Scheduling Lambda Functions on the IoT Edge

BTP Report

Submitted by

Govind Lahoti

Roll Number - 140050020

under the guidance of

Prof. Umesh Bellur

Department of Computer Science & Engineering
Indian Institute of Technology, Bombay

Abstract

Traditionally, all the analytics for the IoT is done in the Cloud. Data is collected from the sensors and shipped all the way to the Cloud through IoT gateways. This approach may not be desirable in several situations due to latency and network cost concerns. We would like to push the analytics and computation models towards the edge devices from the Cloud. We envision an approach where the edge devices provide the same analytics through mutual coordination. This project seek to discover various generic heuristics to schedule the workflow (essentially a DAG) of tasks (Lambda functions) on the edge. Further, we explore the state-of-the-art work done so far in analytics over the edge. We take a practical application of Air Quality Prediction and try to schedule its analytics on the edge of IoT. Key step while doing so would be to come up scheme to train the underlying machine learning model (neural network) in a distributed fashion using the edge devices. In this report, we present various approaches for distributed training of neural networks and their performance. We also present the simulation environment, that was coded as a part of this project, which can be used to analyze the heuristics' performances in different environment settings.

Contents

Chapter 1

Introduction

Internet of Things setup consists of many sensors which continuously generate data at some rate. Many times we wish to perform analytics over this data. Hence, data is sent to the Cloud over the network. This incurs a significant network bandwidth usage. Also, since the Cloud is usually located far away from actual sensors, a significant amount of latency builds up by the time data reaches the Cloud. This is undesirable in many situations. Following figure demonstrates how data flows in tradition IoT setup for analytics -

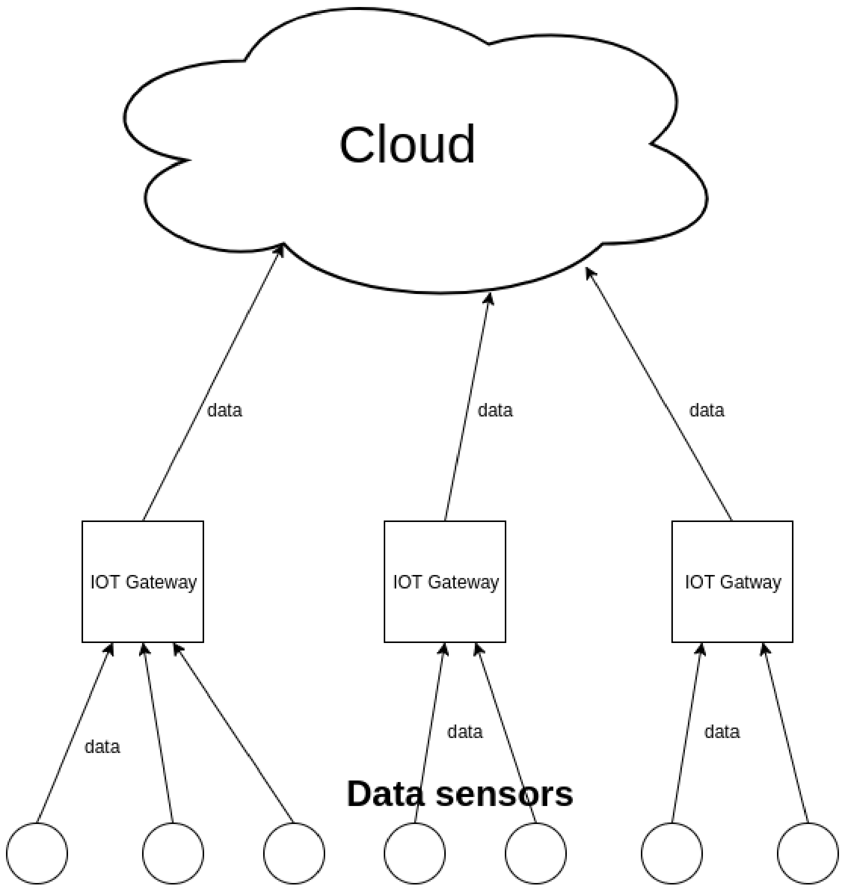


Figure 1.1: Traditional IoT setup

However network costs and latency are two important factors that sometimes make the traditional approach undesirable. We wish to come up with strategies to schedule these analytics on the edge devices itself. Edge devices will communicate and coordinate the computation work among themselves, possibly making use of Cloud once in a while if needed. Following figure demonstrates our envisioned setup for analytics by edge devices in IoT-

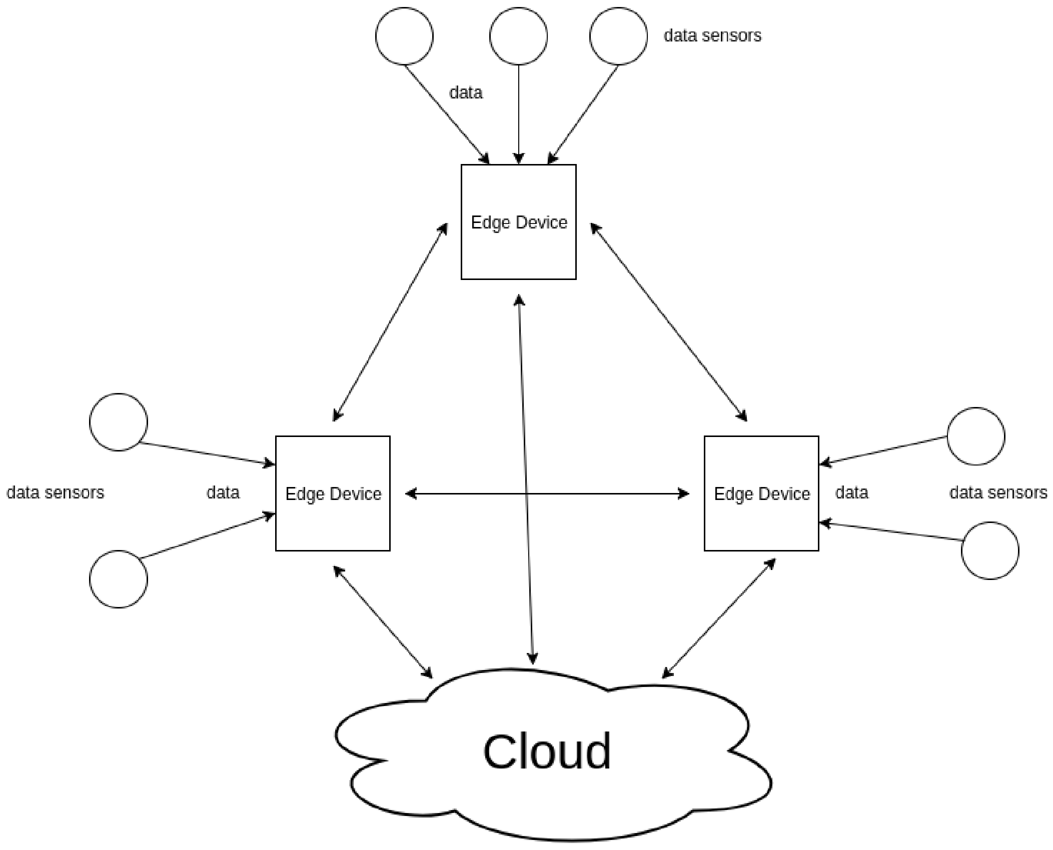


Figure 1.2: Envisioned IoT setup

Since edge devices are not powerful enough, we will have to break down our analytics into small pieces and then schedule each individual piece separately on edge devices. We can to break the analytics into a flowchart of Lambda (stateless) functions. We call such a flowchart as **workflow** and the stateless functions will be called as **tasks**.

The tasks in the workflow will be in some order. We can think of workflow as **directed acyclic graph of tasks**. Each task takes input from output stream of either sensors or some other tasks or both. Each task also emits an output stream. Following figure demonstrates how a workflow looks like -

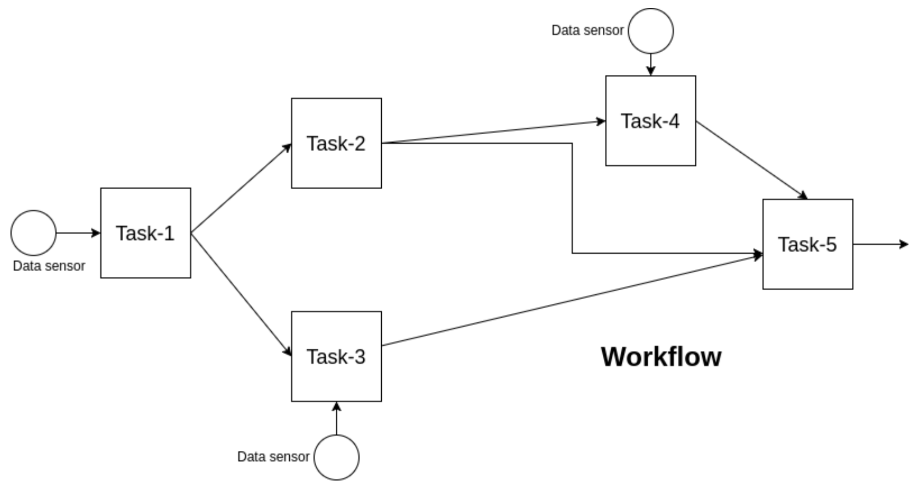


Figure 1.3: An example of how a workflow looks like

1.1 Roadmap of the report

Stage I of this project dealt with mathematical formulation of this problem and coming up with different heuristics for efficient scheduling of workflows. We first mathematically modeled the scheduling problem as a cost minimiza- tion problem under a set of constraints. Then we proposed several heuristics for scheduling. We calculated the performance of those heuristics using self- coded simulator and random data. We also tried genetic algorithms to come up with efficient packing of workflow tasks on the nodes. Work done as part of this stage is covered in Chapter 3.

