Cluster Allocation for Kernel Execution(CAKE) using Artificial Intelligence*

Ajay Ramesh[1] and Chandra Kumar Basavaraju[2]

***Abstract*— A good plan consumes all available facts during decision making. An excellent plan can be intelligent if it takes further inferences from available facts. In short, we are talking about Artificial Intelligence in Scheduling. We see AI is the new electricity for any application. Scheduling is the key for an Application and Organization Success. A good scheduling will make the application faster if and only if the scheduling is fast. Organization benefits if scheduling maximizes the cluster utilization. In short, scheduling's key idea is to maximize cluster utilization use and minimize latency. Existing framework does proper scheduling but uses the only tiny portion of Knowledge. This paper summarizes our study of many scheduler architectures, and our own design of new pluggable scheduler framework called CAKE, which uses Genetic Algorithm, an Artificial Intelligence technique, for scheduling. We have presented our thoughts about why we need CAKE framework and we have also mentioned few self critics for it. We have also given proof of concept for CAKE using Genetic Algorithm. Our research notes section describes why advanced concepts (AI,etc) in scheduling must be unified by all researchers in Apache Open Source project, such that any application or organization will not worry about scheduling at all.**

We started with the study of various schedulers and then we have created a POC in Python (source & docs available at[22]). We concluded our paper with some research notes and ideas for future work.

This paper is organized as -

- Lterature Survey in Section I
- Proof Of Concept of CAKE in Section II
- Research Notes and Conclusion in Section III

We have listed our contributions at the end.

I. Literature Survey

A. *MESOS*

MESOS is a two-level scheduler architecture. In short, Mesos offers resources to another scheduler/framework. The framework will pick/rejects resources from offered resources. It adresses some of the disadvantages of Monolithic Scheduler. Monolithic Schedulers doesn't separate resource allocation and task placement. Mesos, abstracts CPU, memory, storage, and other estimate resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to be built and run efficiently[4].

Design Goals: To create a scalable and efficient scheduling system that supports a broad range of both current and future frameworks.

Critical challenges Mesos addresses: Each framework (ex Hadoop or MPI etc) will have different scheduling needs, based on its programming model, communication pattern, task dependencies, and data placement. Mesos offers the needs of the frameworks.

Design Summary: Mesos is a thin management layer that allows diverse cluster computing frameworks to share resources efficiently. The main design elements of Mesos are: fine-grained sharing model at the level of tasks, and a distributed scheduling mechanism called resource offers, that delegates scheduling decisions to the frameworks.

Disadvantages:

- Priority preemption becomes difficult to implement in the offer based model[23]. The resource offered by running tasks are not visible to the upper-level schedulers in this model.
- Schedulers are unable to consider the interference from running workloads that degrades

*This work (cakeframework.readthedocs.io) was done as part of the academic project for the course CS550 - Advanced Operating Systems at Illinois Institute of Technology

[1]Ajay Ramesh is a Masters Student in Computer Science department at Illinois Institute of Technology, Chicago . aramesh6 at hawk.iit.edu

[2]Chandra Kumar Basavaraju is a Masters Student in Computer Science department at Illinois Institute of Technology, Chicago. cbasavaraju at hawk.iit.edu

quality, because they can't see them. Here, quality can be I/O bandwidth or power etc.
- Application-specific schedulers care about many different aspects of the underlying resources, but their only means of choosing resources is the offer/request interface with the resource manager. This interface can quickly become quite complicated

Research Note: How to make offer/request interface simpler? Integration of many more framework.

B. ***Cooperative Batch Scheduling for HPC Systems***

We learned that batch scheduling is a necessary requirement in supercomputer clusters/HPC clusters. In short batch scheduling allows users to submit their jobs via batch scheduling portal and the batch scheduler makes scheduling decision for each task based on its request for computing sources, i.e. core-hours.

Usually, the jobs submitted to HPC systems are parallel applications, and their lifecycle consists of multiple running phases, such as computation, communication, and I/O. The running of a job could involve different kinds of system resources, such as power, network bandwidth, I/O bandwidth, storage, etc. Traditional batch schedulers rarely take these resource requirements into consideration for making scheduling decisions, this is recognized as one of the primary culprits for system-wide performance loss.[18]

Highlights

In this study, they evaluated the power aware scheduling design via extensive trace-based simulations and a case study on Mira. They presented a series of experiments comparing their plan against the popular first-come, first-serve scheduling policy, with backfilling done in three different aspects (electricity bill saving, scheduling performance, and job performance). Preliminary results demonstrate that the design can cut electricity bill by up to 23% without an impact on overall system utilization.[18]

Takeaways

- Batch scheduling
- Scheduling based on different types of constraints are important in achieving performance and cost reduction. We see below concepts should consider while designingmetrics for a good scheduler design.
  - Energy Cost-aware Scheduling
  - Locality-aware Scheduling
  - Topology-aware Job Scheduling
    - Dragonfly Network
    - Torus Network

Research notes on : How to extend a scheduler to consider Energy Cost, Fragmented Allocation and Network Contention ?

