MASTER-WORKER AND RAY ARCHITECTURE COMPARISON ON COASTAL B.C. ALGAE BLOOM DATA

Benjamin J. Smith
Department of Computer Science
University of Victoria
Victoria, BC, V8P 5C2
benjaminsmith@uvic.ca

August 24, 2019

ABSTRACT

Coastal British Columbia algae bloom affecting ecology and socioeconomic members has been a rising concern over the past decade. Taking in chlorophyll information gathered by Sentinel-1 and Sentinel-2 satellite sensors, the raw data is converted to grey scaled image and then scaled down to allow for cleaner inputs into an Autoencoder. The Autoencoder will learn offline the most essential and important pieces of data. This will allow an LSTM to better predict a temporal forecast of algae bloom movements given historical data. This paper introduces the first step of an ongoing project by analyzing two different base architectures: a Master-Worker pattern and Ray, a system developed by RISE Labs at Berkeley. Analysis is done on the performance of raw information conversion and that output scaled for learning model input.

Keywords dataset; earth data store; geospatial; environment; neural networks; convolutional; autoencoder; lstm; time-series; forecaster; algae bloom; distributed system; master; worker; thread pool; ray;

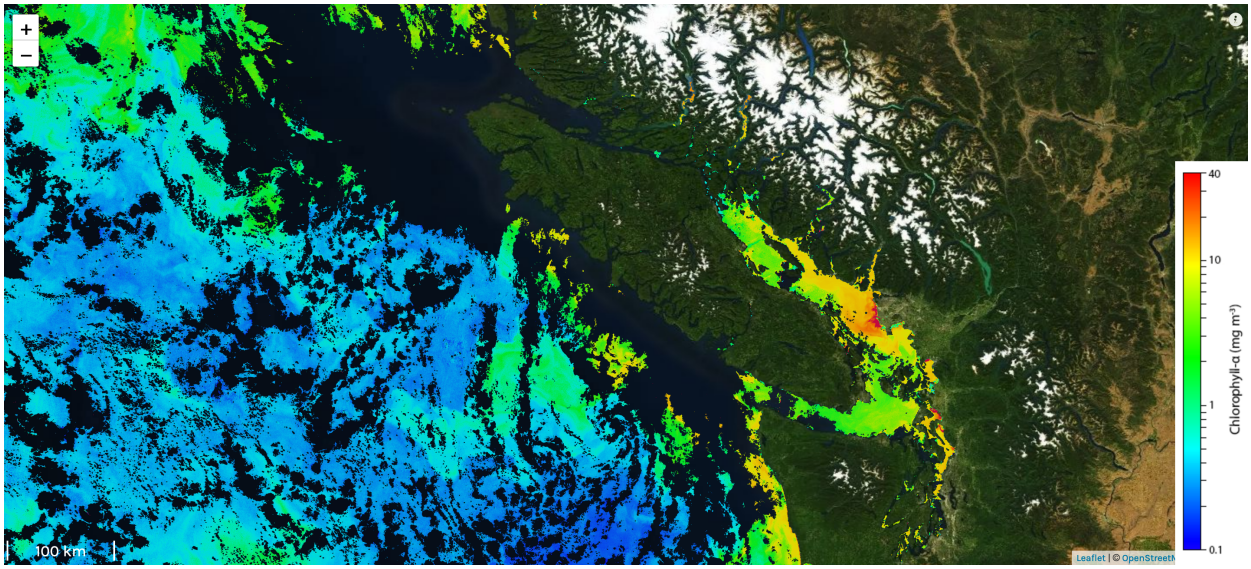


Figure 1: Chlorophyll Concentrations Along Coastal B.C.

