numpywren: serverless linear algebra

Vaishaal Shankar
Karl Krauth
Qifan Pu
Eric Jonas
Shivaram Venkataraman
Ion Stoica
Benjamin Recht
Jonathan Ragan-Kelley

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-137
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-137.html

October 22, 2018

Acknowledgement

This research is supported in part by ONR awards N00014-17-1-2191, N00014-17-1-2401, and N00014-18-1-2833, the DARPA Assured Autonomy (FA8750-18-C-0101) and Lagrange (W911NF-16-1-0552) programs, Amazon AWS AI Research Award, NSF CISE Expeditions Award CCF-1730628 and gifts from
Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel,
Microsoft, Scotiabank, Splunk and VMware as well as by NSF grant DGE-1106400.

We would like to thank Horia Mania, Alyssa Morrow and Esther Rolf for helpful comments while writing this paper.

numpywren: Serverless Linear Algebra

Vaishaal Shankar[1], Karl Krauth[1], Qifan Pu[1],
Eric Jonas[1], Shivaram Venkataraman[2], Ion Stoica[1], Benjamin Recht[1], and Jonathan Ragan-Kelley[1]

[1]UC Berkeley
[2]UW Madison

Abstract

Linear algebra operations are widely used in scientific computing and machine learning applications. However, it is challenging for scientists and data analysts to run linear algebra at scales beyond a single machine. Traditional approaches either require access to supercomputing clusters, or impose configuration and cluster management challenges. In this paper we show how the disaggregation of storage and compute resources in so-called "serverless" environments, combined with compute-intensive workload characteristics, can be exploited to achieve elastic scalability and ease of management.

We present numpywren, a system for linear algebra built on a serverless architecture. We also introduce LAmbdaPACK, a domain-specific language designed to implement highly parallel linear algebra algorithms in a serverless setting. We show that, for certain linear algebra algorithms such as matrix multiply, singular value decomposition, and Cholesky decomposition, numpywren's performance (completion time) is within 33% of ScaLAPACK, and its compute efficiency (total CPU-hours) is up to 240% better due to elasticity, while providing an easier to use interface and better fault tolerance. At the same time, we show that the inability of serverless runtimes to exploit locality *across* the cores in a machine fundamentally limits their network efficiency, which limits performance on other algorithms such as QR factorization. This highlights how cloud providers could better support these types of computations through small changes in their infrastructure.

1 Introduction

As cloud providers push for resource consolidation and disaggregation [16], we see a shift in distributed computing towards greater elasticity. One such example is the advent of *serverless computing* (e.g., AWS Lambda, Google Cloud Functions, Azure Functions) which provides users with instant access to large compute capability without the overhead of managing a complex cluster deployment. While serverless platforms were originally intended for event-driven, stateless functions, and come with corresponding constraints (e.g., small memory and short run-time limits per invocation), recent work has exploited them for other applications like parallel data analysis [25] and distributed video encoding [15]. These workloads are a natural fit for serverless computing as they are either embarrassingly parallel or use simple communication patterns across functions. Exactly how complex the communication patterns and workloads can be and still efficiently fit in a stateless framework remains an active research question.

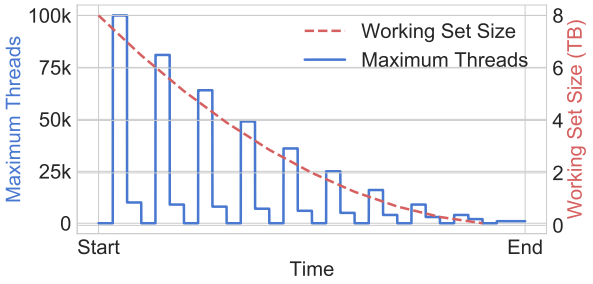


Figure 1: Theoretical profile of available parallelism and required working set size over time in a distributed Cholesky decomposition. Traditional HPC programming models like MPI couple machine parallelism and memory capacity, and require a static allocation for the lifetime of a process. This is inefficient both due to the changing ratio of parallelism to working set, and the sharp decrease in utilization over time.

Linear algebra operations are at the core of many data-intensive applications. Their wide applicability covers both traditional scientific computing problems such as

weather simulation, genome assembly, and fluid dynamics, as well as emerging computing workloads, including distributed optimization [32], robust control [40] and computational imaging [22]. As the data sizes for these problems continue to grow, we see increasing demand for running linear algebra computations at large scale.

