Serverless Map + Reduce with AWS Lambda

Lillian Choung (lchoung) and Audasia Ho (audasiah)

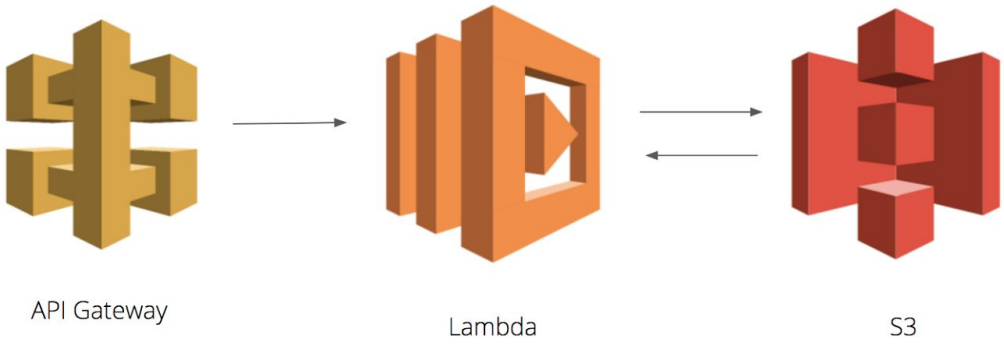
Summary

We explore the power of AWS Lambda's microservice platform for serverless computing. Our focus is upon massively parallel map problems, and we benchmark against typical personal computers (4-core MacBook Pro) and more powerful academic computers (CMU linux.andrew.cmu.edu with 40 cores).

Background

Key Components

- **AWS Lambda**, a serverless compute service that executes (and bills for) code as needed, and scales automatically. Packaged Python code can be executed on each invocation of Lambda.
- **API Gateway**, a service for routing business logic on AWS
- **S3,** a storage system on AWS



For each of our example algorithms, the working dataset was uploaded onto S3, and read in part by each Lambda invocation triggered by API call, with the result written back to S3 or passed back to the caller function.

Challenges

Each Lambda invocation had a limited amount of time (~1 min) to run before our API Gateway request timed out, and a limited amount of memory (up to 1024 MB).

API Gateway and AWS Lambda both implement throttles to protect against spamming/misuse. We had to stay mostly within 100 requests/second.

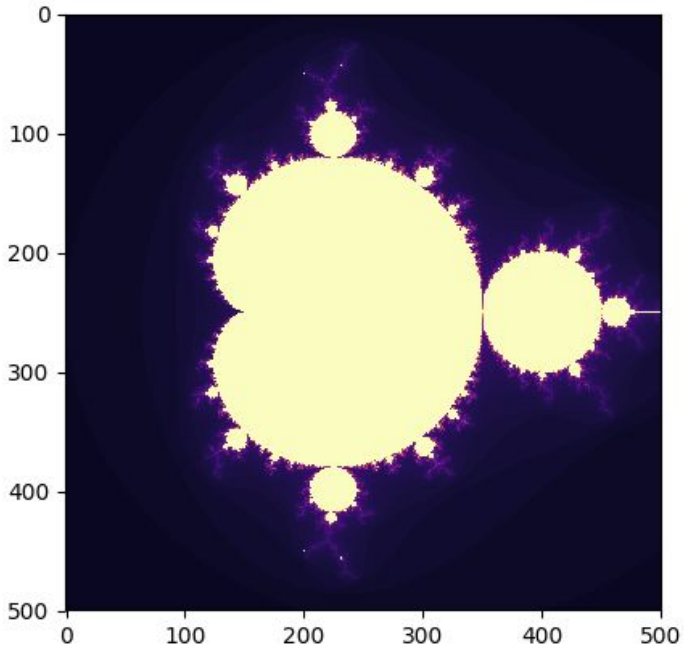
All scientific package dependencies (numpy, scipy) had to be packaged in a zip along with our Lambda code. We had to run a Docker clone image of AWS Lambda runtime to install the dependencies into a virtual environment that would have the same settings when uploaded onto the actual AWS Lambda service.