C. ***IO-Aware Batch Scheduling for Petascale Computing Systems***

This work addresses the I/O congestion problem at the level of batch scheduling. To be more specific, this work aims to answer the following question: if a batch scheduler is aware of ongoing I/O requests from multiple jobs, will it be able to mitigate the I/O congestion issue by coordinating different I/O requests? The batch scheduler is a good candidate to handle I/O congestion due to the following reasons:

- The batch scheduler is a global controller of all user jobs. It has a comprehensive and high-level knowledge of user jobs, and can initiate, suspend, terminate, or restart user jobs when they are submitted to the system.
- Batch scheduler often contains a monitoring component to collect abundant information of the system and user job status (e.g., node utilization, bandwidth usage, sensor data, etc.) from various sources

D. ***Scheduling in High Performance Computing Clusters***

Clusters are divided into two major classes:

- High-Throughput Computing clusters.
- High-Performance computing clusters.

High-throughput clusters usually connect a large number of nodes using low-end interconnects. In contrast, high-performance computing clusters connect more powerful compute nodes using faster interconnects than high-throughput computing clusters. Fast interconnects are designed to provide lower latency and higher bandwidth than low-end interconnects.

These two classes of clusters have different scheduling requirements. In high-throughput computing clusters, the main goal is to maximize throughput, that is jobs completed per unit of time, by reducing load imbalance among compute nodes in the cluster. In high-performance computing clusters, an additional consideration arises: the need to minimize communication overhead by mapping applications appropriately to the available compute nodes.

High-throughput computing clusters are suitable for executing loosely coupled parallel or distributed applications, because such applications do not have high communication requirements among compute nodes during execution time. High-performance computing clusters are more suitable for tightly coupled parallel applications, which have substantial communication and synchronization requirements.

A resource management system manages the processing load by preventing jobs from competing from each other for limited compute resources. Typically, a resource management system comprises a resource manager and a job scheduler.

When a job is submitted to a resource manager, the job waits in a queue until it is scheduled and executed. The time spent in the queue, or wait time, depends on several factors including job priority, load on the system, and availability of requested resources. Turnaround time represents the elapsed time between when job is submitted and when the job is completed; turnaround time includes the wait time as well as the jobs actual execution time. Response time represents how fast a user receives a response from the system after the job is submitted. Resource utilization during the lifetime of the job represents the actual useful work that has been performed. System throughput is defined as the number of jobs completed per unit of time. Mean response time e is an important performance metric for users, who expect minimal response time. System administrators are concerned with overall resource utilization because they want to maximize system throughput and return on investment (ROI), especially in high-throughput computing clusters.

In a typical production environment, many different jobs are submitted to clusters. These jobs can be characterized by factors such as the number of processors requested (also known as job size, or job width), estimated runtime, priority level, parallel or distributed execution, and specific I/O requirements. During execution, large jobs can occupy significant portions of a clusters processing and memory resources.

In high-performance computing clusters, the scheduling of parallel jobs requires special attention because parallel jobs comprise several subtasks. Each subtask is assigned to a unique compute node during execution and nodes constantly communicate among themselves during execution. The manner in which the subtasks are assigned to processors is called mapping. Because mapping affects execution time, the scheduler must map subtasks carefully.

The parallel and distributed computing community has put substantial research effort into developing and understanding job scheduling algorithms. Today, several of these algorithms have been implemented in both commercial and open source job schedulers. Scheduling algorithms can be broadly divided into two classes:

- time-sharing
- space-sharing

Tme-sharing g algorithms divide time on a processor into several discrete intervals, or slots. These slots are then assigned to unique jobs. Hence, several jobs at any given time can share the same compute resource. Conversely, space-sharing algorithms give the requested resources to a single job until the job completes execution. Most cluster schedulers operate in space sharing mode.

Common, simple space-sharing algorithms are:

- First Come, First Served (FCFS)
- First In, First Out (FIFO)
- Round Robin (RR)
- Shortest Job First (SJF)
- Longest Job First (LJF).

As the names suggest, FCFS and FIFO execute jobs in the order in which they enter the queue. This is a very simple strategy to implement, and works acceptably well with a low job load. RR assigns jobs to nodes as they arrive in the queue in a cyclical, round-robin manner. SJF periodically sorts the incoming jobs and executes the shortest job first, allowing short jobs to get a good

turnaround time. However, this strategy may cause delays for the execution of long (large) jobs. In contrast, LJF commits priority jobs. resources to longest jobs first. The LJF approach tends to maximize system utilization at the cost of turnaround time.

Basic scheduling algorithms such as these can be enhanced by combining them with the use of advance reservation and backfill techniques. Advance reservation uses execution time predictions provided by the users to reserve resources (such as CPUs and memory) and to generate a schedule. The backfill technique improves space-sharing scheduling. Given a schedule with advance-reserved, high-priority jobs and a list of low-priority jobs, a backfill algorithm tries to fit the small jobs into scheduling gaps. This allocation does not alter the sequence of jobs previously scheduled, but improves system utilization by running low-priority jobs in between high-priority jobs. To use backfill, the scheduler requires a runtime estimate of the small jobs, which is supplied by the user when jobs are submitted.

E. ***Quasar***

Quasar uses machine learning technique to schedule. This model is similar to movie recommendations. Once application performance constraints are specified, it is up to Quasar to find a resource allocation and assignment that satisfies them.

Quasar uses fast classification techniques to quickly and accurately estimate the impact, different resource allocation and resource assignment decisions have on workload performance. The approach described in the paper[2] takes few amount of data from the application then combine those into the large group of performance data, and then apply an algorithm to find out which resources needs to set aside to get the best performance. They used classification algorithms to determine the impact of different resourse allocations and assignments on workload. What this means is it takes a few samples of a new application, then stops the application, compares the samples against other previously ran applications, and makes resource assignment assumptions based on this combined representation. It takes these samples across four vectors scale-up, scale-out, interference, and heterogeneity, and passes the output to a scheduler. We can think like say we need 1000 numbers to describe the behavior of the program and only get one or two. The way we get the other 998 is using classification.