Stage II of this project deals with trying out the scheduling schemes on a practical application. We first do the survey of state-of-the-art work done so far in scheduling the workflows on the edge. We then pick the problem of Air Quality Prediction. It involves meteorological and AQI sensors deployed across the metropolitan. Edge devices collect the data from sensors and send it to the cloud to train the machine learning model for prediction. We explore the ways by which the training of machine learning models (neural networks in this case) can be done in the edge itself using federation. We develop a simulator as well to test out our heuristics. Work done as part of this stage is covered in Chapters 4-7.

We provide the potential future works in this topic in Chapter 8. We conclude the report in Chapter 9

Chapter 2

Related work

2.1 Scheduling workflows on the edge

Several papers have been published in this area that try to solve this problem using various approaches. However each one of them make certain assumptions or ignore certain real-life constraints rendering the problem still unsolved.

[1] considers multiple workflows (DAGs) and tries to schedule each task in the workflows on the services available in the Cloud under multiple QoS constraints. Objective function used is the success rate of scheduling algorithm where success means meeting the QoS constraints. Here the scheduling is done using centralized scheduler. This paper doesn't consider network delays, communication costs, data transfer etc. Everything is being done in Cloud and data is assumed to be present there.

[2] tries to schedule the application graph on the physical node graph with mathematically provable optimality bounds. Objective function chosen is "load balancing" only. However this paper consider single workflow system only. Node resource capacities is assumed to be infinite. Though it considers the network costs, it assumes that the physical node graph is a tree. This paper provides polynomial time algorithm for the cases when the application graph is linear or when the application graph is tree but placement of non-leaf nodes is already provided.

[3] just introduces the idea of hybrid edge and Cloud computing, and suggests research scope. It identifies following areas for research - Microservice Configuration, Microservice Networking, Microservice Security, Edge Computing, Microservice Workload Contention and Interference Evaluation, Monitoring, Microservice Orchestration and Elasticity Control.

[4] tries to place the in-network computation graph on the physical node graph. Objectives- 1) minimize delay in obtaining function at sink 2) minimize cost of one instance of computation of function. This paper mathematically proves the NP-completeness for arbitrary graph shape. It provides polynomial time algorithm to only specific shapes of workflows (tree, etc.) and it makes the assumption that single workflow is present in the system.

[5] proposes the placement scheme of complex event processing queries on streams of data. Problem statement defined is quite similar to one chosen by us in Stage I of the project. Multiple DAGs, network costs are taken into account. Objective

function is the latency from start to end of the query. But the algorithms proposed works for system with single Cloud VM and single edge device only. The table below summaries the related papers -

Paper	Considers multiple workflows?	Considers network costs?	Works for general DAG?	Objective function	Remarks
[1]	yes	no	yes	Success rate of meeting QoS constraints	-
[2]	no	yes	no	Load balancing	-
[3]	-	-	-	-	Just introduces the idea os osmotic computing
[4]	no	yes	no	Latency and computation cost	-
[5]	yes	yes	yes	Latency	Works for system with single Cloud VM and edge device

Thus, all the related papers make certain assumptions or remove certain constraints from the system, and then solve the simplified problem. Actual scheduling problem with practical constraints is still unsolved. It can be proved to be NP-hard problem. Hence in this project we seek to come up with heuristics for scheduling and evolve them incrementally using variety of techniques.

2.2 Distributed training of neural network

Lot of work has been done on data parallelism than the model parallelism. As memory has increased on GPUs (edge nodes in our case), the majority of distributed training has shifted towards data parallelism.

[6] is the prominent paper in this field. It introduces the idea of model and data parallelism in training of neural networks in distributed fashion.

[7] [6] discuss model parallelism. [6] does not reveal the training algorithm. [7] discusses algorithm for training neural network under model parallelism, assuming a given model partition. It does not discusses the way to obtain the partitions of the

models. It does not solve the cost (bandwidth/computation) minimization problem either.

[8] discusses model parallelism, but not in P2P setting. It considers the hierarchical setting between Cloud, edge and sensors. The model's graph is in a way cut into 3 partitions. Each partition scheduled on Cloud, edge and sensor device. Information is passed to next level in hierarchy if and only if required.

[9] proposes layer-wise parallelism that allows each layer in the network to use an individual parallelization strategy (model parallelism or data parallelism)

The table below summaries the related papers -

Paper	Data parallelism?	Model parallelism?	Remarks
[6]	yes	yes	Does not reveal the algorithm for model parallelism
[7]	no	yes	Assumes that model partition is given
[8]	no	yes	Does not consider P2P setting
[9]	yes	yes	Each layer in the network use a separate parallelization (data/model)

These papers don't considered the setting of geographically distributed edge devices either, the computation capacity of nodes, minimizing network bandwidth and delay. Their focus has primarily been the convergence of models during training.

We in this project try to fill this gap. We seek to evolve the sharing strategies minimizing the communication and computation costs. We would like to see the trade-off between the accuracy of the model and the amount of communication/computation required.

Chapter 3

Stage I: Scheduling Workflows of Lambda Functions on the Edge

We can think of our analytics to be a collection of workflows, and each workflow to be collection of tasks (in some order). Coming up with workflows given an analytical model is problem in itself. In this stage, we assume that we have already been provided with the workflow designs. We address the problem of scheduling these workflows on the edge devices.

3.1 Problem Formualtion

In this section, we propose the mathematical formulation of the problem. Our setup includes some edge nodes with limited power, and some workflows that need to be scheduled on them. We propose the efficient scheduling of lambda functions on the nodes as a cost minimization problem.

3.1.1 Environment constants

Let us define the terminology for our formulation -

- N be the set of nodes available for computation. Let the i^{th} node be denoted as n_i
- T be the set of tasks in the workflows. Let the i^{th} task be denoted as t_i
- S' be the set of input streams. For simplicity let us consider any input stream as output of some imaginary task
- $resource : T \rightarrow \mathbb{R}^d$ such that $resource(t_i)$ denote the vector of amount of resources required for task t_i. In practical setting, CPU, memory, etc could be the resources.
- $capactiy : N \rightarrow \mathbb{R}^d$ such that $capacity(n_i)$ denote the vector of maximum available resources at node n_i.
- $distance : N \times N \rightarrow \mathbb{R}$ such that $distance(n_i, n_j)$ denote the time distance between node n_i and node n_j

- $input : \mathbb{N} \rightarrow 2^{T \cup S'}$ such that $input(i)$ denote the set of tasks (and sources) whose output stream is taken as input by task t_i
- For $i = 1, 2, .., |T|$, $x_i(t_j)$ denote the the number of events on output stream of task t_j needed to trigger task t_i. Clearly, $x_i(t_j) = 0 \ \forall t_j \notin input(i)$
- $s : T \cup S' \rightarrow \mathbb{R}$ such that $s(t_i)$ denote the time period of events in output strem of task t_i
  s function is know to us for all source streams. For other tasks it can be computed by -
  $$s(t_i) = x(t_j) * s(t_j) \qquad \forall t_j \in input(i)$$
  Above equality is also the legality constraint. If the system doesn't follow the above equality $\forall t_j \in input(i)$, then there will be buffer overflows.
- Let us define $S = \underset{t_j \in S' \cup T}{lcm} s(t_j)$
  It is clear that the system of this nodes/workflows is periodic with time period of S. i.e., system will behave identically after every S time units. So we need to solve the scheduling problem for an interval of S time units only.
- $run : T \rightarrow \mathbb{R}$ such that $run(t_i)$ denote the time required to run task t_i

3.1.2 Variables

Now let us define few variable functions that we have to determine for optimal scheduling -
$\forall i = 1, 2, .., |T|$ and $j = 1, 2, .., S/s(t_i)$

- $time_i(j)$ such that $time_i(j)$ denote the time (stating from $t = 0$) at which we receive the j^{th} output of task t_i. Note that we are concerned with time interval of $[0, S]$ only as everything is periodic with time period S
- $delay_i(j)$ such that $delay_i(j)$ denote artificial time delay we introduce for running the task t_i for j^{th} time. Delay might be required for better scheduling of rest of the system.
- $node_i(j)$ such that $node_i(j)$ denote the assignment of node for running the task t_i for the j^{th} time.

Note that $node_i$ and $delay_i$ may not be constant functions. We might have to schedule same task at different nodes within the time period S. For example
Say we have 2 tasks each consuming $> 50\%$ CPU.
Task t_1 occur at $t = 0, 2, 4, ...$
Task t_2 occur at $t = 3, 6, 9, ...$
Now we can schedule t_1 and t_2 at the same machine except for $t = 6, 12, ..$ where we need to schedule both on separate machine.
$\therefore$ $node_i$ need not be a constant function.
If we have only one machine, same example would work to prove that $delay_i$ need

not be cosntant.

