Serverless Linear Algebra

Vaishaal Shankar
UC Berkeley

Karl Krauth
UC Berkeley

Kailas Vodrahalli
Stanford

Qifan Pu
Google

Benjamin Recht
UC Berkeley

Ion Stoica
UC Berkeley

Jonathan Ragan-Kelley
MIT CSAIL

Eric Jonas
University of Chicago

Shivaram Venkataraman
University of Wisconsin-Madison

ABSTRACT

Datacenter disaggregation provides numerous benefits to both the datacenter operator and the application designer. However switching from the server-centric model to a disaggregated model requires developing new programming abstractions that can achieve high performance while benefiting from the greater elasticity. To explore the limits of datacenter disaggregation, we study an application area that near-maximally benefits from current server-centric datacenters: dense linear algebra. We build NumPyWren, a system for linear algebra built on a disaggregated serverless programming model, and LAmbdaPACK, a companion domain-specific language designed for serverless execution of highly parallel linear algebra algorithms. We show that, for a number of linear algebra algorithms such as matrix multiply, singular value decomposition, Cholesky decomposition, and QR decomposition, NumPyWren's performance (completion time) is within a factor of 2 of optimized server-centric MPI implementations, and has up to 15 % greater compute efficiency (total CPU-hours), while providing fault tolerance.

ACM Reference Format:
Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless Linear Algebra. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421287

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10.
https://doi.org/10.1145/3419111.3421287

1 INTRODUCTION

As cloud providers push for datacenter disaggregation [18], we see a shift in distributed computing towards greater elasticity. Datacenter disaggregation provides benefits to both the datacenter operator and application designer. By decoupling resources (CPU, RAM, SSD), datacenter operators can perform efficient bin-packing and maintain high utilization regardless of application logic (e.g., an application using all the cores on a machine & using only 5% of RAM). Similarly the application designer has the flexibility to provision and deprovision resources *on demand* during application runtime (e.g., asking for many cores only during an embarassingly parallel stage of the application). Furthermore, decoupling resources allows each resource technology to evolve independently by side-stepping constraints imposed by current server-centric systems (e.g., the memory-capacity wall making CPU-memory co-location unsustainable) [18, 41]. Current distributed programming abstractions such as MPI and MapReduce rely on the tightly integrated resources in a collection of individual servers. Thus, in order to write applications for a disaggrated datacenter, the datacenter operator must expose a new programming abstraction.

Serverless computing (e.g., AWS Lambda, Google Cloud Functions, Azure Functions) is a programming model in which the cloud provider manages the servers, and also dynamically manages the allocation of resources. Typically these services expose a time limited, stateless, function-as-a-service API for execution of program logic, and an object storage system to manage program state. For application designers who can cleanly separate program state and logic, serverless platforms provide instant access to large compute capability without the overhead of managing a complex cluster deployment. The design constraints of serverless computing is natural fit for disaggregated datacenters, but imposes severe challenges for many performance critical applications, as they close off traditional avenues for performance optimization such as exploiting data locality or hierarchical communication. As a direct consequence, serverless platforms are

mostly used for simple event-driven applications like IoT automation, front-end web serving, and log processing. Recent work has exploited them for broader applications like parallel data analysis [25] and distributed video encoding [17]. These workloads, however, are either embarrassingly parallel or use simple communication patterns across functions. Whether and how complex communication patterns and workloads can be efficiently fit in a serverless application remains an active research question.

We study an application area that near-maximally benefits from current server-centric datacenters: dense linear algebra. State of the art distributed linear algebra frameworks [8, 11, 15] achieve high performance by exploiting locality, network topology, and the tight integration of resources within single servers. Given a static cluster of resources, frameworks such as ScaLAPACK and LINPACK lead in competitions like Top 500 that measure the peak FLOPs that can be achieved for large scale linear algebra tasks. Thus we ask the question: *Can these algorithms can be successfully ported to a disaggrated datacenter?* That is, can we achieve comparable performance to an MPI-based distributed linear algebra framework, but running under the constraints imposed by the serverless programming model?

We find that disaggregation can in fact provide benefits to linear algebra tasks as these workloads have large dynamic range in memory and computation requirements over the course of their execution. For example, performing Cholesky decomposition [5] on a large matrix—one of the most popular methods for solving systems of linear equations—generates computation phases with oscillating parallelism and decreasing working set size. Further, we find that for many linear algebra operations, regardless of their complex structure, computation time often dominates communication for large problem sizes (e.g., $O(n^3)$ compute and $O(n^2)$ communication for Cholesky decomposition). Thus, with appropriate blocking, it is possible to use high-bandwidth but high-latency distributed *storage* as a substitute for large-scale distributed *memory*.

Based on these insights, we design NumPyWren, a system for linear algebra workloads on serverless architectures. NumPyWren executes programs written using LAmbdaPACK, a high level DSL we built that can succinctly express *arbitrary tile-based* linear algebra algorithms. NumPyWren analyzes the data dependencies in LAmbdaPACK and extracts a task graph for parallel execution. NumPyWren then runs parallel computations as stateless functions while storing intermediate state in a distributed object store. One of the main challenges we see is that operating on large matrices at fine granularity can lead to very large task graphs (16M nodes for a 1Mx1M matrix with a block size of 4096). We address this by using ideas from the literature of loop optimization to infer task dependencies on-demand from the

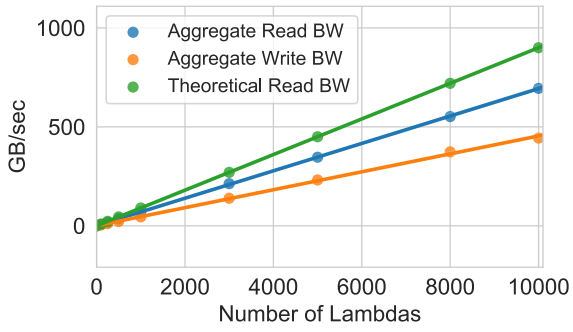


Figure 1: Aggregate S3 Read/Write Bandwidth as a function of number of lambdas. Achieved bandwidth nearly saturates the 25 Gigabit data center network up to at least 10k cores.

original looping program rather than unrolling it, and show that the LAmbdaPACK runtime can scale to large matrix sizes while generating programs of constant size.

We evaluate NumPyWren using a representative set of distributed linear algebra algorithms and compare with MPI-based implementations found in ScaLAPACK. Our experiments show that, for commonly used linear algebra routines (e.g., Cholesky decomposition, matrix multiply, SVD, QR) NumPyWren can come within a factor of 2 in wall-clock time of ScaLAPACK running on a dedicated cluster with the same hardware and number of cores, while using up to 15% less total CPU-hours and providing fault tolerance.

In summary, we make the following contributions:

(1) We provide the first concrete evidence that large scale linear algebra algorithms can be efficiently executed using stateless functions and disaggregated storage.
(2) We design LAmbdaPACK, a domain specific language for linear algebra algorithms that captures fine grained dependencies and can express state of the art communication avoiding linear algebra algorithms in a succinct and readable manner.
(3) We show that NumPyWren can scale to run Cholesky decomposition on a $1M^2$ matrix, and is within a factor of 2 in terms of the completion time compared to ScaLAPACK while using 15% fewer CPU-hours.

