m) \mid \mathbf{s}?(\mathbf{x}_1, ..., \mathbf{x}_m) \ \Rightarrow \ \mathbf{p} \\ & \mid \mathbf{p} \ \Rightarrow \ \mathbf{p} \mid \mathbf{p} \ \Rightarrow \ \mathbf{v}\end{aligned}$$

Following are some clarifications on transitions.

1. Worker and store processes rely on the `do gc od` command in order to invoke commands on incoming data. This is also why we've defined the special commands **seal**$(\cdot)$, **sealed**$(\cdot)$ and **s**$!\mu(\cdot)$, as these are downstream commands to the guarded commands.

2. The semantics for **get**$(\cdot)$, **put**$(\cdot)$, and $\mathbf{R}(f)(\cdot)$ transition through a sequence of communication operations, all of which have defined semantics, and all of which eventually transition to either a value or an object id.

3. **wait**, **snd**, **rcv**, and **seal** are aliases to communication operations to improve the readability of semantics, proofs, etc.

Futures: Communication Semantics

Assume the following.

1. Whenever a command transitions to "nothing," it transitions to **skip**.
2. Whenever a process transitions to "nothing," it transitions to τ.

$$\frac{}{\mathbf{s}\,!\,(\mathbf{v}_1, ..., \mathbf{v}_m) \xrightarrow{\mathbf{s}\,!\,(\mathbf{v}_1,...,\mathbf{v}_m)} \mathbf{p}} \textsc{Send}$$

$$\frac{}{\mathbf{s}\,?\,(\mathbf{x}_1, ..., \mathbf{x}_m) \Rightarrow \mathbf{p} \xrightarrow{\mathbf{s}\,?\,(\mathbf{v}_1,...,\mathbf{v}_m)} \mathbf{p}[\mathbf{v}_1/\mathbf{x}_1, ..., \mathbf{v}_m/\mathbf{x}_m]} \textsc{Rcv}$$

$$\frac{\mu(\mathbf{o}) = \mathbf{v}}{\langle \mathbf{s}\,!\,\mu(\mathbf{o}), (\sigma, \mu)\rangle \xrightarrow{\mathbf{s}\,!\,\mathbf{v}} \langle \mathbf{c}, (\sigma', \mu')\rangle} \textsc{Send-Value-Cmd}$$

$$\frac{}{\langle \mathbf{s}\,?\,(\mathbf{x}_1, ..., \mathbf{x}_m) \Rightarrow \mathbf{c}, (\sigma, \mu)\rangle \xrightarrow{\mathbf{s}\,?\,(\mathbf{v}_1,...,\mathbf{v}_m)} \langle \mathbf{c}[\mathbf{v}_1/\mathbf{x}_1, ..., \mathbf{v}_m/\mathbf{x}_m], (\sigma', \mu')\rangle} \textsc{Rcv-Guard-Cmd}$$

$$\frac{\langle \mathbf{s}_j \Rightarrow \mathbf{c}_j, (\sigma, \mu)\rangle \xrightarrow{\mathbf{s}_j?(\mathbf{v}_1,...,\mathbf{v}_m)} \langle \mathbf{c}_j, (\sigma', \mu')\rangle}{\left\langle \sum_{i=0}^{k} \mathbf{s}_i?(\mathbf{x}_1, ..., \mathbf{x}_m) \Rightarrow \mathbf{c}_i, (\sigma, \mu) \right\rangle \xrightarrow{\lambda} \langle \mathbf{c}_j, (\sigma', \mu')\rangle} \textsc{Rcv-Sum-Guard-Cmd}$$

$$\frac{\langle \mathbf{gc}, (\sigma, \mu)\rangle \xrightarrow{\lambda} \langle \mathbf{c}, (\sigma, \mu)\rangle}{\langle \mathbf{do}\ [\,\mathbf{gc}\,]\ \mathbf{od}, (\sigma, \mu)\rangle \xrightarrow{\lambda} \langle \mathbf{c}; \mathbf{do}\ [\,\mathbf{gc}\,]\ \mathbf{od}, (\sigma', \mu')\rangle} \textsc{Loop-Guard-Cmd}$$

Sub-Futures Syntax

In the following, we'll define the syntax of Sub-Python extended with the futures API, and additional syntax to support inter-process communication and parallelism. Sub-Futures schedules remote functions by selecting worker processes at random, and maintaining a single centralized object store. Much of the design of Sub-Futures is inspired by distributed systems such as Ray and Dask [23, 32].

The syntax for declaring a remote function $\mathbf{r}$ using an existing function $\mathbf{f}$ is defined by the syntax $\mathbf{R}(\mathbf{f})$. When a remote function is invoked with $r(\mathbf{o}_1, \ldots, \mathbf{o}_m)$, the function evaluates on a separate process and execution on the process which invoked the remote function continues to make progress. $\mathbf{put}(\mathbf{e})$ puts data in the object store and returns an object id $\mathbf{o}$, and $\mathbf{get}(\mathbf{o})$ gets an object from the object store and returns a value $\mathbf{v}$. The $\mathbf{id}(\cdot)$ function is not part of the Sub-Futures API, but is needed internally to generate hashes of objects and function

invocations. The Sub-Futures API and $\mathbf{id}(\cdot)$ function are defined as follows.

Sub-Futures Extensions:

$$\begin{aligned}
\mathbf{o} &::= \mathbf{id}(\mathbf{v}_1, ..., \mathbf{v}_m) \mid \mathbf{r}(\mathbf{o}_1, ..., \mathbf{o}_m) \mid \mathbf{put}(\mathbf{e}) \\
\mathbf{c} &::= ... \mid \mathbf{o} = \mathbf{v} \\
\mathbf{r} &::= \mathbf{R}(\mathbf{f}) \\
\mathbf{a} &::= ... \mid \mathbf{rand}(\mathbf{a}) \\
\mathbf{e} &::= ... \mid \mathbf{o} \mid \mathbf{get}(\mathbf{o}) \\
\mathbf{v} &::= ... \mid \mathbf{id}(\mathbf{v}_1, ..., \mathbf{v}_m) \\
\mathbf{h} &::= ... \mid \mathbf{put}(\mathbf{H}) \mid \mathbf{get}(\mathbf{H}) \mid \mathbf{r}(\mathbf{H})
\end{aligned}$$

Parallelism:

$$\mathbf{h} ::= \mathbf{c} \mid \mathbf{c} \parallel \mathbf{h}$$

The syntax $\mathbf{o} = \mathbf{v}$ adds values to μ so that $(\mathbf{o}, \mathbf{v}) \in \mu$ and $\mu(\mathbf{o})$ evaluates to $\mathbf{v}$.

Sub-Futures Semantics

In this section we present semantics for the Sub-Futures API and parallelism. We continue with small-step operational semantics. The state σ is initialized to $\{k = \mathbf{n}\}$, where $\mathbf{n}$ corresponds to the number of Sub-Futures workers.

$$\frac{}{\langle \mathbf{o} = \mathbf{error}, (\sigma, \mu)\rangle \rightarrow \langle \mathbf{skip}, (\sigma, \bot)\rangle}\ \textsc{Store-Assgn-}\infty$$

$$\textsc{Store-Assgn}\ \frac{}{\langle \mathbf{o} = \mathbf{v}, (\sigma, \mu)\rangle \rightarrow \langle \mathbf{skip}, (\sigma, \mu[\mathbf{o} = \mathbf{v}])\rangle}\quad \frac{}{\langle \mathbf{o} = \mathbf{v}, (\sigma, \bot)\rangle \rightarrow \langle \mathbf{skip}, (\sigma, \bot)\rangle}\ \textsc{Store-Assgn-Error}$$

$$\textsc{Store-Read}\ \frac{\mu(o) = \mathbf{v}}{\langle \mu(o), (\sigma, \mu)\rangle \rightarrow \langle \mathbf{v}, (\sigma, \mu)\rangle}\quad \frac{}{\langle \mu(o), (\sigma, \bot)\rangle \rightarrow \langle \mathbf{error}, (\sigma, \bot)\rangle}\ \textsc{Store-Read-Error}$$

Assumptions

The following assumptions are made explicit in the remaining semantics. Recall that $\mathbf{M}$ is the main process, $\mathbf{S} = \mathbf{S}^r \parallel \mathbf{S}^w = \mathbf{S}^r_0 \parallel \mathbf{S}^r_1 \parallel ... \parallel \mathbf{S}^r_k \parallel \mathbf{S}^w$ are store processes, and $\mathbf{W} = \mathbf{W}_1 \parallel ... \parallel \mathbf{W}_k$ are worker processes.

1. $\mathbf{M}$ sends data with **put** to $\mathbf{S}^w$ via communication channel α_0.
2. $\mathbf{W}_i$ sends data with **seal** to $\mathbf{S}^w$ via communication channel α_i for $i = 1, ..., k$.

3. $\mathbf{M}$ receives data with **get** from $\mathbf{S}_0^r$ via communication channel γ_0.
4. $\mathbf{W}_i$ receives data with **get** from $\mathbf{S}_i^r$ via communication channel γ_i for $i = 1, ..., k$.
5. $\mathbf{M}$ triggers RPCs on $\mathbf{W}_i$ via communication channel β_i for $i = 1, ..., k$.
6. Only $\mathbf{M}$ reads from and writes to σ.
7. Only $\mathbf{S}$ reads from and writes to μ.
8. Processes $\mathbf{W}$ and $\mathbf{S}$ have no **local** state.

Parallelism

The store processes $\mathbf{S}_i^r$ are responsible for handling reads from the object store. We parallelize communication from $\mathbf{M}$ and each worker $\mathbf{W}_j$ for $j = 1 \ldots, k$ to the store processes $\mathbf{S}_i^r$ for $i = 0, \ldots, k$, to avoid deadlock, as well as to prevent the main process and worker processes from blocking one another. These semantics are defined as follows:

$$\frac{\langle \mathbf{M}, (\sigma, \mu)\rangle \to \langle \mathbf{M}', (\sigma', \mu)\rangle}{\langle \mathbf{M} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu)\rangle \to \langle \mathbf{M}' \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma', \mu)\rangle} \textsc{Parallel-Main}$$

$$\frac{\langle \mathbf{W}, (\sigma, \mu)\rangle \to \langle \mathbf{W}', (\sigma, \mu)\rangle}{\langle \mathbf{M} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu)\rangle \to \langle \mathbf{M} \parallel \mathbf{W}' \parallel \mathbf{S}, (\sigma, \mu)\rangle} \textsc{Parallel-Worker}$$

$$\frac{\langle \mathbf{S}, (\sigma, \mu)\rangle \to \langle \mathbf{S}', (\sigma, \mu')\rangle}{\langle \mathbf{M} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu)\rangle \to \langle \mathbf{M} \parallel \mathbf{W} \parallel \mathbf{S}', (\sigma, \mu')\rangle} \textsc{Parallel-Store}$$

For non-terminating commands and expressions occurring in $\mathbf{M}$, we have

$$\frac{}{\langle \textbf{skip} \parallel \mathbf{W} \parallel \mathbf{S}, (\bot, \mu)\rangle \to \langle \textbf{skip} \parallel \mathbf{W}' \parallel \mathbf{S}', (\bot, \bot)\rangle} \textsc{Parallel-Main-Error}$$

$$\frac{}{\langle \textbf{skip} \parallel \mathbf{W} \parallel \mathbf{S}^r \parallel \mathbf{o} = \mathbf{d}, (\bot, \mu)\rangle \to \langle \textbf{skip} \parallel \mathbf{W}' \parallel \mathbf{S}', (\bot, \bot)\rangle} \textsc{Parallel-Main-Error-2}$$

For non-terminating expressions occurring in $\mathbf{W}_i$, we have some function transitioning to **error**. In this case, the write process S^w will attempt $\mathbf{o} =$ **error**, which will transition μ to $\bot$, and whenever μ is $\bot$, all assignments $\mathbf{o} = \mathbf{v}$ transition $\bot$ to $\bot$. We therefore define the following parallel semantics to deal with this scenario.

$$\frac{}{\langle \mathbf{M} \parallel \mathbf{W} \parallel \mathbf{S}^r \parallel \mathbf{o} = \textbf{error}, (\sigma, \mu)\rangle \to \langle \mathbf{M} \parallel \mathbf{W}' \parallel \mathbf{S}', (\sigma, \bot)\rangle} \textsc{Parallel-Worker-}\infty$$

$$\frac{}{\langle \mathbf{M} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \bot)\rangle \to \langle \textbf{skip} \parallel \mathbf{W}' \parallel \mathbf{S}', (\bot, \bot)\rangle} \textsc{Parallel-Store-Error}$$

$$\frac{}{\langle \mathbf{x} = \mathbf{d} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \bot)\rangle \to \langle \textbf{skip} \parallel \mathbf{W}' \parallel \mathbf{S}', (\bot, \bot)\rangle} \textsc{Parallel-Store-Error-2}$$

Worker Processes

A worker process $\mathbf{W}_i$ is defined as follows. Operational semantics for these processes appear in the section on remote procedure calls.

$$\mathbf{W}_i = \textbf{do}\ [\, \beta_i \,?\, (\mathbf{x}_f, \mathbf{x}_r, \mathbf{x}_1, \ldots, \mathbf{x}_m) \Rightarrow \textbf{seal}(\mathbf{x}_r, \mathbf{x}_f(\textbf{get}(\mathbf{x}_1), \ldots, \textbf{get}(\mathbf{x}_m)))\,]\ \textbf{od} \tag{5.2}$$

This process receives function calls from the main process, executes them, then sends the result with the appropriate object id to the object store by invoking the **seal** operation.