Unfortunately, running large-scale distributed linear algebra remains challenging for many scientists and data analysts due to accessibility, provisioning, and cluster management constraints. Traditionally, such linear algebra workloads are run on managed high performance computing (HPC) clusters, access to which is often behind walls of paperwork and long job wait queues. To lower the bar for access, providers such as Amazon Web Services (AWS), Google, and Microsoft Azure now provide HPC clusters in the cloud [5, 19, 6]. While the HPC-in-the-cloud model looks promising, it adds extra configuration complexity, since users have to choose from a complex array of configuration options including cluster size, machine types, and storage types [41].

This extends to many existing systems that run large-scale linear algebra on data parallel systems [34, 21, 12] and that are deployed on a cluster of virtual machines (VMs). This complexity is further exacerbated by the fact that many linear algebra workloads have large dynamic range in memory and computation requirements over the course of their execution. For example, performing Cholesky decomposition [7]—one of the most popular methods for solving systems of linear equations—on a large matrix generates computation phases with oscillating parallelism and decreasing working set size (Figure 1). Provisioning a cluster of any static size will either slow down the job or leave the cluster under-utilized.

Our key insight is 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 and pipelining, we find that it is possible to use high-bandwidth but high-latency distributed *storage* as a substitute for large-scale distributed *memory*.

Based on this idea, we design numpywren, a system for linear algebra on serverless architectures. numpywren runs computations as stateless functions while storing intermediate state in a distributed object store. numpywren executes programs written using LAmbdaPACK, a high level DSL we designed that makes it easy to express state-of-the-art communication avoiding linear algebra algorithms [2] with fine-grained parallelism. Importantly, operating on large matrices at fine granularity can lead to very large task graphs (16M nodes for a matrix with 1M rows and columns, even with a relatively coarse block size of 4K), and the lack of a dedicated driver in the serverles setting would mean each worker would need a copy of the task graph to reason about the dependencies in the program. We address this by using ideas from the literature of loop optimization and show that the LAmbdaPACK runtime can scale to large matrix sizes while generating programs of constant size.

Our evaluation shows that for a number of important linear algebra operations (e.g., Cholesky decomposition, matrix multiply, SVD) numpywren can rival the performance of highly optimized distributed linear algebra libraries running on a dedicated cluster.

We also show that in these favorable cases numpywren is more flexible and can consume 32% fewer CPU-hours, while being fault-tolerant. Compared to fault-tolerant data parallel systems like Dask, we find that numpywren is up to 320% faster and can scale to larger problem sizes. We also show that with LAmbdaPACK we can implicitly represent structured task graphs with millions of nodes in as little as 2 KB.

However, for **all** algorithms stateless function execution imposes large communication overhead. Most distributed linear algebra algorithms heavily exploit locality where an instance with n cores can share a single copy of the data. In serverless systems, every a function has a single core and as these functions could be execute on any machine, we need to send n copies of the data to reach n cores. These limitations affect our performance for certain algorithms, such as QR decomposition. We discuss these limitations and potential solutions in Sec 5.

In summary we make the following contributions:

1. We provide the first concrete evidence that certain large scale linear algebra algorithms can be efficiently executed using purely 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 1Mx1M matrix, and is within 36% of the completion time of ScaLAPACK running on dedicated instances, and can be tuned to use 33% 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 300 seconds of compute on a single AVX/AVX2 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, as one would expect, 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 250 GB/s. Having access to such high bandwidth means that we can potentially store intermediate state during computation in a distributed object store. However such object stores typically have high latency ($\sim$10ms) to access any key meaning we need to design our system to perform coarse-grained access. 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 \$0.03 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.

PubSub. 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 bandwidths but can be used for "control plane" state like a task queue that is shared between all serverless function invocations. Providers often offer consistency guarantees for these services, and most services guarantee at least once delivery.

2.2 Linear Algebra Algorithms

Given the motivating applications, in this work, we broadly focus on the case of large-scale *dense* linear algebra. Algorithms in this regime have a rich literature of parallel communication-avoiding algorithms and existing high performance implementations [2, 7, 8, 17].

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 $A = LL^T$ ($\mathcal{O}(n^3)$), then solve two relatively simpler equations of $Ly = b$ ($\mathcal{O}(n^2)$ via forward substitution) and $L^T x = y$ ($\mathcal{O}(n^2)$ 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 [7] 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

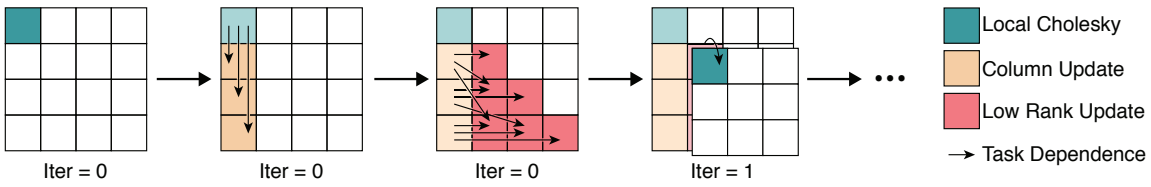


Figure 2: First 4 time steps of parallel Cholesky decomposition: 0) Diagonal block Cholesky decomposition 1) Parallel column update 2) Parallel submatrix update 3) (subsequent) Diagonal block Cholesky decomposition