Our Test Algorithms

We decided to implement two example algorithms, **Mandelbrot set** and **Random forest**, in order to explore strong and weak scaling.

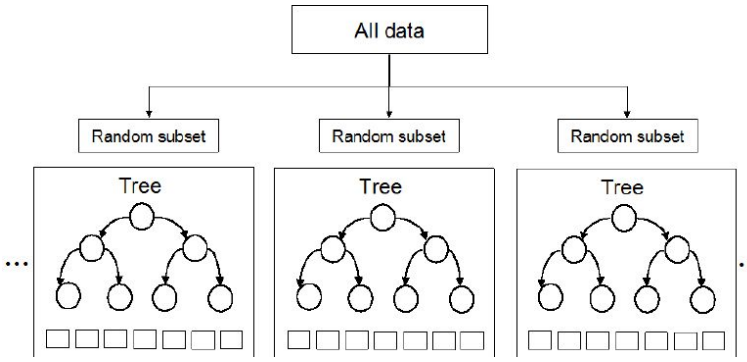
- **Mandelbrot set**, is an example of strong scaling, as we could have a fixed problem, with fixed work, and distribute the work to P processors (Lambda functions) to achieve speedup.
- **Random forest training** is an example of weak scaling, as each processor (Lambda function) has a fixed set of work, but we can have variable amounts of work for the problem.

Mandelbrot Set



We can generate the Mandelbrot set in parallel since each point can be calculated independently of other points. The input of the algorithm are dimensions of the image, as well as the maximum number of iterations. We can split the plane up by row and calculate the values of each point in a row in parallel.

Random Forest (Training)



We train random forests in parallel since trees are independent. Each tree is formed by randomly subsetting the training data (consisting of rows of observed features and the target variable), and building a decision tree from a random subset of features on each level.

Approach

Mandelbrot Set

We implemented the computation of the Mandelbrot set using Python. We used the multiprocessing library in order to implement the parallel multi-core version of generating the Mandelbrot set, which allowed us to utilize every core on the machine to compute the rows of the image in parallel.

The main computational function of our implementation iterates through a row and calculates the value for each index in the row, stopping when the function converges or maximum iterations is reached. Our sequential algorithm uses this function to sequentially iterate through all the rows in the plane, and sequentially calculates the value of each point in the image. Our parallel version uses the multiprocessing library to parallelize the row calculations, assigning the row calculation method to each core on the machine. This generates a 2D array that can then be used the display the image.

Our Lambda version sends HTTP requests through the AWS API Gateway, with a request for each Lambda function that we wanted to run. We used the library grequests in order to send asynchronous requests. Each Lambda function would take in the size of the image, the maximum number of iterations, and a list of the rows that the Lambda function should be computing. The number of rows per function would be NUM_ROWS / NUM_LAMBDAS, with work being evenly distributed to the lambdas. The Lambda function has a method similar to the row computation, and the function calls this method for all of the rows in the list it is given. The function then returns a dictionary, associating each row it was given with the array of the values for that row. On the client side, we parse through the dictionaries

returned by each Lambda function and construct our 2D array needed to create the Mandelbrot set image.

We then used the matplotlib library in order to plot the image for all of our algorithms, in order to ensure that the Mandelbrot image was corrected generated.

