Tony Nguyen

Reading Assignment 1

CPSC 324

Spring 2024

Part 1: Barroso, L.A., Dean, J., Hölze, U. (2003). Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, Volume 23, Issue 2, pp. 22–28.

Notes

1. Architecture overview
    a. Two basic things:
        - Reliability in software over hardware
        - Seek the best request throughput (meaning data flow), not the peak performance
    b. Aim for the best price/performance trade-off
    c. Serving a Google query
        - Whenever a request is sent, Google finds the closest cluster (data center) to deal with that request
            - Each cluster has a few thousand machines
            - Data is processed locally to that server
            - A load balancer allocates the request based on the current performance
            - A Google Web Service (GWS) machine renders the query to the HTML format.
        - Query: two phases

- Phase 1: The index server maps each query word with the matching list of documents (hit list)
  - The index servers determine the relevant documents by performing an intersection of the hit lists with the individual query words
- Phase 2: Compute the relevant score
  - From the algorithm above

- The search process is parallelizable:
  - Divided into shards.
  - Each shard has a random subset of the documents and is served by a pool of machines
  - Each request chooses a pool using an intermediate load balancer
  - Each shard has a replica. If one goes down, the management system will choose another one to fill in
- Final product
  - Ordered list of document identifiers (docids).
  - Document server (docserver) partitions the process by splitting into multiple "instances", creating docserver clusters
  - GWS generates the HTML and returns it to the users

d. Using replication

- Capacity and fault-tolerance
- General idea: creating replicas to have a backup in case the system goes down
- Also good in the way that makes updating feasible, though not necessarily frequent

    - Diverting the traffic away from the updating one
- Leverage inherent parallelism. Transforming larger index into smaller indices. Similar to the query stream
- Parallelizing the search over machines reduces average latency.
- CPU speed does not influence the search query performance. Google uses this point to split the work into smaller loads so that they can be handled by slower machines.
- This also means that Google does not need one machine that can reach peak performance but rather one that can handle excellent request throughput.
- To sum up, really three principles:
    - Software reliability: Leverage software to deal with fault-tolerance
    - Using replication: Better throughput and availability. As Google has to replicate its information already for capacity purposes, replicating its internal services comes at a free cost.
    - Price/performance > Peak performance
    - Commodity PCs can reduce computation costs. Cheaper PCs can create more computing power.
- Price/performance > Peak performance

e. Challenging process

- Large amount of data

2. Leveraging commodity parts

- An Index server has less disk space than a document server because the index one needs more CPU-intensive work.

- The ultimate selection criterion = Cost per query = (Capital expenses (less depreciation) + Operating cost) / Performance
- Equipment is expensive as it depreciates quickly
- Traditional approach: Less appealing – increasing performance but decreasing price/performance ratio
- Drawbacks
    - Significant administration and repair costs since we are operating more PCs.
        - However, all PCs are homogeneous (same types) so the cost is lower/more manageable
        - Because of scale-out, the cost of maintaining more machines is similar to (not significantly more expensive than) a smaller number of machines as they can be batched repairs

3. Power problems
    - Issues of power consumption and cooling
    - Needs special cooling or additional space to reduce power tolerance.
    - Use Watts per unit of performance, not just Watts along
    - Two caveats
        - Reduced power must not reduce performance
        - Cost for a lower-power server must be less than the depreciation cost
        - Seems to not mention environmental costs. I guess this is understandable given this article was written in 2003
4. Hardware characteristics
    a. General notes

- Deals with price/performance ratio (PPR)
- The index server impacts PPR the most
- Index server
    - Decoding compressed information in the inverted index
    - Why inverted?
    - Finding matches against a set of documents to satisfy the search query
- What is instruction-level parallelism
    - Possibly means having multiple requests (like a lot) sent at the same time
- "The level of aggressive out-of-order execution present in modern processors are already beyond the point of diminishing performance returns…"
    - Meaning even newer chips won't make it faster?
- Solution: simultaneous multithreading and multiprocessor → target thread-level parallelism
    - Meaning a bunch of simpler, short-pipeline cores beats one complex core
    - Also lower branch misprediction

b. Memory

- Bandwidth does not seem to be a bottleneck
- In general, not the greatest factor

c. Large-scale multiprocessing

- Note that the approach above refers to inexpensive desktop-class machine
- This is about a smaller number of large-scale shared-memory
- Most useful when:
    - The computation-to-communication ratio is low

    - Communication patterns or data partitioning are hard to predict (dynamic nature)
    - Ownership cost > hardware cost (in terms of license, management overhead, etc.)
- What Google does: None of the above requirements apply
    - Partition the index data and computation → minimize communication and balance load
    - Produce their own software → license-free
    - Minimize management overhead through automation
    - It ends up making hardware the most significant cost since software is essentially "free"
- The large-scale system is not great for redundancy → easy to fail if one fails. Deploying multiprocessors makes it easier to contain errors

Answers to Questions

1. Explain in your own words, with additional detail, the two insights that the paper states arise from Google's software architecture.
    - The company realized that achieving the highest data throughput is much better than reaching the highest single performance. Google can achieve such performance by utilizing parallelism. The cluster management system transforms indexes, fetched into an algorithm for querying, into shards, essentially smaller pieces. Each shard is then performed by a cluster. Parallelizing the search over machines helps reduce the average latency. Google also realizes an important insight: it does not need a super-powerful

machine to run these instances; rather, commodity PCs should be enough as they can generate a higher computation power.