Algorithm 1 Communication-Avoiding Cholesky [7]

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

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 $\mathcal{O}(1)$, $\mathcal{O}(K)$ to $\mathcal{O}(K^2)$, where K is the enclosing sub-matrix size at each step. In addition, as K decreases at each iteration, overall parallelism available for each iteration decreases from $\mathcal{O}(K^2)$ to $\mathcal{O}(1)$ as shown in Figure 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, 13]. Particularly LAmbdaPACK borrows techniques from Dague [14] a DAG execution framework aimed at HPC environments, though we differ in our analysis methods and target computational platform. 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 to compute the tile indices for each.
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 concrete LAmbdaPACK implementations of Cholesky and TSQR are shown in Figures 4 and 5.

```
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.

3.2 Program Analysis

There are no parallel primitives present in LAmbdaPACK, but rather the LAmbdaPACK runtime deduces the underlying 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 $\mathcal{O}(N^3)$, and even fast symbolic enumeration of $\mathcal{O}(N^3)$ operations at runtime as used by systems like MadLINQ [34] can lead to intractable compute times and overheads for large problems.

[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

In contrast, we borrow and extend techniques from the loop optimization community to convert a LAmbdaPACK program into an *implicit* directed acyclic graph. 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 $\mathcal{O}(n^3)$). 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 4. 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 algorithm can be found in Algorithm 2.

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 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

```
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])
```

Figure 4: Sample LAmbdaPACK of Cholesky Decomposition

```
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: Sample LAmbdaPACK of Tall-Skinny QR Decomposition

Algorithm 2 LAmbdaPACK Analysis

Input:
$\mathscr{P}$ - The source of a LAmbdaPACK program
A - a concrete array that is written to
idx - the concrete array index of A writen to
Output:
$O = \{\mathscr{N}_0, ..., \mathscr{N}_k\}$ - A concrete set of program nodes that read from $A[idx]$

$O = \{\}$
for $line \in \mathscr{P}$ **do**
  for $M \in line.read\_matrices$ **do**
    if $M = A$ **then**
      $S = SOLVE(M.symbolic\_idx - idx = 0)$
      $O = O \cup S$
    end if
  end for
end for

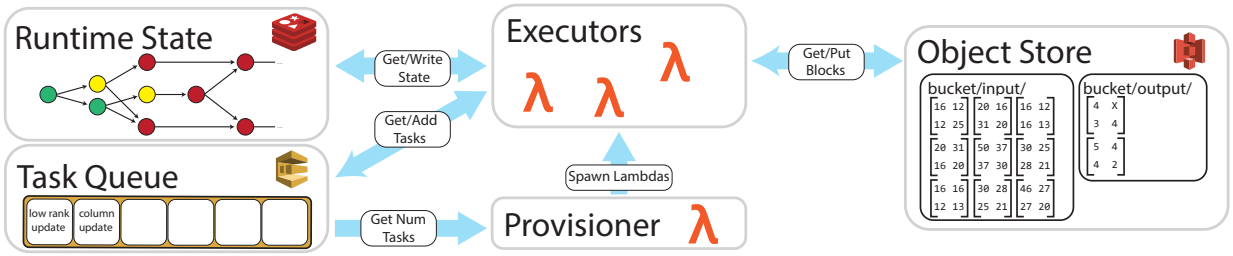


Figure 6: The architecture of the execution framework of numpywren showing the runtime state during a 6×6 cholesky decomposition. The first block cholesky instruction has been executed as well as a single column update.

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*. The exact number of stateless workers that are provisioned depends on the *auto-scaling policy* and we discuss the policy used in Section 4.2. 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 the highest priority 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 (300s 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 4.1. This process of using executors to perform scheduling results in efficient, decentralized, fine grained scheduling of tasks.

4.1 Fault Tolerance

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 [34]. There are many ways to detect and re-run failed tasks. In numpywren we do this via a simple lease mechanism [20], 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 also 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.