2 BACKGROUND

2.1 Serverless Landscape

In the serverless computing model, cloud providers offer the ability to execute functions on demand, hiding cluster configuration and management overheads from end users. In addition to the usability benefits, this model also improves efficiency: the cloud provider can multiplex resources at a

much finer granularity than what is possible with traditional cluster computing, and the user is not charged for idle resources. However, in order to efficiently manage resources, cloud providers place limits on the use of each resource. We next discuss how these constraints affect the design of our system.

Computation. Computation resources offered in serverless platforms are typically restricted to a single CPU core and a short window of computation. For example AWS Lambda provides 900 seconds of compute on a single AVX core with access to up to 3 GB of memory and 512 MB of disk storage. Users can execute a number of parallel functions, and, the aggregate compute performance of these executions scales almost linearly.

The linear scalability in function execution is only useful for embarrassingly parallel computations when there is no communication between the individual workers. Unfortunately, as individual workers are transient and as their start-up times could be staggered, a traditional MPI-like model of peer-to-peer communication will not work in this environment. This encourages us to leverage storage, which can be used as an indirect communication channel between workers.

Storage. Cloud providers offer a number of storage options ranging from key-value stores to relational databases. Some services are not purely elastic in the sense that they require resources to be provisioned beforehand. However distributed object storage systems like Amazon S3 or Google Cloud Storage offer unbounded storage where users are only charged for the amount of data stored. From the study done in [25] we see that AWS Lambda function invocations can read and write to Amazon S3 at an aggregate bandwidth of 800 GB/s or more as shown in Figure 1, roughly saturating the datacenter network bandwidth into each core. This means that we can potentially store intermediate state during computation in a distributed object store and still achieve the same bandwidth as if it were accessed from other nodes' RAM.

Finally, the cost of data storage in an object storage system is often orders of magnitude lower when compared to instance memory. For example on Amazon S3 the price of data storage is $0.04 per TB-hour; in contrast the cheapest large memory instances are priced at $6 per TB-hour. This means that using a storage system could be cheaper if the access pattern does not require instance memory. S3 request are also charged at $4e-6 per read request and $5e-6 per write requests. However in our experiments we found that if the storage granularity is coarse enough the **per request** cost is marginal when compared to the total storage cost.

Control Plane In addition to storage services, cloud providers also offer publish-subscribe services like Amazon SQS or Google Task Queue. These services typically do not support high data access bandwidth but offer consistency guarantees like at least once delivery, and can be used for "control plane" state like a task queue shared between all serverless function invocations. Cloud vendors also offer consistent key-value stores (e.g., DynamoDB) that can be used for storing and manipulating control plane state across serverless function invocations.

2.2 Linear Algebra Algorithms

In this work, we broadly focus on the case of large-scale *dense* linear algebra. This domain has a rich literature of parallel communication-avoiding algorithms and existing high performance implementations [2, 5, 6, 19].

To motivate the design decisions in the subsequent sections we briefly review the communication and computation patterns of a core subroutine in solving a linear system, Cholesky factorization.

Case study: Cholesky factorization is one of the most popular algorithms for solving linear equations, and it is widely used in applications such as matrix inversion, partial differential equations, and Monte Carlo simulations. To illustrate the use of Cholesky decomposition, consider the problem of solving a linear equation $Ax = b$, where A is a symmetric positive definite matrix. One can first perform a Cholesky decomposition of A into two triangular matrices $A = LL^T$ ($O(n^3)$), then solve two relatively simpler equations of $Ly = b$ ($O(n^2)$ via forward substitution) and $L^T x = y$ ($O(n^2)$ via back substitution) to obtain the solution x. From this process, we can see that the decomposition is the most expensive step.

Communication-Avoiding Cholesky [5] is a well-studied routine to compute a Cholesky decomposition. The algorithm divides the matrix into blocks and derives a computation order that minimizes total data transfer. We pick this routine not only because it is one of the most performant, but also because it showcases the structure of computation found in many linear algebra algorithms.

The pseudo-code for communication-avoiding Cholesky decomposition is shown in Algorithm 1. At each step of the outer loop (j), the algorithm first computes Cholesky decomposition of a single block A_{jj} (Fig. 2(a)). This result is used to update the "panel" consisting of the column blocks below A_{ij} (Fig. 2(b)). Finally all blocks to the right of column j are updated by indexing the panel according to their respective positions (Fig. 2(c)). This process is repeated by moving down the diagonal (Fig. 2(d)).

We make two key observations from analyzing the computational structure of Algorithm 1. First, we see that the algorithm exhibits *dynamic parallelism* during execution. The outer loop consists of three distinct steps with different amounts of parallelism, from $O(1)$, $O(K)$ to $O(K^2)$, where K is the enclosing sub-matrix size at each step. In addition, as K decreases at each iteration, overall parallelism available

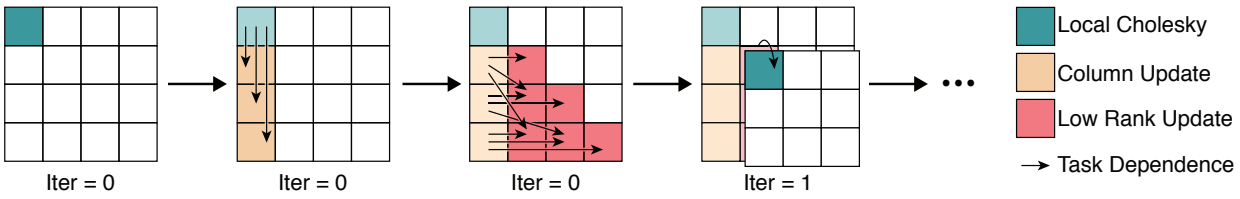


Figure 2: First 4 time steps of parallel Cholesky decomposition: 0) Diagonal block Cholesky decomposition 1) Parallel column update 2) Parallel submatrix update 3) (subsequent) Diagonal block Cholesky decomposition

Algorithm 1 Communication-Avoiding Cholesky [5]

Input:
A - Positive Semidefinite Symmetric Matrix
B - block size
N - number of rows in A
Blocking:
A_{ij} - the ij-th block of A
Output:
L - Cholesky Decomposition of A

1: **for** $j \in \{0 \ldots \lceil \frac{N}{B} \rceil\}$ **do**
2: $L_{jj} \Leftarrow cholesky(A_{jj})$
3: **for all** $i \in \{j+1 \ldots \lceil \frac{N}{B} \rceil\}$ **do in parallel**
4: $L_{ij} \Leftarrow L_{jj}^{-1} A_{ij}$
5: **end for**
6: **for all** $k \in \{j+1 \ldots \lceil \frac{N}{B} \rceil\}$ **do in parallel**
7: **for all** $l \in \{k \ldots \lceil \frac{N}{B} \rceil\}$ **do in parallel**
8: $A_{kl} \Leftarrow A_{kl} - L_{kj}^{T} L_{lj}$
9: **end for**
10: **end for**
11: **end for**

for each iteration decreases from $O(K^2)$ to $O(1)$. Our second observation is that the algorithm has *fine-grained dependencies* between the three steps, both within an iteration and across iterations. For example, A_{kl} in step 3 can be computed as long as L_{kj} and L_{lj} are available (line 8). Similarly, the next iteration can start as soon as $A_{(j+1)(j+1)}$ is updated. Such fine-grained dependencies are hard to exploit in single program multiple data (SPMD) or bulk synchronous parallel (BSP) systems such as MapReduce or Apache Spark, where global synchronous barriers are enforced between steps.