Random Forest (Training)

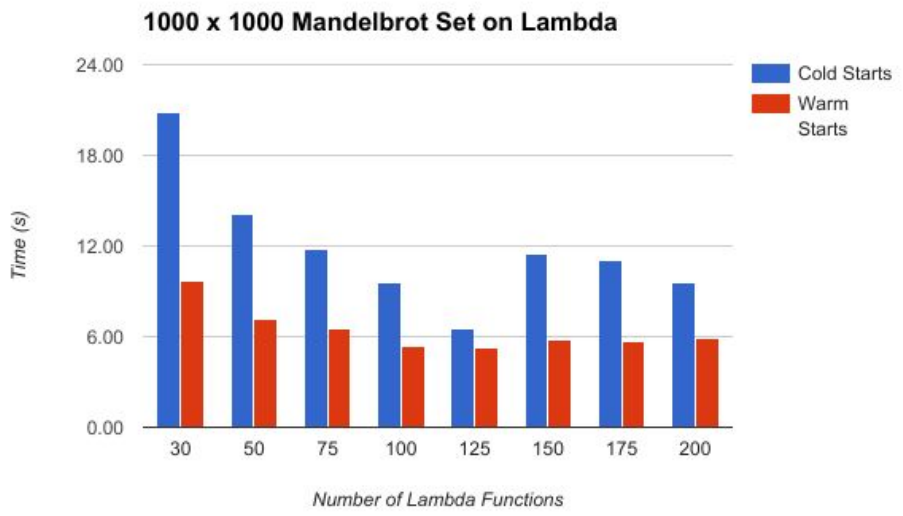
Our dataset was from Lending Club. We used the scipy and numpy libraries in Python to train trees that predicted whether a loan payment would be late based upon many (~15) features such as loan amount, type, and financial situation. We parallelized across training independent trees.

Our 4-core parallel version used multiprocessing, a Python library that allowed us to run tree training threads on every core of our machine, reading from and writing to local memory.

Our Lambda version sends via HTTP request (API Gateway) one request for each tree in the forest, using the asynchronous requests library grequests. The scipy and numpy libraries were zipped and uploaded with the Lambda function, which reads the dataset from S3 and calls the libraries to train each tree.

Results

Mandelbrot Set

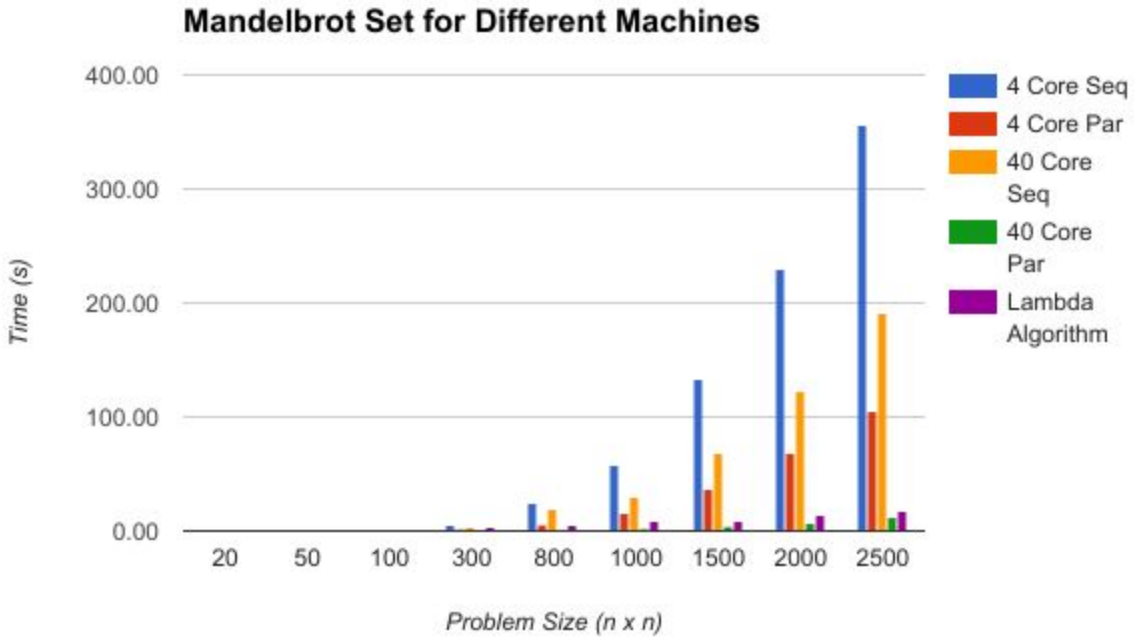


We started off with fixing the problem for Mandelbrot set to the 1000 x 1000 plane case. We wanted to first explore the speedup of the problem over the number of Lambda functions that we call. We were limited to calling up to 200 Lambda functions at a time, as the API Gateway