Straggler Mitigation: The lease mechanism also enables straggler mitigation by default. If a worker stalls or is slow, it can fail to renew a lease before it expires. In this case, a task can be executed by multiple workers. The runtime limit imposed by serverless system act as a global limit for the amount of times a worker can renew their lease, after which the worker will terminate and the task will be handed to a different worker. Because all tasks are idempotent, this has no effect on the correctness, and can speed up execution. numpywren does not require a task queue to have strong guarantees such as exactly-once, in-order delivery, as long as the queue can deliver each task at least once. Such weak "at-least once delivery" guarantee is provided by most queue services.

4.2 Optimizations

We next describe optimizations that improve performance by fully utilizing resources of a worker.

Pipelining: Every LAmbdaPACK instruction block has three execution phases: read, compute and write. To improve CPU utilization and I/O efficiency, we allow a worker to fetch multiple tasks and run them in parallel. The number of parallel tasks is called *pipeline width*. Ideally, with a single-core worker, we can have at most three tasks running in parallel, each doing read, compute and write respectively. With an appropriately chosen block size, we get best utilization when these three phases take approximately same time. We find pipelining to greatly improve overall utilization, and reduce end-to-end completion time when resources are constrained.

Auto Scaling: 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 [37]. In numpywren, we adopt a simple auto-scaling heuristic and find it achieves good utilization while keeping job completion time low. For scaling up, numpywren's auto-scaling framework tracks the number of pending tasks and periodically increases the number of running workers to match the pending tasks 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. If pipeline width is not 1, numpywren also factors in pipeline width. For scaling down, numpywren uses an expiration policy where 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.

Algorithm	ScaLAPACK (sec)	numpywren (sec)	Slow down
SVD	57,919	77,828	1.33x
QR	3,486	25,108	7.19x
GEMM	2,010	2,670	1.33x
Cholesky	2,417	3,100	1.28x

Table 1: A comparison of ScaLAPACK vs numpywren execution time across algorithms when run on a square matrix with N=256K

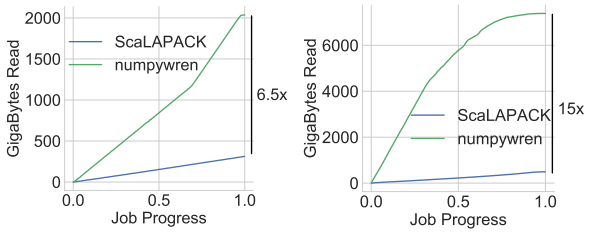
5 Evaluation

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 $\mathcal{O}(N^3)$ but differ in their data access patterns. For all four algorithms we compare to ScalaPACK, an industrial strength Fortran library designed for high performance, distributed dense linear algebra. We then break down the underlying communication overheads imposed by the serverless computing model. We also do a detailed analysis of the scalability and fault tolerance of our system using the Cholesky decomposition. We compare our performance to Dask [36], a python-based fault-tolerant library that supports distributed linear algebra. Finally, we evaluate optimizations in numpywren and how they affect performance and adaptability.

5.1 Setup

Implementation. Our implementation of numpywren is around 6000 lines of Python code and we build on the

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



(a) GEMM (b) QR

Figure 7: Comparing network bytes for GEMM and QR

Algorithm	numpywren (core-secs)	ScaLAPACK (core-secs)	Resource saving
SVD	8.6e6	2.1e7	2.4x
QR	3.8e6	1.3e6	0.31x
GEMM	1.9e6	1.4e6	0.74x
Cholesky	3.4e5	4.3e5	1.26x

Table 2: A comparison of ScaLAPACK vs numpywren total CPU time (in core-secs) across algorithms run on a 256K size square matrix. Resource saving is defined as $\frac{\text{ScaLAPACK core-secs}}{\text{numpywren core-secs}}$.

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 workloads. We used Amazon's simple queue service (SQS) for the task queue, Lambda for function execution, and S3 for object storage. We run ScaLAPACK and Dask on `c4.8xlarge`[3] instances. To obtain a fair comparison with ScaLAPACK and Dask, when comparing to other frameworks we run numpywren on a "emulated" Lambda environment on the same EC2 instances used for other systems [4]. We chose the number of instances for each problem size by finding the minimum number of instances such that ScaLAPACK could complete the algorithm successfully.

5.2 System Comparisons

We first present end-to-end comparison of numpywren to ScaLAPACK on four widely used dense linear algebra methods in Table 1. We compare ScaLAPACK to numpywren when operating on square matrices of size $256K$.

[3] 60 GB of memory, 18 Physical Cores, 10 GBit network link

[4] After imposing all the constraints enforced by AWS Lambda in this emulated environment (memory, disk, runtime limits), we found no performance difference between real Lambda and our emulation.

In table 1 we see that the constraints imposed by the serverless environment lead to a performance penalty between 1.3x to 7x in terms of wall clock time. The difference in the runtime of QR is particularly large, we note that this is primarily due to the high communication penalty our system incurs due to the constraints imposed by the serverless environment.