Quasar can handle dynamic workloads, allowing it to assign new resourses as and when the applications grows or shrinks. The scheduler's primary objective is to allocate the least amount of resources needed to satisfy a workload's performance. That means Scheduler can't comprimise in performance. So it must satisfly the requested workload performance. Quasar improves resource utilization by 47% in the 200-server EC2 cluster, while meeting performance constraints for workloads of all types.

- Take away
  - Model like movie recommendation works for Scheduling also.
  - Amazing results.
  - Performance Constraint is difficult model.
- Disadvantages
  - Complex to genereate those model
  - Usually defining the performance constraints is difficult.

Short summary of Quasar Study Quasar, a cluster management system that performs coordinated resource allocation and assignment. So if you want to tell what performance I need rather than telling I need these many resources, then we can go ahead with this approach. For example the job requires to completed in 1 hr, then scheduler allocates the resources as per that. Quasar currently supports distributed analytics frameworks, web-serving applications, NoSQL datastores, and single-node batch workloads.

F. ***Paragon***

Paragon is an online and scalable Data Center scheduler that is heterogeneity and interference-aware. Paragon is derived from robust analytical methods and instead of profiling each application in detail, it leverages information the system already has about applications it has previously seen. It uses collaborative filtering techniques to

quickly and accurately classify an unknown, incoming workload with respect to heterogeneity and interference in multiple shared re- sources, by identifying similarities to previously scheduled applications. The classification allows Paragon to greedily schedule applications in a manner that minimizes interference and maximizes server utilization. Paragon scales to tens of thousands of servers with marginal scheduling overheads in terms of time or state.

*G. **Borg***

Googles Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.

Borg admits, schedules, starts, restarts, and monitors the full range of applications that Google runs. Borg provides three main benefits: it (1) hides the details of resource management and failure handling so its users can focus on application development instead; (2) operates with very high reliability and availability, and supports applications that do the same; and (3) lets us run workloads across tens of thousands of machines effectively.

Users submit their work to Borg in the form of jobs, each of which consists of one or more tasks that all run the same program (binary). Each job runs in one Borg cell, a set of machines that are managed as a unit. The machines in a cell belong to a single cluster, defined by the high-performance datacenter-scale network fabric that connects them. A cluster lives inside a single datacenter building, and a collection of buildings makes up a site.

Borg jobs properties include its name, owner, and the number of tasks it has. Jobs can have constraints to force its tasks to run on machines with particular attributes such as processor architecture, OS version, or an external IP address. Constraints can be hard or soft; the latter act like preferences rather than requirements. The start of a job can be deferred until a prior one finishes. A job runs in just one cell. A user can change the properties of some or all of the tasks in a running job by pushing a new job configuration to Borg, and then instructing Borg to update the tasks to the new specification.

A Borg cell consists of a set of machines, a logically centralized controller called the Borgmaster, and an agent process called the Borglet that runs on each machine in a cell. A Borg cell consists of a set of machines, a logically centralized controller called the Borgmaster, and an agent process called the Borglet that runs on each machine in a cell.

The Borgmaster is logically a single process but is actually replicated five times. Each replica maintains an inmemory copy of most of the state of the cell, and this state is also recorded in a highly-available, distributed, Paxos-based store on the replicas local disks

The Borglet is a local Borg agent that is present on every machine in a cell. It starts and stops tasks; restarts them if they fail; manages local resources by manipulating OS kernel settings; rolls over debug logs; and reports the state of the machine to the Borgmaster and other monitoring systems.

When a job is submitted, the Borgmaster records it persistently in the Paxos store and adds the jobs tasks to the pending queue. This is scanned asynchronously by the scheduler, which assigns tasks to machines if there are sufficient available resources that meet the jobs constraints. (The scheduler primarily operates on tasks, not jobs.) The scan proceeds from high to low priority, modulated by a round-robin scheme within a priority to ensure fairness across users and avoid head-of-line blocking behind a large job. The scheduling algorithm has two parts: feasibility checking, to find machines on which the task could run, and scoring, which picks one of the feasible machines.

In feasibility checking, the scheduler finds a set of machines that meet the tasks constraints and also have enough available resources which includes resources assigned to lower-priority tasks that can be evicted. In scoring, the scheduler determines the goodness of each feasible machine. The score takes into account user-specified preferences, but is mostly driven by built-in criteria such as minimizing the number and priority of preempted tasks, picking machines that already have a copy of the tasks packages, spreading tasks across power and failure domains, and packing quality including putting a mix of high and low priority tasks onto a single machine to allow the high-priority ones to expand in a load spike.

H. *Omega*

Omega is a parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, to achieve both implementation extensibility and performance scalability. Omega schedulers operate completely in parallel and do not have to wait for jobs in other schedulers, and there is no inter-scheduler head of line blocking. Each scheduler in Omega is granted full access to the entire cluster, wich allows them to compete in a free-for-all manner, and use optimistic concurrency control to mediate clashes when they update the cluster state. This approach eliminates the below issues of the two-level scheduler approach:

- limited parallelism due to pessimistic concurrency control.
- restricted visibility of resources in a scheduler framework.

