Cloudknot: A Python Library to Run your Existing Code on AWS Batch

Adam Richie-Halford[‡*], Ariel Rokem[‡]

https://youtu.be/D9LPzqoZ3f8

Abstract—We introduce Cloudknot, a software library that simplifies cloud-based distributed computing by programmatically executing user-defined functions (UDFs) in AWS Batch. It takes as input a Python function, packages it as a container, creates all the necessary AWS constituent resources to submit jobs, monitors their execution and gathers the results, all from within the Python environment. Cloudknot minimizes the cognitive load of learning a new API by introducing only one new object and using the familiar `map` method. It overcomes limitations of previous similar libraries, such as Pywren, that runs UDFs on AWS Lambda, because most data science workloads exceed the current limits of AWS Lambda on execution time, RAM, and local storage.

Index Terms—Cloud computing, Amazon Web Services, Distributed computing

Introduction

In the quest to minimize time-to-first-result, data scientists are increasingly turning to cloud-based distributed computing with commercial vendors like Amazon Web Services (AWS). Cloud computing platforms have the advantage of linear scalability: users can access limitless computing resources to meet the demands of their computational workloads. At the same time they offer elasticity: resources are provisioned as-needed and can be decomissioned when they are no longer needed. In data-intensive research scenarios in which large computational workloads are coupled with large amounts of data this could, in principle, offer substantial speedups.

But the complexity and learning curve associated with a transition to cloud computing make it inaccessible to beginners. This transition cost has been improving. For example, Dask [Roc15] used to be difficult to run in parallel in a cloud computing environment, but it is now more accessible, thanks in part to tools such as dask-ec2 [Rod17] and kubernetes/helm [Aut18]. Yet despite these improvements, computation in the cloud remains inaccessible to many researchers who have not had previous exposure to distributed computing.

A number of Python libraries have sought to close this gap by allowing users to interact seamlessly with AWS resources from within their Python environment. For example, Cottoncandy allows users to store and access numpy array data on Amazon S3 [NEZH+17]. Pywren [JPV+17] enables users to run their existing Python code on AWS Lambda, providing convenient distributed execution for jobs that fall within the limits of this service[1]. However, these limitations are impractical for many data-oriented workloads, which require more RAM and local storage, longer compute times, and complex dependencies. The AWS Batch service offers a platform for workloads with these requirements. Batch dynamically provisions AWS resources based on the volume and requirements of user-submitted jobs. Instead of provisioning and managing their own batch computing jobs, users specify job constraints, such as the amount of memory required for a single job, and the number of jobs. AWS Batch manages the job distribution to satisfy those constraints. The user can optionally constrain the cost by using Amazon EC2 Spot Instances [AWS18a] and specifying a bid percentage[2].

One of the main advantages of Batch, relative to the provisioning of your own compute instances is that it abstracts away the exact details of the infrastructure that is needed, offering instead relatively straight-forward abstractions:

- a *job*, which is an atomic, independent task to repeat on multiple inputs, encapsulated in a linux executable, a bash script or a Docker container;
- a *job definition*, which connects the job with the compute resources it require;
- a *compute environment*, which defines the configuration of the computational resources needed, such as number of processors, or amount of RAM;
- a *job queue*, where jobs reside until they are run in a compute environment.

While Batch provides useful functional abstractions for processing data in bulk, the user interface provided through the AWS web console still resists automation, requires learning many of the terms that control its execution and does not facilitate scripting and/or reproducibility [AWS18b]. The AWS Python API offers a programming interface that can control the execution of computational tasks in AWS Batch, but it is not currently designed to offer an accessible single point of access to these resources.

Here, we introduce a new Python library with support for Python 2.7 and 3.5+: Cloudknot [RHR18a] [RHR18c], that launches Python functions as jobs on the AWS Batch service, thereby lifting these limitations. Rather than introducing its own

* Corresponding author: richiehalford@gmail.com
‡ University of Washington, Seattle, WA

1. Current limits include a maximum of 300 seconds of execution time, 1.5 GB of RAM, 512 MB of local storage, and no root access.

2. The bid percentage is the maximum price, expressed as a percentage of the on-demand EC2 instance price, with which to bid on unused EC2 capacity.