1 Introduction

Algae defines many different living organisms of which exists in both fresh, brackish, and salt water. In this paper, we focus on the salt water algae, specifically along the coastal region of Western British Columbia (B.C.). Over the last decade, the occurrences of harmful algae blooms (HAB's) have become significant to coastal ecological and socioeconomic members [1, 2, 3, 4]. Though thousands of microaglal organisms exists, about 100 [4] of these emit natural toxins under certain seasonal (notably weather events of spring and summer) [4] circumstances that can be harmful to humans and animals [1, 2, 3, 4], and of specific interest is the Amnesic Shellfish Poisoning (ASM) from the traces of Domoic Acid (DA) [3] or Paralytic Shellfish Poisoning (PSP) [4]. Coastal B.C. first found traces of DA in 1992 in razor clams, native littleneck clams, manila clams, horse clams, Pacific oysters, California mussels, blue mussels, geoducks, Dungeness crabs, and red rock crabs [3]. DA has become even more widely distributed since [3]. PSP occurrences were first documented in 1793 at Poison Cove from contaminated mussels - resulting in five sick and one death (an onset of neurological symptoms resulting in paralysis or death through respiratory arrest [2]) [4]. In B.C. the most prevalent paralytic shellfish toxins (PST) are that of *Alexandrium* blooms [2], though no illnesses have been reported since 2005 due to biotoxin monitoring program, effective closures of oceanic agricultural and fishery areas, and public education about the hazards of shellfish poisoning [2]. Closing down oceanic agriculture and fisheries causes the potential of bankruptcy, layoffs, and lost harvest [2]. Toxins can remain in seafood, even after toxin levels have decreased in the surrounding water [1]. This allows for toxic accumulation in local sea-life (fish, seal varieties, whales, etc) and specifically those that are of most concern exist in the human food chain. This is important since the biotoxins are not destroyed through cooking or processing of seafood [1].

With the access to a chlorophyll level dataset taken by Sentinel-1 and Sentinel-2 satellite sensors, an accurate measurement of algae blooms is possible. Through the compute power harvested in the Compute Canada Cedar cluster and using this temporal history data a distributed system, to make use of as much available resources as possible, will predict a future possibility of algae blooms is proposed. Being able to predict the movement or occurrences of algae blooms can help make insightful decisions into ecological and socioeconomical strategies - such as closures of harvest areas or ocean reclamation remedies. These decisions could minimize the closure times that can keep staff employed and business afloat.

2 The Dataset

The dataset involves a collection of coastal B.C. chlorophyll concentrations over a span of roughly three years gathered by the Sentinel-1 and Sentinel-2 satellites. The raw data was stored as 855 NETCDF4 files and require preprocessing to be usable for learning. The dataset contains many temporal holes as data could only be collected when the satellite was passing over the desired area. Also, the data that was collected also contains holes where cloud coverage obstructed the satellite sensor readings. This causes concerns for learning as the dataset is small and inconsistent. Previous works performed atmospheric corrections on the data. As well as creating 848 RGB PNG files from the corrected dataset that highlight the concentrations of the chlorophyll as a colour spectrum, as seen in Figure 1.

3 System Architecture

The following subsections detail the system architectures that have been developed and compared. Three architectures were implemented and tasks executed on Compute Canada's Cedar cluster. The performance of these implementations were compared to determine the most effective method for being able to preprocess, learn, and predict a forecast of algae bloom events given sample PNG files.

3.1 Compute Canada: Cedar

Compute Canada (CC) Cedar is a heterogeneous high performance compute cluster that houses 58 thousand Broadwell CPU cores and 584 Pascal GPU devices available for researchers and partners to utilize [5]. The tasks performed on the dataset are all performed on the CC Cedar cluster and submitted as batch jobs. This is the mechanism which Cedar executes jobs based on available resources and priority [5].

The following bash script is a typical script used to submit a job to CC Cedar. The options given were the ones used to submit the tasks mentioned below as a means to give a fair comparison between the two architectures, also detailed below.

```
#!/bin/bash
#SBATCH --time=06:00:00
```