I. *Apollo*

Apollo performs scheduling decisions in a distributed manner, utilizing global cluster information via a loosely coordinated mechanism. Each scheduling decision considers future resource availability and optimizes various performance and system factors together in a single unified model. Apollo is robust, which means to cope with unexpected system dynamics, and can take advantage of idle system resources gracefully while supplying guaranteed resources when needed.

J. *Hawk*

Hawk is a hybrid scheduler which addresses the problem of efficient scheduling of large clusters under high load and heterogeneous workloads. A heterogeneous work load typically consists of many short jobs and a small number of large jobs that consume the bulk of the clusters resources.

In Hawk, long jobs are scheduled using a centralized scheduler, while short ones are scheduled in a fully distributed way. Moreover, a small portion of the cluster is reserved for the use of short jobs. Inorder to compensate for the occasional poor decisions made by the distributed scheduler, Hawk uses efficient randomized work-stealing algorithm.

The rationale for Hawk's hybrid approach is as follows. First, there latively small number of long jobs does not overwhelm a centralized scheduler. Hence, scheduling latencies remain modest, and even a moderate amount of scheduling latency does not significantly degrade the performance of long jobs, which are not latency bound. Conversely, the large number of short jobs would overwhelm a centralizedscheduler, and the scheduling latency added by a centralized scheduler would add to what is already a latency-bound job. Second, by scheduling long jobs centrally, and by the fact that these long jobs takeup a large fraction of the cluster resources, the centralized scheduler has a good approximation of the occupancy of nodes in the cluster, eventhough it does not know where the large number of short jobs are scheduled.

The rationale for using randomized workstealing is based on the observation that, in a highly loaded cluster, choosing uniformly at random a loaded node from which to steal a task is very likely to succeed, while finding at random an idle node, as distributed schedulers attempt todo,is increasingly less likely to succeed as the slack in the cluster decreases.

A job is composed of a set of tasks that can run in parallel on different servers. Scheduling a job consists of assigning every task of that job to some server. Terms, long task and short tasks, refer to tasks belonging to long jobs or short jobs respectively. A job completes only after all its tasks finish. Each server has one queue of tasks. When a new task is scheduled on a server that is already running a task, the task is added to the end of the queue. The server queue management policy is FIFO.

Hawks goals are:

1. to run the cluster at highutilization.

2. to improve performance for short jobs, which are the most penalized ones in highly loaded clusters.

3. to sustain or improve the performance for long jobs.

To meet these challenges, Hawk relies on the following mechanisms:

1. reserve a small part of the cluster for short jobs. In other words, short jobs can run anywhere in the cluster, but long jobs can only run on a (large) subset of the cluster.

2. uses distributed scheduling for short jobs.

3. uses randomized work stealing, allowing idle

nodes to steal short tasks that are queued behind long tasks.

4. uses centralized scheduling for long jobs to maintain good performance for them, even in the face of reserving apart of the cluster for short jobs.

K. YARN

YARN is a two level scheduler, used in Apache Hadoop. YARN is used extensively and many big data application exist over on it. *Scheduling policies are limited to*

- First In First Out scheduling.
- FAIR scheduling.
- Dominant Resource Fairness scheduling.

Note: (In future we are planing to add our CAKE framework scheduling policy to YARN - refer Section III).

L. A Review of Machine Learning in Scheduling

We can see many types of learning being explored for scheduling, which includes rote learning, inductive learning (ID3), neural network learning, case-based learning, classifier systems, and others. While each particular methodology offers powerful and negative features, the summary point - these learning systems seem to improve a scheduling systems performance. Also, if the typical production environment is as dynamic as many believe them to be, the learning methods need to be dynamic themselves. Thus, fixed experiments for inductive learning may not be appropriate. Inherently dynamic learning methods are needed. So the experiments should be periodical, so the learning for application time t is not valid at time t+k, where t is the past time, k is gap time again the experiments should learn.

M. I/O-Aware Batch Scheduling for Petascale Computing Systems

I/O congestion caused by Concurrent storage accesses from multiple applications is certain, so it causes the performance issues. Traditional approaches either concentrate on optimizing an applications access pattern individually or handle I/O requests on low-level storage layer without any knowledge of the upper-level applications. In this paper they have considered a view of both system state and jobs activities and can control jobs status on the fly during their execution

N. Prediction of stability of the clusters in Manet using Genetic Algorithm

Clustering in mobile ad hoc network gained importance among researchers for easier maintenance of network. The problem in clustering of Manet is the frequent cluster re-affiliation and lack of stability of the network. In paper we see an brilliant approach to find the stability of the network using Genetic algorithm based on the average number of clusters, load balancing factors and weighted parameters.They have considered the factors the evaluate funtions such as density of the nodes,mobility,power and speed of the node.

O. Genetic Algorithms

A genetic algorithm is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms. Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.

In a genetic algorithm, a population of candidate solutions to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties which can be mutated and altered.

The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

Steps in Genetic Algorithms:

- Initialization
- Selection
- Crossover

- mutation
- Evaluate

Fitness function: allocates a score to each chromosome in the population.

Population: a set of candidate solutions

Selection: is the process of selecting population for the next generation. Candidates in the previous generation with high fitness score are more likely to be selected for the next generation. whereas those with low score are discarded. This phase has an element of randomness just like the survival of organisms in nature.

Most used selection methods are :

- Roulette-wheel
- Rank Selection
- Steady State selection.
- Truncation selection
- Stochastic universal sampling.
- Tournament selection.