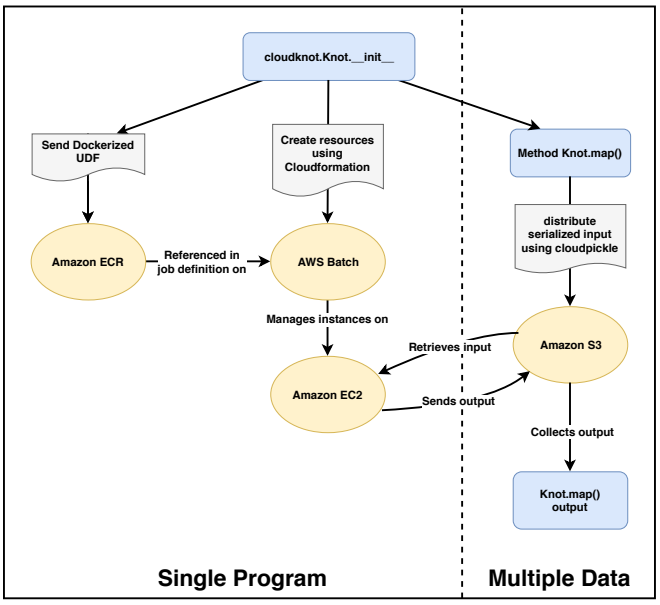

Fig. 1: *Cloudknot's SPMD workflow. The left two columns depict steps Cloudknot takes to create the single program (SP). The right column depicts Cloudknot's management of the multiple data (MD). Blue rounded squares represent components of Cloudknot's user-facing API. Yellow circles represent AWS resources. Grey document shapes represent containers, templates, or data used to communicate with cloud resources.*

set of terms and abstractions, Cloudknot provides a simple abstraction on top of `Executor` objects whose results are returned by concurrent futures. Users of Cloudknot have to familiarize themselves with one new object: the `Knot`. While some of its functionality will initially be new to users of Cloudknot (e.g., the way that resources on AWS are managed), its `map` method should be familiar to most Python users.

The next section discusses Cloudknot's approach to parallelism and the API section describes Cloudknot's user interface. In the Examples section, we demonstrate a few of Cloudknot's use cases, including examples with data ranging from hundreds of GB to several TB. We then summarize the trade-offs between performance and accessibility in the Conclusion.

Design

The primary object in Cloudknot is the `Knot`, which employs the single program, multiple data (SPMD) paradigm to achieve parallelism. In this section, we describe Cloudknot's approach to establishing the single program (SP) and managing the multiple data (MD). `Knot`'s user-facing API and interactions with cloud-based resources are depicted in Figure 1.

Single Program (SP)

The `Knot` object creates the single program on initialization, taking a user-defined function (UDF) as input and wrapping it in a command line interface (CLI), which downloads data from an Amazon Simple Storage Service (S3) bucket specified by an input URL. The UDF is also wrapped in a Python decorator that sends its output back to an S3 bucket. So in total, the resulting command line program downloads input data from S3, executes the UDF, and sends output back to S3. `Knot` then packages the CLI, along with its dependencies, into a Docker container. The container is uploaded into the Amazon Elastic Container Registry (ECR). Cloudknot's use of Docker allows it to handle non-trivial software and data dependencies (see examples below). This is because Docker provides a consistent and isolated environment, allowing complete control over the software dependencies of a particular application, and near-immediate deployment of these dependencies [Boe14].

Separately, `Knot` uses an AWS CloudFormation template to create the AWS resources required by AWS Batch[3]. `Knot` passes the location of the Docker container on AWS ECR to its job definition so that all jobs execute the SP. The user may restrict the compute environment of the `Knot` to only certain instance types (e.g. `c4.2xlarge`) or may choose a specific Amazon Machine Image (AMI) to be loaded on each compute resource. Or, they may simply request a minimum, desired, and maximum number of virtual CPUs and let AWS Batch select and manage the EC2 instances.

`Knot` uses job definition and compute environment defaults that are conservative enough to run most simple jobs, with the goal of minimizing errors due to insufficient resources. The casual user may never need to concern themselves with selecting an instance type or specifying an AMI. Users who want to minimize costs by specifying the minimum sufficient resources or users who need additional resources for intensive jobs can control their jobs' memory requirements, instance types, or AMIs. This might be necessary if the jobs require special hardware (e.g. GPGPU computing) or if the user wants more fine-grained control over which resources are launched.

One of the most complex aspects of AWS is its permissions model[4]. Here, we assume that the user has the permissions needed to run AWS Batch in the console. We also provide users with the minimal necessary permissions in the documentation.

Finally, `Knot` exposes AWS resource tags [AWS18c] to the user, allowing the user to assign metadata key-value pairs to each created resource. This facilitates management of Cloudknot generated resources and allows the user to quickly recognize Cloudknot resources in the AWS console.