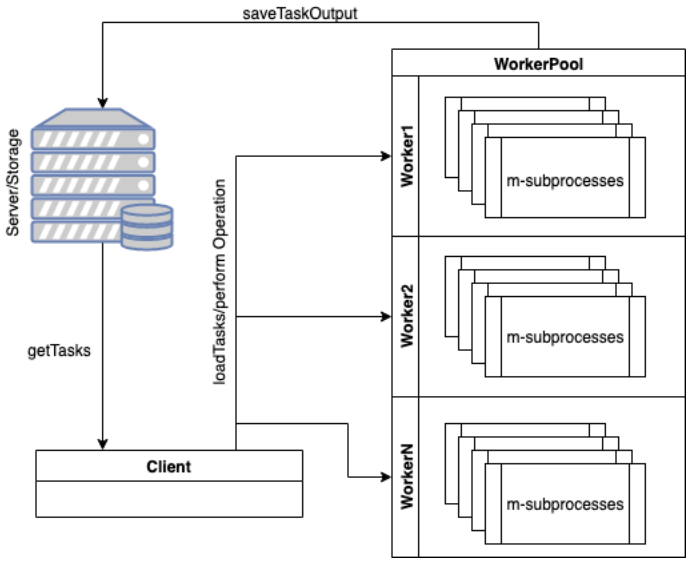


Figure 2: System Architecture

```
#SBATCH --account=ACCT_NAME
#SBATCH --mem-per-cpu=3GB
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=40

module load python/3.6
pip install --user -r requirements.txt
python3 script.py --args...
```

3.2 The Client

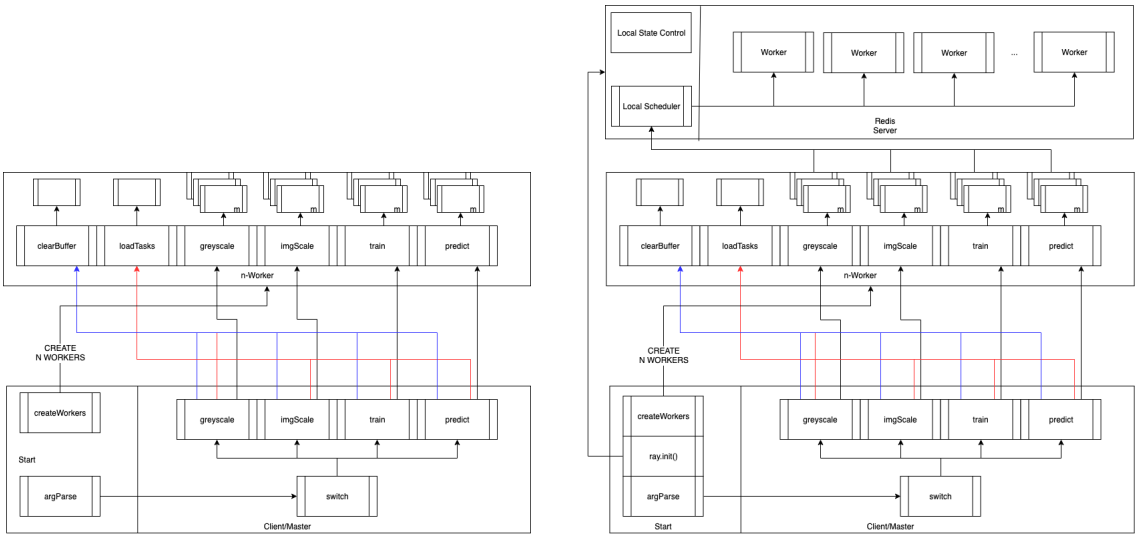
The client is the application which the user interacts with to perform tasks by specifying the appropriate command line arguments. The Client initializes the Worker objects by creating Worker objects (local Master/Worker and Ray Actors).Next, the Client gathers the necessary input file paths and then distribute to the awaiting Workers. File paths were chosen to distribute since they are serialized already as strings and can be passed around with minimal overhead. Once all tasks have been distributed (in a evenly spread manner), the Client then requests the Worker to perform an operation on their buffered task list either by calling the function directly through the Worker (local Master/Worker) or making a remote request (Ray) to the Worker. The Client then proceeds to wait patiently as the Worker(s) process all the tasks in their task buffer. Once all Workers return complete, the Client ends and the job returns complete.

3.3 The Workers

The workers are entities which are supplied with objects and asked to perform various tasks on/with those objects based on the Client command line arguments. They are initially passed objects (file paths) that they buffer locally. Once all available objects have been distributed among the workers, the client then issues the task command to be performed on the task buffer. The Worker(s) are left to process all objects with the designated operation until complete.