Moreover, to improve the performace of Genetic Algorithms, the selection methods are enhanced by Elitism.

Elitism: is a method, which first copies(without any changes) a few of the top scored candidate solutions to the new population and then continue generating the rest of the population, Thus, it prevents loosing the few best solutions found.

Crossover: is the process of combining the bits of one chromosomes with those of another. This is to create an offspring for the next generation that inherits traits of both parensts. A locus is chosen to exchange the subsequences before and after the locus between two chromosomes to create two offsprings.

Mutation: is performed after crossover, to prevent falling all solutoins in the population into a local optimim of the solved problem.

Outline of basic Genetic Algorithm:

- *Start*: Randomly generate a population with N candidate solutions.
- *Fitness*: calculate the fitness of all candidate solutions.
- Create a new population of N candidate solution using below steps:
  - Selection
  - Crossover
  - Mutate
- Replace the current population with new population
- *Test*: Test whether end condition is satisfied: if Yes, return current best candidate solution and Stop. else, go to step2.

Each iteration of this process is called a GENERATION.

This generational process is repeated until a termination condition has been reached. Common terminating conditions are:

- A solution is found that satisfies minimum criteria.
- Fixed number of generations reached.
- Allocated budget (computation time/money) reached.
- The highest ranking solution's fitness is reaching or has reached a plateau such that successive iterations no longer produce better results.
- Manual inspection.
- Combinations of the above.

II. Proof of Concept : CAKE using Genetic Algorithm

A. *Design*

CAKE is the framework we have designed for more intelligent scheduling of jobs on clusters. Our framework accepts jobs from applications with application level constraints and uses Genetic Algorithms to come up with a schedule plan. Once the plan is ready, our framework sends the plan to each machine in the cluster individually to execute the jobs assigned to them.

As shown in Fig.2 CAKE framework has a job queue (using FIFO algorithm). The applications using CAKE, submits jobs onto this queue. CAKE polls on the job queue in every predefined time interval, it takes all the jobs submitted during this interval(which we call a batch) and uses Genetic Algorithm to come up with a plan. CAKE distributes the jobs within a batch evenly (there can be exceptions of number is jobs submitted is not a multiple of number of compute nodes available) across all the available compute nodes, such that each job in the batch is scheduled on the machine where all the job's constraints are satisfied. A example of a CAKE plan is shown is Fig.1, where each row contains a compute node

and the jobs scheduled on the compute node for a single batch of jobs. Upon looking the diagram carefully it is evident that compute nodes 7 and 8 have a job lesser scheduled on them as the number of jobs in this batch was not a multiple of number of compute nodes available, which is 8 in this case.

Fig.3 is a general version of the Fig.1. Fig.3 gives an idea of a CAKE plan and depicts that the plan can grow both vertically and horizontally, based on the number of jobs in a single batch and number of compute nodes available respectively. Fig.4 depicts the sequence od steps followed by CAKE to schedule a batch of jobs.

Fig. 1. An Example of CAKE Plan

CAKE Plan 1 for 75 Jobs

Machine 1	1	9	16	24	32	40	48	56	64	71
Machine 2	2	10	17	25	33	41	49	57	65	72
Machine 3	3	11	18	26	34	42	50	58	66	73
Machine 4	4	12	19	27	35	43	51	59	67	74
Machine 5	5	13	20	28	36	44	52	60	68	75
Machine 6	6	14	21	29	37	45	53	61	69	
Machine 7	7	15	22	30	38	46	54	62	70	
Machine 8	8	16	23	31	39	47	55	63	71	

→ Means in Machine 1, Jobs having ID 1,9, 16, 24, 32, 40, 48, 56, 64, 71 will be executed

In each step of Genetic Algorithm, we can apply domain specific heuristics which will help in finding the optimal solutions faster. The heuristic that we came up is to consider only the candidate solutions those have all the jobs scheduled in the plan. We used this heuristic in the Initialization step of the Genetic Algorithm where we randomly create candidate solutions of specified population size. Our heuristic keeps only those candidates in the initial population which have all the submitted jobs scheduled.

Fig. 2. High-level overview of Knowledge for Scheduling

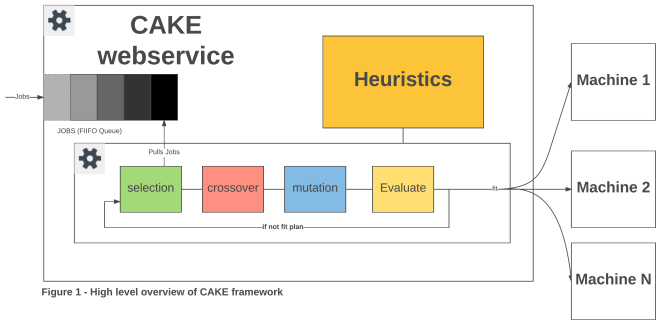


Figure 1 - High level overview of CAKE framework

B. *Implementation*

Below is the pseudocode of our implementation of the CAKE framework:

- Collect all the jobs from the job queue.