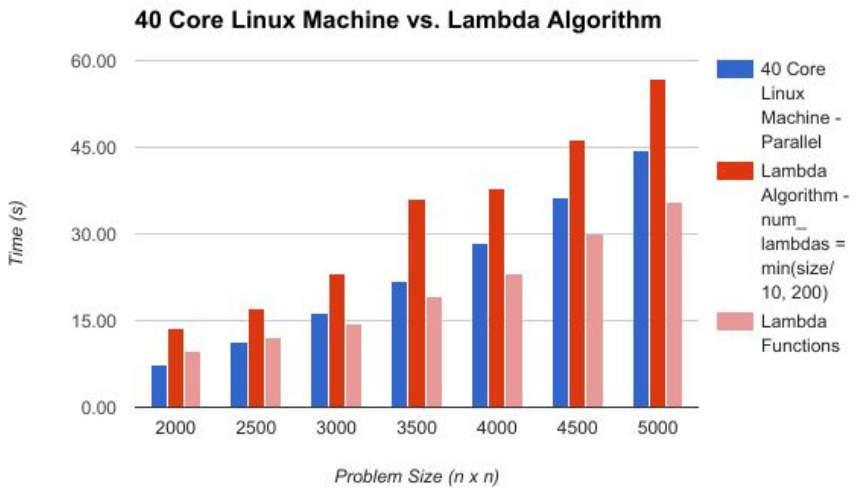
throttled us and would not produce reliable results consistently. From the graph, we see that once the number of Lambda functions is greater than 125, the overall time it takes to run the algorithm starts increasing. We feel that this is due to the number of rows not decreasing greatly as we add more Lambda functions, but there is setup cost to spinning up more functions, and it is not more efficient to have more Lambda functions. We also found that the difference in the amount of time it would take for cold starts and warms starts of the function was significant. This means that if the user wants to run the function multiple times, it may be beneficial to run their function on Lambda, as the more times it is run, the less impactful the cold start will be.

We also found that the price for running 10,000 instances of this problem would range from ~$9 to ~$16, depending on the number of Lambdas. The more Lambda functions that are called, the more it costs overall, however the speedup is best around 100-125 Lambdas. Thus, it costs $0.0013080949 to run one instance of this problem on 125 Lambdas. We compare this to running functions on AWS EC2, which would cost $0.023/hr for a single core machine. If a user doesn't have too many instances to run, it may be more beneficial to run their functions on Lambda, as they will execute faster, and the user will only be paying for their compute time. However, if a user wants to be continuously running functions, it may be more cost efficient to use AWS EC2, as the cost of the core per hour is cheaper overall.

After exploring a single problem instance, we decided to also explore increasing the problem size, to see if a larger problem size would be more efficient on Lambda, compared to 4 core and 40 core machines. We ran our sequential and parallel version of our algorithm on varying problem sizes, on a 4 core Macbook Pro and a 40 core Unix Machine. We then compared this to running the same problems on AWS Lambda, using 200 Lambda functions. We wanted to see in which instances it would be more cost effective and time efficient to run on Lambda, compared to using a local machine.



We found that the time for sequential and parallel solutions on a 4 core machine exponentially increase as the problem size increases, making them not as sustainable as when the problem set is large. However, on a 40 core machine the parallel algorithm is still relatively fast on large problem sets. This makes sense, as there are 40 cores to be utilized, which means speedup is quite large. The Lambda algorithm is faster than all of the algorithms, except for the parallel algorithm on the 40 core machine.