3.4 Local Master/Worker Architecture

A local, "single node", implementation of the Master-Worker pattern encompasses the Thread Pool pattern inside each Worker, as can be seen in Figure 3(a). This architecture was developed this way since CC does not represent typical cloud computing architecture. Embedding the Thread Pool pattern in the Master-Worker pattern utilizes two layers of multiprocessing distribution to distribute tasks. The client creates N-Worker objects and appends them to a worker list. Next the file paths are distributed to workers evenly and each worker appends them to their task buffer. The client then issues a task command and the workers then submit jobs to a process pool of m-subprocesses. Loading tasks and then invoking the operation for the Worker on those tasks are performed serially by the Client. This means that



(a) Master-Worker Grey Scaling Time Completion

(b) Ray Grey Scaling Time Completion

Figure 3: System Architectures

subsequent Workers are not told to perform the same operation until the preceding Worker has submitted all of their tasks to their process pool. This is a source of inefficiency and should be performed asynchronously. Once the Workers have submitted their task to their process pool of m-subprocesses (number of assigned CPU cores assigned to each CC task), they are executed and return when complete.

This architecture is excellent for known strict limitations on hardware and can make efficient use of the number of CPU cores available - as long as the correct hardware availability are implemented correctly. However, it does not automatically scale very well depending on tasks.

3.5 Ray Architecture

Ray is a lightweight distribution framework developed by researchers from RISE Labs at Berkeley [6]. The Ray architecture is similar to the local Master-Worker (as shown in Figure 3(a)), except instead of using a multiprocessing scheme, Ray implements a locally hosted server (Redis) with schedulers (FIFO), object stores (Plasma Object Store) that hold object ID's of futures that have been submitted, and a Global and Local Control State to keep track of tasks/actors [6] as shown in Figure 3(b). Ray requires the decorator "@ray.remote"[6] to be present above remote functions or classes (actors) to be submitted to the server and executed concurrently. The architecture that implements Ray is almost identical to that of the Master-Worker Thread Pool architecture, except that where there are sub-processes in Figure 3(a) we instead have the remote decorator on the handler function and the operation function (what Master-Worker would multi-process). Placing a remote decorator on the handler function as well as the operational function allows the client to invoke the Worker to perform a task asynchronously. The remote operation function is then submitted to the Ray scheduler (a FIFO scheduler [6]) and the Worker waits for futures (reference to an objects return variable in the future) to return. Once all futures have been received, then the Worker has completed and can return safely back to the Client that the operation is finished.

Ray has built in scaling and is handled by the underlying Ray system architecture, this leaves the focus of implementation on the application.

3.6 Autoencoder

An Autoencoder is an unsupervised neural network that applies back propogation which set the target values to be equal to the inputs, $y^i = x^i$, given a training sample $\{x^1, x^2, ..., x^i\}$ where $x^i \epsilon \mathbb{R}^n$. Knowing this, Autoencoders are used to extract important information out of two-dimensional planes of data, such as images [7]. Capturing the most essential information allows for a more accurate prediction by allowing the predictive model to do fewer iterations to learn an effective policy [7]. This is done by building up comparisons of the input to the reconstructed output and determining the rate of loss between the two. The internals of the Autoencoder consist of an encoding phase and a

decoding phase. The encoding phase takes an input M and builds an activation map from a three channeled RGB PNG image to one-dimensional maps. One layer would not suffice in achieving this translation, so the encoder is built of n(hyper-parameter) internal hidden layers. The parameters of these layers are usually optimized by stochastic gradient descent (SGD). The encoding phase can be realized as the following function:

$$f(M) \rightarrow x$$

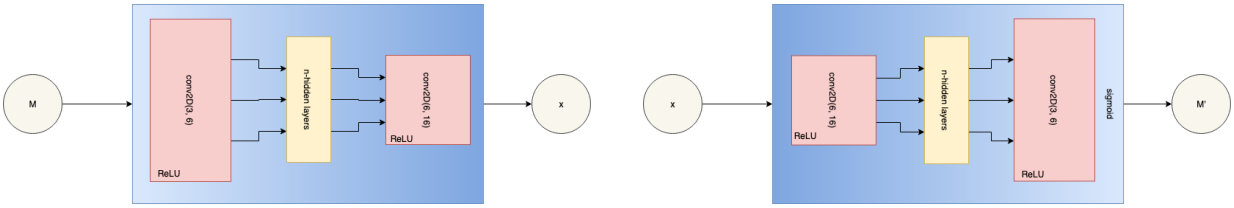
On the other side is the decoder phase. This phase takes in the activation maps in reverse order and attempts to reconstruct the original image.

$$h(x) \rightarrow M'$$

Figure 4 illustrates the structure of these functions. Once reconstructed, this is where it is compared against the original and a loss of accuracy is determined via a particular loss function. For this project, a mean-squared-error loss function was used. This is achieved by the following:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(M\_y_i - M'\_y_i)^2$$

Achieving a maximized accuracy and a minimized loss, the best model (weights of internal layer neurons (activation maps)) has been achieved.