Fig. 3. Data Structure of Plan

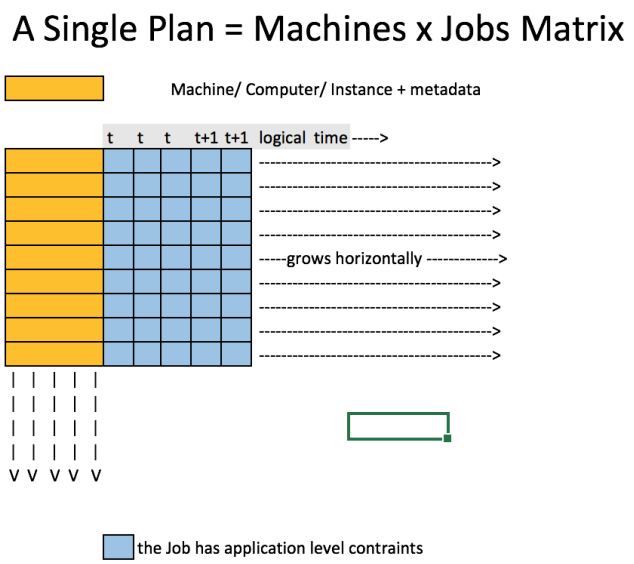
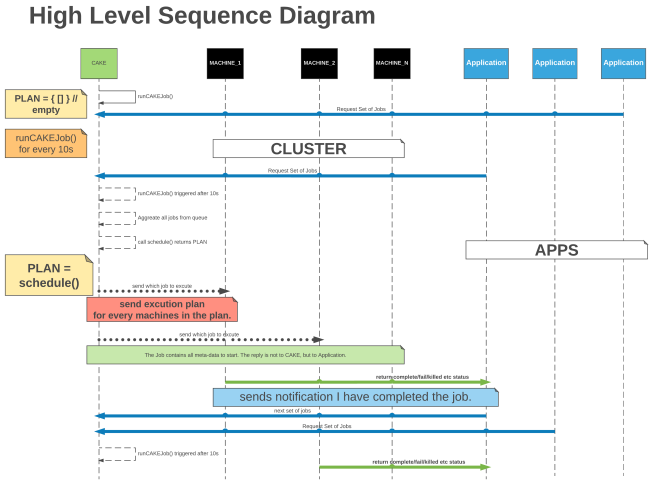


Fig. 4. Sequence Diagram



- Calculate number of jobs per compute node; ceil((number of jobs in this batch)/(number of compute nodes))
- Randomly create N candidates, where N is pre-defined population size. Each candidate should have all the jobs scheduled. Set of this N candidates is called Population.
- Test : Calculate fitness of all N candidates in the current population. If a candidate has fitness score of 100, return this candidate as the CAKE plan.
- Do this step M times, where M is number of generations(pre-defined):

Fig. 5. Test Result 1

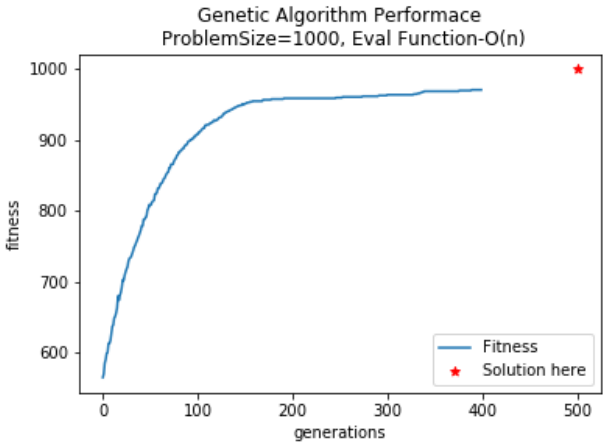
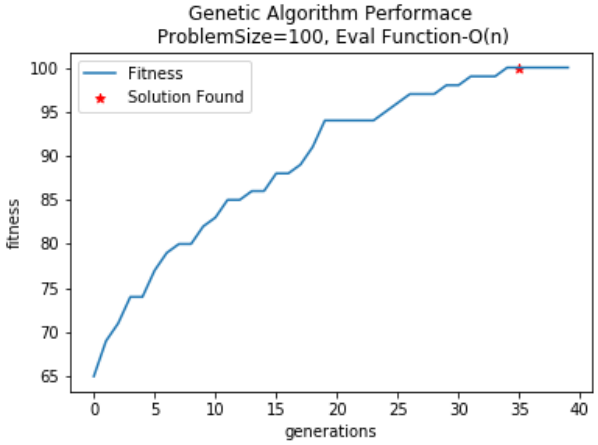


Fig. 6. Test Result 2



- Create New population by performing this step for N/2 times :
  - ∗ Selection : Randomly select two candidates from current population
  - ∗ Crossover : Create 2 new candidates by crossing over characteristics of candidates selected above. Use one of the crossover technique randonly :
    - · cross over jobs of the machine in the second part of the schedule.
    - · cross over jobs on same machine in different schedule.
  - ∗ Mutation : For the new candidates created in above step, change the jobs scheduled on each compute node to some other job, with some pre-defined probability.
- Replace : Drop previous population and replace current population with newly created population.
- Test : Calculate fitness of all N candidates. If a candidate has fitness score of 100, return this candidate as the CAKE plan.

We have implemented the framework in python. The framework could also be implemented in C++ using CUDA libraries for parallelism, which would would make the framework more efficient.