2.3 NumPyWren Overview

We design NumPyWren to target linear algebra workloads that have execution patterns similar to Cholesky decomposition described above. Our goal is to adapt to the amount of parallelism when available and we approach this by decomposing programs into fine-grained execution units that can be run in parallel. To achieve this at scale in a stateless setting, we propose performing dependency analysis in a *decentralized* fashion. We distribute a global dependency graph describing the control flow of the program to every worker. Each worker then locally reasons about its down stream dependencies based on its current position in the global task graph. In the next two sections we will describe LAmbdaPACK the DSL that allows for compact representations of these global dependency graphs, and the NumPyWren execution engine that runs the distributed program.

3 PROGRAMMING MODEL

In this section we present an overview of LAmbdaPACK, our domain specific language for specifying parallel linear algebra algorithms. Classical algorithms for high performance linear algebra are difficult to map directly to a serverless environment as they rely heavily on peer-to-peer communication and exploit locality of data and computation – luxuries absent in a serverless computing cluster. Furthermore, most existing implementations of linear algebra algorithms like ScalaPACK are explicitly designed for stateful HPC clusters.

We thus design LAmbdaPACK to adapt ideas from recent advances in the numerical linear algebra community on expressing algorithms as directed acyclic graph (DAG) based computation [1, 11]. Particularly LAmbdaPACK borrows techniques from Dague [12] a DAG execution framework aimed at HPC environments, though we differ in our analysis methods and target computational platform. Further unlike Dague, LAmbdaPACK does not require the user to pre-specify a DAG of linear algebra kernels, but rather infers the program DAG from an imperative program. By allowing the algorithm to be specified as an imperative program we gain the ability to express intricate communication avoiding algorithms such as those found in [14] that often require hundreds of lines of MPI code, in a succinct and readable manner. All algorithms we implemented in LAmbdaPACK were less than 40 lines of code, and this was often due to the high complexity of the underlying algorithm.

We design LAmbdaPACK to allow users to succinctly express *tiled* linear algebra algorithms. These routines express their computations as operations on matrix *tiles*, small submatrices that can fit in local memory. The main distinction between tiled algorithms and the classical algorithms found in libraries like ScaLAPACK is that the algorithm itself is agnostic to machine layout, connectivity, etc., and only defines

a computational graph on the block indices of the matrices. This uniform, machine independent abstraction for defining complex algorithms allows us to adapt most standard linear algebra routines to a stateless execution engine.

3.1 Language Design

LAmbdaPACK programs are simple imperative routines which produce and consume tiled matrices. These programs can perform basic arithmetic and logical operations on scalar values. They cannot directly read or write matrix values; instead, all substantive computation is performed by calling native kernels on matrix tiles. Matrix tiles are referenced by index, and the primary role of the LAmbdaPACK program is to sequence kernel calls, and compute the tile indices for each call.

LAmbdaPACK programs include simple for loops and if statements, but there is no recursion, only a single level of function calls, from the LAmbdaPACK routine to kernels. Each matrix tile index can be written to only once, a common design principle in many functional languages [1]. Capturing index expressions as symbolic objects in this program is key to the dependence analysis we perform.

These simple primitives are powerful enough to concisely implement algorithms such as Tall Skinny QR (TSQR), LU, Cholesky, and Singular Value decompositions. A description of LAmbdaPACK is shown in Figure 3, and examples of LAmbdaPACK implementations of Cholesky and TSQR are shown in Figure 5.

3.2 Program Analysis

Our program analysis runs in two stages. Naively the uncompressed dag is too large to unroll completely, as the number of nodes in the dag can grows cubically with the input size for algorithm such as Cholesky or QR. Thus, the first stage analyzes a program and extracts a *compressed* DAG of tasks. Each task in the DAG corresponds to an array write, we also extract the kernel computation and array reads that are necessary to execute this task. This process is tractable since each array read has a unique upstream write task. The second analysis stage occurs at runtime when, after a task is executed, the downstream tasks are dynamically discovered. We use information of the current loop variable bindings to query the compressed DAG for downstream tasks. Our compressed DAG representation takes constant space, and supports querying node-edge relationships efficiently. Figure 5 and 4 illustrates an example LAmbdaPACK program and dag respectively.

There are no parallel primitives in LAmbdaPACK, but rather the LAmbdaPACK runtime deduces the underlying

[1] Arbitrary programs can be easily translated into this static single assignment form, but we have found it natural to program directly in this style

```
Uop = Neg| Not| Log| Ceiling| Floor| Log2
Bop = Add| Sub| Mul| Div| Mod| And| Or
Cop = EQ | NE | LT | GT | LE | GE

IdxExpr = IndexExpr(Str matrix_name,
            Expr[] indices)

Expr = BinOp(Bop op, Expr left, Expr right)
       | CmpOp(Cop op, Expr left, Expr right)
       | UnOp(Uop op, Expr e)
       | Ref(Str name)
       | FloatConst(float val)
       | IntConst(int val)

Stmt = KernelCall(Str fn_name,
         IdxExpr[] outputs,
         IdxExpr[] matrix_inputs,
         Expr[] scalar_inputs)
       | Assign(Ref ref, Expr val)
       | Block(Stmt* body)
       | If(Expr cond, Stmt body, Stmt? else)
       | For(Str var, Expr min,
          Expr max, Expr step, Stmt body)
```

Figure 3: A description of the LAmbdaPACK language.

dependency graph by statically analyzing the program. In order to execute a program in parallel, we construct a DAG of kernel calls from the dependency structure induced by the program. Naively converting the program into an executable graph will lead to a *DAG explosion* as the size of the data structure required to represent the program will scale with the size of the input *data*, which can lead to intractable compilation times. Most linear algebra algorithms of interest are $O(N^3)$, and even fast symbolic enumeration of $O(N^3)$ operations at runtime as used by systems like MadLINQ [33] can lead to intractable compute times and overheads for large problems.

In contrast, we borrow and extend techniques from the loop optimization community to convert a LAmbdaPACK program into an *implicit* directed acyclic graph [16].

We represent each node $\mathcal{N}$ in the program's DAG as a tuple of `(line_number, loop_indices)`. With this information any statement in the program's iteration space can be executed. The challenge now lies in deducing the downstream dependencies given a particular node in the DAG. Our approach is to handle dependency analysis at *at runtime*: whenever a storage location is being written to, we determine expressions in $\mathcal{N}$ (all lines, all loop indices) that read from the same storage location.

We solve the problem of determining downstream dependencies for a particular node by modeling the constraints as a system of equations. We assume that the number of lines in a single linear algebra algorithm will be necessarily small.