Even on problems with size up to 5000 x 5000, the 40 core parallel algorithm still runs faster than the Lambda Algorithm. However, we also observe that the Lambda function specific methods take a shorter time than the 40 core parallel solution, and it is the setup and processing time on the client side that makes the Lambda function overall take longer than the 40 core parallel algorithm.

Thus, if the user is able to make the client side code more efficient, it may eventually have the Lambda function run faster than the parallel 40 core algorithm.

So, we would make the following suggestions:

If a user has access to a 40 core or more machine, they are better off using a parallel algorithm on that machine than to implement the algorithm on AWS Lambda.

This is because currently the speedup from a 40 core algorithm to the Lambda function is 0.78x. Thus, the 40 core parallel solution is faster than the Lambda function. However, if the user is able to efficiently process the data that is returned from the Lambda functions, then, at larger datasets the Lambda functions may overtake the 40 core algorithm in time.

It is also beneficial to use the 40 core machine, as there are no billing costs to use it. Using AWS Lambda will cost the user money to run the functions, and it may not be worth it to them. It also takes time to set up AWS Lambda and there is a learning curve, thus if a user only wants to run the function a couple times without having a large learning curve, it may be better for them to use a 40 core machine.

If a user only has access to a 4 core machine, it is beneficial for them to use AWS Lambda to run their algorithms.

This is because the current speedup for large problem sets from 4 core to Lambda is around 6.08x. This is for the 2000 x 2000 problem, and as the problem space increases, the parallel algorithm on a 4 core machine will still exponentially grow, as there are only 4 cores that can be utilized at the same time.

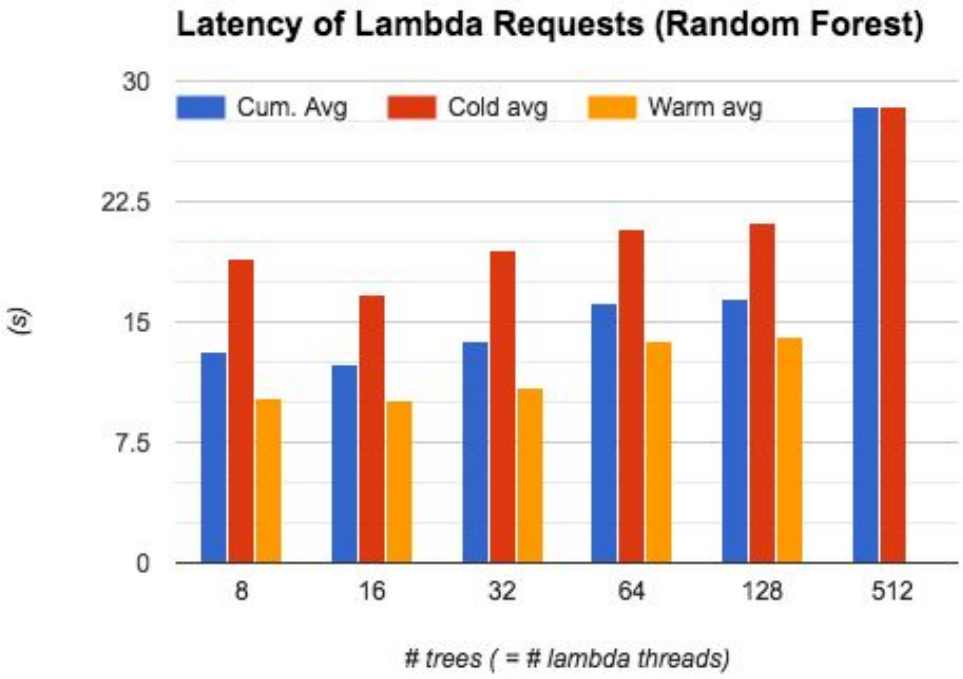
If a user wants to run their function multiple times, and wants to have the results quickly, it is worth the money to run it on Lambda. Running their function on a 4 core machine takes exponentially more time, which means more waiting for them. If they only want to run it a small amount of time, then they could run it on their 4 core machine, it would just take much longer; however it would be free. However Lambda would be a beneficial solution if they anticipate running their function many times, or if the result is time sensitive.