Multiple Data (MD)

To operate on the MD, the `Knot.map()` method uses a simple for loop to iterate over the outer-most dimension of the input array and assign each element to a separate AWS Batch job.

3. The required resources are

- AWS Identity and Access Management (IAM) Roles
  - a batch service IAM role to allow AWS Batch to make calls to other AWS services on the user's behalf;
  - an Elastic Container Service (ECS) instance role to be attached to each container instance when it is launched;
  - an Elastic Cloud Compute (EC2) Spot Fleet role to allow Spot Fleet to bid on, launch, and terminate instances if the user chooses to use Spot Fleet instances instead of dedicated EC2 instances;
- an AWS Virtual Private Cloud (VPC) with subnets and a security group;
- an AWS Batch job definition specifying the job to be run;
- an AWS Batch job queue that holds jobs until scheduled into a compute environment;
- and an AWS Batch compute environment, which is a set of compute resources that will be used to run jobs.

4. https://docs.aws.amazon.com/IAM/latest/UserGuide

The Knot serializes each element in the array and sends it to S3, organizing the data in a schema that is internally consistent with the expectations of the CLI. It then launches an AWS Batch array job (or optionally, separate individual Batch jobs) to execute the program over these data. When run, each batch job selects its own input, executes the UDF, and returns its serialized output to S3.

If the instances and S3 bucket are in the same region, then users do not pay for transfer from S3 to the EC2 instances and back. They pay only for transfer out of the data center (i.e. from their local machine to S3 and back). Transfer speed within the data center also outperforms transfer speed between data centers. So it is both less costly and more performant to colocate the Cloudknot S3 bucket with the EC2 instances. Cloudknot includes utility functions to change regions and S3 buckets for this purpose.

In the last step, `Knot.map()` downloads the output from S3 and returns it to the user. Since AWS Batch allows arbitrarily long execution times, `Knot.map()` returns a list of futures for the results, mimicking Python's concurrent futures' `Executor` objects. If the results are too large to fit on the local machine, the user may augment their UDF to write results to S3 or some other remote storage and then simply return the address at which to retrieve the result.

Under the hood, `Knot.map()` creates a `concurrent.futures.ThreadPoolExecutor` instance where each thread intermittently queries S3 for its returned output. The results are encapsulated in `concurrent.futures.Future` objects, allowing asynchronous execution. The user can use `Future` methods such as `done()` and `result()` to test for success or view the results. This also allows attaching callbacks to the results using the `add_done_callback()` method. For example a user may want to perform a local reduction on results generated on AWS Batch.

API

The above interactions with AWS resources are hidden from the user. The advanced and/or curious user can customize the Docker container or CloudFormation template. But for most use cases, the user interacts only with the `Knot` object. This section provides an example calculating the value of π as a pedagogical introduction to the Cloudknot API.

We first import Cloudknot and define the function that we would like to run on AWS Batch. Cloudknot uses the *pipreqs* [Kra17] package to generate the requirements file used to install dependencies in the Docker container on AWS ECR. So all required packages must be imported in the source code of the UDF itself.

```
import cloudknot as ck

def monte_pi_count(n):
    import numpy as np
    x = np.random.rand(n)
    y = np.random.rand(n)
    return np.count_nonzero(x * x + y * y <= 1.0)
```

Next, we create a `Knot` instance and pass the UDF using the `func` argument. The `name` argument affects the names of resources created on AWS. For example, in this case, the created job definition would be named `pi-calc-cloudknot-job-definition`:

```
knot = ck.Knot(name='pi-calc', func=monte_pi_count)
```

We submit jobs with the `Knot.map()` method:

```
import numpy as np  # for np.ones
n_jobs, n_samples = 1000, 100000000
args = np.ones(n_jobs, dtype=np.int32) * n_samples
future = knot.map(args)
```

This will launch an AWS Batch array job with 20 child jobs, one for each element of the input array. Cloudknot can accomodate functions with multiple inputs by passing the `map()` method a sequence of tuples of input arguments and the `starmap=True` argument. For example, if the UDF signature were `def udf(arg0, arg1)`, one could execute `udf` over all combinations of `arg0` in `[1, 2, 3]` and `arg1` in `['a', 'b', 'c']` by calling

```
args = list(itertools.product([1, 2, 3],
                              ['a', 'b', 'c']))
future = knot.map(args, starmap=True)
```

We can then query the result status using `future.done()` and retrieve the results using `future.result()`, which will block until results are returned unless the user passes an optional `timeout` argument. We can also check the status of all the jobs that have been submitted with this `Knot` instance by inspecting the `knot.jobs` property, which returns a list of `cloudknot.BatchJob` instances, each of which has its own `done` property and `result()` method. So in the example above, `future.done()` is equivalent to `knot.jobs[-1].done` and `future.result()` is equivalent to `knot.jobs[-1].result()`. In this way, users have access to AWS Batch job results that they have run in past sessions.

