1 Supplementary Figures

1.1 Supplementary Figure 1

Figure 1: Overview of the assembly process. A total of 1572 fecal samples were collected and sequenced at various timepoints during the first 3 years of 379 individuals lives. These were assembled with MEGAHIT into 68,181,571 contigs. Across all samples, a total of 62,257,853 genes, 1,000,000 of which were unique, were then annotated using Prokka. Only contigs that were over 1000 bases long were used. The mean length of this group was 4278 bases.
1.2 Supplementary Figure 2

Figure 2: Cost comparison between Aether, standard cloud computing, and user-maintained hardware. Total assembly cost was 18% ($471.60) of what it would have been using on-demand instances. We estimated the upfront cost of a server equivalent to those used to analyze the data being $10,000. Given that we used 30 instances of these servers, the total cost of hardware would be $300,000 according to pricing information from Penguin Computing and Dell, not counting system maintenance and depreciation.
1.3 Supplementary Figure 3

![Expected Value Of Worst Case Job Price For Different Bidding Strategies](image)

Figure 3: Plot of the Expected Value of worst case job price for different bidding strategies to showcase the advantage of Aether using a price "lookback" as a constraint for optimization.

2 Supplementary Methods

2.1 General Implementation

Implementation Details Computational resources and monetary costs are mapped to each available instance type at run-time by querying the cloud providers web-based public APIs. To identify the ideal resource selection, we feed these data, along with constraints provided by the user, into our multi-objective optimization procedure. The user-defined set of jobs is subdivided into computational workloads according to the resources available to each node, and distributed across the worker nodes by a central server. In a single nodes workload, jobs are executed in parallel but may complete asynchronously. Upon completion of a job, the replica node notifies the central server, which then schedules another task for the replica. To prevent scheduling errors, we synchronized changes in the primary nodes job ledger, and used at-least-once message delivery. We controlled access to computational resources and accounts with AWS Identity & Access Management (IAM) security groups and Azure Identity and Access Management (IaAM), which their respective providers recommend for authenti-
cation and authorization. Additional details regarding Aethers implementation are available on the project website (http://aether.kosticlab.org).

2.2 Bidding Algorithm

Due to pricing variability, it can be optimal to bid on non-auctioned instances in certain regions. To properly handle this case, we include additional linear constraints for both an instances on-demand and at-auction prices. The solution vector is bounded by the number of currently running instances as well as limits due to provider capacity. Finally, to avoid bidding on instances that will spike in price, the algorithm looks at pricing history and sets a final constraint corresponding to a users maximum tolerable pricing variability. For each run of the bidder, this system of 140 inequalities is converted to slack (standard) form and then solved with the simplex algorithm as implemented in Pythons scipy.linprog library (Figure 1B). (Jones et al., 2001) This naively outputs suggested compute bids as floats; obviously, a fraction of an instance is not a valid bid and generating integer solutions to linear programming problems is NP-hard. However, a true integer linear programming solution is not required, as the constraints still hold if the oor is taken from each bid, provided that preprocessing is done to remove underutilized instance types and those that cannot process a unary compute job. To reach this optimal integer pseudo-solution, the linear programming solver is run recursively such that these non-feasible fractional bids are iteratively removed. Additionally, adhering to the pricing variability constraint is not guaranteed to yield the optimal value, so the simplex algorithm is applied iteratively, setting the pricing variability from zero to the maximum specified value until either the optimal value is found or it is determined that there is no solution to the system. In the event of finding no solution, the user must re-run the program with a higher maximum cost. This approach results in a tractable average case runtime, which yields essentially instant bidding suggestions given the small size of the system being solved.