However, the iteration space of the program can often be far too large to enumerate directly (as mentioned above, this is often as large as $O(n^3)$). Fortunately the pattern of data accesses in linear algebra algorithms is highly structured. Particularly when arrays are indexed solely by *affine functions of loop variables*—that is functions of the form $ai + b$, where i is a loop variable and a and b are constants known at compile time—standard techniques from loop optimization can be employed to efficiently find the dependencies of a particular node. These techniques often involve solving a small system of integer-valued linear equations, where the number of variables in the system depends on the number of nested loop variables in the program.

Example of linear analysis. Consider the Cholesky program in Figure 5. If at runtime a worker is executing line 7 of the program with $i = 0$, $j = 1$ and $k = 1$, to find the downstream dependencies, the analyzer will scan each of the 7 lines of the program and calculate whether there exists a valid set of loop indices such that $S[1, 1, 1]$ can be read from at that point in the program. If so then the tuple of (`line_number`, `loop_indices`) defines the downstream dependency of such task, and becomes a child of the current task. All index expressions in this program contain only affine indices, thus each system can be solved exactly. In this case the only child is the node $(2, \{i : 1, j : 1, k : 1\})$. Note that this procedure only depends on the size of the **program** and not the size of the data being processed.

Nonlinearites and Reductions. Certain common algorithmic patterns—particularly reductions—involve nonlinear loop bounds and array indices. Unlike traditional compilers, since all our analysis occurs *at runtime*, all loop boundaries have been determined. Thus we can solve the system of linear and nonlinear equations by first solving the linear equations and using that solution to solve the remaining nonlinear equations.

Example of nonlinear analysis. Consider the TSQR program in Figure 5. Suppose at runtime a worker is executing line 6 with $level = 0$ and $i = 6$, then we want to solve for the loop variable assignments for $R[i + 2^{level}, level] = R[6, 1]$ (line 7). In this case one of the expressions contains a nonlinear term involving i and $level$ and thus we cannot solve for both variables directly. However we can solve for $level$ easily and obtain the value 1. We then plug in the resulting value into the nonlinear expression to get a linear equation only involving i. Then we can solve for i and arrive at the solution $(6, \{i : 4, level : 1\})$. We note that the for loop structures defined by Figure 5 define a tree reduction with branching factor of 2. Using this approach we can capture the nonlinear array indices induced by tree reductions in algorithms such as Tall-Skinny QR (TSQR), Communication Avoiding QR (CAQR), Tournament Pivoting LU (TSLU), and Bidiagonal Factorization (BDFAC). The full pseudo code for our analysis

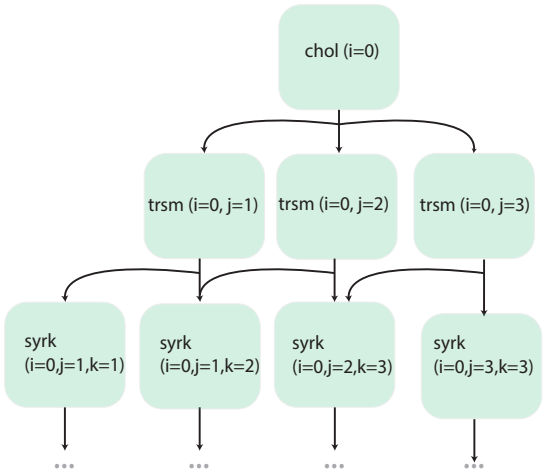


Figure 4: Eexample of a partially unrolled dag for Cholesky decomposition

```
1  def cholesky(O:BigMatrix,S:BigMatrix,N:int):
2    for i in range(0,N)
3      O[i,i] = chol(S[i,i,i])
4      for j in range(i+1,N):
5        O[j,i] = trsm(O[i,i], S[i,j,i])
6        for k in range(i+1,j+1):
7          S[i+1,j,k] = syrk(
8            S[i,j,k], O[j,i], O[k,i])
1  def tsqr(A:BigMatrix, R:BigMatrix, N:Int):
2    for i in range(0,N):
3      R[i, 0] = qr_factor(A[i])
4    for level in range(0,log2(N))
5      for i in range(0,N,2**(level+1)):
6        R[i, level+1] = qr_factor(
7          R[i, level], R[i+2**level, level])
```

Figure 5: LAmbdaPACK code for Cholesky and Tall-Skinny QR decompositions

algorithm can be found in Algorithm 2. The SOLVE call in the algorithm refers to a general purpose symbolic nonlinear equation solver. Our implementation uses the Sympy solver [39]

Implementation. To allow for accessibility and ease of development we embed our language in Python. Since most LAmbdaPACK call into optimized BLAS and LAPACK kernels, the performance penalty of using a high level interpreted language is small.

4 SYSTEM DESIGN

We next present the system architecture of NumPyWren. We begin by introducing the high level components in NumPyWren and trace the execution flow for a computation. Following that we describe techniques to achieve fault tolerance

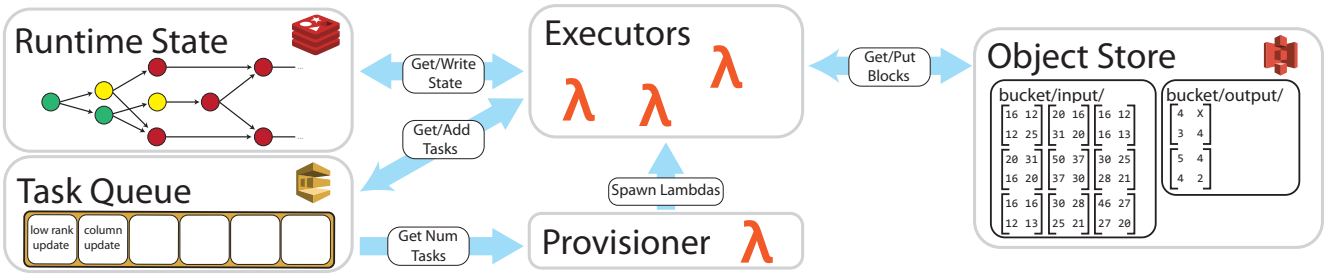


Figure 6: The architecture of the execution framework of NumPyWren showing the runtime state during a 6×6 cholesky decomposition. The first block cholesky instruction has been executed as well as a single column update.

Algorithm 2 LAmbdaPACK Analysis

Input:

$\mathcal{P}$ - The source of a LAmbdaPACK program
A - a concrete array that is written to
idx - the concrete array index of A writen to

Output:

$O = \{\mathcal{N}_0, ..., \mathcal{N}_k\}$ - A concrete set of program nodes that read from $A[idx]$

```
O = {}
for line ∈ P do
    for M ∈ line.read_matrices do
        if M = A then
            S = SOLVE(M.symbolic_idx − idx = 0)
            O = O ∪ S
        end if
    end for
end for
```

and mitigate stragglers. Finally we discuss the dynamic optimizations that are enabled by our design. To fully leverage the elasticity and ease-of-management of the cloud, we build NumPyWren entirely upon existing cloud services while ensuring that we can achieve the performance and fault-tolerance goals for high performance computing workloads. Our system design consists of five major components that are independently scalable: a runtime state store, a task queue, a lightweight global task scheduler, a serverless compute runtime, and a distributed object store. Figure 6 illustrates the components of our system.

The execution proceeds in the following steps:

1. Task Enqueue: The client process enqueues the first task that needs to be executed into the *task queue*. The task queue is a publish-subscribe style queue that contains all the nodes in the DAG whose input dependencies have been met and are ready to execute.