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 ScaLAPACK. This is a direct consequence of each task being stateless, thus all its arguments must be read from a remote object store. Moreover we see that for QR decomposition and GEMM, ScaLAPACK reads 15x and 6x less data respectively than numpywren. We discuss future work to address this in Section 7.

In Table 2 we compute the total amount of core-seconds used by numpywren and ScaLAPACK. For ScaLAPACK the core-seconds is the total amount of cores multiplied by the wall clock runtime. For numpywren we calculate how many cores were actively working on tasks at any given point in time during computation to calculate the total core-seconds. For algorithms such as SVD and Cholesky that have variable parallelism, while our wall clock time is comparable (within a factor of 2), we find that numpywren uses 1.26x to 2.5x less resources. However for algorithms that have a fixed amount of parallelism such as GEMM, the excess communication performed by numpywren leads to a higher resource consumption.

5.3 Scalability

We next look at scalability of numpywren and use the Cholesky decomposition study performance and utilization as we scale. For ScaLAPACK and Dask, 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. Figure 8a shows the completion time when running Cholesky decomposition on each framework, as we increase the problem size. Similar to numpywren, ScaLAPACK has a configurable block size that affects the coarseness of local computation. We report completion time for two different block sizes (4K and 512) for ScaLAPACK in Figure 8a. We use a block size of 4K 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 10 to

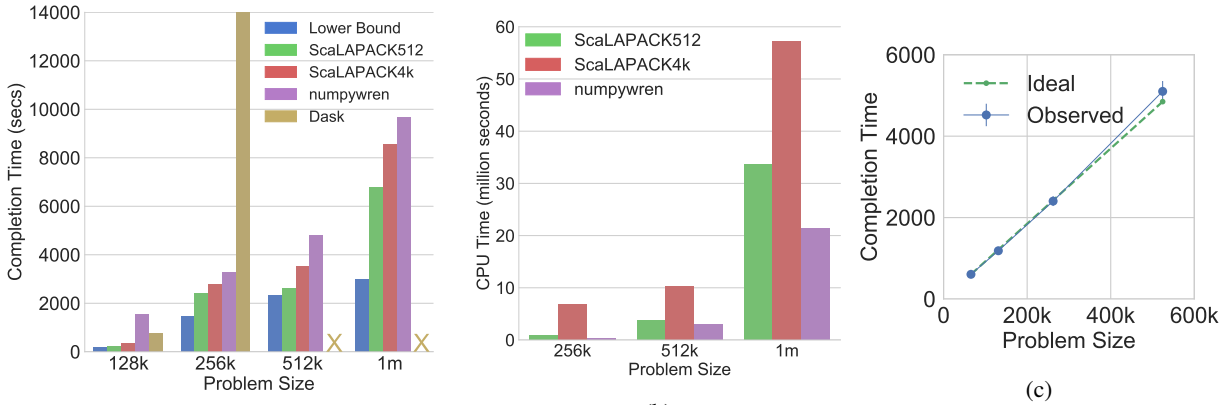


Figure 8: a) Completion time on various problem sizes when numpywren is run on same setup as ScaLAPACK and Dask. b) Total execution core-seconds for Cholesky when the numpywren, ScaLAPACK, and Dask are optimized for utilization. c) Weak scaling behavior of numpywren. Error bars show minimum and maximum time.

15% slower than ScaLAPACK-4K and 36% slower than ScaLAPACK-512. Compared to ScaLAPACK-4K, we perform more communication due to the stateless nature of our execution. ScaLAPACK-512 on the other hand has 64x more parallelism but correspondingly the blocks are only 2MB in size and the small block size does not affect the MPI transfers. While numpywren is 50% slower than Dask at smaller problem sizes, this is because ask execution happens on one machine for small problems avoiding communication. However on large problem sizes, Dask spends a majority of its time serializing and deserializing data and fails to complete execution for the 512k and 1M matrix sizes.

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 65k 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-hours in Figure 8b for different problem sizes. In this experiment we configured all three frameworks, ScaLAPACK, Dask and numpywren to minimize total resources consumed. We note that for ScaLAPACK and Dask this is often the minimum number of machines needed to fit the problem in memory. Compared to ScaLAPACK-512 we find that numpywren uses 20% to 33% lower core hours. Disaggregated storage allows numpywren to have the flexibility to run with 4x **less** cores but increases completion time by 3x. In contrast to numpywren, cluster computation frameworks need a minimum resource allocation to fit the problem in memory, thus such a performance/resource consumption trade-off is not possible on Dask or ScaLAPACK.