C. *Results & Critics*

We have tested our algorithm in Amazon EC2 32GB memory machine. The Source code & document is present here[22]. Our major struggle was to scale the problem size. In future work we are going to scale the problem size, however for now we have tested for relatively very small problem size. That is it will consider only memory, number of core, and hard disk required as constraints. We have tested for 10-20 machine. We want to test it for bigger problem size, so we reduced a *OneMAX* problem to test the algorithm performance for different problem size. OneMAX is also having same Evaluate function complexity as what we have developed. The test results are shown in the graph are for evaluate function $O(n)$. Figure 5 & 6 shows the test results. For small problem size it is performing well but for bigger problem size it is taking lot of time. Some time the solution is too far, as shown in Fig.5, but we took the plan which are 98% fit. We see that when we increased the Genetic Algorithm problem size to 1000 for the same evaluate function it takes a lot of time. It could be because of two reasons-

- Heuristics such as cross over rate, mutation probability rate is not optimized for the problem size 1000.
- We used Python for the implementation of our experimental Proof of concept.

Production Deployment

- Implement in C++ and CUDA. Run in GPU Machine
- For fault tolerance use stand-by installation of CAKE, a zoo keeper for all running instances.

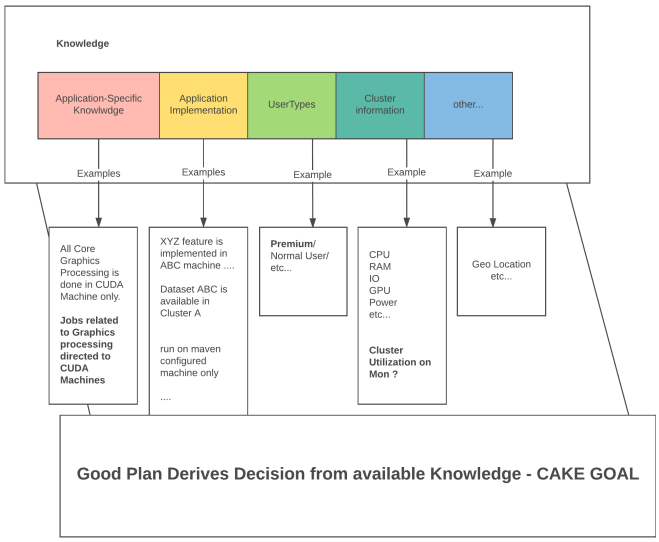
Two most important critics are -

- *Construction of CAKE to production environment is not an easy work. Also writing heuristics requires a lot of engineering work.*

- *A lot of testing is required, simulation of real-world work would be a future work for it.*

III. Research Notes and Conclusion

Fig. 7. High-lever overview of Knowledge for Scheduling



A. *Knowledge*

The application will have different types of users like premium membership or regular users. Consider an application where it provides the feature called searching. For premium users, it provides searching images using advanced image recognition. Say for example if the picture has ”car” in it. Then if the user searches for the car, then it will return the search results, even if it’s not tagged as the car. Whereas for normal users it returns results only if the picture is tagged as car. Now consider a business requirement where in order to promote their product they need premium search results to regular users. But those search processing happens in high-end clusters which are dedicated for those type of search. Imagine we have dataset depicting cluster utilization, suppose on sunday morning there is no search request from premium users but from normal users. So, the scheduler should consider that into account and provide ultra results to normal users. Scheduler allocated those resources to maximize the cluster utilization, because sunday morning no -one is using the cluster or the cluster is low utilized. So normal user will be happy to see premium search results and user might purchase the premium membership. It all happens if scheduler considers cluster utilization period, scheduler knows when it is low utilized. Likewise, it also knows types of users and its application level constraints. So our goal for cake framework is to make it intelligent enough to derive decisions from available knowledge. Fig.6 shows the high-level view of knowledge. Now, think about all the knowledge available to scheduler while decision making - means better decision making for CAKE.

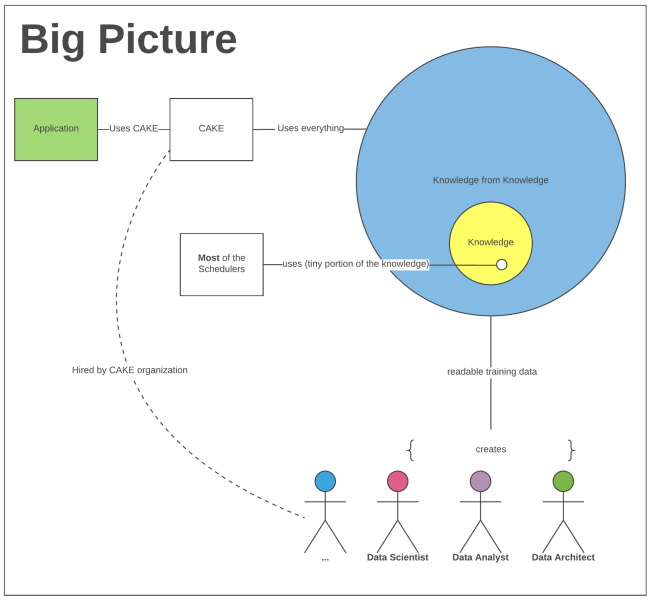
B. *What next?*

We have these in our list as a backlog-

- Cluster simulator (for testing) - for simulating real work load and cluster types for scheduler.
- Training Data Set for many system level information for machine learning.
- Implementation to integrate with YARN and testing it with real world scenario.