2. Executor Provisioning: The length of the task queue is monitored by a *provisioner* that manages compute resources to match the dynamic parallelism during execution. After the first task is enqueued, the provisioner launches an *executor*, and maintains the number of active executors based on task queue size. As the provisioner's role is only lightweight it can also be executed periodically as a "serverless" cloud function.

3. Task Execution: Executors manage executing and scheduling NumPyWren tasks. Once an executor is ready, it polls the task queue to fetch a task available and executes the instructions encoded in the task. Most tasks involve reading input from and writing output to the *object store*, and executing BLAS/LAPACK functions. The object store is assumed to be a distributed, persistent storage system that supports read-after-write consistency for individual keys. Using a persistent object store with a single static assignment language is helpful in designing our fault tolerance protocol. Executors self terminate when they near the runtime limit imposed by many serverless systems (900s for AWS Lambda). The provisioner is then left in charge of launching new workers if necessary. As long as we choose the coarsness of tasks such that many tasks can be successfully completed in the allocated time interval, we do not see too large of a performance penalty for timely worker termination. Our fault tolerance protocol keeps running programs in a valid state even if workers exceed the runtime limit and are killed mid-execution by the cloud provider.

4. Runtime State Update: Once the task execution is complete and the output has been persisted, the executor updates the task status in the *runtime state store*. The runtime state store tracks the control state of the entire execution and needs to support fast, atomic updates for each task. If a completed task has children that are "ready" to be executed the executor adds the child tasks to the task queue. The atomicity of the state store guarantees every child will be scheduled. We would like to emphasize that we only need transactional semantics within the runtime state store, we do not need the runtime state store and the child task enqueuing to occur atomically. We discuss this further in Section 4. This process of using executors to perform scheduling results in efficient, decentralized, fine grained scheduling of tasks.

Fault tolerance in NumPyWren is much simpler to achieve due to the disaggregation of compute and storage. Because all writes to the object store are made durable, no recomputation is needed after a task is finished. Thus fault tolerance in NumPyWren is reduced to the problem of recovering failed tasks, in contrast to many systems where all un-checkpointed tasks have to be re-executed [33]. In NumPyWren we re-execute failed tasks via a lease mechanism [21], which allows the system to track task status without a scheduler periodically communicating with executors.

Task Lease: In NumPyWren, all the pending and executable tasks are stored in a task queue. We maintain a invariant that a task can only be deleted from the queue once it is completed (i.e., the runtime state store has been updated and the output persisted to the object store). When a task is fetched by a worker, the worker obtains a lease on the task. For the duration of the lease, the task is marked invisible to prevent other workers from fetching the same task. As the lease length is often set to a value that is smaller than task execution time, e.g., 10 seconds, a worker is responsible for renewing the lease and keeping a task invisible when executing the task.

Failure Detection and Recovery: During normal operation, the worker will renew lease of the task using a background thread until the task is completed. If the task completes, the worker deletes the task from the queue. If the worker fails, it can no longer renew the lease and the task will become visible to any available workers. Thus, failure detection happens through lease expiration and recovery latency is determined by lease length.

Garbage Collection: Since NumPyWren stores all intermediate state to a persistent object store, it is imperative we clear the state when it is no longer necessary. However due to the extremely low cost of storing bytes in an object store ($ 0.04 per TB-hour), compared to the compute cost for working on problems with terabytes of intermediate state, it suffices to do garbage collection at the end of the program. We tag all allocations in the object store for a single program execution with a unique id associated with the program. After the program execution terminates, NumPyWren asynchronously cleans the object store by launching a set of parallel serverless tasks to clean up all objects associated with a given program id.

Autoscaling: In contrast to the traditional serverless computing model where each new task is assigned a new container, task scheduling and worker management is decoupled in NumPyWren. This decoupling allows auto-scaling of computing resources for a better cost-performance trade-off. Historically many auto-scaling policies have been explored [36]. In NumPyWren, we adopt a simple auto-scaling heuristic and find it achieves good utilization while keeping job completion time low.

Algorithm	MPI (sec)	NumPyWren (sec)	Slow down
SVD	5.8e4	4.8e4	N/A
QR	9.9e3	1.4e4	1.5x
GEMM	5.0e3	8.1e3	1.6x
Cholesky	1.7e3	2.5e3	1.5x

Table 1: A comparison of MPI vs NumPyWren execution time across algorithms when run on a square matrix with N=256K on a cluster with 512 virtual cores. Median performance of 3 runs reported.

For scaling up, NumPyWren's auto-scaling framework tracks the number of pending tasks and periodically increases the number of running workers to match thetasks with a scaling factor sf. For instance, let $sf = 0.5$, when there are 100 pending tasks, 40 running workers, we launch another $100 * 0.5 - 40 = 10$ workers. For scaling down, each worker shuts down itself if no task has been found for the last $T_{timeout}$ seconds. At equilibrium, the number of running workers is sf times the number of pending tasks. All of the auto-scaling logic is handled by the "provisioner" in Figure 6.

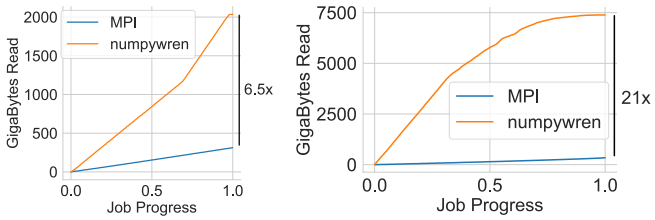
5 EVALUATION

We evaluate NumPyWren on 4 linear algebra algorithms Matrix Multiply (GEMM), QR Decomposition (QR) , Singular Value Decomposition (SVD) [2] and Cholesky Decomposition (Cholesky). All of the algorithms have computational complexity of $O(N^3)$ but differ in their data access patterns. For all four algorithms we compare to state of the art MPI implementations. For Cholesky, GEMM and SVD[3]we use ScaLAPACK, an industrial strength Fortran library designed for high performance, distributed dense linear algebra. For QR decomposition we use an optimized implementation of a communication-avoiding QR decomposition [14] algorithm, specifically the implementation found in [13]. We also do a detailed analysis of the scalability and fault tolerance of our system using the Cholesky decomposition.

5.1 Setup

Implementation. Our implementation of NumPyWren is around 6000 lines of Python code and we build on the Amazon Web Service (AWS) platform. For our runtime state store we use Redis, a key-value store offered by ElasticCache. Though ElasticCache is a provisioned (not "serverless") service we find that using a single instance suffices for all our

[2]Only the reduction to banded form is done in parallel for the SVD



(a) GEMM **(b) QR**

Figure 7: Network traffic for GEMM and QR

Algorithm	MPI (core-secs)	NumPyWren (core-secs)	Resource saving
SVD	2.1e7	6.2e6	3.4x
QR	2.6e6	2.2e6	1.15x
GEMM	1.2e6	1.9e6	0.63x
Cholesky	4.5e5	3.9e5	1.14x

Table 2: A comparison of MPI vs NumPyWren total CPU time (in core-secs) across algorithms run on a 256K size square matrix. Resource saving is defined as $\frac{\text{MPI core-secs}}{\text{NumPyWren core-secs}}$. We compute "active" core-secs for NumPyWren as a 10 second window around a job execution to account for startup & teardown latency.

workloads. We found that we could replace Redis with a managed vendor provided key value store such as DynamoDB, with a slight performance degradation. We used Amazon's simple queue service (SQS) for the task queue, Lambda or EC2. We use Amazon S3 as our remote object store.