(a) AE Encoder (b) AE Decoder

Figure 4: Autoencoder

The Autoencoder developed as part of this project was built using PyTorch [8] and it's Model libraries. It is a Convolutional Autoencoder implementing two-dimensional convolution layers with recifier linear unit (ReLU) activation functions. The decoder implements one last sigmoid activation function to normalize the output data from the last convolutional layer. The Autoencoder is fed the scaled down grey scaled images (763201 image files) in batches of 10 files with an epoch of 25 iterations. After every epoch, the batch is shuffled and reexamined. After every 1000 iterations the model is analyzed and if it is performing better then the last recorded model (or if it's the first) then it is saved (checkpoint) to storage. This saved model can be loaded and reevaluated given an increase of data or adjustment of parameters.

4 Performance Analysis

To evaluate architecture performance fairly, pre-processing operations were limited to 40 CPU cores located on the same compute node. Use of memory was not restricted beyond a minimum of 3GB requested. The header of the bash submission script can be referenced from the 'Compute Canada: Cedar' section. Performance metrics were gathered and analyzed based on the number of Workers and the time taken to complete the given operation.

4.1 Grey Scaling

In order to build usable inputs for training, the 855 NETCDF4 files were transposed into 849 (some files were corrupt) grey scaled PNG files - this isolates the chlorophyll concentrations without corrupting their accuracy. The RBG PNG files skewed the accuracy of the chlorophyll concentrations to satisfy a referential colour spectrum that makes sense for human interaction.

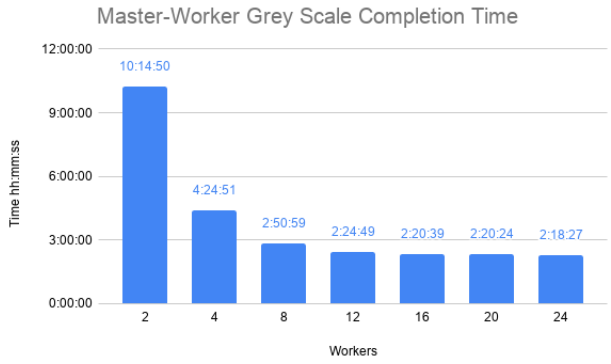
Figure 5 and Figure 6 detail the output of each trial run, dependent on the number of Workers, for the Master-Worker and Ray architecture respectively.

numWorkers	time	memAvail (GB)	memUsed (GB)	memUtilized	cpuUtilized	cpuTotal	cpuEfficiency	cpuCores	nodes
2	10:14:50	120	4.01	3.34%	3-15:42:10	17-01:53:20	21.40%	40	1
4	4:24:51	120	7.35	6.13%	3-18:44:35	7-08:34:00	51.39%	40	1
8	2:50:59	120	13.12	10.93%	3-18:02:09	4-17:59:20	78.99%	40	1
12	2:24:49	120	19.48	16.23%	3-17:42:46	4-00:32:40	92.92%	40	1
16	2:20:39	120	25.07	20.89%	3-16:32:37	3-21:46:00	94.43%	40	1
20	2:20:24	120	31.25	26.04%	3-16:17:45	3-21:36:00	94.33%	40	1
24	2:18:27	120	37.25	31.04%	3-16:44:31	3-20:18:00	96.15%	40	1