Only $delay_i$ and $node_i$ are in our hand. $time_i$ will automatically get decided for a particular choice of $delay$ and $node$ functions according to following recurrence condition -

$$time_i(j) = \max_{t_k \in input(i)} \left[\max_{z=1,2,..,x_i(k)} \left\{ time_k(m) + distance(node_i(j), node_k(m) \right\} \right] + delay_i(j) + run(i)$$

where $m = x_i(k) * (j-1) + z$

If we assume that network time distances are very small as compared to event rates, then we can simplify above equation (by assuming that output events of a task come in order)

$$time_i(j) = \max_{t_k \in input(i)} \left\{ time_k(m) + distance(node_i(j), node_k(m) \right\} + delay_i(j) + run(i)$$

where $m = x_i(k) * j$
Base cases of above recurrences are $time_i(j)\ \forall t_i \in S'$. They are known to us.

3.1.3 Constraints

We also have few resource constraints in the system. They can be either explicitly stated or included in the cost function itself with ∞ cost.

Let us define $usage_i : \mathbb{R} \to \mathbb{R}\ \forall i = 1, 2, ..., |N|$ s.t. $usage_i(t)$ denotes the amount of resources being used at node n_i at time t. This function will be periodic with period S. We can formulate it as -

$$usage_i(t) = \sum_{t_k \in T} \sum_{j=1}^{S/s(t_k)} inuse(n_i, t_k, j, t) * resource(k)$$

where,

$$inuse(n_i, t_k, j, t) = \begin{cases} 1 & node_k(j) = n_i,\ t \in [time_k(j) - run(k), time_k(j)] \\ 0 & otherwise \end{cases}$$

So our constrains (based on resources) are -

$$usage_i(t) \leq capacity(i) \qquad \forall i = 1, 2, .., |N| \quad \forall t \in [0, S]$$

3.1.4 Cost Functions

In this project, we wish to obtain an optimal scheduling of the workflows. Optimality of a scheduling can be quantified in terms of cost functions. We can think of minimizing following two category of cost functions. However it is non-trivial to mathematically solve the resulting optimization problems.

Average latency

We define latency of a workflow as the time difference between the output generation of the final task and input arrival at the first node. Mathematical formulation of cost function is -

$$\frac{1}{S/s(t_F)} \sum_{j=1}^{S/s(t_F)} \Big\{ time_F(j) - \big(time_I(j') - delay_I(j') - run_I(j') \big) \Big\}$$

where t_I and t_F are the initial and final tasks of the workflow and j' is the index of first run of t_I corresponding to j^{th} run of t_F.
An appropriately weighted sum of average latencies of all the workflows in system would be our final cost function. We can give weights on the basis of priority of each workflow.

Power consumption

We can look for scenarios where we can turn OFF few nodes for certain interval of time (by packing all the other tasks on remaining nodes). This will save power. Assuming power usage of a machine remains same if it is ON no matter how much % CPU is being used, cost function will look like -

$$\frac{1}{S} \sum_{n_i \in \mathbb{N}} \int_0^S I_{usage_i(t)>0} \; dt$$

3.2 Types of schedulers

Following are the two kind of schedulers we can attempt to come up with. Both these schedulers will be aware of all the environment constants mentioned in section 2.1. However they will differ upon the amount of real time information (during the actual scheduling) they will have access to.

3.2.1 Centralized "all aware" scheduler

To simplify the problem in hand, we will start with a centralized and "all aware" scheduler. This scheduler will be a central entity. This scheduler will be aware of entire state of the IoT system at all the times. It will be perfectly aware of resource occupancy of each edge device. It will also be aware of future incoming tasks. This scheduler is impractical, but we wish to start off with a simpler problem and then incrementally build upon it. In rest of the report, unless specified, scheduler is assumed to be this.

3.2.2 Distributed "partially aware" scheduler

This is the ideal scheduler we wish to come up with. This scheduler will be present in all the edge devices. So the end user can approach any of the edge node to get his/her workflow scheduled. This scheduler will have partial observability of the system. Also the information it receives will be with appropriate network latencies. Work on this scheduler is in progress and it has been included under future works section.

3.3 Heuristic Approaches

Cost optimization problem mentioned in section 2 is highly non-trivial to solve. Given high amount of non-linearity and number of variables involved, a closed form solution is highly unlikely to exist. We can think of various intuition based heuristics to obtain good-enough, if not optimal, solution. We can try many heuristics and their combinations in attempt to minimize the cost functions as much as possible.
In order to get to the solution for problem this complex, it is important that we start off with a very simple version of the problem. We make certain assumptions to reduce the complexity of the problem. We find the solution to this reduced version first. We then remove the assumptions one by one, making newer versions more and more closer towards the original version.

Each of the following experiments are carried out in self coded **simulation environment**. We randomly generate workflows with randomly determined tasks' resource requirements. Similarly edge nodes are randomly generated with varying resource capabilities.
In all the approaches, we maintain the queue of pending tasks that need to be scheduled. We add and remove tasks from this queue as and when required. The order in which tasks are to be removed from the queue and the node they are to be sent to run, is what we need to determine.

3.3.1 Version 1: Single task workflows

This is the simplest version we start with. In this version, we assume that all the workflows consist of single task only. This assumption basically removes the time ordering constraint (due to directed nature of workflows) between tasks. We also assume that there are no network delays. We further assume that time required by a task is constant (irrespective of the amount of CPU and memory provided to it), ans the task demands fixed amount of CPU and memory (determined randomly)
Since in this version all workflows are nothing but single tasks, there is no time ordering between tasks. So all the tasks can be put in the pending tasks queue right at the beginning. Following three metrics were used to measure the performance of any heuristic -

1. Average duration for which nodes were occupied.

2. Maximum duration for which any node was occupied

3. Average occupancy of resources at all the node

Choosing node first, then task

We run the following iteratively till we have scheduled all the tasks. We try several methods to select nodes and tasks. These methods are outlined here. First we choose a node on which we will schedule some task. Node is chosen in following to manners -

1. Node with least amount of work. i.e., node which becomes partially free earliest
2. Node with highest available resources at time when node becomes partially free

Once the node is chosen, we chose the task(to be scheduled on that node) in following manners -

1. Random task
2. Task with least time requirement
3. Task with highest time requirement
4. Task with least norm of resource requirement
5. Task with highest norm of resource requirement
6. Task with Highest volume ($cpu * memory * time$) requirement
7. Task with Highest area $norm(cpu, memory) * time$ requirement
8. Task with Highest cosine similarity between required and available resources
9. Task with highest weighted sum of point 7 and 8 (also known as tetris)

Every possible combination of above two lists was tried out in the simulator, and **choosing tasks greedily in area** (approach 7) while choosing node by approach 1 gave the best results. Detailed results can be looked up in Appendix.

Choosing task first, then node

Similar experiments as 4.1.1 were repeated, but with slight modification. This time, we first chose the task we want to schedule, and then we find a suitable machine on which it could be run. Task was chosen on the basis of -

1. Highest time requirement
2. Highest norm of resource requirement
3. Highest area $norm(cpu, memory) * time$ requirement

Once the task was chosen, node on which it could be scheduled was chosen in following manner. Node was chosen which provided the least

1. Earliest possible time (EPT) at which task could be scheduled

2. (1 - cosine similarity b/w resources free and required) * EPT
3. (1 - 0.5 * cosine similarity b/w resources free and required) * EPT
4. Earliest possible time * fraction of resources free
5. (1 + 0.5 * cosine similarity b/w resources free and required) * EPT
6. Random

After several experiments it was found that **choosing task greedily in norm of resource requirement** worked the best. This supports intuition. Coarse tasks should fill the nodes' resource space first, then the gaps can be filled by finer tasks. **Choosing nodes which gives the earliest possible time** for scheduling gave the best results. Detailed results can be found in Appendix.

3.3.2 Version 2: Multi task workflows

In this version we drop the assumption of single task workflows we made in Version 1 (Section 4.1). This brings time ordering between the tasks. A task can not be scheduled until the output of its preceding tasks has not be obtained. In order to compare our heuristics we introduce another metric named **%extra latency**. We define it as -

%extra latency of workflow = (actual time taken - min time required) / min time required * 100 %

We use average extra latency of all the runs of all the workflows as our metric for comparison. Average resource occupancy is also used as secondary metric to ensure that results are not abnormal. Heuristics mentioned in section 4.1.2 (choosing task first, then node) were repeated again in this case.

After several experiments it was found that choosing task with least resource norm first, and then choosing node with least {(1 - 0.5 * cosine similarity b/w resources free and required) * earliest possible time} gave the best results.
Please refer to Appendix for detailed results.

3.3.3 Version 3: Including network delays

In this version, we drop the assumption of having no network costs. We introduce network delays (determined randomly) between the nodes. Now while calculating the earliest possible time (used in heuristics above) for any task on any node, network delays from the nodes of the task's predecessors were also taken into account. All the experiments performed in Version 2 (Section 4.2) are repeated. It was found that it merely increased the performance metrics by some factor. It did not change relative performance of heuristics. The conclusion of Version 2 still holds true even after the inclusion of network delays.