Simulating AWS Lambda. Since we cannot easily control the number of concurrent Lambda executions or the type of hardware provided by AWS, for experimental control reasons, we do the bulk of our evaluations by *mimicking* a serverless runtime on EC2 for all experiments that compare to other systems. Our Lambda simulation was based on "standalone mode" in the PyWren framework [25]. PyWren uses a separate SQS queue to simulate the Lambda job queue, and time limited processes to simulate short function invocations. Using SQS induces nondeterminism while controlling for the underlying hardware (AVX, NIC, etc.). Additionally the current pricing of Lambda is 10x more expensive than EC2 spot instances, making the large-scale experiments in this paper unaffordable on Lambda. As shown in Table 3 we found minimal performance differences from running on a simulated serverless environment on EC2 vs AWS Lambda. The simulated environment also lets us modify certain system parameters that are out of the users control in modern serverless environments such as function timeout, which we study in Figure 10.

5.2 System Comparisons

We first present end-to-end comparison of NumPyWren to MPI on four widely used dense linear algebra methods in Table 1. We compare MPI to NumPyWren when operating on square matrices of size 256k (262144) while having access to the exact same hardware (256 physical cores across 8 $r4.16xlarge$ instances).

In Table 1 we see that the constraints imposed by the serverless environment lead to a performance penalty between 1.4x to 1.6x in terms of wall clock time. To understand the overheads, in Figure 7 we compare the number of bytes read over the network by a single machine for two algorithms: GEMM and QR decomposition. We see that the amount of bytes read by NumPyWren is always greater than MPI. This is a direct consequence of each task being stateless, thus all its arguments must be read from a remote object store (Amazon S3 in our experiments). Moreover we see that for QR decomposition and GEMM, MPI reads 21x and 6x less data respectively than NumPyWren. However we note that even though NumPyWren reads more than 21x bytes over the network when compared to MPI, our end to end completion time is only 47% slower.

In Table 2 we compare the total core-seconds used by NumPyWren and MPI. For MPI the core-seconds is the total amount of cores multiplied by the wall clock runtime. For NumPyWren we wish to only account for "active cores" in our core-second calculation, as the free cores can be utilized by other tasks.

We calculate total core seconds by adding a startup latency of γ to the total compute time for each of the kernels executed in the course of computation to account for startup and cooldown of the serverless cores. We calculate γ based on the serverless startup latency measured in [40]. The measured cold-startup latency in is under 10 seconds for all but one of the serverless function providers. For our core second measurement we use $\gamma = 20s$ to get a conservative performance measurement of our system.

For algorithms such as QR and Cholesky that have variable parallelism, while our wall clock time is comparable (within a factor of 2), we find that NumPyWren uses 1.15x less core-seconds. For SVD we see a resource savings of over 3x, but this difference is partially due to difference in SVD algorithms used[4]. However for algorithms that have a fixed amount of parallelism (e.g., GEMM), the excess communication in NumPyWren leads to higher resource consumption.

[4]We used a different SVD algorithm in ScaLAPACK as the baseline (2 step factorization), as the MPI implementation of the algorithm used in NumPyWren (3 step factorization) did not terminate over a period of 2 days, and after personal correspondence with the author we were informed that the implementation was not designed for high performance.

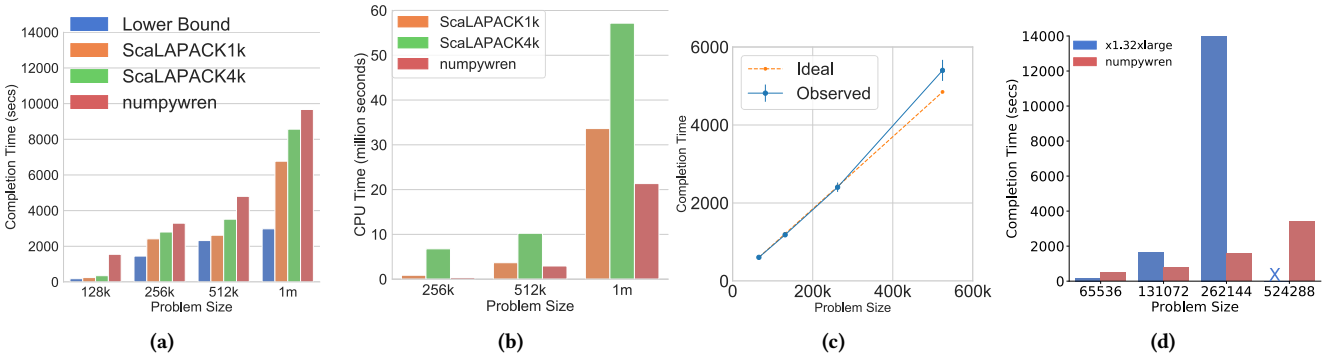


Figure 8: a) Completion time on various problem sizes when NumPyWren is run on same setup as ScaLAPACK. b) Total execution core-seconds for Cholesky when the NumPyWren and ScaLAPACK are optimized for utilization. c) Weak scaling behavior of NumPyWren. Error bars show minimum and maximum time. d) Comparison of NumPyWren with 128 core single node machine running Cholesky decompositions of various sizes

5.3 Scalability

We next look at scalability of NumPyWren and use the Cholesky decomposition study performance and utilization as we scale. For MPI we start with 2 instances for the smallest problem size. We scale the number of instances by 4x for a 2x increase in matrix dimension to ensure that the problem fits in cluster memory. For this experiment we used the c4.8xlarge instance type. Figure 8a shows the completion time when running Cholesky decomposition on each framework, as we increase the problem size. Similar to NumPyWren, MPI has a configurable block size that affects the coarseness of local computation. We report completion time for two different block sizes (4096 and 512) for MPI in Figure 8a. We use a block size of 4096 for NumPyWren. To get an idea of the communication overheads, we also plot a lower bound on completion time based on the clock-rate of the CPUs used.

From the figure we see that NumPyWren is 20 to 25% faster than MPI-4096 and 40 to 60% slower than MPI-512. Compared to MPI-4K, we perform more communication due to the stateless nature of our execution. MPI-512 on the other hand has 64x more parallelism but correspondingly the blocks are only 2MB in size.

Weak Scaling. In Figure 8c we focus on the weak-scaling behavior of NumPyWren. Cholesky decomposition has an algorithmic complexity of $O(N^3)$ and a maximum parallelism of $O(N^2)$, so we increase our core count quadratically from 57 to 1800 as we scale the problem from 64k to 512k. We expect our ideal curve (shown by the green line in Figure 8c) to be a diagonal line. We see that our performance tracks the ideal behavior quite well despite the extra communication overheads incurred.