There is a lack of speedup, as we are unable to call more than 200 Lambda functions at once, as the API Gateway limits us. This means that eventually there will be a linear increase in time it takes for the Lambda functions to run, as we cannot spin up more than 200 functions at a time. Furthermore, the API Gateway also has a timeout, which means that all Lambda functions must finish and return in <1 minute. If not, there is a possibility of the API Gateway stopping the function, and we would be unable to get a response.

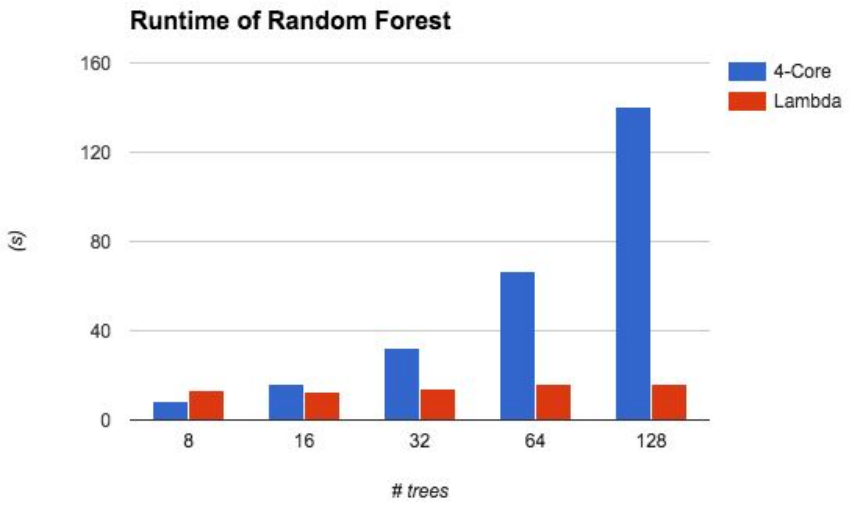
For Mandelbrot, as the problem space got larger, there was more communication and overhead from the client side, as it had to process more and more responses from the Lambda functions. Thus, we would not recommend running large Mandelbrot sets on AWS

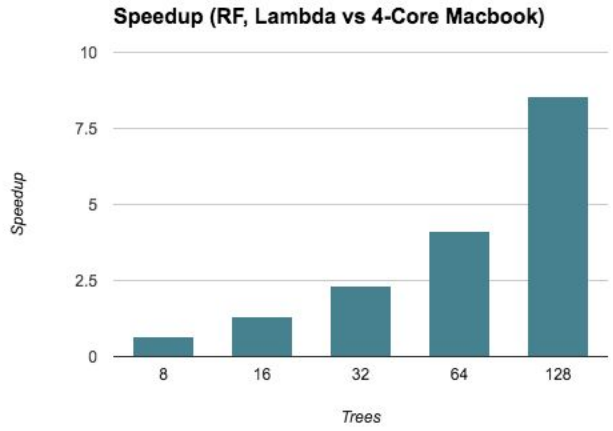
Lambda. Thus, we cannot currently recommend strong scaling problems to be run on AWS Lambda, due to the constraints on how many Lambda functions can be called, and having larger and larger problems to run.