3.3.4 Version 4: Including CPU-time correlation

So far we had assumed that any task would require fixed amount of CPU and time. However in practical setting, OS tries to assign all the free CPU to process (if there are no other processes). So CPU% being used up by a task should be kept variable. And then the amount of time a task requires becomes some function of the %CPU alloted to it. Practically, if a process is CPU bounded, then time required and %CPU used by process are inversely correlated. If the process is not CPU bound, then correlation is negligible.
Simulator was thus modified to drop this assumption. Simulator now tries to make maximum use of available CPU. At any time, free CPU is distributed among the running tasks to make efficient use of resources.
The previous heuristics can still be applied on this version (by estimating the %CPU available at any node, thereby finding the time that a task will take to execute on that node).

3.4 Evolutionary Approaches (Genetic Algorithm)

3.4.1 Introduction

Genetic algorithms are commonly used to generate solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection. We start with a population of candidate solutions to an optimization problem. This population is then evolved toward better solutions by various alterations. Genetic Algorithms are sufficiently randomized in nature, but they perform much better than random local search, as they exploit historical information as well.

3.4.2 Experimental Setup

The setup of nodes and workflows was same as the one used in above experiments. Details specific to genetic algorithm are provided here. We start with population size of 50 genes. Each gene encodes the assignment of machines for different runs of each task. In each iteration, for the next population, we keep the top 25 genes as it is. Obtain 10 new genes via mutation, 10 new genes via crossover, and 5 new random genes. We monitor the performance of top 5 genes. We stop the iterations if the top 5 genes don't improve for a fairly long time. Performance metric chosen for comparison was the same %extra latency as chosen in above experiments.
The experiments were performed in following 3 sets -

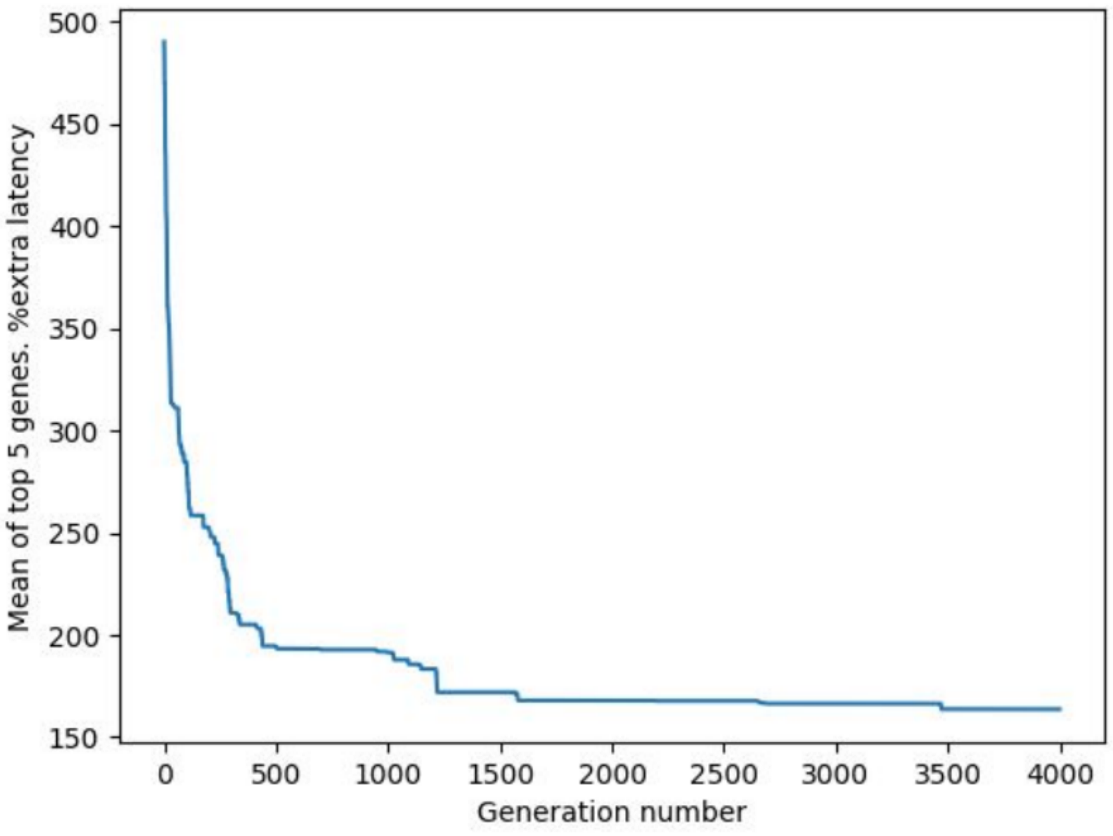
1. Starting genes were chosen randomly.

2. Starting genes were obtained by using heuristics (those discussed in section 4), and also intelligent mutations were performed.

3. Heuristics used in section 4 involved various parameters like cosine similarity, area, time requirement, fraction of resources free, etc. GA was applied to find out optimal weights of each of the parameters

3.4.3 Results

Upon running the first set of experiments with random start, following graph between the mean performance of top 5 genes and iteration number was obtained -



As it can be observed from the graph, the rate of improvement decreases rapidly with generation number. Also, the performance(%extra latency) converges to ∼170%. This is very poor in comparison to heuristics mentioned in the above sections where the best extra latency was just ∼10%.

In second set of experiments (with starting genes determined using heuristics), a similar graph was obtained but it converged to ∼50%. This is still poor as compared to the original heuristics, but much better than the random starts.

In third set of experiments, Generic Algorithm was able to achieve significantly better performance over the original heuristics. It converged to %extra latency metric of ∼8%. GA was able to identify appropriate weights for each of the parameters in the heuristic.

3.4.4 Inference

First two set of experiments gave very poor performance in comparison to heuristics mentioned in Section 4. It was due to the fact that even slight change in gene could bring drastic decrease in the performance (as a slight change in scheduling of a particular task could stall all the future tasks, thereby increasing the average

latency by a lot). In defense of Genetic Algorithm, it made huge improvement over random node assignment(from 490% to 170%)
However, the result of the third experiment was encouraging. It gives way for more use of machine learning based approaches for the given problem.

Chapter 4

Application to Air Quality Prediction

4.1 Stage II goals

Stage II of this project deals with trying out the scheduling schemes on a practical application. We first do the survey of state-of-the-art work done so far in scheduling the workflows on the edge. We then pick the problem of **Air Quality Prediction**. It involves meteorological and AQI sensors deployed across the metropolitan. Edge devices collect the data from sensors and send it to the Cloud to train the machine learning model for prediction. We explore the ways by which the training of machine learning models (neural networks in this case) can be done in the edge itself using federation. We develop a simulator as well to test out our heuristics.

4.2 Introduction

Air Quality in metro cities across the world is deteriorating day by day due to pollution. Hence for the health and other reasons it becomes essential to find the air quality in region of interest. For example, one may want to travel to the desired destination using the route having least air pollution. Air Quality can be predicted using data from various meteorological and AQI sensors deployed across the metropolitan. Traditionally the data from these sensors used to be sent to Cloud for training the machine learning models for AQI prediction. We try to do same analytics using the edge devices. We define the problem statement more formally in the next section.

4.3 Problem statement

The problem statement pertaining to AQI Prediction is explained in great detail in [10]. A abridged version of the same with modifications is being provided here with the permission from the author. Given a geographic area divided into a grid of cells [g1,g2,g3,...gn] of equal size where each grid cell contains sensors to collect weather information and an edge device for local computation and communication with edge devices in the neighborhood, and AQI sensors in few grid cells, schedule the AQI queries on edge devices and answer these queries such that computational and network costs are minimized. The area is divided into grid cells of equal sizes.