Utilization. We next look at how resource utilization varies with scale. We compare aggregate core-seconds in Figure 8b for different problem sizes. In this experiment we configured MPI and NumPyWren to minimize total resources consumed. We note that for MPI this is often the minimum number of machines needed to fit the problem in memory. Compared to MPI-512 we find that NumPyWren uses 20% to 33% lower core seconds. Disaggregated storage allows NumPyWren to have the flexibility to run with 4x **less** cores but increases completion time by 3x. In contrast to NumPyWren, MPI frameworks need a minimum resource allocation to fit the problem in memory, thus such a performance/resource consumption trade-off is not possible on MPI.

Single Node Comparisons. Cloud providers now offer many-core high memory machines that can fit large problem instances in memory. We use parallel MKL code to run a sequence of Cholesky decompositions on series of exponentially spaced problem sizes on the largest single node offered by AWS: an x1.32xlarge. This instance type has 2 TB of RAM and the largest problem size we try on this machine is a 256k Cholesky decomposition (working set of 1.2 TB). In Figure 8d, we compare the performance of a Cholesky decomposition running on a single x1.32xlarge with NumPyWren which will use as many cores as necessary to solve the problem. We see that after a problem size of 128k, NumPyWren begins to overtake the single machine performance. This is simply because NumPyWren can summon far more cores than are available to the single node. For the maximally parallel stage of the 512k problem instance, NumPyWren uses up to 10,000 lambdas. The 512k problem instance however does not fit in memory of the x1.32xlarge and thus cannot run successfully. This shows that NumPyWren can provide good performance even for problems that fit on a single machine, while also scaling to larger problem sizes.

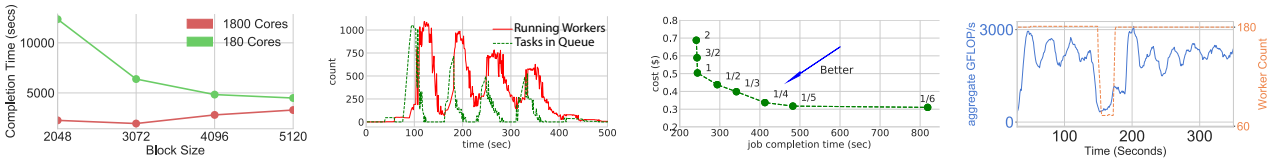


Figure 9: a) Effect of block size on completion time b) Our auto-scaling policy in action. The number of workers increases as the task queue builds up, decreases as the queue is being cleared c) Cost performance trade-off when varying auto-scaling factor (as labeled next to the data points) d) Graceful degradation and recovery of system performance with failure of 80% of workers

Problem Size	Lambda Time (s)	Simulated Lambda Time (s)
64k	414s	420s
128k	820s	840s
256k	2180s	2533s

Table 3: Runtime of a 256k Cholesky on actual AWS Lambda versus a simulated variant on EC2. (both with access to a maximum of 512 virtual cpus)

5.4 System Ablations

We next study certain ablations of NumPyWren to see how our system performs in a variety of conditions.

Fault Recovery. We measure performance of NumPyWren under intermittent failures of the cloud functions. Failures can be common in this setting as cloud functions can get preempted or slow down due to contention. In the experiment in Figure 9d we start with 180 workers and after 150 seconds, we inject failures in 80% of the workers. The disaggregation of resources and the fine grained computation performed by our execution engine leads to a performance penalty linear in the amount of workers that fail. Using the autoscaling technique discussed in **??**, Figure 9d also shows that we can replenish the worker pool to the original size in 20 seconds. We find there is an extra 20 second delay before the computation picks back up due the the startup communication cost of reading program arguments from the object store.

Auto-scaling. Figure 9b shows our auto-scaling policy in action. We ran the first 5000 instructions of a 256k Cholesky solve on AWS Lambda with $sf = 1.0$ (as mentioned in subsection 4). We see that NumPyWren adapts to the dynamic parallelism of the workload. Another important question is how to set the parameters, i.e., scaling factor sf and $T_{timeout}$. We use simple heuristics and empirical experiments to decide these two parameters and leave more rigorous investigation for future work. We set $T_{timeout} = 10s$, which is the average start-up latency of a worker. For sf, we want to make sure that when a new worker (started during scaling up) becomes ready, the task queue should not be completely empty, so the worker can be utilized. Figure 9c shows the trade-off between cost-vs-completion time as we vary sf. From the figure we see that as sf decreases we waste fewer resources but the completion time becomes worse. At higher values of sf the job finishes faster but costs more. Finally we see that there are a range of values of sf (1/4, 1/3, 1/2, 1) that balance the cost and execution time. Outside of the range, either there are always tasks in the queue, or overly-aggressive scaling spawns workers that do not execute any tasks. The balanced range is determined by worker start-up latency, task graph, and execution time.

Blocksize Sensitivity A parameter that is of interest in performance tuning of distributed linear algebra algorithms is the *block size* which defines the coarseness of computation. Blocksize provides the programmer between *arithmetic intensity* and *parallelism*. A larger blocksize will allow each of the tasks to have a greater arithmetic intensity but reduces the total parallelism of the program. The maximal blocksize is also limited by the memory of each of the individual workers.

We evaluate the effect of block size on completion time in Figure 9a. We run the same workload (a $256K$ Cholesky decomposition) at two levels of parallelism, 180 cores and 1800 cores. We see that in the 180 core case, larger block size leads to significantly faster completion time as each task performs more computation and can hide communication overheads. With higher parallelism, we see that the largest block size is slowest as it has the least opportunity to exploit the parallelism available. However, we also see that the smallest block size (2048) is affected by latency overheads in both regimes.

For our main experimental results in Figure 8a and Table 1 we swept the blocksize parameter for both MPI and NumPyWren and found optimal settings of 512 and 4096 for MPI and NumPyWren respectively.

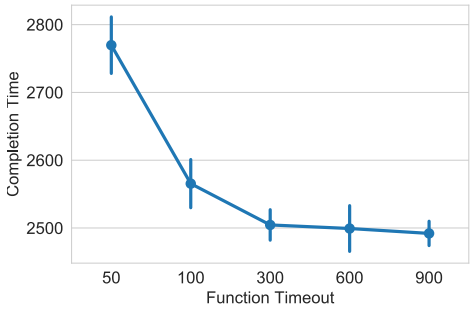


Figure 10: Effect of function time out on end to end completion time of a 256k Cholesky decomposition

5.5 Timeout Sensitvity

Often serverless runtimes impose a limitation of *maximum execution time* for a function. Figure 10 plots the effect of this maximum execution time on performance of a 256k Cholesky Decomposition. After the timeout the underlying serverless framework (PyWren) releases control of the core, and the cloud provider is free to schedule other work at that location. However in figure 10 we see that though performance degrades with a short timeout, we only pay an execution penalty of 20% in completion time for the shortest timeout studied.

6 RELATED WORK

Distributed Linear Algebra Building distributed systems for linear algebra has long been an active area of research. Initially, this was studied in the context of High Performance Computing (HPC), where frameworks like ScaLAPACK [8], DPLASMA [11] and Elemental [32] run on statically-provisioned clusters. However, many users do not have access to HPC clusters. While one can run ScaLAPACK or DPLASMA in the cloud, but they lack fault tolerance and require static cluster provisioning.