In this pedagogical example, we are estimating π using the Monte Carlo method. `Knot.map()` returns a future for an array of counts of random points that fall within the circle enclosed by the unit square. To get the final estimate of π, we need to sum all the elements of this array and divide by four, a simple use case for `future.add_done_callback()`:

```
PI = 0.0
n_total = n_samples * n_jobs
def pi_from_future(future):
    global PI
    PI = 4.0 * np.sum(future.result()) / n_total

future.add_done_callback(pi_from_future)
```

Lastly, without navigating to the AWS console, we can get a quick summary of the status of all jobs submitted with this `Knot` using

```
>>> knot.view_jobs()
Job ID          Name            Status
-----------------------------------------
fcd2a14b...     pi-calc-0       PENDING
```

Examples

In this section, we will present a few use cases of Cloudknot. We will start with examples that have minimal software and data dependencies, and increase the complexity by adding first data dependencies and subsequently complex software and resource dependencies. These and other examples are available in Jupyter Notebooks in the Cloudknot repository [RHR18b].

Solving differential equations

Simulations executed with Cloudknot do not have to comply with any particular memory or time limitations. This is in contrast

to Pywren's limitations, which stem from the use of the AWS Lambda service. On the other hand, Cloudknot's use of AWS Batch increases the overhead associated with creating AWS resources and uploading a Docker container to ECR. While this infrastructure setup time can be minimized by reusing AWS resources that were created in a previous session, this setup time suits use-cases for which execution time is much greater than the time required to create the necessary resources on AWS.

To demonstrate this, we used Cloudknot and Pywren to find the steady-state solution to the two-dimensional heat equation by the Gauss-Seidel method [BBC+94]. The method chosen is suboptimal, as is the specific implementation of the method, and serves only as a benchmarking tool. In this unrealistic example, we wish to parallelize execution both over a range of different boundary conditions and over a range of grid sizes.

First, we hold the grid size constant at 10 x 10 and parallelize over different temperature constraints on one edge of the simulation grid. We investigate the scaling of job execution time as a function of the size of the argument array. In Figure 2 we show the execution time as a function of n_{args}, the length of the argument array (with both on $\log_2$ scales). We tested scaling using Cloudknot's default parameters and also using custom parameters[5]. Regardless of the `Knot` parameters, Pywren outperformed Cloudknot at all argument array sizes. Indeed, Pywren appears to achieve constant scaling between $2^2 \leq n_{\text{args}} \leq 2^9$, revealing AWS Lambda's capabilities for massively parallel computation. For $n_{\text{args}} > 2^9$, Pywren appears to conform to linear scaling with a constant of roughly 0.25. By contrast, Cloudknot exhibits noisy linear scaling for $n_{\text{args}} \gtrapprox 2^5$, with constants of roughly 2 for the custom configuration and roughly 4 for the default configuration. Precise determination of these scaling constants would require more data for a larger range of argument sizes.

For the data in Figure 3, we still parallelized over only five different temperature constraints, but we did so for increasing grid sizes. Grid sizes beyond 125 x 125 required an individual job execution time that exceeded the AWS Lambda execution limit of 300s. So Pywren was unable to compute on the larger grid sizes. There is a crossover point around 80 x 80 where Cloudknot outperforms Pywren. Before this point, AWS Lambda's fast triggering and continuous scaling surpass the AWS Batch queueing system. Conversely, past this point the compute power of each individual EC2 instance launched by AWS Batch is enough to compensate for the difference in queueing performance.

Taken together, Figures 2 and 3 indicate that if a UDF can be executed within AWS Lambda's five minute execution time and 1.5 GB memory limitations and does not have software and data dependencies that would prohibit using Pywren, it should be parallelized on AWS using Pywren rather than Cloudknot. However, when simulations are too large or complicated to fit well into Pywren's framework, Cloudknot is the appropriate tool to simplify their distributed execution on AWS. Pywren's authors note that the AWS Lambda limits are not fixed and are likely to improve. We agree and note only that EC2 and AWS Batch limitations are likely to improve as well. So long as there exists a computational regime between the two sets of limitations, Cloudknot can offer researchers a simple platform with which to execute their scientific workloads.