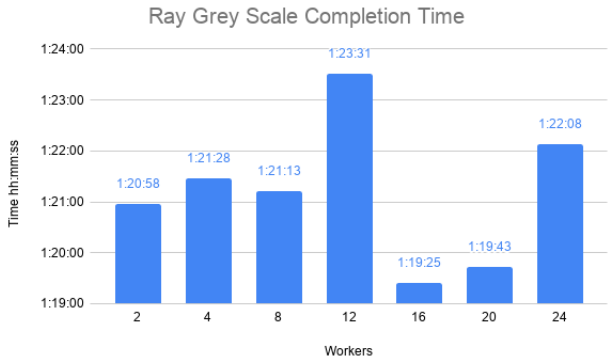
Figure 5: Master-Worker Grey Scale Output

numWorkers	time	memAvail (GB)	memUsed (GB)	memUtilized	cpuUtilized	cpuTotal	cpuEfficiency	cpuCores	nodes
2	1:20:58	120	15.91	13.26%	2-02:30:35	2-05:58:40	93.58%	40	1
4	1:21:28	120	18.33	15.28%	2-03:03:54	2-06:18:40	94.02%	40	1
8	1:21:13	120	18.62	15.52%	2-03:50:27	2-06:08:40	95.75%	40	1
12	1:23:31	120	18.86	15.72%	2-03:18:44	2-07:40:40	92.16%	40	1
16	1:19:25	120	17.24	14.37%	2-00:40:35	2-04:56:40	91.94%	40	1
20	1:19:43	120	16.76	13.97%	2-00:38:25	2-05:08:40	91.52%	40	1
24	1:22:08	120	20.67	17.23%	2-03:22:56	2-06:45:20	93.84%	40	1

Figure 6: Ray Grey Scale Output



(a) Master-Worker Grey Scaling Time Completion



(b) Ray Grey Scaling Time Completion

Figure 7: Grey Scale Completion Time

Figure 7 illustrates the time of completion for each architecture depended on the number of workers.

4.1.1 Discussion

The efficiency of Master-Worker is largely dependent on the number of Workers as seen in the amount of time to complete all tasks. This is also apparent from Figure 5 "cpuEfficiency" column noting that the use of two and four Workers greatly under utilize the available CPU cores. Compared this to Ray, which has an almost standard time to completion despite the number of Workers and also utilizing over 90% of available CPU cores. Ray manages to complete all tasks in 50% of the time it takes the best version of the Master-Worker architecture. This is a significant performance difference. The Ray architecture makes a more efficient use of it's environment given the same distribution of number of tasks to each individual Worker as the Master-Worker architecture. This could be due to the underlying system architecture of Ray that processes separate scheduling and Global and Local Control States for task tracking and it's ability to scale available resources; whereas, the Master-Worker architecture is limited to the defined number of processes each Worker is permitted to have. Since the time of completion and the CPU utilization are similar with 12 plus Workers for the Master-Worker architecture and all variations of the Ray architecture, this implies that given enough resources and with efficient use of those resources the completion time will be linearly comparable. However, Ray could make use of such resources as efficiently with less Workers as an ever increasing number of Master-Worker Workers to utilize the given resources. Understanding these differences, this makes Ray the more efficient architecture with compute based tasks.

The real power of Ray comes from utilizing multiple nodes - only one node and 40 CPU cores was used as to give a fair comparison against the Master-Worker architecture. Implementing a multi-node architecture on Compute Canada is possible but was not investigated due to the time constraint of the study. Introducing multiple nodes allows each Worker to possibly have more available resources, which could improve efficiency of task completion due to the direct relationship of having more processing power - horizontally scaling the environment.

4.2 Scaling

From the 849 grey scaled images, the next step is to scale these images down to a standard size that is compatible with the autoencoder. The original images were 6493 x 7823 pixels, which is far too large to train on, as the batches would not fit efficiently into the GPU. To resize the images, we developed a method called poolscaling. Given a scaling factor S, poolscaling will select every i+S, j+S th pixel and map it to the output image, providing a scaling factor of S squared. What makes poolscaling different from regular scaling is that the unused pixels are not discarded, and are instead used to create more output images, with the next image sampling every i+S+1, j+S+1 th pixel, and so on. For each input poolscale generates S squared outputs with similar value distributions to the original. This is useful for training as the initial dataset was only 849 images which is quite small for deep learning. Once all 849 original sized PNG files were scaled down and output with a scale factor of 30, there exists a total of 763201 PNG files ready for the autoencoder to train on.