Store Processes

Store processes are defined as $\mathbf{S} = \mathbf{S}^r \parallel \mathbf{S}^w = \mathbf{S}_0^r \parallel \mathbf{S}_1^r \parallel \ldots \parallel \mathbf{S}_k^r \parallel \mathbf{S}^w$, where for $i = 0, \ldots, k$ we have

$$\mathbf{S}_i^r = \textbf{do}\ [\, \gamma_i \,?\, \mathbf{x} \Rightarrow \textbf{sealed}(\mathbf{x}) \Rightarrow \gamma_i \,!\, \mu(\mathbf{x})\,]\ \textbf{od} \tag{5.3}$$

$$\mathbf{S}^w = \textbf{do}\ \left[\, \sum_{j=0}^{k} \alpha_j \,?\, (\mathbf{x}_o, \mathbf{x}_v) \Rightarrow \mathbf{x}_o = \mathbf{x}_v \right]\ \textbf{od} \tag{5.4}$$

The **sealed** operation checks whether an object is currently stored in the object store:

$$\frac{\exists\, \mathbf{v}.(\mathbf{o}, \mathbf{v}) \in \mu}{\langle \textbf{sealed}(\mathbf{o}) \Rightarrow \mathbf{c}, (\mu)\rangle \xrightarrow{\lambda} \langle \mathbf{c}, (\mu')\rangle} \textsc{Sealed-True}$$

$$\frac{\nexists\, \mathbf{v}.(\mathbf{o}, \mathbf{v}) \in \mu}{\langle \textbf{sealed}(\mathbf{o}) \Rightarrow \mathbf{c}, (\mu)\rangle \xrightarrow{\lambda} \langle \textbf{sealed}(\mathbf{o}) \Rightarrow \mathbf{c}, (\mu')\rangle} \textsc{Sealed-False}$$

$$\frac{}{\langle \textbf{sealed}(\mathbf{o}) \Rightarrow \mathbf{c}, (\bot)\rangle \xrightarrow{\lambda} \langle \mathbf{c}, (\bot)\rangle} \textsc{Sealed-Error}$$

Note that when $\mu = \bot$, all S_i^r processes transmit **error**. Together with semantics for assignment to μ and σ, this behavior causes worker processes to immediately evaluate to **error**, and the main process to enter **skip** with $\sigma =$ **error**.

Put

The **put** operation immediately transitions to the **seal** operation as defined by the following semantics.

$$\frac{\mathbf{e} \rightarrow^* \mathbf{v} \qquad \mathbf{o} = \mathbf{id}(\mathbf{v})}{\langle \mathbf{put}(\mathbf{e}) \parallel \mathbf{S}^w, (\sigma, \mu) \rangle \rightarrow \langle \mathbf{seal}(\mathbf{o}, \mathbf{v}) \parallel \mathbf{S}^w, (\sigma, \mu') \rangle} \textsc{Put}$$

The **seal** operation may be triggered by the main process $\mathbf{M}$ or any of the worker processes $\mathbf{W}_j$, $j = 1, \ldots, k$. An invocation of **seal** requires communication with $\mathbf{S}^w$, which is done over the channel α_0 if **seal** is invoked from $\mathbf{M}$ and α_j if **seal** is invoked from $\mathbf{W}_j$. For this reason, we only need to define the parallel semantics of **seal** between $\mathbf{P}_i$ and $\mathbf{S}^w$, where $\mathbf{P}_0 = \mathbf{M}$ and $\mathbf{P}_j = W_j$ for $j = 1, \ldots, k$.

It is important to note that while processes other than $\mathbf{P}_i$ may be invoking **seal**, S^w handles those communications serially. Thus, while the **intermediate** shared state μ may be non-deterministic during execution due to non-deterministic selection of which incoming request to process next, the **final** shared state μ is deterministic as all requests will be processed serially. The semantics of **seal** are defined as follows for $i = 0, \ldots, k$.

$$\frac{(\alpha_i \,!\, (\mathbf{o}, \mathbf{d}) \Rightarrow \mathbf{o}) \xrightarrow{\alpha_i \,!\, (\mathbf{o},\mathbf{d})} \mathbf{o} \qquad \langle \mathbf{S}^w, (\mu) \rangle \xrightarrow{\alpha_i \,?\, (\mathbf{o},\mathbf{d})} \langle \mathbf{o} = \mathbf{d}; \mathbf{S}^w, (\mu) \rangle}{\langle \mathbf{seal}(\mathbf{o}, \mathbf{d}) \parallel \mathbf{S}^w, (\sigma, \mu) \rangle \xrightarrow{\lambda} \langle \mathbf{o} \parallel \mathbf{o} = \mathbf{d}; \mathbf{S}^w, (\sigma', \mu) \rangle} \textsc{Seal}$$

The Store-Assignment command will transition the resulting program configuration μ to a configuration $\mu' = \mu[\mathbf{o} = \mathbf{v}]$ if $\mathbf{d} \neq$ **error**, and to $\perp$ otherwise. It is impossible for μ to have changed after a **seal** transition because the $\mathbf{S}^w$ process only lets one **seal** through at a time [1].

Get

Get may be invoked from two separate process types: The main process and worker processes. Each process which may invoke **get** has a dedicated store process which carries out the read for that process. Thus, it is sufficient to define the semantics for a particular process $\mathbf{P}_i$ where $\mathbf{P}_0 = \mathbf{M}$ and $\mathbf{P}_i = \mathbf{W}_j$ for $j = 1, \ldots, k$. A **get** invocation transitions over the following communication configurations before evaluating to $\mathbf{d}$, which may be $\mathbf{v}$ or .

1. $\mathbf{get}(\mathbf{o})$, which submits a request over channel γ_i. This request is received by dedicated store process $\mathbf{S}_i^r$, which is listening for such a request on channel γ_i. This operation always makes progress as each $\mathbf{P}_i$ is able to make at most a single request at a time.

2. $\mathbf{get}(\mathbf{o})$ transitions to $\mathbf{rcv}(\mathbf{o})$ on $\mathbf{P}_i$ while the store process $\mathbf{S}_i^r$ transitions to $\mathbf{wait}(\mathbf{o})$. $\mathbf{P}_i$ cannot make progress until $\mathbf{o}$ becomes available in the object store.

[1] A similar **seal** operation for commands is defined for remote procedure calls. This is to ensure the **seal** operation transitions to the appropriate program configuration when invoked as a command.

3. $\mathbf{S}_i^r$ transitions to $\mathbf{snd}(\mathbf{o})$ once the object is available in the object store. $\mathbf{P}_i$ makes no progress during this transition.

4. With $\mathbf{P}_i$ in a $\mathbf{rcv}(\mathbf{o})$ configuration and $\mathbf{S}_i^r$ in a $\mathbf{snd}(\mathbf{o})$ configuration, $\mathbf{P}_i$ is able to transition to $\mathbf{d}$, which is **error** if $\mu = \bot$, or the value $\mu(\mathbf{o}) = \mathbf{v}$. In both cases, $\mathbf{S}_i^r$ transitions to its original listening state.

$$\frac{\gamma_i \,!\, \mathbf{o} \Rightarrow \mathbf{rcv}(\mathbf{o}) \xrightarrow{\gamma_i \,!\, \mathbf{o}} \mathbf{rcv}(\mathbf{o}) \qquad \langle \mathbf{S}_i^r, \mu \rangle \xrightarrow{\gamma_i \,?\, \mathbf{o}} \langle \mathbf{sealed}(\mathbf{o}) \Rightarrow \gamma_i \,!\, \mu''(\mathbf{o}); \mathbf{S}_i^r, \mu' \rangle}{\langle \mathbf{get}(\mathbf{o}) \parallel \mathbf{S}, (\sigma, \mu) \rangle \xrightarrow{\lambda} \langle \mathbf{rcv}(\mathbf{o}) \parallel \ \ldots \ \parallel (\mathbf{wait}(\mathbf{o}); \mathbf{S}_i^r) \parallel \ \ldots \ \parallel \mathbf{S}^w, (\sigma', \mu') \rangle} \textsc{Get}$$

$$\frac{\langle \mathbf{sealed}(\mathbf{o}) \Rightarrow \gamma_i \,!\, \mu'(\mathbf{o}); \mathbf{S}_i^r, (\mu) \rangle \xrightarrow{\lambda} \langle \gamma_i \,!\, \mu'(\mathbf{o}), (\mu') \rangle}{\langle \mathbf{rcv}(\mathbf{o}) \parallel \ \ldots \ \parallel (\mathbf{wait}(\mathbf{o}); \mathbf{S}_i^r) \parallel \ \ldots \ \parallel \mathbf{S}^w, (\sigma, \mu) \rangle \xrightarrow{\lambda} \langle \mathbf{rcv}(\mathbf{o}) \parallel \ \ldots \ \parallel (\mathbf{snd}(\mathbf{o}); \mathbf{S}_i^r) \parallel \ \ldots \ \parallel \mathbf{S}^w, (\sigma', \mu') \rangle} \textsc{Wait-True}$$

$$\frac{\langle \mathbf{sealed}(\mathbf{o}) \Rightarrow \gamma_i \,!\, \mu'(\mathbf{o}); \mathbf{S}_i^r, (\mu) \rangle \xrightarrow{\lambda} \langle \mathbf{sealed}(\mathbf{o}) \Rightarrow \gamma_i \,!\, \mu''(\mathbf{o}); \mathbf{S}_i^r, (\mu') \rangle}{\langle \mathbf{rcv}(\mathbf{o}) \parallel \ \ldots \ \parallel (\mathbf{wait}(\mathbf{o}); \mathbf{S}_i^r) \parallel \ \ldots \ \parallel \mathbf{S}^w, (\sigma, \mu) \rangle \xrightarrow{\lambda} \langle \mathbf{rcv}(\mathbf{o}) \parallel \ \ldots \ \parallel (\mathbf{wait}(\mathbf{o}); \mathbf{S}_i^r) \parallel \ \ldots \ \parallel \mathbf{S}^w, (\sigma', \mu') \rangle} \textsc{Wait-False}$$

$$\frac{\langle \gamma_0 \,!\, \mu(\mathbf{o}); \mathbf{S}_0^r, (\mu) \rangle \xrightarrow{\gamma_0 \,!\, \mathbf{d}} \langle \mathbf{S}_0^r, (\mu') \rangle \qquad \langle \gamma_0 \,?\, \mathbf{x} \Rightarrow \mathbf{x} \rangle \xrightarrow{\gamma_0 \,?\, \mathbf{d}} \mathbf{d}}{\langle \mathbf{rcv}(\mathbf{o}) \parallel \ \ldots \ \parallel (\mathbf{snd}(\mathbf{o}); \mathbf{S}_i^r) \parallel \ \ldots \ \parallel \mathbf{S}^w, (\sigma, \mu) \rangle \xrightarrow{\lambda} \langle \mathbf{d} \parallel \mathbf{S}, (\sigma', \mu') \rangle} \textsc{Snd-Rcv}$$

In the above and in general, $\mu \subseteq \mu' \subseteq \mu''$, which is proven by Lemma 5.5.

Remote Function Call

To simplify the analysis, we assume function calls are randomly assigned to workers. The definition of a remote function call serves only to direct operational behavior during invocation. Semantics for sampling random numbers and remote function definitions are given as follows.

$$\frac{\mathbf{n}_2 \sim U(1, ..., \mathbf{n}_1)}{\langle \mathbf{rand}(\mathbf{n}_1), (\sigma, \mu) \rangle \to \langle \mathbf{n}_2, (\sigma, \mu) \rangle} \textsc{Random}$$

$$\frac{}{\langle \mathbf{R}(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_m)\{\mathbf{e}\}), (\sigma, \mu) \rangle} \textsc{Remote-Def}$$

Only the main process may invoke a remote function call. Remote function calls transmit a function definition, as well as a collection of object ids to a random worker. Recall that a worker process is defined as follows

$$\mathbf{W}_i = \mathbf{do}\ [\, \beta_i \,?\, (\mathbf{x}_f, \mathbf{x}_r, \mathbf{x}_1, \ldots, \mathbf{x}_m) \Rightarrow \mathbf{seal}(\mathbf{x}_r, \mathbf{x}_f(\mathbf{get}(\mathbf{x}_1), \ldots, \mathbf{get}(\mathbf{x}_m)))\,]\ \mathbf{od}$$

The worker invokes the provided function and stores the result in the object store. Communication between the main process $\mathbf{M}$ and workers $\mathbf{W}_i$ is carried out over the communication channels β_i, where $i = 1, ..., k$. The semantics of remote function calls need only be defined in parallel for processes $\mathbf{M}$ and $\mathbf{W}$.

$$\frac{\begin{array}{c}\sigma(\mathbf{r}) = \mathbf{R}(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_m)\{\mathbf{e}\}) \qquad \mathbf{o}_r = \mathbf{id}(\mathbf{f}, \mathbf{o}_1, \ldots, \mathbf{o}_m) \qquad i = \mathbf{rand}(k) \\ (\beta_i\,!\,(\mathbf{f}, \mathbf{o}_r, \mathbf{o}_1, \ldots, \mathbf{o}_m) \Rightarrow \mathbf{o}_r) \xrightarrow{\beta_i\,!\,(\mathbf{f}, \mathbf{o}_r, \mathbf{o}_1, \ldots, \mathbf{o}_m)} \mathbf{o}_r \\ \mathbf{W}_i \xrightarrow{\beta_i\,?\,(\mathbf{o}_f, \mathbf{o}_r, \mathbf{o}_1, \ldots, \mathbf{o}_m)} \mathbf{seal}(\mathbf{o}_r, \mathbf{f}(\mathbf{get}(\mathbf{o}_1), \ldots, \mathbf{get}(\mathbf{o}_m)))\end{array}}{\langle \mathbf{r}(\mathbf{o}_1, \ldots, \mathbf{o}_m) \parallel \mathbf{W}, (\sigma, \mu)\rangle \rightarrow \left\langle \mathbf{o}_r \parallel \ \ldots \ \parallel \mathbf{seal}(\mathbf{o}_r, \mathbf{f}(\mathbf{get}(\mathbf{o}_1), \ldots, \mathbf{get}(\mathbf{o}_m))); \mathbf{W}_i \parallel \ \ldots, (\sigma', \mu')\right\rangle}\ \textsc{R-Call}$$