5. Default settings are `min_vcpus=0`, `desired_vcpus=8`, and `max_vcpus=256`. Custom settings are `desired_vcpus=2048`, `max_vcpus=4096`, and `min_vcpus=512`. Both default and custom Cloudknot cases were also limited by the EC2 service limits for our region and account, which vary by instance type but never exceeded 200 instances.

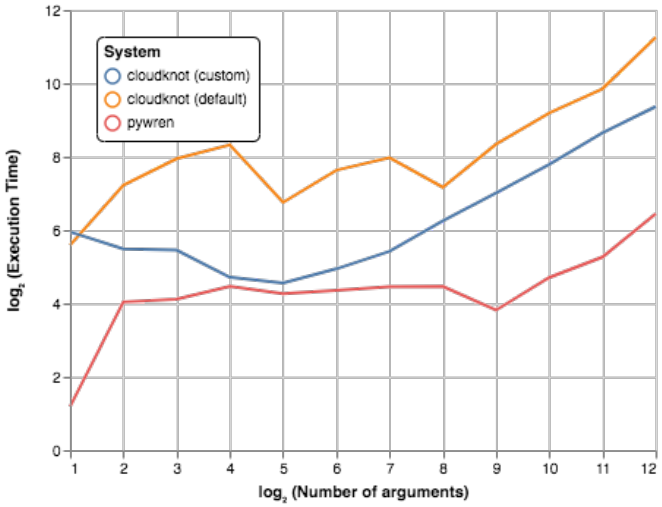

Fig. 2: *Execution time to find solutions of the 2D heat equation for many different temperature constraints on a 10 x 10 grid. We show execution time scaling as a function of the number of constraints for Pywren, the default Cloudknot configuration, and a Cloudknot configuration with more available vCPUs. Pywren outperforms Cloudknot in all cases. We posit that the additional overhead associated with building the Docker image, along with EC2 service limits affected Cloudknot's throughput.*

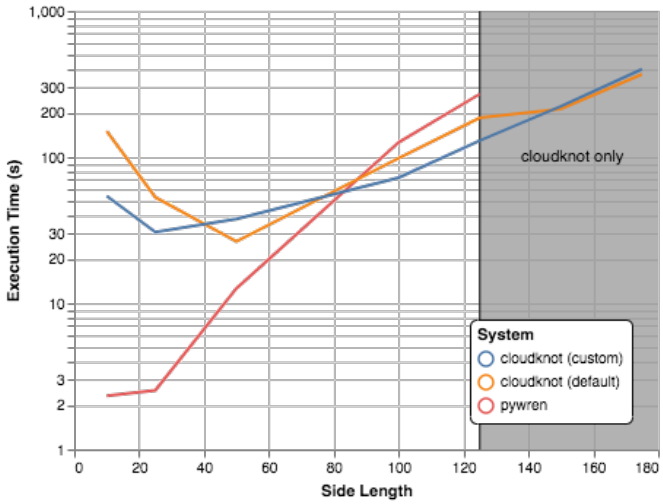

Fig. 3: *Execution time to find five solutions to the 2D heat equation as a function of grid size. Grid sizes above 125 x 125 exceed Pywren's limit on execution time of 300 sec. The cross-over point at around 80 x 80 occurs when it is more beneficial to have the more powerful EC2 instances provided by Cloudknot with AWS Batch than the massively parallel execution provided by Pywren with AWS Lambda.*

Data Dependencies: Analysis of magnetic resonance imaging data

Because Cloudknot is run on the standard AWS infrastructure, it allows specification of complex and large data dependencies. Dependency of individual tasks on data can be addressed by preloading the data into object storage on S3, and then downloading of individual bits of data needed to complete each task into the individual worker machines.

As an example, we implemented a pipeline for analysis of human MRI data. Human MRI data is a good use-case for a system such as Cloudknot because much of the analysis proceeds in a parallel manner. Even for large datasets with multiple subjects, a large part of the analysis is conducted first at the level of each individual brain. Aggregation of information across brains is typically done after many preprocessing and analysis stages at the level of each individual subject.

For example, diffusion MRI (dMRI) is a method that measures the properties of the connections between different regions of the brain. Over the last few decades, this method has been used to establish the role of these connections in many different cognitive and behavioral properties of the human brain, and to delineate the role that the biology of these connections plays in neurological and psychiatric disorders [Wan16]. Because of the interest in these connections, several large consortium efforts for data collection have aggregated large datasets of human dMRI data from multiple different subjects [GSM+16].