2.3 Comparisons

2.3.1 Comparison With Other Theoretical Algorithms

While we did mention that there are other bidding strategies available, direct comparisons are difficult as universally every method cited either provides a theoretical algorithm and not an implementation (Zheng et al., Andrzejak et al.) or an implementation with an algorithm that is closed source (i.e. AWS Batch with Spot Fleet Pricing). Aether already incorporates ideas proposed by existing algorithms that do not have implementations that we were able to empirically validate (i.e. incorporation of SLA constrains as in Andrzejak et al.). In the case where there was a closed source implementation (AWS Batch), we were able to make both qualitative and empirical comparisons to Aether.
2.3.2 Steps Needed To Run AWS Batch

1) An AWS Batch user one must first manually set up IAM user permissions, which is a 12 step process.
2) Subsequently, an AWS Batch user needs to create IAM roles for these IAM users, which Amazons most detailed batch instructions do not provide clear instructions for.
3) Subsequently, an AWS Batch user then needs to follow an up to 13 step process to manually create a key pair for the AWS Batch backend to authenticate to provisioned resources that the batch process job will end up using.
4) Subsequently, an AWS Batch user must manually change non-default networking settings to create a VPC (a closed virtual network) where the provisioned AWS Batch jobs will run. This is a 5 step manual process.
5) Subsequently, an AWS Batch user must manually create a security group to allow correct port access to provisioned resources that will be utilized. This is a manual 6 step process.
6) Subsequently, an AWS Batch user must configure their job options for each job they run. This is a manual 3 step process.
7) Subsequently, an AWS Batch user must specify the run time environment on their batch job via parameterizing a virtual container. This is a manual 3 step process. Note that this has nothing to do with how long the job runs but rather run time environment.
8) Subsequently, an AWS Batch user must specify resources that their compute job will need (Aether also asks for this but also asks for anticipated length of time that a job will run for). This is a manual 4 step process.
9) Subsequently, an AWS Batch user must configure their computer environment type. This is a manual 3 step process.
10) Subsequently, an AWS Batch user must configure instances. This is a manual 6 step process.
11) Subsequently, an AWS Batch user must configure networking for the instances they configured in step 10. This is a manual 3 step process.
12) Subsequently, an AWS Batch user must tag their networked instances. This is a manual 3 step process.
13) Finally, an AWS Batch user must manually create their job queue.

2.3.3 Comparing Aether With AWS Batch

The problem of comparing two bidding algorithms on AWS cloud is difficult—one could easily sample outliers to paint a comparison picture that is not true (in either direction). Thus, as authors the burden of proof is on us to prove mathematically that Aethers bidding algorithm yields better prices than the closed source algorithm in AWS Batch. It is difficult to accurately compare two algorithms (and easy to fake—both algorithms running at the same time would affect the other; if they are running at separate times they are subject to different conditions. It is financially not feasible to run the same Batch process at realistic scale on both platforms with sufficient replicates to ensure statistical
significance). Thus, we have attempted to logically showcase the features that make Aether a superior algorithm through examples comparing mechanisms of querying the Expectation of batch job runtime in both bidding approaches. The reasons that Aethers bidding approach is superior is quite simple—AWS Batch never asks the user how long they expect each of their batch jobs to take. The goal of any arbitrary bidding algorithm looking at AWS spot instances (those that can terminate if you are outbid on price; Aethers advantage comes from bidding on these types of instances efficiently) is to minimize the amount of wasted compute that occurs when you bid on an instance, run you job for an arbitrary amount of time, and have somebody outbid you before your job completes. Two important things to note here are 1) that AWS benefits if you bid incorrectly—they can very easily make more money having users bidding incorrectly and use more compute on spot instances than anticipated compared with initially utilizing on a fixed price instance and 2) unlike the stock market, past prices in a region do correlate well with future prices—it is quite common for the price of a region to fall into a local minima where it is underutilized for some reason. Besides parameterizing its Linear Programming (LP) bidding strategy with the characteristics of the compute that a user will require for a batch job, Aether also asks the user how long that they estimate each batch job will take. AWS Batch does not inquire as to an estimate of how long each batch job will take. The Aether bidding backend subsequently takes this estimate to parameterize a lookback that assigns essentially serves as a regularization parameter penalizing regions that are not underutilized with price in a stable local minima. Without this lookback, it is not feasible to have a spot instance bidding strategy that provides more utility per dollar than utilizing fixed price instances. Giving Amazon the benefit of the doubt, they could possibly be using extra compute to run machine learning algorithms that predict the Expectation of the amount of time a job will run for any AWS Batch user; we would argue non-objectively that the variability of runtime between bioinformatics tasks (i.e. de novo assembly versus alignment) would make such a hypothetical implication useless. Finally, it is worth noting that there is value in having a fully open source bidding algorithm as it holds AWS accountable in the context of providing the best service to their users by making the potential effects from the conflicts of financial interest in AWS running a tool to compete in a market that they fully control less opaque.