N	Full DAG Time (s)	LambdaPack time (s)	DAG Size (# nodes)	Expanded DAG (MB)	Compiled Program (MB)
65k	3.56	0.019	4k	0.6	0.027
128k	4.02	0.027	32k	4.6	0.027
256k	12.57	0.065	256k	36.3	0.027
512k	49.0	0.15	2M	286	0.027
1M	450	0.44	16M	2270	0.027

Table 3: Benefits of LAmbdaPACK analysis in terms of program size and time to enumerate DAG dependencies.

5.4 Optimizations and Adaptability

We next evaluate optimizations in numpywren (Section 4) and how those affect performance and adapatbility.

Pipelining Benefits. We measured the benefits of pipelining using Cholesky decomposition of a matrix of size $256K$. We see that pipelining drastically improves the resource utilization profile as shown in Figure 9a. The *average flop rate* on a 180 core cluster is 40% higher with pipelining enabled.

Fault Recovery. We next measure performance of numpywren under intermittent failures of the cloud functions. Failures can be common in this setting as cloud

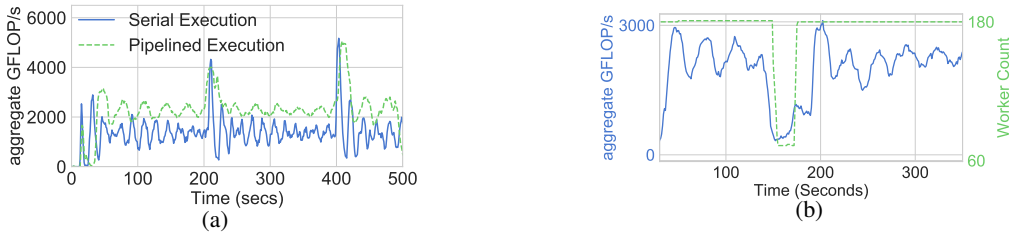


Figure 9: a) Runtime profile with and without pipelining. b) Graceful degredation and recovery of system performance with failure of 80% of workers.

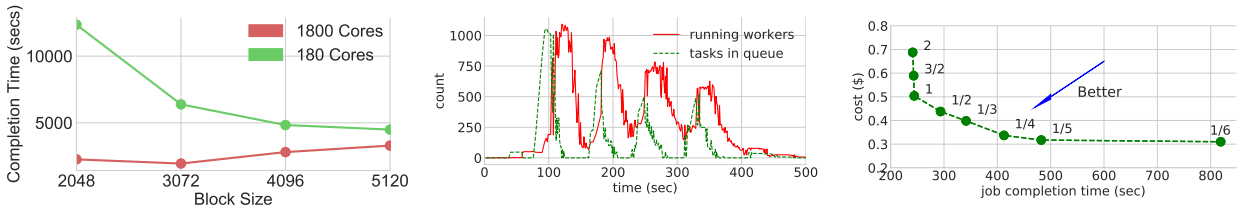


Figure 10: a) Effect of block size on completion time b) Our auto-scaling policy in action. The number of workers increases as the task queue builds up, decreases as the queue is being cleared c) Cost performance trade-off when varying auto-scaling factor (as labeled next to the data points)

functions can get preempted or slow down due to contention. In the experiment in Figure 9b 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 4.2, Figure 9b 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 10b 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 4.2) and pipeline width = 1. 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 10c 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. As described in Section 4.2, the balanced range is determined by worker start-up latency, task graph, execution time and pipeline width.

DAG Compression. In Table 3 we measure the LAmbdaPACK's ability to express large program graphs with constant space, and moreover that we can compile such programs quickly. This is crucial for efficient execution since memory is scarce in the serverless environment, and distributing a large program DAG to each worker can dramatically affect performance. We see that as matrix sizes grow to 1Mx1M the DAG takes over 2 GB of memory LAmbdaPACK lowers this to 2 KB making it feasible to execute on large matrices.

Blocksize A parameter that is of interest in performance tuning of distributed linear algebra algorithms is the *block size* which defines the coarseness of computation. We evaluate the effect of block size on completion time in Figure 10a. 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.