The following seal command for worker processes transitions the program to its final state by evaluating all **get** operations, evaluating the function, and actually invoking **seal** on the resulting value.

$$\frac{\mathbf{f}(\mathbf{d}_1, \ldots, \mathbf{d}_m) \rightarrow^* \mathbf{d} \qquad (\alpha_i\,!\,(\mathbf{o}, \mathbf{d}) \Rightarrow \mathbf{o}) \xrightarrow{\alpha_i\,!\,(\mathbf{o}, \mathbf{d})} \mathbf{o} \qquad \langle \mathbf{S}^w, (\mu)\rangle \xrightarrow{\alpha_i\,?\,(\mathbf{o}, \mathbf{d})} \langle \mathbf{o} = \mathbf{d}; \mathbf{S}^w, (\mu)\rangle}{\langle \mathbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{d}_1, \ldots, \mathbf{d}_m)) \parallel \mathbf{S}^w, (\sigma, \mu)\rangle \xrightarrow{\lambda} \left\langle \mathbf{skip} \parallel \mathbf{o} = \mathbf{d}; \mathbf{S}^w, (\sigma', \mu)\right\rangle}\ \textsc{Seal-Cmd}$$

5.4 Compilation of Basic Python Operations

In this section, we formally present a procedure for translating serially executing Python programs to programs which execute concurrently. We provide a proof of correctness for this procedure by way of induction over all operations presented in chapter 5. Let $\mathbf{T}$ denote the

translation operator, which is inductively defined as follows.

$$
\begin{aligned}
\mathbf{T}(\mathbf{skip}) &\equiv \mathbf{skip} \\
\mathbf{T}(\mathbf{c}_1; \mathbf{c}_2) &\equiv (\mathbf{T}(\mathbf{c}_1); \mathbf{T}(\mathbf{c}_2)) \\
\mathbf{T}(\mathbf{x} = \mathbf{e}) &\equiv \mathbf{x} = \mathbf{T}(\mathbf{e}) \\
\mathbf{T}(\mathbf{if\ b\ then\ c}_1\ \mathbf{else\ c}_2) &\equiv \mathbf{if\ get}(\mathbf{T}(\mathbf{b}))\ \mathbf{then}\ \mathbf{T}(\mathbf{c}_1)\ \mathbf{else}\ \mathbf{T}(\mathbf{c}_2) \\
\mathbf{T}(\mathbf{while\ b\ do\ c}) &\equiv \mathbf{while\ get}(\mathbf{T}(\mathbf{b}))\ \mathbf{do}\ \mathbf{T}(\mathbf{c}) \\
\mathbf{T}(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_m)\{\mathbf{e}\}) &\equiv \mathbf{R}(\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_m)\{\mathbf{e}\}) \\
\mathbf{T}(\mathbf{f}(\mathbf{e}_1, \ldots, \mathbf{e}_m)) &\equiv \mathbf{R}(\mathbf{f})(\mathbf{T}(\mathbf{e}_1), \ldots, \mathbf{T}(\mathbf{e}_m)) \\
\mathbf{T}(\mathbf{R}(\mathbf{f})) &\equiv \mathbf{R}(\mathbf{f}) \\
\mathbf{T}(-\mathbf{a}) &\equiv \mathbf{R}(-)(\mathbf{T}(\mathbf{a})) \\
\mathbf{T}(\mathbf{not\,b}) &\equiv \mathbf{R}(\mathbf{not})(\mathbf{T}(\mathbf{b})) \\
\mathbf{T}(\mathbf{a}_1 \bullet_2 \mathbf{a}_2) &\equiv \mathbf{R}(\bullet_2)(\mathbf{T}(\mathbf{a}_1), \mathbf{T}(\mathbf{a}_2)) \\
\mathbf{T}(\mathbf{b}_1 \bullet_2 \mathbf{b}_2) &\equiv \mathbf{R}(\bullet_2)(\mathbf{T}(\mathbf{b}_1), \mathbf{T}(\mathbf{b}_2)) \\
\mathbf{T}(\mathbf{v}) &\equiv \mathbf{put}(\mathbf{v}) \\
\mathbf{T}(\mathbf{x}) &\equiv \mathbf{x}
\end{aligned}
$$

In the above, $\bullet_2$ is shorthand for arbitrary binary operations conditioned on the operands. For instance, $\bullet_2$ in $\mathbf{a}_1 \bullet_2 \mathbf{a}_2$ stands for all binary arithmetic operations. $\mathbf{R}(\bullet_2)$ is the remote function which takes two arguments and returns the result of applying $\bullet_2$ on those arguments. In general, for an $\mathbf{n}$-ary operation $\bullet_\mathbf{n}$, $\mathbf{R}(\bullet_\mathbf{n})$ is the remote function which takes $\mathbf{n}$ arguments and returns the result of applying the $\mathbf{n}$-ary operation.

5.5 Correctness Proofs

The notion of correctness of translated programs is captured by the following theorem.

$$
\begin{aligned}
&\mathbf{if}\ \langle \mathbf{c}, \sigma \rangle \to^* \sigma'\ \mathbf{and}\ \sigma' \neq \bot \\
&\mathbf{then}\ \langle \mathbf{T}(\mathbf{c}) \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu) \rangle \to^* \langle \mathbf{skip} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma'', \mu') \rangle \\
&\mathbf{such\ that}\ \forall (\mathbf{x}, \mathbf{v}) \in \sigma'.\, (\sigma''(\mathbf{x}), \mathbf{v}) \in \mu'.
\end{aligned}
\tag{5.5}
$$

Note that in this theorem we assume $\mathbf{c}$ is terminating. We deal with the case of non-terminating programs in Lemma 5.5. We need to prove this for all Sub-Python commands, so we proceed by structural induction on the derivation of Sub-Python commands and Sub-Futures commands. Specifically, these are the Sub-Futures operation transitions, as well as the resulting configuration transitions, that appear in translated Sub-Python programs. In our proofs, we make the following simplifying assumptions:

1. The number of worker processes $k = 1$.

Remark: Object Hashes are Unique

The function $\mathbf{id}(\cdot)$ generates a hash of its input. For any function $\mathbf{f}(\mathbf{x}_1 \ldots, \mathbf{x}_m)\{\mathbf{e}\}$ the identifier $\mathbf{o} = \mathbf{id}(\mathbf{f}, \mathbf{o}_1, \ldots, \mathbf{o}_m)$ uniquely identifies the invocation of $\mathbf{f}$ on $\mathbf{v}_1, \ldots, \mathbf{v}_m$ where $\mathbf{o}_i = \mathbf{id}(\mathbf{v}_i)$. This is trivially true as the expressions $\mathbf{e}$ are deterministic. If an expression is non-terminating, the entire context in which the expression occurs transitions to **error**.

Lemma: Object Store Does Not Lose Information

Some of the following proofs require that concurrent write requests sent to $\mathbf{S}^w$ from $\mathbf{M}$ and the $\mathbf{W}_j$ never result in a configuration whereby the store loses previously written information. We only care about the case where expressions evaluate to values. We deal with the non-terminating case in Lemma 5.5. Formally,

$$\begin{aligned}&\textbf{if } \langle \mathbf{put}(\mathbf{v}_0) \parallel \mathbf{put}(\mathbf{v}_1) \parallel \ \ldots \ \parallel \mathbf{put}(\mathbf{v}_k) \parallel \mathbf{S}^w, (\sigma, \mu)\rangle \rightarrow^* \langle \mathbf{skip} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu')\rangle \\ &\textbf{then } \forall\, i = 0, \ldots, k\,.\, (\mathbf{id}(\mathbf{v}_i), \mathbf{v}_i) \in \mu'.\end{aligned}$$

Consider the process $\mathbf{S}^w$, which can be in one of two configurations: $\mathbf{S}^w$, or $\mathbf{o}_i = \mathbf{v}_i; \mathbf{S}^w$. In the configuration $\mathbf{S}^w$, the summation over incoming messages non-determinisitcally accepts a single message from the incoming messages over channels α_i. This transition yields the configuration $\mathbf{o}_i = \mathbf{v}_i; \mathbf{S}^w$. Since $\mathbf{S}^w$ is the only process listening on channels $\alpha_0, \ldots, \alpha_k$, other processes P_j where $j \neq i$ are blocking until $\mathbf{o}_i = \mathbf{v}_i; \mathbf{S}^w \rightarrow \mathbf{S}^w$, at which point $\mu[\mathbf{o}_i = \mathbf{v}_i]$, and the next incoming write request will be written to $\mu[\mathbf{o}_i = \mathbf{v}_i]$. Thus, the cardinality of the store μ is monotonically increasing, and μ does not lose previously written information. $\square$

Correctness of Translated Expressions

We start by proving correctness of all translated expressions $\mathbf{e}$. The translation of an expression $\mathbf{e}$ is simply $\mathbf{T}(\mathbf{e})$. We need to prove the following.

$$\textbf{if } (\mathbf{e} \rightarrow^* \mathbf{v}) \textbf{ then } \langle \mathbf{T}(\mathbf{e}) \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu)\rangle \rightarrow^* \langle \mathbf{o} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu')\rangle \textbf{ such that } (\mathbf{o}, \mathbf{v}) \in \mu.$$

We proceed by structural induction on the derivation of expressions.

Base Case

Our base cases are the following:

1. $\mathbf{e} \equiv \mathbf{v}$. We have $\mathbf{T}(\mathbf{v}) = \mathbf{put}(\mathbf{v})$.
2. $\mathbf{e} \equiv \mathbf{x}$. This follows immediately from $\mathbf{e} = \mathbf{v}$ since $\mathbf{T}(\mathbf{x}) = \mathbf{x}$ and $\sigma(\mathbf{T}(\mathbf{x}))$ will be an object id by definition of the translation operator.

$\mathbf{put}(\mathbf{v})$ provides a derivation of the step to $\mathbf{seal}(\mathbf{o}, \mathbf{v})$, and the derivation of **seal** yields

$$\langle \mathbf{seal}(\mathbf{o}, \mathbf{v}) \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu) \rangle \xrightarrow{\lambda} \langle \mathbf{o} \parallel \mathbf{W} \parallel \mathbf{S}^r \parallel \mathbf{o} = \mathbf{v}; \mathbf{S}^w, (\sigma, \mu) \rangle \,.$$

The proof for the base case follows immediately from the derivation of $\mathbf{o} = \mathbf{v}$:

$$\langle \mathbf{o} \parallel \mathbf{W} \parallel \mathbf{S}^r \parallel \mathbf{o} = \mathbf{v}; \mathbf{S}^w, (\sigma, \mu) \rangle \to \langle \mathbf{o} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu[\mathbf{o} = \mathbf{v}]) \rangle \,.$$

Functions

The rest of expressions follow immediately from a proof of function invocation, as the binary operations of arithmetic and boolean expressions, such as $a_1 + a_2$, are translated to remote functions $\mathbf{R}(+)(\mathbf{put}(a_1), \mathbf{put}(a_2))$, where $\mathbf{R}(+) \equiv \mathbf{R}(\mathbf{f}(\mathbf{x}_1, \mathbf{x}_2)\{\mathbf{x}_1 + \mathbf{x}_2\})$. We must show that

$$\begin{aligned} &\textbf{if } (\mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{v}_m) \to \mathbf{v}) \\ &\textbf{then } \langle \mathbf{T}(\mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{v}_m)) \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu) \rangle \to^* \langle \mathbf{o} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu') \rangle \\ &\textbf{such that } (\mathbf{o}, \mathbf{v}) \in \mu'. \end{aligned}$$

$\mathbf{T}(\mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{v}_m)) \equiv \mathbf{R}(\mathbf{f})(\mathbf{T}(\mathbf{v}_1), \ldots, \mathbf{T}(\mathbf{v}_m))$. By the induction hypothesis, we have that $\mathbf{T}(\mathbf{v}_i) \equiv \mathbf{put}(\mathbf{v}_i) \to^* \mathbf{o}_i$, and that by Lemma 5.5, the entire program will transition to a configuration

$$\langle \mathbf{R}(\mathbf{f})(\mathbf{o}_1, \ldots, \mathbf{o}_m) \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu'') \rangle \,,$$

where $\forall\, i = 1, \ldots, m.\, (\mathbf{o}_i, \mathbf{v}_i) \in \mu''$. The rule for remote function call provides a derivation of the transition to

$$\langle \mathbf{o} \parallel \mathbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{get}(\mathbf{o}_1), \ldots, \mathbf{get}(\mathbf{o}_m))); W_1 \parallel \mathbf{S}, (\sigma, \mu'') \rangle \,.$$

For $\mathbf{get}(\mathbf{o}_1)$, the derivation of **get** provides the following transition:

$$\begin{aligned} &\langle \mathbf{o} \parallel \mathbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{get}(\mathbf{o}_1), \ldots, \mathbf{get}(\mathbf{o}_m))); \mathbf{W}_1 \parallel \mathbf{S}, (\sigma, \mu'') \rangle \\ \xrightarrow{\lambda} &\langle \mathbf{o} \parallel \mathbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{rcv}(\mathbf{o}_1), \ldots, \mathbf{get}(\mathbf{o}_m))); \mathbf{W}_1 \parallel (\mathbf{wait}(\mathbf{o}_1); \mathbf{S}_0^r) \parallel \mathbf{S}^w, (\sigma, \mu'') \rangle \,. \end{aligned}$$

For **wait** to make progress, $\mathbf{sealed}(\mathbf{o}_1)$ must evaluate to **True**. The derivation of $\mathbf{sealed}(\mathbf{o}_1)$ requires that $\exists\, \mathbf{v}\,.\, (\mathbf{o}_1, \mathbf{v}) \in \mu''$. This is satisfied since $\mu''(\mathbf{o}_1) = \mathbf{v}_1$. Thus, the derivation of **wait** provides the remaining transitions:

$$\begin{aligned} &\langle \mathbf{o} \parallel \mathbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{rcv}(\mathbf{o}_1), \ldots, \mathbf{get}(\mathbf{o}_m))); \mathbf{W}_1 \parallel (\mathbf{wait}(\mathbf{o}_1); \mathbf{S}_0^r) \parallel \mathbf{S}^w, (\sigma, \mu'') \rangle \\ \xrightarrow{\lambda} &\langle \mathbf{o} \parallel \mathbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{rcv}(\mathbf{o}_1), \ldots, \mathbf{get}(\mathbf{o}_m))); \mathbf{W}_1 \parallel (\mathbf{snd}(\mathbf{o}_1); \mathbf{S}_0^r) \parallel \mathbf{S}^w, (\sigma, \mu'') \rangle \\ \xrightarrow{\lambda} &\langle \mathbf{o} \parallel \mathbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{get}(\mathbf{o}_m))); \mathbf{W}_1 \parallel \mathbf{S}, (\sigma, \mu'') \rangle \,. \end{aligned}$$

The same holds for the remaining **get** operations, which evaluate left-to-right as specified by the evaluation order of function arguments and yield $\langle \mathbf{o} \parallel \mathbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{v}_m)) \parallel \mathbf{S}, (\sigma, \mu'') \rangle$.