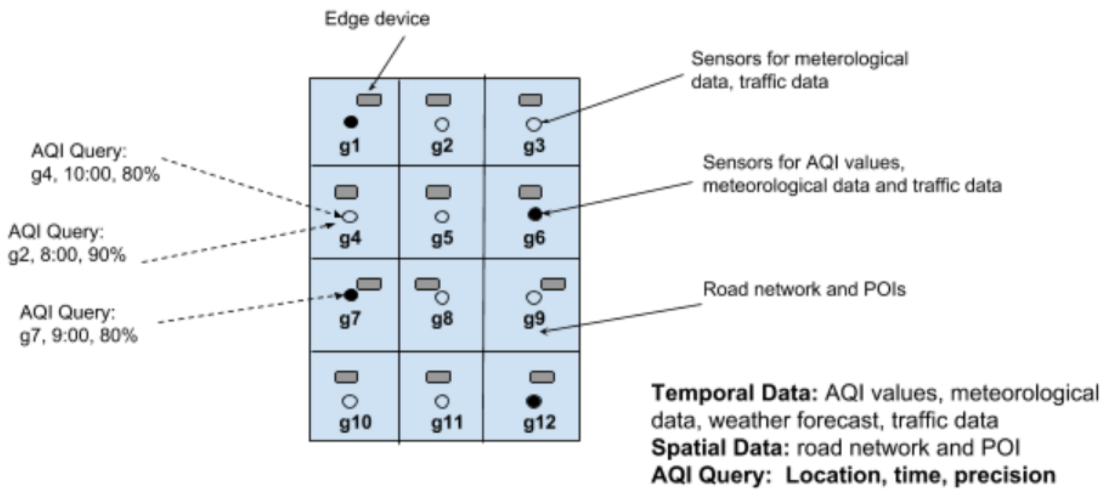

Edge device
Sensors for meterological data, traffic data
Sensors for AQI values, meteorological data and traffic data
Road network and POIs
AQI Query: g4, 10:00, 80%
AQI Query: g2, 8:00, 90%
AQI Query: g7, 9:00, 80%
g1
g2
g3
g4
g5
g6
g7
g8
g9
g10
g11
g12
Temporal Data: AQI values, meteorological data, weather forecast, traffic data
Spatial Data: road network and POI
AQI Query: Location, time, precision

Figure 4.1: Example grid of cells generating data and receving queries [10]

Assumptions:

1. Each grid cell contains sensors to get the meteorological data. These sensors gives following information at regular time interval: *Weather, Temperature, Pressure, Humidity, Wind Speed, Wind Direction.*

2. Very few grid cells have sensors to collect air quality data. These sensors collect following information at regular interval: *PM2.5, PM10, NO2, CO, O3, SO2.*

3. Each grid cell has one edge device.

The raw data collected from these sensors need to be preprocessed because, all sensors may not provide data exactly at the same time. Some values may also be missing. Moreover, there may be multiple sensors to collect weather data. In such case, aggregation and interpolation is required.

Some locations may be queried many times while some locations may rarely be queried. In such cases, queries on edge devices will not be equally distributed. Some edge devices may face many queries and some edge devices may not face any query. Some queries may be repeated again frequently.

To answer a query, the following information is required:

1. Weather data of the queried grid cell, road network and POI information from the the grid and all the grid cells in its neighborhood.

2. AQI values, if grid cell has AQI sensor. Otherwise, AQI values from neighborhood grid cells where AQI sensors are located.

The collected data is fed into the learned model to compute AQI value. If we assume that each edge device has the model for its own grid cell and has capability to communicate with devices in the neighborhood. In such case, if query is given to an edge device which does not belong to the queried grid cell then extra cost is required for sending the model and all the data to compute AQI values.

As mentioned before, the model is traditionally learnt in the Cloud. We would like to explore if the same can be done using the federation of edge devices. We present in chapter 5 the approaches to distribute the training of underlying machine learning models in the AQI problem which are nothing but neural networks.

4.4 Literature Survey

[11] [12] [13] provide approaches towards forecasting the air quality index. [11] [12] provide the machine learning models for computing AQI values of any unlabeled locations at the current time interval. The model given in [13] can be used for computing forecasting model of each grid cell.

Chapter 5

Distributed training of neural network

The key step in federating the AQI prediction problem to the edge is to figure the ways to learn its underlying machine learning model (neural network) in a distributed manner. With the advent of deep learning, the size and depth of neural networks have increased tremendously. It requires more and more computation for forward pass and back-propagation. What it also means is that more data is required to train the model as more number of parameters are present now.

5.1 Approaches

There are two major approaches to parallelize/distribute the neural network training in the literature -

1. **Model parallelism** - In this approach different machines in the distributed system are responsible for the computations of different parts of a single model. For example, each layer in the neural network may be assigned to a different machine

2. **Data parallelism** - In this approach different machines have a complete copy of the model. Each machine simply gets a different portion of the data and results from each are combined using some logic guaranteeing the convergence.

The idea behind both the approaches can be depicted well using the figure below:

As memory has increased on GPUs (edge nodes in our case), the majority of distributed training has shifted towards data parallelism. Model parallelism is however useful when the deep neural network is huge and does not fit in the memory of edge device.

We experiment with each approach using real amazon AWS EC2 VMs and neural similar to one used for AQI prediction using toy data. We present our findings in the next section.

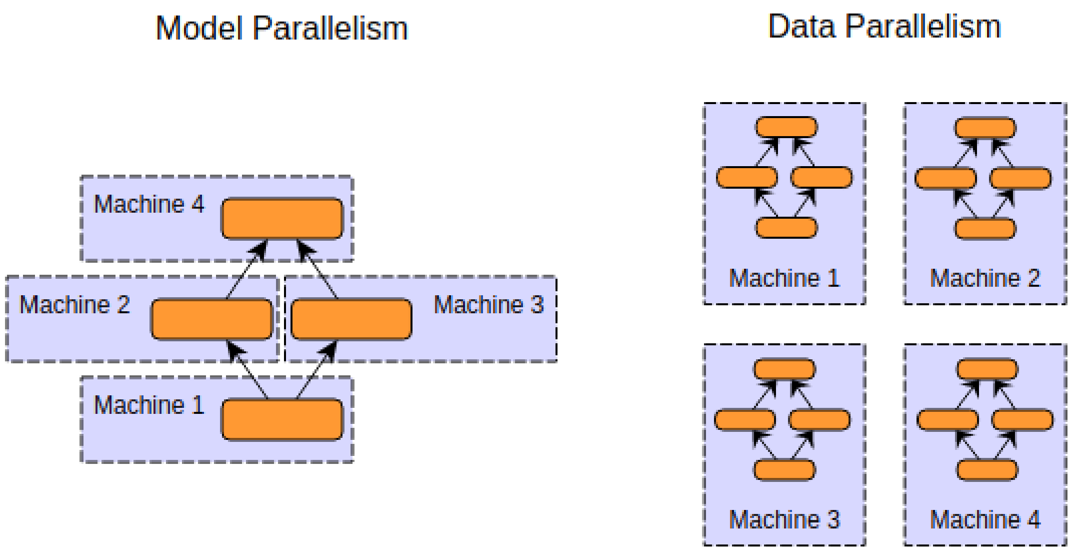


Figure 5.1: Pictorial representation of Model and Data Parallelism [14]

Chapter 6

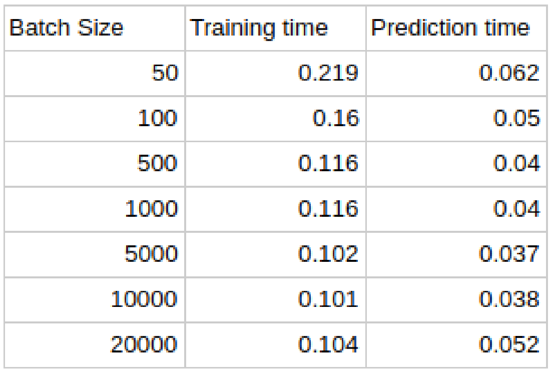
Experiments

6.1 Setup

We run all our experiments using the amazon AWS EC2 c5.x instances. These instances has 4 vCPU cores and 4 GiB of memory. The structure of the Neural network looks like - Input layer having 729 nodes, 3 hidden layers each having 729 nodes and output layer having 48 nodes. Each layer is fully connected. The model is quite similar to AQI prediction model found in literature except that the loss we use is simply squared loss, while the actual loss function used in original AQI model has spatial and temporal error components as well. The data fed to this network is generated randomly. We use tensorflow for our implementation.

6.2 Single VM

We first set the benchmark with training the model on a single VM (depicting the Cloud). We measure the average time taken per data point to be consumed in training. Following are the results -



Batch Size	Training time	Prediction time
50	0.219	0.062
100	0.16	0.05
500	0.116	0.04
1000	0.116	0.04
5000	0.102	0.037
10000	0.101	0.038
20000	0.104	0.052

Figure 6.1: Single VM. Time in *milliseconds per data point*

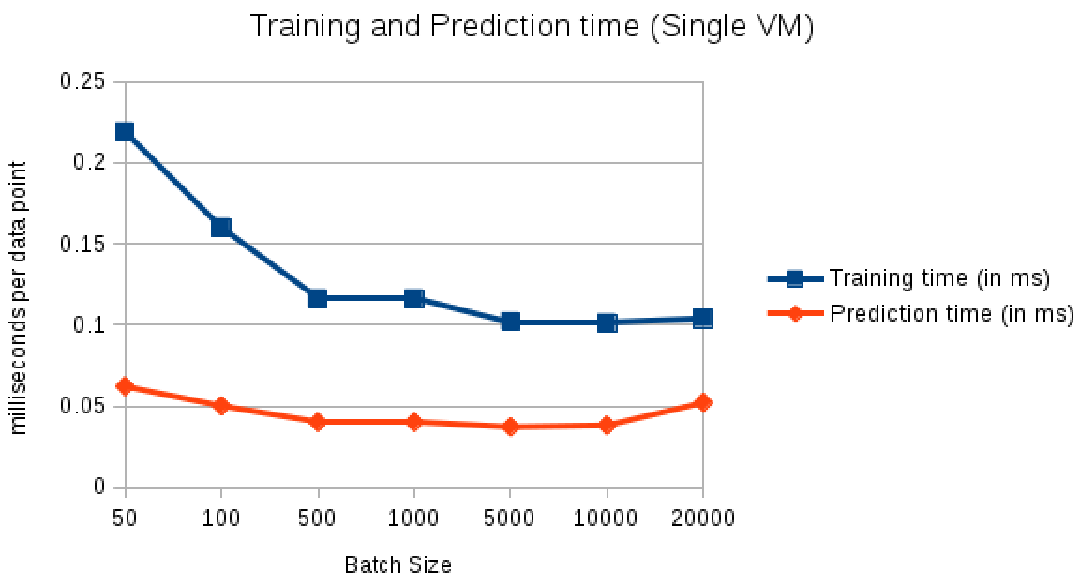
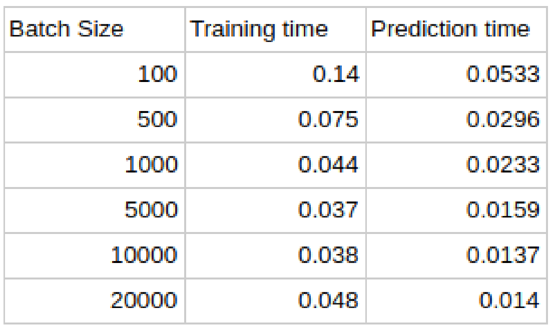