In the analysis of dMRI data, the first few steps are done at the individual level. For example, the selection of regions of interest within each image and the denoising and initial modeling of the data can all be completed at the individual level in parallel. In a previous study, we implemented a dMRI analysis pipeline that contained these steps and we used it to compare several Big Data systems as a basis for efficient scientific image processing [MDZ+17]. Here, we reused this pipeline. This allows us to compare the performance of Cloudknot directly against the performance of several alternative systems for distributed computing that were studied in our previous work: Spark [ZCF+10], Myria [HTdAC+14] and Dask [Roc15].

In Cloudknot, we used the reference implementation from this previous study written in Python and using methods from Dipy [GBA+14], which are implemented in Python and Cython. In contrast to the other systems, essentially no changes had to be made to the reference implementation when using Cloudknot, except to download the part of the data required for an individual job from S3 into the individual instances. Parallelization was implemented only at the level of individual subjects, and a naive serial approach was taken at the level of each individual.

We found that with a small number of subjects this reference implementation is significantly slower with Cloudknot compared to the parallelized implementation in these other systems. But the relative advantage of these systems diminshes substantially as the number of subjects grows larger (Figure 4), and the benefits of parallelization across subjects starts to be more substantial. With the largest number of subjects used, Cloudknot processed 25 subjects 10% slower than Spark and Myria; however, it was 25% slower than Dask, the fastest of the tools that we previously benchmarked.

There are two important caveats to this analysis: the first is that the analysis with the other systems was conducted on a cluster with a fixed allocation of 16 nodes (each node was an AWS r3.2xlarge instance with 8 vCPUs). The benchmark code does run faster with more nodes added to the cluster [MDZ+17]. The largest amount of data that was benchmarked was for 25 subjects, corresponding to 105 GB of input data and a maximum of 210 GB of intermediate data. Notably, even for this amount of data, Cloudknot deployed only two instances of the r4.16xlarge type -- each with 64 vCPUs and 488 GB of RAM. In terms of RAM, this is the equivalent of a 16 node cluster of r3.2xlarge instances, but the number of CPUs deployed to the task is about half. In general, users can choose to scale vertically (i.e., larger instance types, with more CPUs) or horizontally (i.e., more machines of smaller instance types) through the `instance_types` keyword argument to `Knot`. Additional scaling can also be reached by expanding the cluster with `min_vcpus`. The second caveat to these results is that that the comparison timing data for the other systems is from early 2017, and these systems may have evolved and improved since.

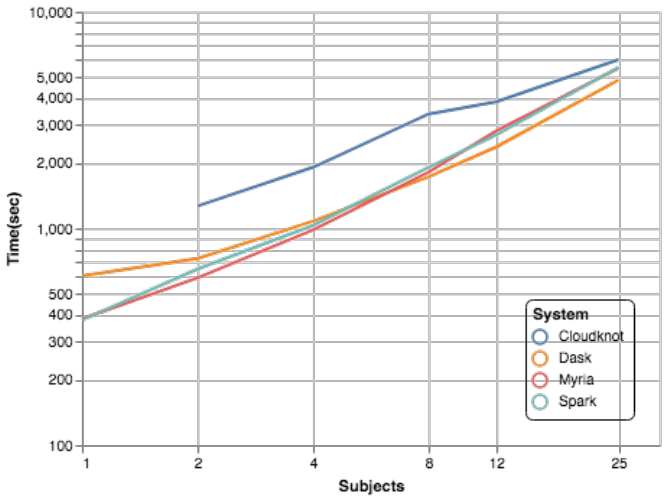

Fig. 4: *MRI analysis pipeline with data requirements. A comparison of Cloudknot performance to other parallel computing systems: Dask, Spark and Myria, based on a previous benchmark [MDZ+17]. Cloudknot is orders of magnitude slower for small amounts of data, but reaches within 10-25 % of these systems' performance for large amounts of data.*

Data and software dependencies: analysis of microscopy data

The MRI example demonstrates the use of a large and rather complex dataset. In addition, Cloudknot can manage complex software dependencies. Researchers in cell biology, molecular engineering and nano-engineering are also increasingly relying on methods that generate large amounts of data and on analysis that requires large amounts of computing power. For example, in experiments that evaluate the mobility of synthetically designed nano-particles in biological tissue [Nan17], [NWS+12], researchers may record movies of microscopic images of the tissue at high spatial and temporal resolution and with a wide field of view, resulting in large amounts of image data, often stored in multiple large files. These collections often reach several TB in size.

To analyze these experiments, researchers rely on software implemented in ImageJ for particle segmentation and tracking, such as TrackMate [TPS+17]. However, when applied to large amounts of data, using TrackMate serially in each experiment can be prohibitively time consuming. One solution is to divide the movies spatially into smaller field of view movies, and analyze them in parallel.