6 Related Work

Distributed Linear Algebra Libraries 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 [10], DPLASMA [13] and Elemental [33] run on a multi-core, shared-memory architecture with high performance network interconnect. However, on-demand access to a HPC cluster can be difficult. While one can run ScaLAPACK or DPLASMA in the cloud, it is undesirable due to their lack of fault tolerance. 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 [12, 28, 26, 21, 38]. However, the BSP-style programming API is ill-suited for expressing the fine-grained dependencies in linear algebra algorithms, and imposing global synchronous barriers often greatly slows down a job. Thus not surprisingly, none of these systems [12, 28, 26, 21] have an efficient implementation of distributed Cholesky decomposition that can compare with numpywren or ScaLAPACK. The only dataflow-based system that supports fine grained dependencies is MadLINQ [34]. numpywren differs from MadLINQ in that it is designed for a serverless architecture and achieves recomputation-free failure (since the previously computed blocks will remain in the object store) recovery by leveraging resource disaggregation, compared to MadLINQ where lost blocks need to be recomputed during recovery. SystemML [12] takes a similar approach to LAmbdaPACK in providing a high level framework for numerical computation, however they target a BSP backend and focus on machine learning algorithms as opposed to linear algebra primitives.

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 [9, 3, 18, 35]. Serverless general computing platforms (OpenLambda [23], AWS Lambda, Google Cloud Functions, Azure Functions, etc.) have led to new computing frameworks [4, 15, 25]. Even a complex analytics system such as Apache Spark has been ported to run on AWS Lambda [39]. 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 [11]. For almost all BSP and dataflow systems[30, 24, 29], recomputation is required to restore stateful workers or datasets that have not been checkpointed. MadLINQ [34] 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 Discussion and Future Work

Collective Communication and Colocation. One of the main drawbacks of the serverless model is the high communication needed due to the lack of locality and efficient broadcast primitives. One way to alleviate this would be to have coarser serverless executions (e.g., 8 cores instead of 1) that process larger portions of the input data. Colocation of lambdas could also achieve similar effects if the colocated lambdas could efficiently share data with each other. Finally, developing services that provide efficient collective communication primitives like broadcast will also help address this problem.

Higher-level libraries. The high level interface in numpywren paves way for easy algorithm design and we believe modern convex optimization solvers such as CVXOPT can use numpywren to scale to much larger problems. Akin to Numba [31] we are also working on automatically translating `numpy` code directly into LAmbdaPACK instructions than can be executed in parallel.

In conclusion, we have presented numpywren, a distributed system for executing large-scale dense linear algebra programs via stateless function executions. We show that the serverless computing model can be used for computationally intensive programs with complex communication routines while providing ease-of-use and

seamless fault tolerance, through analysis of the intermediate LAmbdaPACK language. Furthermore, the elasticity provided by serverless computing allows our 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.

8 Acknowledgements

This research is supported in part by ONR awards N00014-17-1-2191, N00014-17-1-2401, and N00014-18-1-2833, the DARPA Assured Autonomy (FA8750-18-C-0101) and Lagrange (W911NF-16-1-0552) programs, Amazon AWS AI Research Award, NSF CISE Expeditions Award CCF-1730628 and gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware as well as by NSF grant DGE-1106400.

We would like to thank Horia Mania, Alyssa Morrow and Esther Rolf for helpful comments while writing this paper.

References

[1] AGULLO, E., DEMMEL, J., DONGARRA, J., HADRI, B., KURZAK, J., LANGOU, J., LTAIEF, H., LUSZCZEK, P., AND TOMOV, S. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series* (2009), vol. 180, IOP Publishing, p. 012037.

[2] ANDERSON, M., BALLARD, G., DEMMEL, J., AND KEUTZER, K. Communication-avoiding qr decomposition for gpus. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* (2011), IEEE, pp. 48–58.

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

[4] Serverless Reference Architecture: MapReduce. `https://github.com/awslabs/lambda-refarch-mapreduce`.

[5] Amazon AWS High Performance Clusters. `https://aws.amazon.com/hpc`.

[6] Microsoft Azure High Performance Computing. `https://azure.microsoft.com/en-us/solutions/high-performance-computing`.

[7] BALLARD, G., DEMMEL, J., HOLTZ, O., AND SCHWARTZ, O. Communication-optimal parallel and sequential cholesky decomposition. *SIAM Journal on Scientific Computing 32*, 6 (2010), 3495–3523.

[8] BALLARD, G., DEMMEL, J., HOLTZ, O., AND SCHWARTZ, O. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications 32*, 3 (2011), 866–901.

[9] Google BigQuery. `https://cloud.google.com/bigquery/`.

[10] BLACKFORD, L. S., CHOI, J., CLEARY, A., PETITET, A., WHALEY, R. C., DEMMEL, J., DHILLON, I., STANLEY, K., DONGARRA, J., HAMMARLING, S., HENRY, G., AND WALKER, D. Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance. In *Proceedings of ACM/IEEE Conference on Supercomputing* (1996).

[11] BLAND, W., DU, P., BOUTEILLER, A., HERAULT, T., BOSILCA, G., AND DONGARRA, J. A checkpoint-on-failure protocol for algorithm-based recovery in standard mpi. In *European Conference on Parallel Processing* (2012), Springer, pp. 477–488.