C. *Big Picture*

Fig. 8. Big Picture for Business



During our study, we found that most of the schedulers are using a tiny portion of data to making decisions for scheduling. So many papers have taken the further step to induce active learning and machine learning concepts. We see if you use knowledge from all the available sources we can give better PLAN. As we stated in critics, it’s

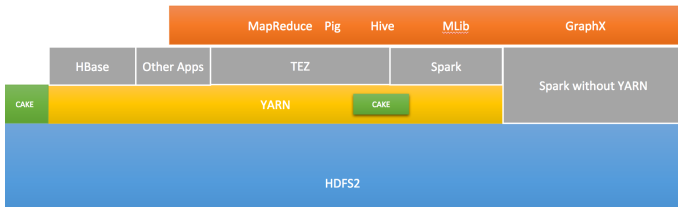
difficult to create a dataset which encapsulates all the available data in a dynamic environment.

So we decided to organize everything in one place, called CAKE framework, it will contain all the necessary tools, designs, methods, etc. for scheduling to an application or data center. Like how we have the design pattern for different problems in software development, we are tailoring CAKE by working with data scientists and solution architects. Apple & Nokia both manufacture mobile; it's up to users to choose which one. Similarly, in future many organization will use CAKE Framework as a cookbook for their scheduling - so the user will choose that company based on their need and demand.

D. *Where does CAKE fit in?*

A common question asked by many is where does it fit in, so a short answer is it can be plugging to the existing framework, or it can independently exist. Figure 9 tells where it fits. It can be used as a plug in to other schedulers like Mesos or YARN etc., or it can independently exist.

Fig. 9. Tech Stack



E. *Conclusion & Future Work*

Scheduler plays critical part in organization success. Scheduling is NP-Complete problem, finding approximate solution is tricky and great research work is being done in this area. This field is constantly improving field, some algorithm works for particular problem, and some or not. As a application developers, we see difficult to see which one to implement for our application - this is common problem across different applications or organizations. YARN, MESOS and BORG are intelligent, but the policies at present are limited. So, we designed CAKE framework with Genetic algorithm for scheduling, but in future it will have all the open source scheduler implementation-details, design and critics for it.

Future work: Our experiment(POC) and research shows that improving the quality of heuristics will lead to better decisions. Many papers have already proven that for different scenarios. Now in this project, we implemented genetic algorithm for scheduling but for small problem size. Future work in this is to extend this for various test environment and different test cases. We also look forward to add work on this as open source project. May be down the line cake framework look like a the best pluggable framework for scheduling. In short we are aiming this open-source project as super set of all schedulers.

ACKNOWLEDGMENT

Our sincere thanks to our professor Dr. Zhiling Lan, for reviewing our design and supporting us throughout the duration of this project. We also would like to thank our department and the university for providing the infrastructure for the study of this project. Gratitude to all researches whose papers we have studied (complete list mentioned in the references section) during the course of this project.

LIST OF CONTRIBUTION BY EACH INDIVIDUALS

We worked together and periodically reviewed each others work.

Ajay Ramesh -

- Survey
- Design of CAKE
- Research Notes and Critics

Chandra Kumar

- Survey
- POC
- Results and Future Works

REFERENCES

[1] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune and John Wilkes, Large-scale cluster management at Google with Borg.

[2] Christina Delimitrou and Christos Kozyrakis, Quasar: Resource-Efficient and QoS-Aware Cluster Management.

[3] Christina Delimitrou and Christos Kozyrakis, Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters.

[4] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker and Ion Stoica, Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.

[5] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen OMalley, Sanjay Radia, Benjamin Reed and Eric Baldeschwieler, Apache Hadoop YARN: Yet Another Resource Negotiator.

[6] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek and John Wilkes, Omega: flexible, scalable schedulers for large compute clusters.

[7] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou, Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing.

[8] Armon Dadgar, Nomad and next-generation application architectures.

[9] Kay Ousterhout, Patrick Wendell, Matei Zaharia and Ion Stoica, Sparrow: Distributed, Low Latency Scheduling.

[10] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec and Willy Zwaenepoel, Hawk: Hybrid Datacenter Scheduling.

[11] Christina Delimitrou, Daniel Sanchez and Christos Kozyrakis, Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters.

[12] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan and Sarvesh Sakalanaga, Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters.

[13] Matthew Bartschi Wall, A Genetic Algorithm for Resource-Constrained Scheduling

[14] Haldun Aytug, Siddhartha Battacharyya, Gary J Koehler and Jane L Snowdon, A Review of Machine Learning in Scheduling.

[15] V.Preetha and K.Chitra, Prediction of Stability of the clusters in Manet using Genetic Algorithm.

[16] R. Gowri and R. Rathipriya, MR-GABiT: Map reduce based genetic algorithm for biclustering time series data.

[17] Saeed Iqbal, Rinku Gupta and Yung-Chin Fang, Planning Considerations for Job Scheduling in HPC Clusters.

[18] Xu Yang and Zhiling Lan, Cooperative Batch Scheduling for HPC Systems.

[19] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich y, Wei Tang y, Jia Wang and Zhiling Lan, I/O-Aware Batch Scheduling for Petascale Computing Systems.

[20] Xingwu Zheng, Zhou Zhou, Xu Yang, Zhiling Lan and Jia Wang, Exploring Plan-Based Scheduling for Large-Scale Computing Systems.

[21] Sean Wallace, Xu Yang, Venkatram Vishwanath, William E. Allcock, Susan Coghlan, Michael E. Papka and Zhiling Lan, A Data Driven Scheduling Approach for Power Management on HPC Systems.

[22] Project Source Code and Documentation - http://cakeframework.readthedocs.io/en/latest/

[23] - http://firmament.io/blog/scheduler-architectures.html