ImageJ and Trackmate are written in Java and can be scripted using Jython. This implies complex software dependencies, because the software requires installation of the ImageJ Jython runtime. Because Cloudknot relies on docker, this installation can be managed using the command line interface (i.e., `wget`). Once a docker image is created that contains the software dependencies for a particular analysis, Python code can be written on top of it to

execute system calls that will run the analysis. This approach was recently implemented in [Cur18].

Additional complexity in this use-case is caused by the volume of data. Because of the data size in this case, a custom AMI had to be created from the AWS Batch AMI, that includes a larger volume (Batch AMI volumes are limited to 30 GB of disk-space).

Conclusion

Cloudknot simplifies cloud-based distributed computing by programmatically executing UDFs in AWS Batch. This lowers the barrier to cloud computing and allows users to launch massive workloads at scale from within their Python environment.

We have demonstrated Cloudknot's ability to execute complex algorithms over vast quantities of data using real-world examples from neuroimaging and microscopy. And we've included analyses that show Cloudknot's performance compared to other distributed computing frameworks. On one hand, scaling charts like the ones in Figures 2, 3, and 4 are important because they show potential users the relative cost in execution time of using Cloudknot compared to other distributed computing platforms.

On the other hand, the timing results in this paper, indeed most benchmark results in general, measure the bare execution time, capturing only partial information about the time that it takes to reach a computational result. This is because all the distributed systems currently available require some amount of systems administration and often incur non-trivial setup time. In addition, most of the existing systems currently require some amount of rewriting of the original code [MDZ+17]. If the amount of time that a user will spend learning a new queueing system or batch processing language, administering this system, and rewriting their code for this system exceeds the time savings due to reduced execution time, then it will be advantageous to accept Cloudknot's suboptimal execution time in order to use its simplified API. Once they gain access to AWS Batch, beginning Cloudknot users simply add an extra import statement, instantiate a `Knot` object, call the `map()` method, and wait for results. And because Cloudknot is built using Docker and the AWS Batch infrastructure, it can accommodate the needs of more advanced users who want to augment their Docker files or specify instance types.

Cloudknot trades runtime performance for development performance and is best used when development speed matters most. Its simple API makes it a viable tool for researchers who want distributed execution of their computational workflow, from within their Python environment, without the steep learning curve of learning a new platform. It may have business applications as well since data scientists performing exploratory analysis would benefit from short development times.

Future Work

Cloudknot can benefit from several enhancements:

- In future developments, we will focus our attention on domain-specific applications (in neuroimaging, for example) and include enhancements and bug-fixes that arise from use in our own research.
- Unlike Dask, Cloudknot does not support computational pipelines that define dependencies between different tasks. Future releases may support job dependencies so that specific jobs can be scheduled to wait for the results of previously submitted jobs.
- Cloudknot could also provide a simple way to connect to EC2 instances to allow in-situ monitoring of running jobs. To do this now, a user must look up an EC2 instance's address in the AWS console and connect to that instance using an SSH client. Future releases may launch this SSH terminal from within the Python session.
- `Knot` uses hard-coded defaults for the configuration of its job definition and compute environment. Future Cloudknot releases could intelligently estimate these defaults based on the UDF and the input data. For example, `Knot` could estimate its resource requirements by executing the UDF on one element of the input array many times using a variety of EC2 instance types. By recording the execution time, memory consumption, and disk usage for each trial, `Knot` could then adopt the configuration parameters of the best[6] run and apply those to the remaining input.

In addition to these capability enhancements, Cloudknot could benefit from performance enhancements designed to address the performance gap with other distributed computing platforms. This might involve prebuilding certain Docker containers or intelligently selecting an AWS region to minimize cost or queueing time. Lastly, we claimed that Cloudknot's simple API likely gives it a gentler learning curve than other distributed computing platforms, but we did not rigorously compare the time investment required to learn how to use Cloudknot, relative to other systems. Future work may seek to fill this gap with a comparative human-computer interaction (HCI) study.

Acknowledgements

This work was funded through a grant from the Gordon & Betty Moore Foundation and the Alfred P. Sloan Foundation to the University of Washington eScience Institute. Thanks to Chad Curtis and Elizabth Nance for the collaboration on the implementation of a Cloudknot pipeline for analysis of microscopy data.

References

[Aut18] The Kubernetes Authors. Helm: The package manager for kubernetes. https://helm.sh/, 2018.

[AWS18a] Inc. Amazon Web Services. Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot, 2018.

[AWS18b] Inc. Amazon Web Services. Getting started with aws batch. https://docs.aws.amazon.com/batch/latest/userguide/Batch_GetStarted.html, 2018.