Function evaluation yields $\langle \mathbf{o} \parallel \mathbf{seal}(\mathbf{o}, \mathbf{v}); \mathbf{W}_1 \parallel \mathbf{S}, (\sigma, \mu'') \rangle$, and finally the **seal** operation yields the following sequence of configurations

$$\begin{aligned} & \langle \mathbf{o} \parallel \mathbf{seal}(\mathbf{o}, \mathbf{v}); \mathbf{W}_1 \parallel \mathbf{S}, (\sigma, \mu'') \rangle \\ \xrightarrow{\lambda} & \langle \mathbf{o} \parallel \mathbf{skip}; \mathbf{W}_1 \parallel \mathbf{S}^r \parallel \mathbf{o} = \mathbf{v}; \mathbf{S}^w, (\sigma, \mu'') \rangle \\ \xrightarrow{\lambda} & \langle \mathbf{o} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu''[\mathbf{o} = \mathbf{v}]) \rangle . \end{aligned}$$

This concludes the proof for expressions. □

Lemma: Get is Correct

We want to show that if $\mathbf{e} \rightarrow^* \mathbf{v}$ then $\mathbf{get}(\mathbf{T}(\mathbf{e})) \rightarrow^* \mathbf{v}$. We know that $\mathbf{T}(\mathbf{e})) \rightarrow^* \mathbf{o}$ yields a state $\mu'(o) = \mathbf{v}$ from the proof of correctness of expressions. For $\mathbf{get}(\mathbf{o})$, the derivation of **get** provides the following transitions:

$$\begin{aligned} & \langle \mathbf{get}(\mathbf{o}) \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu') \rangle \\ \xrightarrow{\lambda} & \langle \mathbf{rcv}(\mathbf{o}) \parallel \mathbf{W} \parallel (\mathbf{wait}(\mathbf{o}); \mathbf{S}_0^r) \parallel \mathbf{S}^w, (\sigma, \mu') \rangle \\ \xrightarrow{\lambda} & \langle \mathbf{rcv}(\mathbf{o}) \parallel \mathbf{W} \parallel (\mathbf{snd}(\mathbf{o}); \mathbf{S}_0^r) \parallel \mathbf{S}^w, (\sigma, \mu') \rangle \\ \xrightarrow{\lambda} & \langle \mathbf{v} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu') \rangle . \end{aligned}$$

To make progress, $\mathbf{wait}(\mathbf{o})$ requires that $\mathbf{sealed}(\mathbf{o})$ evaluates to **True**, and we have by I.H. $\exists\, \mathbf{v} \,.\, (\mathbf{o}, \mathbf{v}) \in \mu'$.

Correctness of Translated Commands

Assignment

$\mathbf{c} \equiv \mathbf{x} = \mathbf{e}$. We have $\mathbf{T}(\mathbf{x} = \mathbf{e}) \equiv \mathbf{x} = \mathbf{T}(\mathbf{e})$. We have $\langle \mathbf{x} = \mathbf{e}, \sigma \rangle \rightarrow^* \langle \mathbf{v}, \sigma' \rangle$. By definition of the operator $\mathbf{T}$ and the induction hypothesis, we have $\mathbf{M} \equiv \mathbf{x} = \mathbf{o}$ and μ' such that $\mu'(\mathbf{o}) = \mathbf{v}$. Assignment yields $\langle \mathbf{x} = \mathbf{o} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu') \rangle \rightarrow \langle \mathbf{skip} \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma'', \mu') \rangle$ where $\sigma'' = \sigma[\mathbf{x} = \mathbf{o}]$. Thus, we have $(\mathbf{x}, \mathbf{v}) \in \sigma'$ and $(\sigma''(\mathbf{x}), \mathbf{v}) \in \mu'$.

Sequence

$\mathbf{c} \equiv \mathbf{c}_1; \mathbf{c}_2$. Let σ'_0 be the intermediate state after executing $\mathbf{c}_1$ and before executing $\mathbf{c}_2$, and σ''_0 be the intermediate state after executing $T(\mathbf{c}_1)$ and before executing $T(\mathbf{c}_2)$. By induction hypothesis, for $\mathbf{c}_1$ we have $\forall (\mathbf{x}, \mathbf{v}) \in \sigma'_1 . (\sigma''_1(\mathbf{x}), \mathbf{v}) \in \mu'_1$. By induction hypothesis, for $\mathbf{c}_2$ we have $\forall (\mathbf{x}, \mathbf{v}) \in \sigma'_2 . (\sigma''_2(\mathbf{x}), \mathbf{v}) \in \mu'_2$. By definition of the sequence command, we have $\sigma'_0 = \sigma'_1$, $\sigma''_0 = \sigma''_1$. Thus, $\sigma' = \sigma'_2$, $\sigma'' = \sigma''_2$, $\mu' = \mu'_2$, and $\forall (\mathbf{x}, \mathbf{v}) \in \sigma' . (\sigma''(\mathbf{x}), \mathbf{v}) \in \mu'$.

If-Then-Else

$\mathbf{c} \equiv \textbf{if b then } \mathbf{c}_1 \textbf{ else } \mathbf{c}_2$. We have by the I.H. and Lemma 5.5 that $\mathbf{b} = \mathbf{get}(\mathbf{T}(b))$. Thus, if $\mathbf{b} = \mathbf{get}(\mathbf{T}(\mathbf{b})) = \mathbf{True}$ then by I.H. the theorem holds for $\mathbf{c}_1$. Likewise, $\mathbf{b} = \mathbf{get}(\mathbf{T}(\mathbf{b})) = \mathbf{False}$, the theorem holds for $\mathbf{c}_2$ by I.H.

While Loops

Note here we only prove the terminating case. We prove the non-terminating cases in Lemma 5.5, where $\langle \textbf{while b do c}, \sigma\rangle \to \langle \textbf{skip}, \bot\rangle$, and in such cases we have that μ also transitions to $\bot$. We want to show Theorem 5.5 for $\mathbf{c} \equiv \textbf{while b do c}'$.

By I.H. and Lemma 5.5, if $\langle \mathbf{b}, \sigma\rangle \to^* \langle \mathbf{True}, \sigma\rangle$ then $\langle \mathbf{get}(\mathbf{T}(\mathbf{b})), \sigma, \mu\rangle \to^* \langle \mathbf{True}, \sigma, \mu'\rangle$. The same holds when $\mathbf{b}$ evaluates to $\mathbf{False}$. If $\mathbf{b}$ evaluates to $\mathbf{False}$, so does $\mathbf{get}(\mathbf{T}(\mathbf{b}))$, which means $\sigma = \sigma' = \sigma''$ and $\mu = \mu'$, proving the result for $\mathbf{False}$. If $\mathbf{b}$ evaluates to $\mathbf{True}$, then

$$\langle \textbf{while get}(\mathbf{T}(\mathbf{b})) \textbf{ do } \mathbf{T}(\mathbf{c}'), \sigma, \mu\rangle \to \langle \mathbf{T}(\mathbf{c}'); \textbf{while get}(\mathbf{T}(\mathbf{b})) \textbf{ do } \mathbf{T}(\mathbf{c}'), \sigma, \mu\rangle\,.$$

We have already proven the theorem for sequence, thus, we know that if $\mathbf{c}'$ and $\mathbf{T}(\mathbf{c}')$ execute k times and terminate in state σ' and (σ'', μ'), respectively, then $\forall(\mathbf{x}, \mathbf{v}) \in \sigma'$, $(\sigma''(\mathbf{x}), \mathbf{v}) \in \mu'$.

We now want to show that if the serial loop body $\mathbf{c}'$ executes k times, the parallel loop body $\mathbf{T}(\mathbf{c}')$ also executes k times. We show this by proving that $\mathbf{b} = \mathbf{get}(\mathbf{T}(\mathbf{b}))$ after every loop iteration. We have this result by the I.H. for $k = 1$. We can prove the theorem for arbitrary k by showing that the theorem holds after executing $\mathbf{c}'$. Let $\langle \mathbf{c}', \sigma_0\rangle \to^* \langle \mathbf{skip}, \sigma_0'\rangle$ and $\langle \mathbf{T}(\mathbf{c}'), \sigma_0\rangle \to^* \langle \mathbf{skip}, \sigma_0'', \mu_0'\rangle$. We have by the I.H. that $\forall(\mathbf{x}, \mathbf{v}) \in \sigma_0'.\, (\sigma_0''(\mathbf{x}), \mathbf{v}) \in \mu_0'$. Thus, after the kth execution of $\mathbf{c}'$ and $\mathbf{T}(\mathbf{c}')$, we have again by the I.H. and Lemma 5.5 that if $\langle \mathbf{b}, \sigma_0'\rangle \to^* \langle \mathbf{True}, \sigma_0'\rangle$ then $\langle \mathbf{get}(\mathbf{T}(\mathbf{b})), \sigma_0'', \mu_0'\rangle \to^* \langle \mathbf{True}, \sigma_0'', \mu_0''\rangle$. The same holds when $\mathbf{b}$ evaluates to $\mathbf{False}$. This concludes the proof of correctness for translated programs that terminate. □

Corollary: Non-Terminating Programs Translate to Non-Terminating Programs

To show that non-terminating Sub-Python programs are translated to non-terminating Sub-Futures programs, we need to prove the following corollary for expressions and while loops.

$$\begin{aligned} &\textbf{if } \langle \mathbf{c}, \sigma\rangle \to^* \langle \mathbf{skip}, \bot\rangle \\ &\textbf{then } \langle \mathbf{T}(\mathbf{c}) \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu)\rangle \to^* \langle \mathbf{skip} \parallel \mathbf{W} \parallel \mathbf{S}, (\bot, \bot)\rangle \end{aligned} \tag{5.6}$$

Expressions

Here we want to show that, for some non-terminating expression $\mathbf{e}$,

$$\begin{aligned} &\textbf{if } \langle \mathbf{e}, \sigma \rangle \to^* \langle \textbf{error}, \bot \rangle \\ &\textbf{then } \langle \mathbf{T}(\mathbf{e}) \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu) \rangle \to^* \langle \mathbf{o} \parallel \mathbf{W} \parallel \mathbf{S}, (\bot, \bot) \rangle \end{aligned}$$

The only relevant case is when $\mathbf{e} = \mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{v}_m)$, and in such cases, $\mathbf{T}(\mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{v}_m))$ is a remote function. Let $\mathbf{r}(\mathbf{o}_1, \ldots, \mathbf{o}_m) = \mathbf{T}(\mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{v}_m))$. By the derivation of remote function calls, on worker i we transition to a **seal** operation of the form $\textbf{seal}(\mathbf{o}, \mathbf{f}(\mathbf{v}_1, \ldots, \mathbf{v}_m))$ and yield the object id $\mathbf{o}$ corresponding to the result of executing $\mathbf{f}$ [2]. The derivation of **seal** relies on the evaluation of $\mathbf{f}$, which we know evaluates to **error**. Thus, the **seal** command transitions to a state where $\mathbf{S}^w \equiv \mathbf{o} = \textbf{error}; \mathbf{S}^w$. The derivation of $\mathit{Store} - \mathit{Assign} - \infty$ handles this state by transitioning μ to $\bot$. The derivations for Parallel-Store-Error and Parallel-Store-Error-2 handle this state by transitioning σ to $\bot$, and the main process to **skip**. This proves the theorem for expressions and shows that whenever we translate a non-terminating recursive function, the translated remote function is also non-terminating and handled appropriately.