Random Forest



We fixed the amount of work each lambda thread does (1 tree each), and grow the number of trees. The above graph shows the result of doubling the amount of work done by the system as whole on each test. Note that in general, time taken stays fairly constant as the work increases, up until we try 512 threads. This is because at 512 threads, we are reaching the upper boundary of requests per second that Lambda can handle, and many computations fail to return. So this is an upper bound on the number of invocations of this problem we can run. Also note that there is a small increase in runtime from 32 to 64 to 128 threads. This is due to a mixture of overhead on the caller side, keeping all the connections open, and significantly Lambda threads just running slower when there are more of them called at the same time, probably due to AWS scheduling rules which are not disclosed to us.



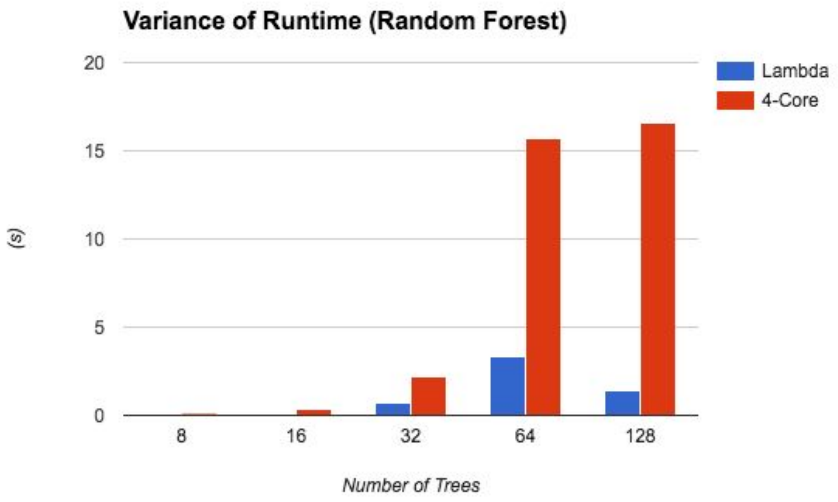


Unsurprisingly, the runtime of Lambda is much better than that of the 4-core machine, since Lambda can run each tree "simultaneously" but the 4-core machine can only compute as many trees at a given moment as it has contexts available.

On the left, notice that we have sub-linear speedup for Lambda over 4-core Macbook below n_trees = 16. Then it gets better.

Why? A tree can typically be computed on Lambda in 16s, on local machine in 8s. The extra 8s includes start-up times and communication overhead. On warm start, in fact, we can compute a single tree on Lambda in 10s!

** We wanted to compare against linux.andrew.cmu.edu too, but had permissions trouble installing alternate Python packages and distributions there.



Interestingly, Lambda exhibited less variance in runtime than running on our local machine. This was unexpected since we thought that the HTTP connection + black box AWS scheduler would vary more. The personal computer may sometimes schedule our Python program behind that of other user processes.

Conclusion

In conclusion, we found that there is a small window of problems that may be useful to run on AWS Lambda. We suspected that memory and time limitations on the platform would be problematic, as well as relatively high cost of HTTP communication, excluding many complex algorithms where “threads” may need to talk to each other.

Putting $$ costs into account, we concluded that for problems that we’ve identified to be well parallelizable on Lambda, if they are medium sized one-off computations that don’t require thousands of threads, and the user has no access to large-core machines or GPUs, then Lambda could be a good choice. It does, however, have a large learning curve since the platform is not built for scientific parallel computing purpose. Furthermore, code that runs for longer periods of time or more often should be done on a EC2 instance (or you can maintain your own servers for it) because it is more cost efficient in that case.

Equal work was performed by both project members.

References

1. https://en.wikipedia.org/wiki/Mandelbrot_set
2. https://www.ibm.com/developerworks/community/blogs/jfp/entry/How_To_Compute_Mandelbrodt_Set_Quickly?lang=en
3. https://docs.python.org/2/library/multiprocessing.html
4. https://aws.amazon.com/documentation/lambda/
5. https://medium.freecodecamp.com/escaping-lambda-function-hell-using-docker-40b187ec1e48