On the other hand, with the wide adoption of MapReduce or BSP-style data analytics in the cloud, a number of systems have implemented linear algebra libraries [10, 22, 26, 29, 37]. However, BSP programming models are ill-suited for expressing the fine-grained dependencies in linear algebra algorithms, and imposing global synchronous barriers often greatly slows down a job. As a result, none of these systems [10, 22, 26, 29] have an implementation of distributed Cholesky decomposition that can compare with NumPyWren or ScaLAPACK.

Dataflow-based linear algebra libraries, such as MadLINQ [33] and Dask, support fine grained dependencies. We could not find an opensource implementation of MadLINQ, and could not get Dask to complete for large problem instances.

SystemML [10] takes a similar approach to LAmbdaPACK in providing a high level framework for numerical computation, but it targets a BSP backend and focuses on machine learning algorithms as opposed to linear algebra primitives. In Sec. 3 we delve into the differences between DAGUE, a distributed execution engine for linear algebra libraries, and LAmbdaPACK. Execution Templates [28] explores similar ideas to our decentralized DAG execution (Sec. 4), where each worker maintains a full copy of the program and only receives small messages for each task. However, it focuses on eliminating scheduling overheads, which is less relevant to high arithmetic intensity linear algebra workloads.

Halide [34] is a DSL for writing optimized array processing algorithms. It uses dependency analysis techniques similar to LAmbdaPACK, but more restricted for linear algebra algorithms. Halide is designed for dense local computation on a single machine, whereas the main design of LAmbdaPACK is to orchestrate computation on thousands of decentralized cores. The languages are complementary: the local kernels that LAmbdaPACK calls could be optimized Halide kernels.

Serverless Frameworks The paradigm shift to serverless computing has brought new innovations to many traditional applications. One predominant example is SQL processing, which is now offered in a serverless mode by many cloud providers [3, 7, 20, 35]. Serverless general computing platforms (OpenLambda [23], AWS Lambda, Google Cloud Functions, Azure Functions, etc.) have led to new computing frameworks [4, 17, 25]. Even a complex analytics system such as Apache Spark has been ported to run on AWS Lambda [38]. However, none of the previous frameworks deal with complex communication patterns across stateless workers. NumPyWren is, to our knowledge, the first large-scale linear algebra library that runs on a serverless architecture.

Auto-Scaling and Fault Tolerance Efforts that add fault tolerance to ScaLAPACK has so far demonstrated to incur significant performance overhead [9]. For almost all BSP and dataflow systems[24, 30, 31], recomputation is required to restore stateful workers or datasets that have not been checkpointed. MadLINQ [33] also uses dependency tracking to minimize recomputation for its pipelined execution. In contrast, NumPyWren uses a serverless computing model where fault tolerance only requires re-executing failed tasks and no recomputation is required. NumPyWren's failure detection is also different and we use a lease-based mechanism. The problem of auto-scaling cluster size to fit dynamic workload demand has been both studied [27] and deployed by many cloud vendors. However, due to the relatively high start-up latency of virtual machines, its cost-saving capacity has been limited. NumPyWren exploits the elasticity of serverless computing to achieve better cost-performance trade-off.

7 CONCLUSION & FUTURE WORK

NumPyWren is a distributed system for executing large-scale dense linear algebra programs via stateless function executions. It shows that the disaggregated serverless computing model can be used for computationally intensive programs with complex communication routines through analysis of the intermediate LAmbdaPACK language. Furthermore, the elasticity provided by serverless computing allows the system to dynamically adapt to the inherent parallelism of common linear algebra algorithms. As datacenters continue their push towards disaggregation, platforms like NumPyWren open up a fruitful area of research for applications that have long been dominated by traditional HPC.

REFERENCES

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012037.

[2] Michael Anderson, Grey Ballard, James Demmel, and Kurt Keutzer. 2011. Communication-avoiding QR decomposition for GPUs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 48–58.

[3] athena [n.d.]. Amazon Athena. http://aws.amazon.com/athena/.

[4] aws-lambda-mapred [n.d.]. Serverless Reference Architecture: MapReduce. https://github.com/awslabs/lambda-refarch-mapreduce.

[5] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2010. Communication-optimal parallel and sequential Cholesky decomposition. *SIAM Journal on Scientific Computing* 32, 6 (2010), 3495–3523.

[6] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901.

[7] bigquery [n.d.]. Google BigQuery. https://cloud.google.com/bigquery/.

[8] Laura Susan Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. 1996. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Proceedings of ACM/IEEE Conference on Supercomputing*.

[9] Wesley Bland, Peng Du, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. 2012. A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI. In *European Conference on Parallel Processing*. Springer, 477–488.

[10] Matthias Boehm, Arvind C. Surve, Shirish Tatikonda, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, and Prithviraj Sen. 2016. SystemML: declarative machine learning on spark. *Proceedings of the VLDB Endowment* 9 (09 2016), 1425–1436. https://doi.org/10.14778/3007263.3007279

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. 2011. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 1432–1441. https://doi.org/10.1109/IPDPS.2011.299

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. 2012. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Comput.* 38, 1-2 (2012), 37–51.

[13] candmc [n.d.]. Communication Avoiding Numerical Dense Matrix Computations . https://github.com/solomonik/CANDMC.

[14] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing* 34, 1 (2012), A206–A239.

[15] Jack J Dongarra, James R Bunch, Cleve B Moler, and Gilbert W Stewart. 1979. *LINPACK users' guide*. Siam.

[16] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

[17] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads.. In *NSDI*. 363–376.

[18] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation.. In *OSDI*, Vol. 16. 249–264.

[19] Evangelos Georganas, Jorge Gonzalez-Dominguez, Edgar Solomonik, Yili Zheng, Juan Tourino, and Katherine Yelick. 2012. Communication avoiding and overlapping for numerical linear algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 100.

[20] glue [n.d.]. Amazon Glue. https://aws.amazon.com/glue/.

[21] C. Gray and D. Cheriton. 1989. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. 202–210.

[22] R. Gu, Y. Tang, C. Tian, H. Zhou, G. Li, X. Zheng, and Y. Huang. 2017. Improving Execution Concurrency of Large-Scale Matrix Multiplication on Distributed Data-Parallel Platforms. In *IEEE Transactions on Parallel & Distributed Systems*.

[23] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with openLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'16)*.

[24] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS operating systems review* 41, 3 (2007), 59–72.

[25] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 445–451.

[26] mahout [n.d.]. Apache Mahout. https://mahout.apache.org.

[27] Ming Mao and Marty Humphrey. 2011. Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*.

[28] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. 2017. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 513–526.

[29] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *arXiv preprint arXiv:1712.05889* (2017).

[31] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 113–126.

[32] Jack Poulson, Bryan Marker, Robert A Van de Geijn, Jeff R Hammond, and Nichols A Romero. 2013. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)* 39, 2 (2013), 13.

[33] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. 2012. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 197–210.

[34] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[35] redshift [n.d.]. Amazon Redshift Spectrum. https://aws.amazon.com/redshift/spectrum/.

[36] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 500–507.

[37] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. 2010. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *CLOUDCOM*.

[38] sparkonlambda [n.d.]. Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda/.

[39] sympy [n.d.]. A computer algebra system written in pure Python . https://github.com/sympy/sympy.

[40] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 133–146.

[41] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.