While Loops

We now prove Theorem 5.6 for $\mathbf{c} \equiv \textbf{while b do c}'$, which is the only command that may not terminate. By the derivation of While-∞, if $\textbf{while b do c}'$ is non-terminating, we have $\langle \textbf{while b do c}', \sigma \rangle \to \langle \textbf{skip}, \bot \rangle$, which relies on the premise

$$\forall k.\ \langle \textbf{while}_k \textbf{ b do c}', \sigma \rangle \to \langle \textbf{skip}, \bot \rangle .$$

If we prove that this premise is preserved under the translation operator, then we have proven our result. We want to prove that

$$\begin{aligned} &\textbf{if } \forall k_1.\ \langle \textbf{while}_{k_1} \textbf{ b do c}', \sigma \rangle \to^{k_1+1} \langle \textbf{skip}, \bot \rangle \\ &\textbf{then } \forall k_2.\ \langle \textbf{while}_{k_2} \textbf{ get}(\mathbf{T}(\mathbf{b})) \textbf{ do } \mathbf{T}(\mathbf{c}') \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu) \rangle \to^{k_2+1} \langle \textbf{skip} \parallel \mathbf{W} \parallel \mathbf{S}, (\bot, \bot) \rangle , \end{aligned}$$

We proceed by induction on k_2. If $k_2 = 0$, then by Theorem 5.5 we have that $\langle \textbf{get}(\mathbf{T}(\mathbf{b})), \sigma, \mu \rangle$ evaluates to the same value as $\langle \mathbf{b}, \sigma \rangle$. Since $\langle \mathbf{b}, \sigma \rangle$ evaluates to **True** for all k_1,

$$\langle \textbf{while}_0 \textbf{ get}(\mathbf{T}(\mathbf{b})) \textbf{ do } \mathbf{T}(\mathbf{c}') \parallel \mathbf{W} \parallel \mathbf{S}, (\sigma, \mu) \rangle \to^1 \langle \textbf{skip} \parallel \mathbf{W} \parallel \mathbf{S}, (\bot, \mu) \rangle .$$

By the derivation of Parallel-Main-Error and Parallel-Main-Error-2, the Sub-Futures program transitions to a state where $\mu = \bot$.

For $k_2 > 0$, we'll show that

$$\langle \textbf{while}_{k_2} \textbf{ get}(\mathbf{T}(\mathbf{b})) \textbf{ do } \mathbf{T}(\mathbf{c}'), \sigma, \mu \rangle \to^1 \langle \mathbf{T}(\mathbf{c}'); \textbf{while}_{k_2-1} \textbf{ get}(\mathbf{T}(\mathbf{b})) \textbf{ do } \mathbf{T}(\mathbf{c}'), \sigma, \mu' \rangle .$$

[2] A **get** invocation on such an object id will yield **error**. This is not needed in the proof but is worth noting.

The derivation of While-k-True provides the transition

$$\langle \mathbf{while}_k\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}', \sigma\rangle \to^1 \langle \mathbf{c}'; \mathbf{while}_{k-1}\ \mathbf{b}\ \mathbf{do}\ \mathbf{c}', \sigma\rangle$$

and requires that $\mathbf{b}$ evaluates to **True**. By theorem 5.5 and Lemma 5.5 we have that if $\langle \mathbf{b}, \sigma\rangle$ evaluates to **True**, then $\langle \mathbf{get}(\mathbf{T}(\mathbf{b})), \sigma, \mu\rangle$ must also evaluate to **True**. Since $\mathbf{b}$ evaluates to **True** for all k_1, $\langle \mathbf{while}_{k_2}\ \mathbf{get}(\mathbf{T}(\mathbf{b}))\ \mathbf{do}\ \mathbf{T}(\mathbf{c}'), \sigma, \mu\rangle \to \langle \mathbf{T}(\mathbf{c}'); \mathbf{while}_{k_2-1}\ \mathbf{get}(\mathbf{T}(\mathbf{b}))\ \mathbf{do}\ \mathbf{T}(\mathbf{c}'), \sigma, \mu'\rangle$. The derivation of sequence yields the transition

$$\langle \mathbf{T}(\mathbf{c}'); \mathbf{while}_{k_2-1}\ \mathbf{get}(\mathbf{T}(\mathbf{b}))\ \mathbf{do}\ \mathbf{T}(\mathbf{c}'), \sigma, \mu'\rangle \to \langle \mathbf{while}_{k_2-1}\ \mathbf{get}(\mathbf{T}(\mathbf{b}))\ \mathbf{do}\ \mathbf{T}(\mathbf{c}'), \sigma', \mu''\rangle\,.$$

By the I.H., we have that $\langle \mathbf{while}_{k_2-1}\ \mathbf{get}(\mathbf{T}(\mathbf{b}))\ \mathbf{do}\ \mathbf{T}(\mathbf{c}'), \sigma', \mu''\rangle \to^{k_2} \langle \mathbf{skip} \parallel \mathbf{W} \parallel \mathbf{S}, (\bot, \bot)\rangle$. □

5.6 Extension to Distributed Memory

While we do not model distributed memory explicitly, we informally show that our approach can be extended to the distributed-memory setting. Consider the key/value stores introduced in Section 5.1: σ, σ', and μ. We introduce a new key/value store, ω, which maps futures $\mathbf{o}$ to nodes η, where η serves as an identifier corresponding to a set of workers which operate locally on the key/value store μ. An RPC call with a result associated with future $\mathbf{o}$ is assigned to a worker on node η. We therefore set $(\mathbf{o}, \eta) \in \omega$. During dependency resolution, a worker will block until $\mathbf{o}$ exists in ω instead of blocking until $\mathbf{o}$ exists in μ. We make this change because the data may no longer be co-located with the worker executing the RPC. To obtain all future and value pairs $(\mathbf{o}, \mathbf{v})$ required to execute an RPC, a worker initiates a read from a non-local object store μ' and writes $\mathbf{o}$ with associated value $\mathbf{v}$ to its local object store μ. The worker then executes the RPC.

Our correctness theorem is extended to include ω as follows. Consider a cluster with p nodes $\eta_1, \ldots, \eta_p$. We have p stores local to each node, defined as $\mu_1, \ldots, \mu_p$. If some program from the source language terminates with store content σ, then the target language must also terminate with store contents σ', μ_i, and ω such that for any key/value pair $(\mathbf{x}, \mathbf{v}) \in \sigma$, there exists $(\mathbf{o}, \eta_i) \in \omega$, $(\mathbf{o}, \mathbf{v}) \in \mu_i$, and $(\mathbf{x}, \mathbf{o}) \in \sigma'$. With minor modifications to the semantics of RPCs as described above, the proof of correctness for the distributed memory setting is very similar to our existing proof.

Chapter 6

Compilation of Block-Partitioned Array Operations

In this chapter, we extend our formalism from Chapter 5 to support 2-dimensional arrays. We provide a brief summary and intuition before presenting formal syntax and semantics of our formulation of arrays.

Consider a snippet of our NumPy-inspired array syntax. We focus on the matrix multiplication operator, which is the only binary operation which does not trivially generalize to n dimensions.

$$\mathbf{A} \ ::= \ \mathbf{N}(n,m) \mid \mathbf{x} \mid \ \ldots \ \mid \mathbf{A}_1 @ \mathbf{A}_2$$

We have $\mathbf{A} \in \mathcal{A}$, where $\mathcal{A}$ is the set defined above in BNF form, and $\mathbf{N}(n,m) \in \mathbb{R}^{n \times m}$ is a real-valued 2-dimensional array. To translate arrays to our target language, we will need a way to represent them as futures. We define the syntax for a 2-dimensional array of futures as follows.

$$\mathbf{o} \ ::= \ \ldots \mid \mathbf{o}(\kappa_1, \kappa_2)$$

The syntax $\mathbf{o}(\kappa_1, \kappa_2)$ denotes a 2-dimensional futures encoding of 2-dimensional arrays, where κ_1, κ_2 define the number of blocks along axes 1 and 2. To ease the exposition of our treatment of arrays, assume that $\kappa_1 = n$ and $\kappa_1 = m$, so that every value $\mathbf{v}_{i,j}$ in $\mathbf{N}(n,m)$ is translated to a future $\mathbf{o}_{i,j}$ in $\mathbf{o}(n,m)$. The structure of 2-dimensional futures is as follows:

$$\mathbf{o}(n,m) = \begin{bmatrix} \mathbf{o}_{1,1} & \ldots & \mathbf{o}_{1,m} \\ \vdots & \ddots & \vdots \\ \mathbf{o}_{n,1} & \ldots & \mathbf{o}_{n,m}. \end{bmatrix} \tag{6.1}$$

We call 2-dimensional arrays of futures *2-dimensional futures* for short. For our translation of matrix multiplication, we denote by $\mathbf{x}_1, \mathbf{N}_1(n_1, m_1), \mathbf{o}_1(n_1, m_1)$ the operands corresponding to the left-hand side of the @ operation, and by $\mathbf{x}_2, \mathbf{N}_2(n_2, m_2), \mathbf{o}_2(n_2, m_2)$ the right-hand

side. The result is denoted by $\mathbf{N}(n, m)$ and $\mathbf{o}(n, m)$, where $n = n_1$ and $m = m_2$. We use the notation $\mathbf{o}_1(n_1, m_1)_{i,:}$ and $\mathbf{o}_1(n_1, m_1)_{:,j}$ to denote the ith row and jth column of $\mathbf{o}_1$, respectively. The local function $\mathbf{g}(\mathbf{args}_{i,j})$ computes the i, j entry of the result $\mathbf{N}(n, m)$, where $\mathbf{args}_{i,j} = (\mathbf{o}_1(n_1, m_1)_{i,:}, \mathbf{o}_2(n_2, m_2)_{:,j}, i, j)$. We define $\mathbf{g}$ as follows:

$$\begin{aligned}\mathbf{g}(\mathbf{args}_{i,j}) \equiv &\mathbf{R}(+)(\\ &\mathbf{R}(@)(\mathbf{o}_1(n_1, m_1)_{i,1}, \mathbf{o}_2(n_2, m_2)_{1,j}) , \\ &\dots , \\ &\mathbf{R}(@)(\mathbf{o}_1(n_1, m_1)_{i,k}, \mathbf{o}_2(n_2, m_2)_{k,j})),\end{aligned}$$

where $k = m_1 = n_2$. The translation operator for matrix multiplication is as follows:

$$\begin{aligned}\mathbf{T}(\mathbf{o}_1(n_1, m_1) \,@\, \mathbf{o}_2(n_2, m_2)) \equiv \\ \begin{bmatrix} \mathbf{g}(\mathbf{args}_{1,1}) & \dots & \mathbf{g}(\mathbf{args}_{1,m}) \\ \vdots & \ddots & \vdots \\ \mathbf{g}(\mathbf{args}_{n,1}) & \dots & \mathbf{g}(\mathbf{args}_{n,m}) \end{bmatrix} \\ \mathbf{T}(\mathbf{x}_1 \,@\, \mathbf{x}_2) \equiv \mathbf{T}(\mathbf{T}(\mathbf{x}_1) \,@\, \mathbf{T}(\mathbf{x}_2)).\end{aligned}$$

Each entry $\mathbf{o}_{i,j}$ in array $N(n, m)$ is computed using remote functions. While this is inefficient due to the RPC overhead problem, it illustrates our approach to parallelizing array operations.

The correctness of matrix multiplication follows from our proof of correctness of functions. Each entry of the matrix is computed by a composition of basic linear algebra operations on futures. Our translation operator generates $n \times m$ such remote function compositions to generate each future entry of $\mathbf{o}(n, m)$. The resulting 2-dimensional futures object therefore contains entries such that for each value $\mathbf{v}_{i,j}$ in $\mathbf{N}(n, m)$ computed by a Python program, we have $(\mathbf{o}_{i,j}, \mathbf{v}_{i,j}) \in \mu$.

Data Dependency Resolution and Concurrency

The 2-dimensional futures object solves the problem of establishing data dependencies within a matrix multiplication operation by translating all values to futures, and all binary operations to RPCs. The semantics of our execution model is defined on k worker processes. Futures generated by RPCs are computed on workers, which block until the futures on which they depend are made available. Let k go to infinity. In this configuration, all operations which have no dependencies execute concurrently, and all operations which have dependencies block until their dependencies are resolved. Thus, our formulation of 2-dimensional futures simultaneously solves the data dependency and concurrency problem.

In this chapter, We extend the language presented in Chapter 5 to support 2-dimensional arrays and basic linear algebra operations. Our syntax, semantics, and translation operators are defined for a core subset of the NumPy API, which illustrates our general approach

to translating NumPy syntax to our representation of multi-dimensional futures and their operations. The definition of Assignment differs slightly from what we implement in practice.

6.1 Syntax

$$
\begin{array}{rcl}
\mathbf{c} & ::= & \ldots \mid \mathbf{x}[\mathbf{i}_1{:}\mathbf{i}_2, \mathbf{j}_1{:}\mathbf{j}_2] = \mathbf{e} \mid \mathbf{x}[\mathbf{i}, :] = \mathbf{e} \mid \mathbf{x}[:, \mathbf{j}] = \mathbf{e} \mid \mathbf{x}[\mathbf{i}, \mathbf{j}] = \mathbf{e} \\
\mathbf{e} & ::= & \ldots \mid \mathbf{A} \\
\mathbf{A} & ::= & \mathbf{N}(\mathbf{n}, \mathbf{m}) \mid \mathbf{x} \\
 & & \mid \mathbf{Zeros}(\mathbf{n}, \mathbf{m}) \mid \mathbf{Read}(\mathbf{s}, \mathbf{n}, \mathbf{m}) \mid \mathbf{Read}(\mathbf{s}, \mathbf{i}_1{:}\mathbf{i}_2, \mathbf{j}_1{:}\mathbf{j}_2) \\
 & & \mid \mathbf{A}[\mathbf{i}_1{:}\mathbf{i}_2, \mathbf{j}_1{:}\mathbf{j}_2] \mid \mathbf{A}[\mathbf{i}, :] \mid \mathbf{A}[:, \mathbf{j}] \mid \mathbf{A}[\mathbf{i}, \mathbf{j}] \\
 & & \mid \mathbf{A}.\mathbf{T} \mid \mathbf{A}_1 @ \mathbf{A}_2 \mid \mathbf{A}_1 + \mathbf{A}_2 \mid \mathbf{A}_1 - \mathbf{A}_2 \\
 & & \mid \mathbf{A} * \mathbf{a} \mid \mathbf{A} + \mathbf{a} \mid \mathbf{A} - \mathbf{a} \mid \mathbf{A}/\mathbf{a} \\
\mathbf{a} & ::= & \ldots \mid \mathbf{A}.\mathbf{shape}[0] \mid \mathbf{A}.\mathbf{shape}[1] \mid \mathbf{pow}(\mathbf{x}, \mathbf{n}) \mid \mathbf{sqrt}(\mathbf{n}) \mid \mathbf{norm}(\mathbf{A}_1, \mathbf{A}_2) \\
\mathbf{v} & ::= & \ldots \mid \mathbf{N}(\mathbf{n}, \mathbf{m})
\end{array}
$$

We have $\mathbf{A} \in \mathcal{A}$, where $\mathcal{A}$ is the set defined above in BNF form, and $\mathbf{N}(n, m) \in \mathbb{R}^{n \times m}$ is a real-valued 2-dimensional array. **Zeros** and **Read** are the only constructors for real-valued arrays. We introduce $\omega \subseteq \mathcal{S} \times \mathbb{N} \times \mathbb{N} \times \mathbb{R}$, which corresponds to a read-only object containing entries of 2-dimensional arrays. We access entries using $\omega(\mathbf{s}, \mathbf{n}_1, \mathbf{n}_2) \in \mathbb{R}$. $\mathbf{s} \in \mathcal{S}$ and $\mathcal{S}$ corresponds to the space of all strings. For brevity, we exclude ω in program configurations.