2.4 Aether Command Line Options

Usage: aether [OPTIONS]

The Aether Command Line Interface

Options:
- -1, --interactive Enables interactive mode.
--dry_run Runs Aether in dry-run mode. This shows what cloud computing resources Aether would use,
but does not actually use them or perform any computation.

--ilp Runs the LP algorithm as a dry run with the CPLEX solver instead of the default solver

-A, --input-file TEXT The name of a text file, wherein each line corresponds to an argument passed to one of the distributed batch jobs.


-P, --processors TEXT The number of cores that each batch job requires

-M, --memory TEXT The amount of memory, in Gigabytes, that each batch job will require.

-N, --name TEXT The name of the project. This should be unique, as an S3 bucket is created on Amazon for this project, and they must have unique names.

-E, --key-ID TEXT Cloud CLI Access Key ID.

-K, --key TEXT Cloud CLI Access Key.

-R, --region TEXT The region/datacenter that the pipeline should be run in (e.g. "us-east-1").

-B, --bin-dir TEXT The directory with applications runnable on the cloud image that are dependencies for your batch jobs. Paths in your scripts must be reachable from the top level of this directory.

-S, --script TEXT The script to be run for every line in input-file and distributed across the cluster.

-D, --data TEXT The directory of any data that the job script will need to access.

--help Show this message and exit.

2.5 Tutorial

We have provided a tutorial to help users with migrating their computational workflows to Aether. The tutorial can be accessed here: http://aether.kosticlab.org/tutorials/.

Additionally, the tutorial page provides information about using Aether across multiple cloud compute service providers or with your own hardware.

2.6 Optimization Problem Specification

For bidding on the Amazon cloud, Aether’s approach minimizes

\[ \sum_{i=1}^{n} Q_i O_i + \sum_{j=1}^{m} W_j S_j \]

where there are \( n \) types of on demand instances, \( m \) types of spot instances available (there is not a spot instance for every type of on demand instance),
\( O = \{o_1, o_2, \ldots, o_n\} \) is the set of prices for on demand instances, \( S = \{s_1, s_2, \ldots, s_m\} \) is the set of maximum prices for spot instance types over a user specified "look-back" period, \( Q = \{q_1, q_2, \ldots, q_n\} \) is the set of "bids" (number of instances requested) for on demand instances, and \( S = \{s_1, s_2, \ldots, s_m\} \) is the set of "bids" (number of instances requested) for spot instances. This minimization is constrained by:

\[
\begin{align*}
\sum_{i=1}^{n} Q_i O_i^{ram} + \sum_{j=1}^{m} W_j S_j^{ram} & \geq R \\
\sum_{i=1}^{n} Q_i O_i^{cpu} + \sum_{j=1}^{m} W_j S_j^{cpu} & \geq P \\
\sum_{i=1}^{n} Q_i O_i^{ephemeral, storage} + \sum_{j=1}^{m} W_j S_j^{ephemeral, storage} & \geq E
\end{align*}
\]

where \( R \) is the minimum amount of total ram, \( P \) is the total number of processors, and \( E \) is the amount of scratch space available without need for purchasing EBS storage. To reach the 140 inequalities mentioned in the paper, an additional constraint is added for each instance type that bounds the number that can be bid upon (SLA Agreement constraints inspired by the work in Andrzejak et. al.) to the number of that type of instance the user currently has spun up subtracted for the users service limit for that type of instance. As mentioned previously in the supplemental methods section, the default Aether solve utilizes a recursive set reduction heuristic to reach an estimated ILP solution (code is available here: https://github.com/kosticlab/aether/blob/master/lp/lp.py). To allow for reproducibility, the ILP solution can also be generated with the CPLEX solver (code is available here: https://github.com/kosticlab/aether/blob/master/lp/ilp.py).

3 Supplemental References