[12] BOEHM, M., DUSENBERRY, M. W., ERIKSSON, D., EVFIMIEVSKI, A. V., MANSHADI, F. M., PANSARE, N., REINWALD, B., REISS, F. R., SEN, P., SURVE, A. C., ET AL. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment 9*, 13 (2016), 1425–1436.

[13] BOSILCA, G., BOUTEILLER, A., DANALIS, A., FAVERGE, M., HAIDAR, A., HERAULT, T., KURZAK, J., LANGOU, J., LEMARINIER, P., LTAIEF, H., ET AL. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on* (2011), IEEE, pp. 1432–1441.

[14] BOSILCA, G., BOUTEILLER, A., DANALIS, A., HERAULT, T., LEMARINIER, P., AND DONGARRA, J. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing 38*, 1-2 (2012), 37–51.

[15] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI* (2017), pp. 363–376.

[16] GAO, P. X., NARAYAN, A., KARANDIKAR, S., CARREIRA, J., HAN, S., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Network requirements for resource disaggregation. In *OSDI* (2016), vol. 16, pp. 249–264.

[17] GEORGANAS, E., GONZALEZ-DOMINGUEZ, J., SOLOMONIK, E., ZHENG, Y., TOURINO, J., AND YELICK, K. Communication avoiding and overlapping for numerical linear algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 100.

[18] Amazon Glue. `https://aws.amazon.com/glue/`.

[19] Google Cloud High Performance Computing. `https://cloud.google.com/solutions/architecture/highperformancecomputing`.

[20] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (1989), SOSP '89, pp. 202–210.

[21] GU, R., TANG, Y., TIAN, C., ZHOU, H., LI, G., ZHENG, X., AND HUANG, Y. Improving execution concurrency of large-scale matrix multiplication on distributed data-parallel platforms. In *IEEE Transactions on Parallel & Distributed Systems* (2017).

[22] HEIDE, F., DIAMOND, S., NIESSNER, M., RAGAN-KELLEY, J., HEIDRICH, W., AND WETZSTEIN, G. Proximal: Efficient image optimization using proximal algorithms. *ACM Transactions on Graphics (TOG) 35*, 4 (2016), 84.

[23] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (2016), HotCloud'16.

[24] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS operating systems review 41*, 3 (2007), 59–72.

[25] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 445–451.

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

[27] MAO, M., AND HUMPHREY, M. 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* (2011).

[28] MENG, X., BRADLEY, J., YAVUZ, B., SPARKS, E., VENKATARAMAN, S., LIU, D., FREEMAN, J., TSAI, D., AMDE, M., OWEN, S., XIN, D., XIN, R., FRANKLIN, M. J., ZADEH, R., ZAHARIA, M., AND TALWALKAR, A. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research 17*, 34 (2016), 1–7.

[29] MORITZ, P., NISHIHARA, R., WANG, S., TUMANOV, A., LIAW, R., LIANG, E., PAUL, W., JORDAN, M. I., AND STOICA, I. Ray: A distributed framework for emerging ai applications. *arXiv preprint arXiv:1712.05889* (2017).

[30] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. Ciel: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation* (2011), pp. 113–126.

[31] Numba. `https://numba.pydata.org/`.

[32] PARIKH, N., BOYD, S., ET AL. Proximal algorithms. *Foundations and Trends in Optimization 1*, 3 (2014), 127–239.

[33] POULSON, J., MARKER, B., VAN DE GEIJN, R. A., HAMMOND, J. R., AND ROMERO, N. A. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS) 39*, 2 (2013), 13.

[34] QIAN, Z., CHEN, X., KANG, N., CHEN, M., YU, Y., MOSCIBRODA, T., AND ZHANG, Z. Madlinq: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), ACM, pp. 197–210.

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

[36] ROCKLIN, M. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference* (2015).

[37] ROY, N., DUBEY, A., AND GOKHALE, A. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on* (2011), IEEE, pp. 500–507.

[38] SEO, S., YOON, E. J., KIM, J., JIN, S., KIM, J.-S., AND MAENG, S. Hama: An efficient matrix computation with the mapreduce framework. In *CLOUDCOM* (2010).

[39] Apache Spark on AWS Lambda. `https://www.qubole.com/blog/spark-on-aws-lambda/`.

[40] TU, S., BOCZAR, R., PACKARD, A., AND RECHT, B. Non-asymptotic analysis of robust control from coarse-grained identification. *arXiv preprint arXiv:1707.04791* (2017).

[41] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI* (2016).