6.2 Semantics

Declaration

$$\frac{0 < n, m \qquad 0 \leq i < n \qquad 0 \leq j < m \qquad \mathbf{N}(n, m)[i, j] = 0}{\langle \mathbf{Zeros}(n, m), \sigma \rangle \to \langle \mathbf{N}(n, m), \sigma \rangle} \textsc{Zeros}$$

$$\frac{0 < n, m \qquad 0 \leq i < n \qquad 0 \leq j < m \qquad \mathbf{N}(n, m) = \mathbf{Read}(\mathbf{s},\ 0{:}n,\ 0{:}m)}{\langle \mathbf{Read}(\mathbf{s}, n, m), \sigma \rangle \to \langle \mathbf{N}(n, m), \sigma \rangle} \textsc{Read}$$

$$\frac{\begin{array}{c} n = i_2 - i_1 \qquad m = j_2 - j_1 \qquad 0 < n, m \qquad i_1 \leq i < i_2 \qquad j_1 \leq j < j_2 \\ (\mathbf{s},\ n - 1,\ m - 1) \in \omega \qquad \mathbf{N}(n, m)[i, j] = \omega(\mathbf{s}, i, j) \end{array}}{\langle \mathbf{Read}(\mathbf{s},\ i_1{:}i_2,\ j_1{:}j_2), \sigma \rangle \to \langle \mathbf{N}(n, m), \sigma \rangle} \textsc{Read-Block}$$

$$\frac{\sigma(x) = \mathbf{N}(n, m)}{\langle \mathbf{x}, \sigma \rangle \to \langle \mathbf{N}(n, m), \sigma \rangle} \textsc{Array-Lookup}$$

Selection

Selection

$$\frac{\langle \mathbf{A}, \sigma\rangle \to \langle \mathbf{N}'(n', m'), \sigma\rangle \qquad 0 \le i_1 < i_2 \le n' \qquad 0 \le j_1 < j_2 \le m' \qquad n = i_2 - i_1 \qquad m = j_2 - j_1 \qquad 0 < n, m \qquad 0 \le i < n \qquad 0 \le j < m \qquad \mathbf{N}(n, m)[i, j] = \mathbf{N}'(n', m')[i_1 + i, j_1 + j]}{\langle \mathbf{A}[i_1{:}i_2,\ j_1{:}j_2], \sigma\rangle \to \langle \mathbf{N}(n, m), \sigma\rangle}\ \textsc{Select-Block}$$

$$\frac{\langle \mathbf{A}, \sigma\rangle \to \langle \mathbf{N}'(n, m), \sigma\rangle \qquad 0 \le i < n \qquad \langle \mathbf{N}'(n, m)[i{:}i+1,\ 0{:}m], \sigma\rangle \to \langle \mathbf{N}(1, m), \sigma\rangle}{\langle \mathbf{A}[i, :], \sigma\rangle \to \langle \mathbf{N}(1, m), \sigma\rangle}\ \textsc{Select-Row}$$

$$\frac{\langle \mathbf{A}, \sigma\rangle \to \langle \mathbf{N}'(n, m), \sigma\rangle \qquad 0 \le j < m \qquad \langle \mathbf{N}'(n, m)[0{:}n,\ j{:}j+1], \sigma\rangle \to \langle \mathbf{N}(n, 1), \sigma\rangle}{\langle \mathbf{A}[:, j], \sigma\rangle \to \langle \mathbf{N}(n, 1), \sigma\rangle}\ \textsc{Select-Col}$$

$$\frac{\langle \mathbf{A}, \sigma\rangle \to \langle \mathbf{N}'(n, m), \sigma\rangle \qquad 0 \le i < m \qquad 0 \le j < m \qquad \langle \mathbf{N}'(n, m)[i{:}i+1,\ j{:}j+1], \sigma\rangle \to \langle \mathbf{N}(1, 1), \sigma\rangle}{\langle \mathbf{A}[i, j], \sigma\rangle \to \langle \mathbf{N}(1, 1), \sigma\rangle}\ \textsc{Select-Entry}$$

Array Assignment

We model array assignments as pure functions: Assigning new values to entries of an existing array produces a new array object.

$$\frac{\langle \mathbf{x}, \sigma\rangle \to \langle \mathbf{N}'(n', m'), \sigma\rangle \qquad 0 \le i_1 < i_2 \le n' \qquad 0 \le j_1 < j_2 \le m' \qquad \langle \mathbf{e}, \sigma\rangle \to \langle \mathbf{N}''(n'', m''), \sigma\rangle \qquad n'' = i_2 - i_1 \qquad m'' = j_2 - j_1 \qquad 0 \le i'' < n'' \qquad 0 \le j'' < m'' \qquad 0 \le i' < n' \wedge i' \ne i'' \qquad 0 \le j' < m' \wedge j' \ne j'' \qquad n = n' \qquad m = m' \qquad \mathbf{N}(n, m)[i_1 + i'', j_1 + j''] = \mathbf{N}''(n'', m'')[i'', j''] \qquad \mathbf{N}(n, m)[i', j'] = \mathbf{N}'(n', m')[i', j']}{\langle \mathbf{x}[i_1{:}i_2,\ j_1{:}j_2] = \mathbf{e}, \sigma\rangle \to \langle \mathbf{N}(n, m), \sigma\rangle}\ \textsc{Assign-Block}$$

$$\frac{\langle \mathbf{x}, \sigma\rangle \to \langle \mathbf{N}'(n, m), \sigma\rangle \qquad i_1 = i \qquad i_2 = i + 1 \qquad j_1 = 0 \qquad j_2 = m \qquad \langle \mathbf{x}[i_1{:}i_2,\ j_1{:}j_2] = \mathbf{e}, \sigma\rangle \to \langle \mathbf{N}(n, m), \sigma\rangle}{\langle \mathbf{x}[i, :] = \mathbf{e}, \sigma\rangle \to \langle \mathbf{N}(n, m), \sigma\rangle}\ \textsc{Assign-Row}$$

$$\frac{\langle \mathbf{x}, \sigma\rangle \to \langle \mathbf{N}'(n, m), \sigma\rangle \qquad i_1 = 0 \qquad i_2 = n \qquad j_1 = j \qquad j_2 = j + 1 \qquad \langle \mathbf{x}[i_1{:}i_2,\ j_1{:}j_2] = \mathbf{e}, \sigma\rangle \to \langle \mathbf{N}(n, m), \sigma\rangle}{\langle \mathbf{x}[:, j] = \mathbf{e}, \sigma\rangle \to \langle \mathbf{N}(n, m), \sigma\rangle}\ \textsc{Assign-Col}$$

$$\frac{\langle \mathbf{x}, \sigma\rangle \to \langle \mathbf{N}'(n, m), \sigma\rangle \qquad i_1 = i \qquad i_2 = i + 1 \qquad j_1 = j \qquad j_2 = j + 1 \qquad \langle \mathbf{x}[i_1{:}i_2,\ j_1{:}j_2] = \mathbf{e}, \sigma\rangle \to \langle \mathbf{N}(n, m), \sigma\rangle}{\langle \mathbf{x}[i, j] = \mathbf{e}, \sigma\rangle \to \langle \mathbf{N}(n, m), \sigma\rangle}\ \textsc{Assign-Entry}$$

Operators

We rely on underlying implementations of array operators, such as the @ operator implemented by NumPy [25]. We denote such reliance by coloring the operator blue. We denote the binary operations corresponding to Array-Array operators, and Array-Real operators as $\bullet$.

$$\frac{\langle \mathbf{A}, \sigma \rangle \to \langle \mathbf{N}'(n,m), \sigma \rangle \qquad 0 \le i < n \qquad 0 \le j < m \qquad \mathbf{N}(m,n)[j,i] = \mathbf{N}'(n,m)[i,j]}{\langle \mathbf{A}.\top, \sigma \rangle \to \langle \mathbf{N}(m,n), \sigma \rangle} \textsc{Transpose}$$

$$\frac{\begin{array}{c}\langle \mathbf{A}_1, \sigma \rangle \to \langle \mathbf{N}_1(n_1,m_1), \sigma \rangle \qquad \langle \mathbf{A}_2, \sigma \rangle \to \langle \mathbf{N}_2(n_2,m_2), \sigma \rangle \\ m_1 = n_2 \qquad n = n_1 \qquad m = m_2 \qquad \mathbf{N}(n,m) = \mathbf{N}_1(n_1,m_1) \mathbin{@} \mathbf{N}_2(n_2,m_2)\end{array}}{\langle \mathbf{A}_1 \mathbin{@} \mathbf{A}_2, \sigma \rangle \to \langle \mathbf{N}(n,m), \sigma \rangle} \textsc{Matrix-Multiply}$$

$$\frac{\begin{array}{c}\langle \mathbf{A}_1, \sigma \rangle \to \langle \mathbf{N}_1(n_1,m_1), \sigma \rangle \qquad \langle \mathbf{A}_2, \sigma \rangle \to \langle \mathbf{N}_2(n_2,m_2), \sigma \rangle \\ n = n_1 = n_2 \qquad m = m_1 = m_2 \qquad \mathbf{N}(n,m) = \mathbf{N}_1(n_1,m_1) \bullet \mathbf{N}_2(n_2,m_2)\end{array}}{\langle \mathbf{A}_1 \bullet \mathbf{A}_2, \sigma \rangle \to \langle \mathbf{N}(n,m), \sigma \rangle} \textsc{Array-Array-Binary-Op}$$

$$\frac{\langle \mathbf{A}, \sigma \rangle \to \langle \mathbf{N}'(n,m), \sigma \rangle \qquad \langle \mathbf{a}, \sigma \rangle \to \langle \mathbf{n}, \sigma \rangle \qquad \mathbf{N}(n,m) = \mathbf{N}'(n,m) \bullet \mathbf{n}}{\langle \mathbf{A} \bullet \mathbf{a}, \sigma \rangle \to \langle \mathbf{N}(n,m), \sigma \rangle} \textsc{Array-Real-Binary-Op}$$

Additional Array and Arithmetic Expressions

$$\frac{\begin{array}{c}\langle \mathbf{A}_1, \sigma \rangle \to \langle \mathbf{N}_1(n_1,1), \sigma \rangle \qquad \langle \mathbf{A}_2, \sigma \rangle \to \langle \mathbf{N}_2(n_2,1), \sigma \rangle \qquad n_1 = n_2 \\ \langle \mathbf{N}_1(n_1,1).\top \mathbin{@} \mathbf{N}_2(n_2,1), \sigma \rangle \to \langle \mathbf{N}(1,1), \sigma \rangle\end{array}}{\langle \mathbf{norm}(\mathbf{A}_1, \mathbf{A}_2), \sigma \rangle \to \langle \mathbf{N}(1,1), \sigma \rangle} \textsc{Vector-Norm}$$

$$\frac{\mathbf{n} = \mathbf{N}(1,1)[0,0]}{\langle \mathbf{N}(1,1), \sigma \rangle \to \langle \mathbf{n}, \sigma \rangle} \textsc{Array-to-Real}$$

$$\frac{\mathbf{n} = \mathbf{n}_1^{\mathbf{n}_2}}{\langle \mathbf{pow}(\mathbf{n}_1, \mathbf{n}_2), \sigma \rangle \to \langle \mathbf{n}, \sigma \rangle} \textsc{Power} \qquad \frac{\mathbf{n}' = \sqrt{n}}{\langle \mathbf{sqrt}(\mathbf{n}), \sigma \rangle \to \langle \mathbf{n}', \sigma \rangle} \textsc{Square Root}$$

6.3 Translation

In our target language, we assume a grid layout for 2-dimensional arrays. For each array $\mathbf{A}^i(n_i, m_i)$ in a program, a block shape $(\mathbf{c}_1^i, \mathbf{c}_2^i)$ is chosen. The block shape is comprised of two integers corresponding to the block's length along each axis of $\mathbf{A}^i(n_i, m_i)$ with the constraint that if $n_i = n_j$ then $c_1^i = c_1^j$ and if $m_i = m_j$ then $c_2^i = c_2^j$. Let $n//k$ be the integer division of n by k. Then $\mathbf{divup}(n,k) = (n + k - 1)//k$ is the integer division of n

by k rounded up. For each array, we have a grid of $\mathbf{divup}(n, \mathbf{c}_1) \times \mathbf{divup}(m, \mathbf{c}_2)$ blocks. We denote the 2-dimensional grid layout of array $\mathbf{A}^i$ by $\kappa_1^i \times \kappa_2^i$. We drop the superscripts i when the array in question is obvious. An additional object id structure is given below to represent futures for arrays. Below, we denote the object ids corresponding to blocks of an array by $\mathbf{o}_{i,j}$ for $1 \leq i \leq \kappa_1 \wedge 1 \leq j \leq \kappa_2$.

$$\mathbf{o}(\kappa_1, \kappa_2) = \begin{bmatrix} \mathbf{o}_{1,1} & \dots & \mathbf{o}_{1,\kappa_2} \\ \vdots & \ddots & \vdots \\ \mathbf{o}_{\kappa_1,1} & \dots & \mathbf{o}_{\kappa_1,\kappa_2} \end{bmatrix} \tag{6.2}$$

We extend the syntax of object ids below to include the above data structure.

```
o ::= ... | o(\kappa_1, \kappa_2)
```

The index set corresponding to the indices of the i, j block of $\mathbf{A}$ is given by

$$\mathbf{I}(\kappa_1, \kappa_2)_{i,j} = \{(i', j') \mid i' = (i-1)\mathbf{c}_1, \dots, i\mathbf{c}_1 - 1\,;\, j' = (j-1)\mathbf{c}_2, \dots, j\mathbf{c}_2 - 1\}.$$

We assume for simplicity that the size of the last block along the first and second axes are $\min(\kappa_1\mathbf{c}_1, n)$, and $\min(\kappa_2\mathbf{c}_2, m)$, respectively. Translations for arrays and their operators are defined as follows.