Figure 6.2: Performance in Single VM case

6.3 Data parallelism using parameter servers

In this approach, we use data parallelism as explained in previous section. We used parameter server - worker system provided inbuilt by tensorflow. Workers are the actual computation units which train their model using the incoming data. While parameter servers act as relay between the workers to communicate. Each worker has its own copy of the entire neural network. Instead of each worker communicating with every other worker (where the communication complexity will be O(n*n)), workers send their model updates to relay (Parameter Server) and it merges all updates and sends it to all other nodes (communication complexity becomes O(n)). We need more parameter servers for faster communication. Following are the results when we use system of 2 parameter servers and 3 workers -



Batch Size	Training time	Prediction time
100	0.14	0.0533
500	0.075	0.0296
1000	0.044	0.0233
5000	0.037	0.0159
10000	0.038	0.0137
20000	0.048	0.014

Figure 6.3: 2 Parameter servers and 3 workers. Time in *milliseconds per data point*

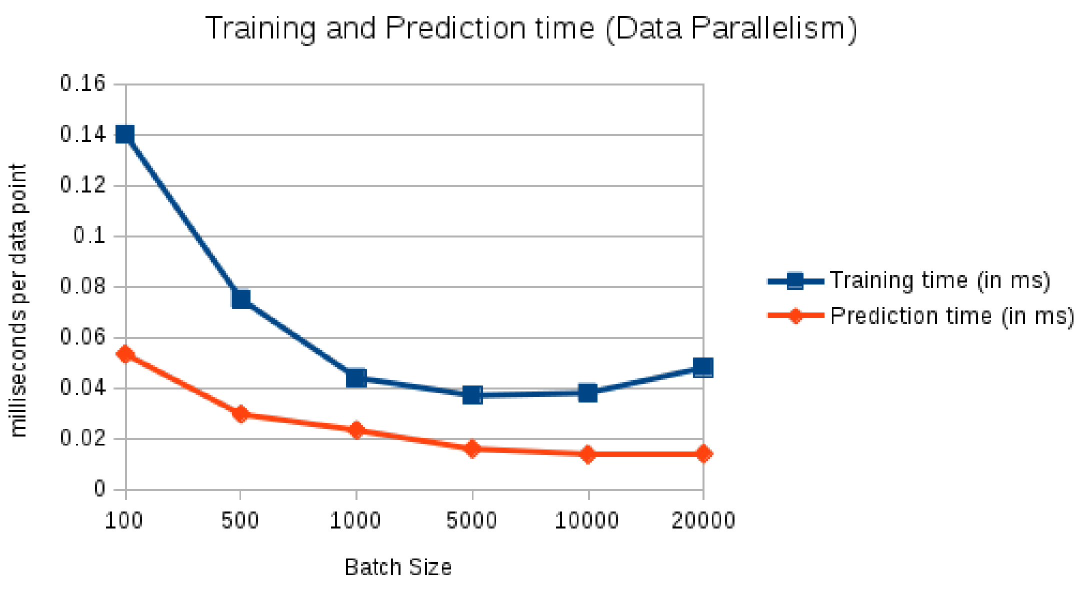
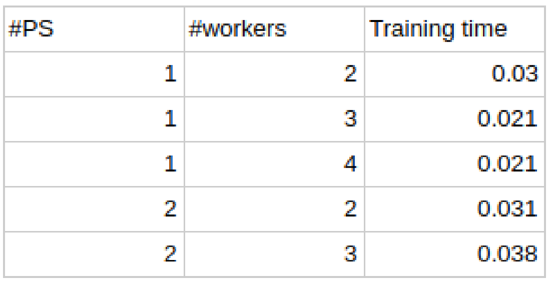


Figure 6.4: Performance in data parallelism case (2 Parameter servers and 3 workers)

We next vary the number of parameter servers and workers (keeping the batch size constant) to figure out their effect on the training time. Following are the results when batch size was fixed at 10000 -



#PS	#workers	Training time
1	2	0.03
1	3	0.021
1	4	0.021
2	2	0.031
2	3	0.038

Figure 6.5: Batch size = 10000. Time in *milliseconds per data point*

6.4 Model parallelism (layerwise partition)

As mentioned previously, with increase in memory on GPUs (edge nodes in our case), the majority of distributed training has shifted towards data parallelism. Model parallelism is however useful when the deep neural network is huge and does not fit in the memory of edge device. The model for AQI prediction is sufficiently small to fit into the memory of edge devices. We however run experiments to see the performance of this approach. We partition the neural network layer-wise. We use a total of 5 VMs and assign one layer to each VM. For a batch size of 100 data points, it took *68.32 ms* for training and *18.31 ms* for prediction per data point.

6.5 Hierarchical data parllelism

In the data parallelism implementation provided by tensorflow, every worker shares their model with every other worker effectively. We anticipate thousands of geographically distributed workers each having a model trained by local data. Approach using the parameter servers may not scale well. What we would prefer is to have groups of workers who share their model amongst themselves via a parameter server. Less frequently than this, these groups can share with other groups and so on organized hierarchically. Tensorflow limits our flexibility and it is not possible to have such custom sharing rules in it. Also we would like to look at costs of network communication. We would prefer greater flexibility and control over the environment and training/sharing algorithm. Hence there was a need to implement our own simulation environment. The simulator should to configurable to accommodate customization in every aspect of data sharing and distributed training if neural networks. We present the simulator implemented so far in the next section.

Chapter 7

Hierarchal data parallelism (Simulator)

7.1 Idea

Tensorflow limits our control over the data/model sharing schemes in distributed training of neural networks. It was not possible to have the hierarchy of parameter servers for data parallelism as explained towards the end of last section. We would like to have greater flexibility in changing the aspects of simulation environment. Hence we built a simulator which can be used test the performance of sharing strategy for training a hierarchy of nodes.

7.2 Training the hierarchy of nodes

The nodes (VMs) are arranged in a tree like structure. The leaf nodes receive the data from various sources. They train their own copy of the models. After every specified number of epochs, they send the gradient acquired so far to their parent node and obtain a latest copy of the model from the parent. The parent nodes repeat the same process with their respective parents and so on.

7.2.1 Algorithm

[6] provides an algorithm named Downpour Stochastic Gradient Descent for data parallelism. We use it for sharing the models/gradients up and down the levels in the tree.

```
Algorithm 1.1: DownpourSgdClient($\alpha, n_{fetch}, n_{push}$)

procedure StartAsynchronouslyFetchingParameters($parameters$)
  $parameters \leftarrow$ GetParametersFromParamServer()

procedure StartAsynchronouslyPushingGradients($accruedgradients$)
  SendGradientsToParamServer($accruedgradients$)
  $accruedgradients \leftarrow 0$

main
  global $parameters, accruedgradients$
  $step \leftarrow 0$
  $accruedgradients \leftarrow 0$
  while $true$
     do
        if ($step$ mod $n_{fetch}$) == 0
          then StartAsynchronouslyFetchingParameters($parameters$)
        $data \leftarrow$ GetNextMinibatch()
        $gradient \leftarrow$ ComputeGradient($parameters, data$)
        $accruedgradients \leftarrow accruedgradients + gradient$
        $parameters \leftarrow parameters - \alpha * gradient$
        if ($step$ mod $n_{push}$) == 0
          then StartAsynchronouslyPushingGradients($accruedgradients$)
        $step \leftarrow step + 1$
```

Figure 7.1: Downpour SGD algorithm [6]

7.3 Simulator - Inside Look

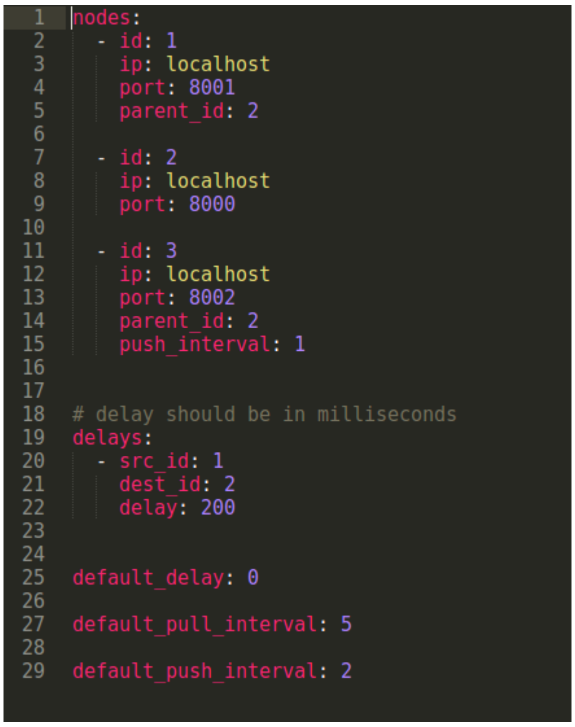
7.3.1 Configurations

Simulator takes in a yaml configuration file as an input. This file mentions all the customizable variables of the simulation environment.