[AWS18c] Inc. Amazon Web Services. Tagging your amazon ec2 resources. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Using_Tags.html, 2018.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[Boe14] Carl Boettiger. An introduction to docker for reproducible research, with examples from the R environment. *CoRR*, abs/1410.0846, 2014. URL: http://arxiv.org/abs/1410.0846, arXiv:1410.0846.

[Cur18] Chad Curtis. diff_classifier. https://github.com/ccurtis7/diff_classifier, 2018.

[GBA+14] Eleftherios Garyfallidis, Matthew Brett, Bagrat Amirbekian, Ariel Rokem, Stefan Van Der Walt, Maxime Descoteaux, and Ian Nimmo-Smith. Dipy, a library for the analysis of diffusion mri data. *Frontiers in Neuroinformatics*, 8:8, 2014. doi:10.3389/fninf.2014.00008.

6. The "best" configuration could be specified by the user on `Knot` instantiation as either the one which minimizes cost to the user or that which minimizes the wall time required to process the input data.

[GSM+16] Matthew F Glasser, Stephen M Smith, Daniel S Marcus, Jesper L R Andersson, Edward J Auerbach, Timothy E J Behrens, Timothy S Coalson, Michael P Harms, Mark Jenkinson, Steen Moeller, Emma C Robinson, Stamatios N Sotiropoulos, Junqian Xu, Essa Yacoub, Kamil Ugurbil, and David C Van Essen. The human connectome project's neuroimaging approach. *Nat. Neurosci.*, 19(9):1175–1187, August 2016.

[HTdAC+14] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 881–884, New York, NY, USA, 2014. ACM.

[JPV+17] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the Cloud: Distributed Computing for the 99%. *ArXiv e-prints*, February 2017. arXiv:1702.04024.

[Kra17] Vadim Kravcenko. pipreqs. https://github.com/bndr/pipreqs, 2017.

[MDZ+17] Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan, Alvin Cheung, Magdalena Balazinska, Ariel Rokem, Andrew Connolly, Jacob Vanderplas, and Yusra AlSayyad. Comparative evaluation of big-data systems on scientific image analytics workloads. *Proceedings of the VLDB Endowment*, 10(11):1226–1237, 2017.

[Nan17] Elizabeth Nance. Brain-Penetrating nanoparticles for analysis of the brain microenvironment. *Methods Mol. Biol.*, 1570:91–104, 2017.

[NEZH+17] Anwar O Nunez-Elizalde, Tianjiao Zhang, Alexander G Huth, James S Gao, Storm Slivkoff, Mark D Lescroart, Fatma Deniz, Carson McNeil, Robert Gibboni, Sara F Popham, Ariel Rokem, Michael D Oliver, and Jack L Gallant. cottoncandy: scientific python package for easy cloud storage, October 2017. URL: https://doi.org/10.5281/zenodo.1034342, doi:10.5281/zenodo.1034342.

[NWS+12] Elizabeth A Nance, Graeme F Woodworth, Kurt A Sailor, Ting-Yu Shih, Qingguo Xu, Ganesh Swaminathan, Dennis Xiang, Charles Eberhart, and Justin Hanes. A dense poly (ethylene glycol) coating improves penetration of large polymeric nanoparticles within brain tissue. *Sci. Transl. Med.*, 4(149):149ra119–149ra119, 2012.

[RHR18a] Adam Richie-Halford and Ariel Rokem. Cloudknot documentation. https://richford.github.io/cloudknot/index.html, 2018.

[RHR18b] Adam Richie-Halford and Ariel Rokem. Cloudknot examples. https://github.com/richford/cloudknot/tree/master/examples, 2018.

[RHR18c] Adam Richie-Halford and Ariel Rokem. Cloudknot repository. https://github.com/richford/cloudknot, 2018.

[Roc15] M Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference (Scipy 2015)*, 2015.

[Rod17] Daniel Rodriguez. dask-ec2 repository. https://github.com/dask/dask-ec2, 2017.

[TPS+17] Jean-Yves Tinevez, Nick Perry, Johannes Schindelin, Genevieve M Hoopes, Gregory D Reynolds, Emmanuel Laplantine, Sebastian Y Bednarek, Spencer L Shorte, and Kevin W Eliceiri. TrackMate: An open and extensible platform for single-particle tracking. *Methods*, 115:80–90, February 2017.

[Wan16] Brian A Wandell. Clarifying human white matter. *Annu. Rev. Neurosci.*, April 2016.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10. static.usenix.org, 2010.