Declaration

$$\mathbf{T}(\mathbf{Zeros}(n, m)) \equiv \begin{bmatrix} \mathbf{R}(\mathbf{Zeros})(n, m, 1, 1) & \dots & \mathbf{R}(\mathbf{Zeros})(n, m, 1, \kappa_2) \\ \vdots & \ddots & \vdots \\ \mathbf{R}(\mathbf{Zeros})(n, m, \kappa_1, 1) & \dots & \mathbf{R}(\mathbf{Zeros})(n, m, \kappa_1, \kappa_2) \end{bmatrix}$$

$$\mathbf{T}(\mathbf{Read}(\mathbf{s}, n, m)) \equiv \begin{bmatrix} \mathbf{R}(\mathbf{Read})(\mathbf{s}, n, m, 1, 1) & \dots & \mathbf{R}(\mathbf{Read})(\mathbf{s}, n, m, 1, \kappa_2) \\ \vdots & \ddots & \vdots \\ \mathbf{R}(\mathbf{Read})(\mathbf{s}, n, m, \kappa_1, 1) & \dots & \mathbf{R}(\mathbf{Read})(\mathbf{s}, n, m, \kappa_1, \kappa_2) \end{bmatrix}$$

For brevity, we extend **Zeros** and **Read** in their remote counterparts to take block indices.

Selection

Let n', m' denote the dimensions of the array $\mathbf{A}'$ from which entries are being selected, and n, m be the dimensions of the resulting array $\mathbf{A}$. We denote by $\mathbf{o}'$ and $\mathbf{o}$ the object id to which $\mathbf{A}'$ and $\mathbf{A}$ are translated, respectively. We know $n = i_2 - i_1$, $m = j_2 - j_1$, and we also denote by κ_1', κ_2' the number of blocks in $\mathbf{A}'$, and κ_1, κ_2 the number of blocks in $\mathbf{A}$. We can now derive the index sets $\mathbf{I}'(\kappa_1', \kappa_2')_{i',j'}$ for the block representation of $\mathbf{A}'$ and $\mathbf{I}(\kappa_1, \kappa_2)_{i,j}$ for the block representation of $\mathbf{A} = \mathbf{A}'[i_1{:}i_2, j_1{:}j_2]$. We define the selection set $\mathbf{S}_{i,j} = \{(i_1{+}i'', j_1{+}j'' \mid$

$(i'', j'') \in \mathbf{I}(\tau_1, \tau_2)_{i,j}\}$ to obtain $\mathbf{args}_{i,j} = \{(i', j', o'_{i',j'}) \mid \mathbf{I}'(\kappa'_1, \kappa'_2)_{i',j'} \cap \mathbf{S}_{i,j} \neq \emptyset\}$, the arguments provided to the operator $\mathbf{Sel}(\mathbf{args}, i, j)$, where $\mathbf{args} = (\mathbf{args}_{i,j}, i_1, i_2, j_1, j_2)$. The selection operator returns an object id $\mathbf{o}(\kappa_1, \kappa_2)_{i,j}$ corresponding to block i, j of $\mathbf{A} = \mathbf{A}'[i_1{:}i_2, j_1{:}j_2]$.

$$\mathbf{T}(\mathbf{o}'(\kappa'_1, \kappa'_2)[i_1{:}i_2, j_1{:}j_2]) \equiv \begin{bmatrix} \mathbf{R}(\mathbf{Sel})(\mathbf{args}, 1, 1) & \dots & \mathbf{R}(\mathbf{Sel})(\mathbf{args}, 1, \kappa_2) \\ \vdots & \ddots & \vdots \\ \mathbf{R}(\mathbf{Sel})(\mathbf{args}, \kappa_1, 1) & \dots & \mathbf{R}(\mathbf{Sel})(\mathbf{args}, \kappa_1, \kappa_2) \end{bmatrix}$$

$$\begin{aligned} \mathbf{T}(\mathbf{A}'[i_1{:}i_2, j_1{:}j_2]) &\equiv \mathbf{T}(\mathbf{T}(x)[i_1{:}i_2, j_1{:}j_2]) \\ \mathbf{T}(\mathbf{A}'[i', :]) &\equiv \mathbf{T}(\mathbf{A}'[i'{:}i'+1, 0{:}m']) \\ \mathbf{T}(\mathbf{A}'[:, j']) &\equiv \mathbf{T}(\mathbf{A}'[0{:}n', j'{:}j'+1]) \\ \mathbf{T}(\mathbf{A}'[i', j']) &\equiv \mathbf{T}(\mathbf{A}'[i'{:}i'+1, j'{:}j'+1]) \end{aligned}$$

Assignment

Let n', m' be the dimensions of $\mathbf{A}'$, the array to which new values are being assigned, and $n'' = i_2 - i_1, m'' = j_2 - j_1$ the dimensions of the array $\mathbf{A}''$ containing the new values. We further define $\mathbf{o}'(m', n')$ and $\mathbf{o}''(m'', n'')$ as the object ids corresponding to $\mathbf{A}'$ and $\mathbf{A}''$, respectively. The number of blocks in $\mathbf{A}'$ and $\mathbf{A}''$ are κ'_1, κ'_2 and κ''_1, κ''_2, respectively. We use $\mathbf{Asgn}(\mathbf{args}, i, j)$, where $\mathbf{args} = (\mathbf{args}_{i,j}, \mathbf{o}''(\kappa''_1, \kappa''_2)_{i,j}, i_1, i_2, j_1, j_2)$, to denote the assignment $\mathbf{A}'[i_1{:}i_2,\ j_1{:}j_2]_{i,j} = \mathbf{A}''_{i,j}$. We compute $\mathbf{args}_{i,j}$ in a similar fashion to what is done for the selection operator.

$$\mathbf{o}'(n', m')[i_1{:}i_2,\ j_1{:}j_2] = \mathbf{o}''(n'', m'') \equiv \mathbf{o}'(n', m') = \begin{bmatrix} \mathbf{R}(\mathbf{Asgn})(\mathbf{args}, 1, 1) & \dots & \mathbf{R}(\mathbf{Asgn})(\mathbf{args}, 1, \kappa'_2) \\ \vdots & \ddots & \vdots \\ \mathbf{R}(\mathbf{Asgn})(\mathbf{args}, \kappa'_1, 1) & \dots & \mathbf{R}(\mathbf{Asgn})(\mathbf{args}, \kappa'_1, \kappa'_2) \end{bmatrix}$$

$$\begin{aligned} \mathbf{x}[i_1{:}i_2,\ j_1{:}j_2] = \mathbf{e} &\equiv \mathbf{T}(\mathbf{T}(\mathbf{x})[i_1{:}i_2,\ j_1{:}j_2]) = \mathbf{T}(\mathbf{e}) \\ \mathbf{x}[i', :] = \mathbf{e} &\equiv \mathbf{T}(\mathbf{T}(\mathbf{x})[i'{:}i'+1, 0{:}m']) = \mathbf{T}(\mathbf{e}) \\ \mathbf{x}[:, j'] = \mathbf{e} &\equiv \mathbf{T}(\mathbf{T}(\mathbf{x})[0{:}n', j'{:}j'+1]) = \mathbf{T}(\mathbf{e}) \\ \mathbf{x}[i', j'] = \mathbf{e} &\equiv \mathbf{T}(\mathbf{T}(\mathbf{x})[i'{:}i'+1, j'{:}j'+1]) = \mathbf{T}(\mathbf{e}) \end{aligned}$$

The **Asgn** operator updates the entries of $\mathbf{A}'$ in block i, j which intersect the selection operator $[i_1{:}i_2,\ j_1{:}j_2]$. The resulting updated block is returned. If a block has no intersection with the selection operator, then the unchanged object id corresponding to that block is returned.

Operators

We denote by $\mathbf{x}_1, \mathbf{N}_1(n_1, m_1), \mathbf{o}_1(n_1, m_1)$ the operands corresponding to the left-hand side of the @ operation, and by $\mathbf{x}_2, \mathbf{N}_2(n_2, m_2), \mathbf{o}_2(n_2, m_2)$ the right-hand side. The result is denoted

by $\mathbf{N}(n,m)$ and $\mathbf{o}(n,m)$, where $n = n_1$ and $m = m_2$. We use the notation $\mathbf{o}_1(n_1,m_1)_{i,:}$ and $\mathbf{o}_1(n_1,m_1)_{:,j}$ to correspond to the ith "row block" and jth "column block" of $\mathbf{o}_1$, respectively. The remote function $\mathbf{R}(@)(\mathbf{args}_{i,j})$ computes the i,j block of the result $\mathbf{N}(n,m)$, where $\mathbf{args}_{i,j} = (\mathbf{o}_1(n_1,m_1)_{i,:}, \mathbf{o}_2(n_2,m_2)_{:,j}, i, j)$. In short, the $\mathbf{R}(@)$ operator computes the result of the operation $\mathbf{N}_1(n_1,m_1)_{i,:} \mathbin{@} \mathbf{N}_2(n_2,m_2){:},j$, which corresponds to block i,j of $\mathbf{N}(n,m)$.

For binary array operations, $\mathbf{R}(\bullet)(\mathbf{o}^1_{i,j}, \mathbf{o}^1_{i,j}, i, j)$ denotes the parallel application of $\bullet$ on block i,j of $\mathbf{o}_1(n_1,m_1)$ and $\mathbf{o}_2(n_2,m_2)$;

For binary array-scalar operations, $\mathbf{R}(\bullet)(\mathbf{o}^1_{i,j}, \mathbf{o}, i, j)$ denotes the parallel application of $\bullet$ on block i,j of $\mathbf{o}_1(n_1,m_1)$, $\mathbf{o}$ corresponds to the scalar $\mathbf{n}$.

Unary operations, such as $\mathbf{x}.\top$, are applied in parallel in the obvious way and are ranged over with the notation $\mathbf{Unary}$.

$$\mathbf{T}(\mathbf{o}_1(n_1,m_1) \mathbin{@} \mathbf{o}_2(n_2,m_2)) \equiv \begin{bmatrix} \mathbf{R}(@)(\mathbf{args}_{1,1}) & \dots & \mathbf{R}(@)(\mathbf{args}_{1,\kappa_2}) \\ \vdots & \ddots & \vdots \\ \mathbf{R}(@)(\mathbf{args}_{\kappa_1,1}) & \dots & \mathbf{R}(@)(\mathbf{args}_{\kappa_1,\kappa_2}) \end{bmatrix}$$

$$\mathbf{T}(\mathbf{x}_1 \mathbin{@} \mathbf{x}_2) \equiv \mathbf{T}(\mathbf{T}(\mathbf{x}_1) \mathbin{@} \mathbf{T}(\mathbf{x}_2))$$

$$\mathbf{T}(\mathbf{o}_1(n_1,m_1) \bullet \mathbf{o}_2(n_2,m_2)) \equiv \begin{bmatrix} \mathbf{R}(\bullet)(\mathbf{o}_1,\mathbf{o}_2,1,1) & \dots & \mathbf{R}(\bullet)(\mathbf{o}_1,\mathbf{o}_2,1,\kappa_2) \\ \vdots & \ddots & \vdots \\ \mathbf{R}(\bullet)(\mathbf{o}_1,\mathbf{o}_2,\kappa_1,1) & \dots & \mathbf{R}(\bullet)(\mathbf{o}_1,\mathbf{o}_2,\kappa_1,\kappa_2) \end{bmatrix}$$

$$\mathbf{T}(\mathbf{o}_1(n_1,m_1) \bullet \mathbf{o}) \equiv \begin{bmatrix} \mathbf{R}(\bullet)(\mathbf{o}_1,\mathbf{o},1,1) & \dots & \mathbf{R}(\bullet)(\mathbf{o}_1,\mathbf{o},1,\kappa_2) \\ \vdots & \ddots & \vdots \\ \mathbf{R}(\bullet)(\mathbf{o}_1,\mathbf{o},\kappa_1,1) & \dots & \mathbf{R}(\bullet)(\mathbf{o}_1,\mathbf{o},\kappa_1,\kappa_2) \end{bmatrix}$$

$$\mathbf{T}(\mathbf{x}_1 \bullet \mathbf{x}_2) \equiv \mathbf{T}(\mathbf{T}(\mathbf{x}_1) \bullet \mathbf{T}(\mathbf{x}_2))$$

$$\mathbf{T}(\mathbf{Unary}(\mathbf{o}(n,m))) \equiv \begin{bmatrix} \mathbf{R}(\mathbf{Unary})(\mathbf{o}(n,m)_{1,1}) & \dots & \mathbf{R}(\mathbf{Unary})(\mathbf{o}(n,m)_{1,\kappa_2}) \\ \vdots & \ddots & \vdots \\ \mathbf{R}(\mathbf{Unary})(\mathbf{o}(n,m)_{\kappa_1,1}) & \dots & \mathbf{R}(\mathbf{Unary})(\mathbf{o}(n,m)_{\kappa_1,\kappa_2}) \end{bmatrix}$$

$$\mathbf{T}(\mathbf{Unary}(\mathbf{x})) \equiv \mathbf{T}(\mathbf{Unary}(\mathbf{T}(\mathbf{x})))$$