The configuration file lists all the nodes present in the system. For each node it provides a unique id, its IP address (eg. *localhost*) and the port on which the RPC server corresponding to that node is to be run. The ID of the parent node (if present) is also specified.

The delays between the nodes (simulating network latencies) can also be mentioned. Also the interval for pulling the updated model from parent node and interval for pushing the acquired gradients to the parent node can also be customized.

The detailed documentation of the spec file can be found in the sample specification file present in the code repository.



```
nodes:
  - id: 1
    ip: localhost
    port: 8001
    parent_id: 2

  - id: 2
    ip: localhost
    port: 8000

  - id: 3
    ip: localhost
    port: 8002
    parent_id: 2
    push_interval: 1


# delay should be in milliseconds
delays:
  - src_id: 1
    dest_id: 2
    delay: 200


default_delay: 0

default_pull_interval: 5

default_push_interval: 2
```

Figure 7.2: A sample specification file

7.3.2 Implementation

Code of the current implementation of the simulator can be found at:
https://github.com/govindlahoti/TreeNN

The simulator has been coded in python. It does not use any external library as of now. At present, the code of the simulator does the following-

1. Reads in the configuration file passed to it as command line argument.

2. After reading the specs from the configuration file, it spawns the nodes on separate threads. Each node thread further spawns threads for its RPC server. We use built-in SimpleXMLRPCServer library for this purpose. RPC is used for all the inter-node communication. Functionalities of nodes like pushing/pulling the gradients/model has been abstracted out using RPC functions.

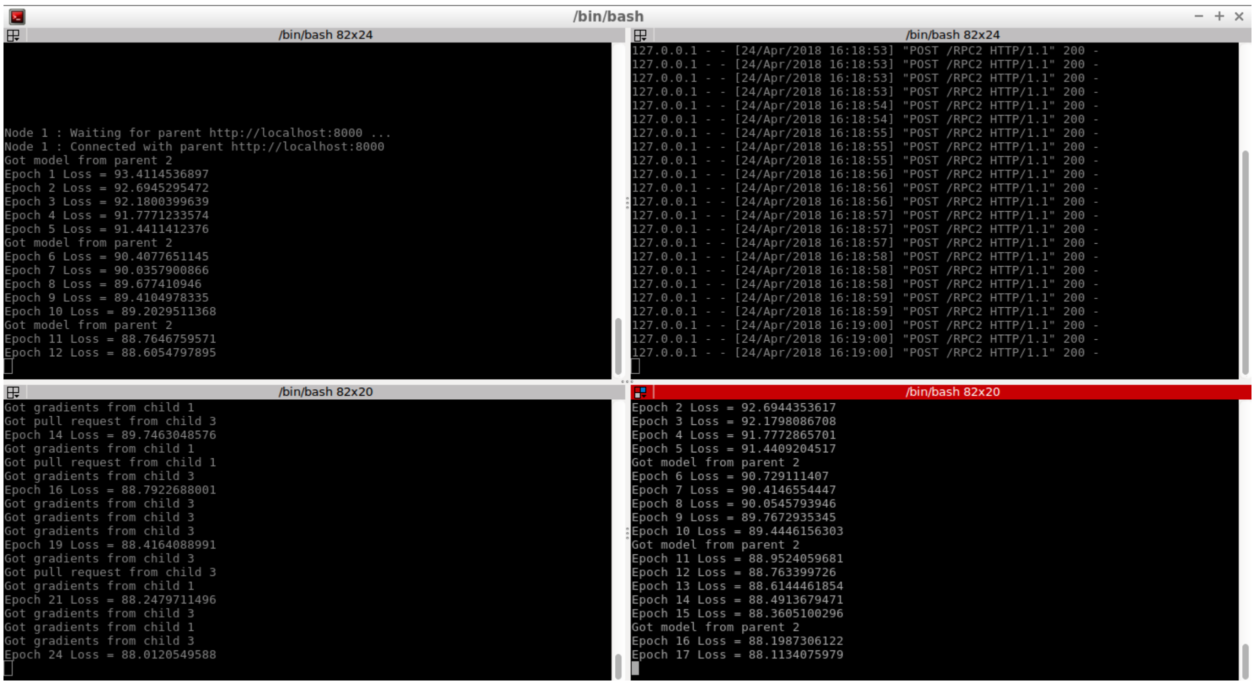
3. Each leaf node begins reading from the stream provided to it (as mention in the configuration file). Each node trains their model using the downpour SGD algorithm as explained previously.

4. Each node pushes its gradients to the parent node and pulls the model from the parent node at intervals specified in the configruation file.

5. Delays are inserted in the RPC call to simulate the network latencies between the nodes.

6. Each node logs all the events taking place on it. These logs can be processed later after the simulation run is over to obtain the desirable analytics.

Detailed documentation of the code can be found in the repository itself. Following is the screenshot of a sample run of the simulator -



```
Node 1 : Waiting for parent http://localhost:8000 ...
Node 1 : Connected with parent http://localhost:8000
Got model from parent 2
Epoch 1 Loss = 93.4114536897
Epoch 2 Loss = 92.6945295472
Epoch 3 Loss = 92.1800399639
Epoch 4 Loss = 91.7771233574
Epoch 5 Loss = 91.4411412376
Got model from parent 2
Epoch 6 Loss = 90.4077651145
Epoch 7 Loss = 90.0357900866
Epoch 8 Loss = 89.677410946
Epoch 9 Loss = 89.4104978335
Epoch 10 Loss = 89.2029511368
Got model from parent 2
Epoch 11 Loss = 88.7646759571
Epoch 12 Loss = 88.6054797895
```

```
127.0.0.1 - - [24/Apr/2018 16:18:53] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:53] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:53] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:53] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:54] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:54] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:55] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:55] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:55] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:56] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:56] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:56] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:57] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:57] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:57] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:58] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:58] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:58] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:59] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:18:59] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:19:00] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:19:00] "POST /RPC2 HTTP/1.1" 200 -
127.0.0.1 - - [24/Apr/2018 16:19:00] "POST /RPC2 HTTP/1.1" 200 -
```

```
Got gradients from child 1
Got pull request from child 3
Epoch 14 Loss = 89.7463048576
Got gradients from child 1
Got pull request from child 1
Got gradients from child 3
Epoch 16 Loss = 88.7922688001
Got gradients from child 3
Got gradients from child 3
Got gradients from child 3
Epoch 19 Loss = 88.4164088991
Got gradients from child 3
Got pull request from child 3
Got gradients from child 1
Epoch 21 Loss = 88.2479711496
Got gradients from child 3
Got gradients from child 1
Got gradients from child 3
Epoch 24 Loss = 88.0120549588
```

```
Epoch 2 Loss = 92.6944353617
Epoch 3 Loss = 92.1798086708
Epoch 4 Loss = 91.7772865701
Epoch 5 Loss = 91.4409204517
Got model from parent 2
Epoch 6 Loss = 90.729111407
Epoch 7 Loss = 90.4146554447
Epoch 8 Loss = 90.0545793946
Epoch 9 Loss = 89.7672935345
Epoch 10 Loss = 89.4446156303
Got model from parent 2
Epoch 11 Loss = 88.9524059681
Epoch 12 Loss = 88.763399726
Epoch 13 Loss = 88.6144461854
Epoch 14 Loss = 88.4913679471
Epoch 15 Loss = 88.3605100296
Got model from parent 2
Epoch 16 Loss = 88.1987306122
Epoch 17 Loss = 88.1134075979
```

Figure 7.3: View of test run of the simulator showing the logs of all the nodes being generated simultaneously.

Chapter 8

Future Work

The work done in this project can be potentially extended in several ways. Few directions are listed below -

1. Further research on all the ways of model and data sharing for training the neural network is required. The simulator built as part of this project can be used to analyze the performance of heuristics. Experiments need to be conducted to figure out the optimal sharing strategy minimizing the communication costs and maximizing the accuracy of the system.

2. The simulator code needs to be modified further to cater to our needs. It misses out several aspects in simulation which otherwise become evident in practical setting. Also the model being used in simulation (i.e. DNN) needs to be modified to the actual AQI model.

3. There is also a need for trying out the proposed heuristics practically. We picked up AQI monitoring problem to practically try out our heuristics. However we are still using the self-built simulator to test our heuristics. There is need to practically deploy sensors and edge devices and experiment.