Since the methodology of scaling images compared to grey scaling is identical in the operations of the architectures handling such requests, a pool of only twelve Workers was used. The worst case of Ray and the average case of Master-Worker. This is to minimize possible contention of requesting futures jobs on the batch request system of CC. The architectures scaled down the original grey scaled images and the performances of each architecture was compared. Again like the grey scaling comparison, 40 cores located on the same node was used to give a fair comparison, but only 80GB of memory was allocated as not to completely abuse the request system on CC.

	numWorkers	time	memAvail (GB)	memUsed (GB)	memUtilized	cpuUtilized	cpuTotal	cpuEfficiency	cpuCores	nodes
Ray	12	0:06:57	80	10.36	12.95%	1:11:51	4:38:00	25.85%	40	1
MW	12	0:07:48	80	7.81	9.76%	0:59:29	5:12:00	19.07%	40	1

Figure 8: Scaling Comparison of the Two Architectures

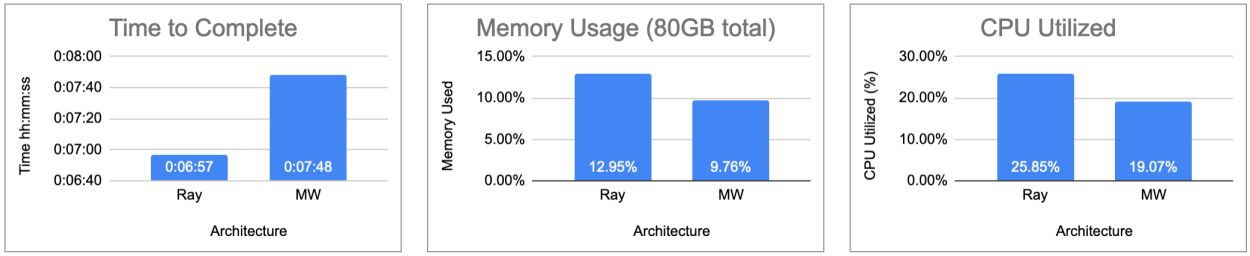


Figure 9: Scaling Comparison of the Two Architectures

4.2.1 Discussion

As observed in Figure 8 and Figure 9, Ray completes the task in less time and produces a greater utilization of resources that were available to the job. Both architectures do under-utilize all the available resources compared to the grey scaling operation - this is due to the image scaling being a much simpler and less intensive operation per task. Resources could have been more efficiently requested - under utilization like this causes a drop in priority for future jobs, this is why a minimal set of tests were run. This analysis further reinforces that Ray is a more efficient architecture to implement and offers a better performance on distributed systems.

4.3 Autoencoder

No fully operational implementation of an Autoencoder was developed - determining errors could not be done locally due to the resource overhead of this type of model. This meant testing was exclusively performed on CC. A version of the Autoencoder with a batch job and epoch round of one would run, but that would provide no usable information.

Also, running a one batch and one epoch on 763K files would not be an efficient use of time. Since CC is a batch based system that runs jobs based on priority and resources available, this constrained the time to run tests as they sat pending execution. This delayed error/bug reporting (generally over an hour at a time before bugs were noted and the job failed) during implementation. Unfortunately, no performance metrics were recorded in the time available of this project. Future work into developing and analyzing this model will be carried out and applied to other domains of data.

5 Future Work

The Master-Worker under performed, even disregarding the performance comparison to Ray. The architecture requires asynchronous operation invocation of each of it's Workers. Serially invoking and waiting for the Worker to submit its entire task list to its process pool before invoking the next Worker is poor implementation. If one Worker is slow then it becomes the bottleneck for the entire system as others cannot start until the slow Worker is complete. This brings up the issue of task assignment. These architectures implement an even distribution of tasks to workers as a pre-loading phase before the operation is invoked. This causes issues with slow workers again. Tasks should be requested by the Workers and submitted via serialized struct containing the task and operation for the Worker to execute. Performing task distribution this way could possibly increase efficiency by having the faster Workers complete more tasks. This can also be coupled with a limited task buffer to make efficient use of pre-loading without communication overhead incurred by excessive pre-loading. Automating this task request on the Workers minimizes the effort by the Client and the effort is placed on the compute nodes (not an issue for CC, but potentially for other cloud services).

Ray was designed to be distributed over multiple nodes. Restricting the Ray architecture for performance comparison sake limited the flexibility that Ray has to offer. Future work into deploying Ray Actors on multiple nodes could see an increase in performance and throughput - as long as the communication links do not serve as the bottleneck (not likely in high performance clusters such as CC and single datacenter deployments). Initializing Ray, its Redis server, and objects (store, schedulers, and control states) offer a large overhead up front. Being able to perform operations with Workers already up and waiting could see an increase in performance.