Additional Array and Arithmetic Expressions

$$\mathbf{T}(\mathbf{pow}(\mathbf{x}_1,\mathbf{x}_2)) \equiv \mathbf{R}(\mathbf{pow})(\mathbf{T}(\mathbf{x}_1),\mathbf{T}(\mathbf{x}_2))$$

$$\mathbf{T}(\mathbf{sqrt}(\mathbf{x})) \equiv \mathbf{R}(\mathbf{sqrt})(\mathbf{T}(\mathbf{x}))$$

$$\mathbf{T}(\mathbf{norm}(\mathbf{x}_1,\mathbf{x}_2)) \equiv \mathbf{T}(\mathbf{sqrt}(\mathbf{T}(\mathbf{x}.\top \mathbin{@} \mathbf{x})))$$

$$\mathbf{T}(\mathbf{o}) \equiv \mathbf{o}$$

6.4 Proof Of Correctness

Our proof of correctness for arrays follows trivially from our existing proof of correctness for expressions. Consider the case where block shape and array shape are $(1, 1)$. Our proof of correctness for n-ary operations covers this case. Recall the proof of n-ary operations follows from the proof for arbitrary functions. If an array A has shape equivalent to its block shape (one large block), then its proof of correctness is covered by our proof of correctness for arbitrary functions. Now consider the case for arbitrary array sizes with arbitrary block shape. The expressions that comprise each block in the translated arrays are each covered in our proof of theorem 5.5. The proof is by induction on the translations of operations for 2-dimensional arrays. The proof approach follows the same process as the other proofs of correctness.

We do not provide formal solutions for the n-dimensional array and distributed memory setting, but we do provide the intuition to these extensions in this section.

6.5 Extension to N-Dimensions

The correctness of all but matrix multiplication trivially extends to the n-dimensional setting. We now describe how our translation approach and proof of correctness generalizes to the $\mathbf{tensordot}(\mathbf{A}_1, \mathbf{A}_2, n)$ operation. Consider the output shape of a matrix multiply of two arrays $\mathbf{N}_1(n_1, m_1)$ and $\mathbf{N}_2(n_2, m_2)$. The output shape (n_1, m_2) provides a structure whereby the correct value of each entry $\mathbf{N}(n_1, m_2)_{i,j}$ is a basic sum of products. The same idea applies to the **tensordot** operation: The output structure (shape) of a tensor dot operation is predefined. Consider the output of the **tensordot** operation, an n-dimensional array $\mathbf{N}$. Let $\mathbf{I}$ denote the index tuple which accesses arbitrary entries of an n-dimensional array, so that $\mathbf{v_I} \in \mathbb{R}$ is a value in $\mathbf{N}$. The **tensordot** operation is a straightforward generalization of matrix multiplication. Like matrix multiplication, each entry $\mathbf{v_I}$ is computed by a composition of binary functions. Thus, each entry $\mathbf{o_I}$ in an n-dimensional futures object is a composition of remote functions. The proof argument is the same as the proof for 2-dimensional arrays.

Chapter 7

Conclusion

This thesis shows that, by combining state-of-the-art techniques from distributed memory numerical linear algebra, discrete local search optimization techniques typically employed by numerical optimization algorithms for machine learning, and automatic parallelization of Python programs using Python-based distributed systems APIs, we enable the ease of programming provided by Python tools, as well as the ability to achieve performance competitive with state-of-the-art techniques through application-specific performance tuning.

As with related solutions in this space, our solution is not without its limitations. In particular, centralized control of our system limits the number of partitions possible without introducing control overhead. Thus, NumS is not suitable for fine-grained tasks. For large-scale problems, where number of blocks are on the order of hundreds of thousands, we expect NumS will not perform as well as SLATE or ScaLAPACK.

NumS is also unable to provide a speedup over serially executing solutions for very small arrays. For such applications, we recommend using NumPy and scikit-learn.

The following sections provide some exciting applications of NumS, as well as ways in which we believe NumS can be improved.

7.1 Applications in Climate Science

One exciting application of NumS is in the climate sciences. CMIP6 [19, 15, 18] persisted as block-partitioned ZARR [38] data in the cloud can be read into NumS using a simple set of APIs, bringing the scale and performance provided by NumS to climate scientists.

One such application in the climate sciences is in wildfire danger forecasting. A group of researchers at the National Observatory of Athens develops a method for wildfire forcasting using deep learning [30], but their method is limited to Greece. NumS is being used to scale their method globally. The primary bottleneck is one of sampling, which NumS is able to overcome with recent optimizations to its solution to sub-sampling multi-dimensional arrays on distributed memory. For deep learning models to train effectively using stochastic gradient descent, multi-dimensional data needs to be permuted along a particular axis. We

reformulate this procedure as a *shuffle* operation, a technique used to reorder data in map reduce frameworks. By enabling fast subsampling, we exceed the speed of updating model parameters, which alleviates the bottleneck associated with scaling this work.

7.2 Potential Improvements and Open Problems

Array Programming. Although novel, our solution to automatic partitioning is still relatively basic. One future direction in this area is to *compile* a NumS program, and optimize array partitioning according to the operations performed within the NumS program. Compilation can also be used to enhance scheduling. Beyond partitioning and scheduling, reshape operations can be optimize by leveraging the same kind of shuffle procedure used for array permutation.

Scalable Sparse Tensor Programming. One potential future direction of NumS is to provide support for scalable sparse tensor programming. Sparsity support would enable models such as graph neural networks, sparse GLMs, and collaborative filtering models via sparse matrix factorization. Some interesting problems in this area include efficiently reading tabular data in a balanced manner into a distributed memory sparse array representation, and optimizing certain expressions by mapping them to optimized sparse einsum kernels at runtime.

Second Order Optimization. The Newton optimizer for NumS achieves very high performance compared to all other methods we evaluated. However, it is limited to small feature spaces. With proper implementation of a scalable matrix inversion, this bottleneck could be alleviated. There are known methods for symmetric semi-definite matrices, such as the Hessian matrix. Fusing Hessian inversion with other operations involved in updating parameters could also avoid the memory overhead associated with constructing a large Hessian.

Deep Learning Models. We can expand the set of machine learning models supported by NumS by adding support for automatic differentiation, or updating model parameters directly using a distributed memory solution to backpropagation. NumS also lacks the necessary operations (stencil and convolution operations) to support image classification.

Distributed Systems. NumS is based on immutable objects, which prevents it from matching NumPy's view semantics. One area of improvement for the concurrently executing language of futures is mutability. The centralized control bottleneck which we described may also be solved within Ray. Ray actors may potentially be used as a mechanism for control replication and partitioning.

Bibliography

[1] Martin Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[2] *Accelerate Fast Math with Intel® oneAPI Math Kernel Library*. en. URL: `https://www.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html` (visited on 05/05/2021).

[3] Claire Adam-Bourdarios et al. "The Higgs Boson Machine Learning Challenge". In: *Proceedings of the 2014 International Conference on High-Energy Physics and Machine Learning - Volume 42*. HEPML'14. JMLR.org, 2014, pp. 19–55.

[4] E. Anderson et al. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[5] Austin R Benson, David F Gleich, and James Demmel. "Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures". In: *2013 IEEE international conference on big data*. IEEE. 2013, pp. 264–272.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[7] L Susan Blackford et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[8] L Susan Blackford et al. *ScaLAPACK users' guide*. Vol. 4. Siam, 1997.

[9] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.1.55. 2018. URL: `http://github.com/google/jax`.

[10] *Breeze: A numerical processing library for Scala*. 2017. URL: `https://github.com/scalanlp/breeze`.

[11] Tianqi Chen et al. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint arXiv:1512.01274* (2015).

[12] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. "The MPI Message Passing Interface Standard". In: *Programming Environments for Massively Parallel Distributed Systems*. Ed. by Karsten M. Decker and René M. Rehmann. Basel: Birkhäuser Basel, 1994, pp. 213–218. ISBN: 978-3-0348-8534-8.

[13] Paul G. Constantine and David F. Gleich. "Tall and Skinny QR Factorizations in MapReduce Architectures". In: *Proceedings of the Second International Workshop on MapReduce and Its Applications*. MapReduce '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 43–50. ISBN: 9781450307000. DOI: 10.1145/1996092.1996103. URL: https://doi.org/10.1145/1996092.1996103.

[14] Huseyin Melih Elibol et al. "Cross-corpora unsupervised learning of trajectories in autism spectrum disorders". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 4597–4634.

[15] V. Eyring et al. "Overview of the Coupled Model Intercomparison Project Phase 6 (CMIP6) experimental design and organization". In: *Geoscientific Model Development* 9.5 (2016), pp. 1937–1958. DOI: 10.5194/gmd-9-1937-2016. URL: https://gmd.copernicus.org/articles/9/1937/2016/.

[16] Mark Gates et al. "SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356223. URL: https://doi.org/10.1145/3295500.3356223.

[17] Robert A. van de Geijn and Jerrell Watts. *SUMMA: Scalable Universal Matrix Multiplication Algorithm*. Tech. rep. USA, 1995.

[18] C. D. Jones et al. "C4MIP – The Coupled Climate–Carbon Cycle Model Intercomparison Project: experimental protocol for CMIP6". In: *Geoscientific Model Development* 9.8 (2016), pp. 2853–2880. DOI: 10.5194/gmd-9-2853-2016. URL: https://gmd.copernicus.org/articles/9/2853/2016/.

[19] B. Kravitz et al. "The Geoengineering Model Intercomparison Project Phase 6 (GeoMIP6): simulation design and preliminary results". In: *Geoscientific Model Development* 8.10 (2015), pp. 3379–3392. DOI: 10.5194/gmd-8-3379-2015. URL: https://gmd.copernicus.org/articles/8/3379/2015/.

[20] Jennifer Listgarten et al. "Prediction of off-target activities for the end-to-end design of CRISPR guide RNAs". In: *Nature biomedical engineering* 2.1 (2018), pp. 38–47.

[21] Dong C. Liu and Jorge Nocedal. "On the Limited Memory BFGS Method for Large Scale Optimization". In: *Math. Program.* 45.1–3 (Aug. 1989), pp. 503–528. ISSN: 0025-5610.

[22] Xiangrui Meng et al. "MLlib: Machine Learning in Apache Spark". In: *Journal of Machine Learning Research* 17.34 (2016), pp. 1–7. URL: http://jmlr.org/papers/v17/15-237.html.

[23] Philipp Moritz et al. "Ray: A distributed framework for emerging {AI} applications". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 561–577.

[24] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. second. New York, NY, USA: Springer, 2006.

[25] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[26] Ji Hwan Park et al. "Machine learning prediction of incidence of Alzheimer's disease using large-scale administrative health data". In: *npj Digital Medicine* 3.1 (Mar. 2020), p. 46. ISSN: 2398-6352. DOI: 10.1038/s41746-020-0256-0. URL: https://doi.org/10.1038/s41746-020-0256-0.

[27] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *arXiv preprint arXiv:1912.01703* (2019).

[28] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[29] Devin Petersohn et al. "Towards Scalable Dataframe Systems". In: *arXiv preprint arXiv:2001.00888* (2020).

[30] Ioannis Prapas et al. *Deep Learning Methods for Daily Wildfire Danger Forecasting*. 2021. DOI: 10.48550/ARXIV.2111.02736. URL: https://arxiv.org/abs/2111.02736.

[31] Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. *Introduction to Tensor Decompositions and their Applications in Machine Learning*. 2017. DOI: 10.48550/ARXIV.1711.10781. URL: https://arxiv.org/abs/1711.10781.

[32] Matthew Rocklin. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: *Proceedings of the 14th Python in Science Conference*. Ed. by Kathryn Huff and James Bergstra. 2015, pp. 130–136.

[33] Anton Rodomanov and Yurii Nesterov. "Rates of superlinear convergence for classical quasi-Newton methods". In: *Mathematical Programming* (Feb. 2021). DOI: 10.1007/s10107-021-01622-5. URL: https://doi.org/10.1007%2Fs10107-021-01622-5.

[34] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 2009.

[35] Vatsal Sharan and Gregory Valiant. *Orthogonalized ALS: A Theoretically Principled Tensor Decomposition Algorithm for Practical Use*. 2017. DOI: 10.48550/ARXIV.1703.01804. URL: https://arxiv.org/abs/1703.01804.

[36] Noam Shazeer et al. "Mesh-TensorFlow: Deep Learning for Supercomputers". In: 2018. URL: https://arxiv.org/pdf/1811.02084.pdf.

[37] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[38] ZARR Team. *ZARR Library*. URL: https://github.com/zarr-developers/zarr-python (visited on 05/13/2022).

[39] C. Tebaldi et al. "Extreme Metrics and Large Ensembles". In: *Earth System Dynamics Discussions* 2021 (2021), pp. 1–59. DOI: 10.5194/esd-2021-53. URL: https://esd.copernicus.org/preprints/esd-2021-53/.

[40] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[41] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0262231697.

[42] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0262231697.

[43] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, p. 10.

[44] Hui Zou and Trevor Hastie. "Regularization and variable selection via the Elastic Net". In: *Journal of the Royal Statistical Society, Series B* 67 (2005), pp. 301–320.