4. Further work is required on distributed and partially aware schedulers. The scale of IoT setup can be huge. It may not be feasible for any node (or any central entity) to be aware of state of entire IoT setup. This leads to partial observability of nodes. Since there are communication costs in the network, each node would like to further keep its observability to the minimum. Schedulers(running on nodes) need to smartly communicate and coordinate among themselves to make efficient resource allocation.

5. Further work is also required in removing the left out assumptions in our setup. Work in previous stage of BTP assumed that the meta information of the upcoming workflows in the future is known in advance. This may not be the case practically. A more dynamic scheduler catering to this aspect needs to be developed.

Chapter 9

Conclusion

There is wide scope of research in federating analytics towards the edge. With improvement in technology, edge devices are becoming more powerful and cheaper with easy passing day. They can be efficiently used to carry out analytics which otherwise are done in Cloud. This would help us reduce the latency in decision making as well as the cost for network communication. The work done in this project touches upon various aspects of scheduling the Lambda functions on the edge of IoT. We evolve the strategies for simpler models to work for more complex realistic models. We then take a practical application of AQI monitoring and explore ways to federate the training of underlying neural networks on the edge. We explore various ways of data and model parallelism and their performances. We also build a simulation environment to test the performance of heuristics we can come up with in the future. The simulation environment can be further extended to bring in all the practical constraints present in real life IoT.

Acknowledgement

I wish to express my sincere gratitude and indebtedness to Prof. Umesh Bellur and Prof. Vinayak Naik for their constant guidance throughout the project.

Bibliography

[1] M. Xu, L. Cui, H. Wang, and Y. Bi. A multiple qos constrained scheduling strategy of multiple workflows for cloud computing. In *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 629–634, Aug 2009.

[2] S. Wang, M. Zafer, and K. K. Leung. Online placement of multi-component applications in edge computing environments. *IEEE Access*, 5:2514–2533, 2017.

[3] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, Nov 2016.

[4] Pooja Vyavahare, Nutan Limaye, and D. Manjunath. Efficient embedding of functions in weighted communication networks. *CoRR*, abs/1401.2518, 2014.

[5] Nithyashri Govindarajan, Yogesh L. Simmhan, Nitin Jamadagni, and Prasant Misra. Event processing across edge and the cloud for internet of things applications. In *COMAD*, 2014.

[6] Jeffrey Dean, Gregory S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[7] D. Shrivastava, S. Chaudhury, and D. Jayadeva. A Data and Model-Parallel, Distributed and Scalable Framework for Training of Deep Networks in Apache Spark. *ArXiv e-prints*, August 2017.

[8] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. Distributed deep neural networks over the cloud, the edge and end devices. *CoRR*, abs/1709.01921, 2017.

[9] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. *CoRR*, abs/1802.04924, 2018.

[10] Alka Bhushan. Problem statement - aqi prediction, url: http://bit.ly/2i06k6f. 2018.

[11] Yu Zheng, Furui Liu, and Hsun-Ping Hsieh. U-air: When urban air quality inference meets big data. August 2013.

[12] Hsun-Ping Hsieh, Shou-De Lin, and Yu Zheng. Inferring air quality for station location recommendation based on urban big data. In *KDD*, 2015.

[13] Yu Zheng, Xiuwen Yi, Ming Li, Ruiyuan Li, Zhangqing Shan, Eric Chang, and Tianrui Li. Forecasting fine-grained air quality based on big data. In *KDD*, 2015.

[14] Blog: An introduction to distributed training of neural networks, url: http://bit.ly/2hk2ygx. 2017.

Chapter 10

Appendix

Following are the results for **Version 1 (node first, then task)**. 50 nodes where generated at random with capabilities chosen from $\mathcal{N}(75, 15)$. 1000 tasks were generated at random with resource requirements $\mathcal{N}(50, 10)$. Similarly 2500 tasks from $\mathcal{N}(30, 15)$ and 1500 tasks from $\mathcal{N}(10, 2)$ were generated.
For the following table, node was selected on the basis of least amount work load. Please refer to main text for more context.

Strategy to select task	**Average time on machines**	**Max time on any machine**	**Avg. occupancy of resources on machines**
Random	8001.50	8261	0.78
Lowest time	7835.10	8046	0.80
Highest time	7139.32	7149	0.87
Lowest norm of resources required	8528.49	8930	0.73
Highest norm of resources required	6883.94	7029	0.90
Highest volume (cpu*mem*time)	6889.20	6905	0.90
Highest Area norm(cpu, mem) * time	6843.24	6854	0.91
Highest cosine similarity of required and available resources	7371.56	7567	0.84
Tetris	6953.89	7057	0.90

For the following table, node was selected on the basis of highest free resources. Please refer to main text for more context.

Strategy to select task	**Average time on machines**	**Max time on any machine**	**Avg. occupancy of resources on machines**
Random	8278.0	11083	0.77
Lowest time	8103.10	10335	0.79
Highest time	7138.56	15343	0.89
Lowest norm of resources required	8813.80	11463	0.72
Highest norm of resources required	7424.96	18630	0.89
Highest volume (cpu*mem*time)	20000	Very large	Very large
Highest area norm(cpu, mem) * time	7207.84	18868	0.90
Highest cosine similarity of required and available resources	7745.98	13144	0.83

Following are the results for **Version 1 (task first, then node)** The experimental setup was same as above. Please refer to main text for more context.

How was node selected?	**How was task selected?**	**Avg. busy time of machine**	**maximum busy time of machine**	**Avg. occupancy of machine**
Random	Highest time	5920.62	12290	0.77
	Highest norm	5576.34	10541	0.79
	Highest area	5920.62	12290	0.77
Least earliest possible time at any machine	Highest time	5350.36	7024	0.9
	Highest norm	4167.56	7049	0.89
	Highest area	4518.8	6950	0.9
Least (1 - cosine similarity between resources left and required) * earliest possible time	Highest time	5694.2	11830	0.87
	Highest norm	5238.02	12981	0.86
	Highest area	4998.66	10810	0.87
Least (1 - 0.5 *cosine similarity between resources left and required) * earliest possible time	Highest time	6018.66	7274	0.89
	Highest norm	4148.74	7297	0.89
	Highest area	4922.36	7377	0.9
Least (1 - 0.5 * fraction of resources free) * earliest possible time	Highest time	6691.24	8825	0.78
	Highest norm	4244.06	8316	0.83
	Highest area	5992.9	9096	0.8
Least earliest possible time * fraction of resources free	Highest time	4271.58	6940	0.89
	Highest norm	3774.58	7288	0.88
	Highest area	5108.46	7474	0.89
Least (1 + 0.5 * fraction of resources free) * earliest possible time	Highest timee	4972.78	6960	0.89
	Highest norm	3929.04	7124	0.88
	Highest area	4930.74	7384	0.9

Following are the results for **Version 2**. Experimental setup had 10 workflows and 110 machines. Number of tasks in workflows were [10, 10, 10, 10, 5, 5, 5, 5, 2, 2]. Intervals at which each workflow were triggered are [3, 3, 2, 2, 2, 2, 1, 1, 2, 1]. Machine capabilities were drawn from $\mathcal{N}(80, 15)$. While task requirements were drawn from $\mathcal{N}(20, 10)$. Task time requirement was drawn from $\mathcal{N}(10, 8)$ Task was selected first, then the node. Please refer to main text for more context.

Task selection: Highest duration first

Machine selection	**Avg. % extra latency of workflows**	**Avg. % occupancy of machines**
Random	190.88	64.39
Least earliest possible time at any machine	26.70	81.77
Least (1 - 0.5 * cosine similarity between resources left and required) * earliest possible time	15.28	83.76
Least (1 - 0.5 * fraction of resources free) * earliest possible time	31.79	81.10
Least (1 + 0.5 * fraction of resources free) * earliest possible time	27.21	81.62

Task selection: Least duration first

Machine selection	**Avg. % extra latency of workflows**	**Avg. % occupancy of machines**
Random	190.91	64.47
Least earliest possible time at any machine	29.69	81.27
Least (1 - 0.5 * cosine similarity between resources left and required) * earliest possible time	16.63	83.56
Least (1 - 0.5 * fraction of resources free) * earliest possible time	29.86	81.55
Least (1 + 0.5 * fraction of resources free) * earliest possible time	28.42	81.38

Task selection: Highest norm first

Machine selection	**Avg. % extra latency of workflows**	**Avg. % occupancy of machines**
Random	195.83	64.21
Least earliest possible time at any machine	31.69	81.26
Least (1 - 0.5 * cosine similarity between resources left and required) * earliest possible time	23.75	82.52
Least (1 - 0.5 * fraction of resources free) * earliest possible time	36.22	80.74
Least (1 + 0.5 * fraction of resources free) * earliest possible time	29.90	81.41

Task selection: Least norm first

Machine selection	**Avg. % extra latency of workflows**	**Avg. % occupancy of machines**
Random	190.04	64.62
Least earliest possible time at any machine	18.95	83.04
Least (1 - 0.5 * cosine similarity between resources left and required) * earliest possible time	**9.95**	**84.66**
Least (1 - 0.5 * fraction of resources free) * earliest possible time	19.10	83.07
Least (1 + 0.5 * fraction of resources free) * earliest possible time	17.13	83.25