The Autoencoder requires more implementation in order to fully achieve a predictive operation. It needs to be built as a feeder into a Long-Short Term Memory neural network that will be making the predictions. The current Autoencoder offers a preliminary implementation to test the viability of running such a system on CC. It also offers up an investigation of distributing tasks to the Autoencoder and multiple Autoencoders and how this sort of distribution affects the accuracy of the model. The scheduling of machine/deep learning tasks is an ongoing field of research. Future work into developing this system and building a predictive model will be carried out. Future work into investigating a Variational Autoencoder versus Convolutional Autoencoder as viable compression solutions to learn the input data more effectively.

Future work into applying this system to other domains, such as image data of oil spills, will be performed.

6 Conclusion

Given the performance results between the two system architectures, Ray appears to outperform the Master-Worker Thread Pool system. This is most likely due to the scalability of the Ray system - being able to employ more Ray Workers to fill resources as necessary to complete the given tasks. Realized as the usage of CPU cores ran above 90% even for a minimal number of Workers. Running Workers operations on a local Redis shard employs the tactics of scalable multiprocessing as individual Ray system worker objects to process the submitted task can be created and destroyed within the shard as necessary. Ray employs the use of Local and Global Control States which manage the tracking and progress of tasks - if a task fails due to a lost worker, then it can be restarted. The Master-Worker architecture would incur an increase in complexity for this feature to be implemented - a trade-off that was dismissed due to time constraints of the project. Moving forward, Ray will be utilized as the base architecture while development of future work items will be incorporated into the design.

Continued work will employ the implementation of Ray to achieve a distributed predictive system capable of determining temporal estimates of the B.C. coastal algae blooms within reasonable service level agreement expectations.

References

[1] Berdalet, E., Fleming, L., Gowen, R., Davidson, K., Hess, P., Backer, L., . . . Enevoldsen, H. (2016). Marine harmful algal blooms, human health and well-being: Challenges and opportunities in the 21st century. Journal of the Marine Biological Association of the United Kingdom, 96(1), 61-91. doi:10.1017/S0025315415001733

[2] Lewitus, A.J., Horner, R.A., Caron, D.A., Garcı a-Mendoza,E., Hickey, B.M., Hunter, M.,Huppert, D.D., Kelly, D., Kudela, R.M., Langlois, G.W.Largier, J.L.,Lessard, E.J., RaLonde, R.,Rensel, J.,Strutton, P.G.,Trainer, V.L., Tweddle, J.F. Harmful Algal blooms along the North American West Coast Region: history, trends, causes,and impacts. Harmful Algae, in press 19 (2012) pp. 133–15

[3] Trainer, V., Bates, S., Lundholm, N., Thessen, A., Cochlan, W., Adams, N., and Trick C. 2012. Pseudo-nitzschia physiological ecology, phylogeny, toxicity, monitoring and impacts on ecosystem health. Harmful Algar 14 (2012) pp. 271-300.

[4] Horner, R., Garrison, L, and Plumley, F. Harmful algal blooms and red tide problems on the U.S. west coast. 2003. Limnology and Oceanography, vol 42, 5part2, pp. 1076-1088 doi: 10.4319/lo.1997.42.5_part_2.1076

[5] Compute Canada. 2019. Cedar. (May 2019). Retrieved August 19, 2019 from https://docs.computecanada.ca/wiki/Cedar

[6] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M., and Stoica, I. Ray: A Distributed Framework for Emerging AI Applications eprint arXiv:1712.05889, December 2017

[7] Prakash, B., Horton, M., Waytowich, N., Hairston, W., Oates, T., and Mohsenin, T. On the use of Deep Autoencoders for Efficient Embedded Reinforcement Learning. GLSVLSI '19 Proceedings of the 2019 on Great Lakes Symposium on VLSI Pages 507-512. doi:10.1145/3299874.3319493

[8] Ketkar, N. Deep Learning with Python: A Hands-on Introduction. 2017. Apress. pp. 195-208. doi:10.1007/978-1-4842-2766-4_12