- Google also invests in software reliability over hardware. As the data centers operate non-stop, the machines are constantly under an intense workload. Therefore, failure happens more often than we think, which is inevitable. Google then develops fault tolerance software, which can salvage the work if things go wrong. In order to do so, they create replicas. This is nice because it allows fault resistance ability – the server always has a backup. Having replicas also comes at a free cost since they would have to replicate their services anyway to increase their capacity. Furthermore, since Google chooses to use commodity PCs as described above, it can update its computer more quickly since it can just divert traffic away from the updating one.

2. In your own words, summarize the main benefits described in the paper of using mid- range PCs over more powerful servers. Also describe one of the challenges of the approach described in the paper (there are multiple challenges raised).

    - Google has found that peak CPU performance is less important than having the highest price-to-performance ratio since it parallelizes many of its internal services. One of the distinctions of parallelization is that it needs to handle lots of data simultaneously, which requires a significant amount of clusters and, hence, machines. Parallelizing its services essentially splits the task into smaller ones, then processing all of them at once on different clusters. This suggests we need a solution to run many tasks simultaneously rather than having one machine run a super-intensive workload, as each task now is less intensive to need a powerful machine. Therefore, we need multiple machines to run many tasks rather than one that runs everything.

- However, machines depreciate quickly, and the quick development of new technologies makes them costly to maintain. Google came up with a solution by using commodity PCs – readily available PCs in the market – as they are much cheaper to buy. Google bases its selection criterion on the cost per query, which includes the capital expenses of the machines, less depreciation, and the operating cost divided by the performance. The higher the ratio, the better it is, which is the case of using commodity PCs over more expensive machines.
- However, using commodity PCs also has some challenges. One is the significant cost of operating, repairing, and upgrading. This is understandable as commodity PCs are less durable than more powerful ones, in addition to the new development of new technology. With that, upgrading and repairing occur often. However, all PCs are homogeneous, making maintenance costs lower since Google can stock up on parts in advance. Because clusters can be scaled out by adding more machines, the cost of maintaining more machines is not significantly more expensive than smaller machines since they can be batch-repaired.

Part 2: Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-Scale Cluster Management at Google with Borg. *Proceedings of the 10th European Conference on Computer Systems* (EuroSys), pp. 1–17.

Notes

1. Overview
    - Google Borg:
        - Cluster management system
        - Runs jobs and applications
        - Each cluster has around 10,000+ machines
    - Able to achieve high utilization thanks to
        - Combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation
        - Support high-availability applications with features that minimize fault recovery time
        - Scheduling possibility → reducing the probability of correlated failure
2. User's Perspective
    a. Overview
        - Submit Borg's works in the form of jobs
        - Each job has tasks
        - Each job runs on the Borg cell
    b. Workload
        - Borg cell runs a heterogeneous workload:

- Long-running services that should not go down – Handle short-lived latency-sensitivity request
- Batch jobs that take seconds or days to complete

c. Clusters and cells

- Machines in a cell belong to a single cluster
- Machines are heterogeneous, meaning not everything is the same
- Borg determines which machine to run a task, hiding it from the users
- It is good since resources are balanced

d. Jobs and tasks

- A Borg's job has a name, owner, and a number of tasks
  - Can have constraints to force a job to run on specific machines
  - Can select a time to start a job
  - A job runs in a cell
  - A task has resource requirements and an index within a job. It also has the ability to be overridden
    - Task-specific command-line flags
    - Resource dimension is specified independently at fine granularity
  - Borg are statically linked: reduced dependencies on their runtime environment
- Operates on jobs by calling RPC (remote procedure call)
  - Written in BCL, a derive of GCL
  - Has lambda functions
  - Able to trigger "submit", "kill", and "update"

e. Allocs (Allocations)

- A reserved set of resources on a machine where one or more tasks can be run. It remains assigned whether they are used or not
- An alloc set is like a job

f. Priority, quota, and admission control

- Priority
  - Every job has a priority, shown as a positive integer
  - Higher priority tasks can take up more resources than smaller ones, even allowing themselves to kill (preempt) smaller ones if needed
  - However, we don't allow high-priority tasks to kill each other
- Quota
  - Used to decide which jobs to admit for scheduling
  - Expressed as a vector of resource quantities (CPU, RAM, etc.) at a given priority for a period of time
  - The quantities specify the maximum amount of resources that a job can request (e.g., 20TB of RAM at prod priority from now until the end of July)
  - Jobs with insufficient quota are immediately rejected upon submission
  - Higher-priority quota costs more than lower ones. Users are only encouraged to purchase the amount of data that they need. All users are offered priority zero quotas for free

g. Naming and monitoring

- Not just create and place tasks; we need to be able to find them
- Borg creates a stable "Borg name service" (BNS) name for each task. It includes the cell name, job name, and task number

- Borg writes the task's hostname and port into a consistent file in Chubby, a highly available and persistent distributed lock service and configuration manager, to find the task endpoint
- Every task has a server that contains information about it, which users can run logs to diagnose.
- Generally good for diagnostic and analysis

3. Architecture
    a. Overview
        - Borg has a set of machines, a Borgmaster and a Borglet
        - All are written in C++
    b. Borgmaster
        - A logically centralized controller
        - Has two processes
            - The main Borgmaster process
                - Handles RPCs that either mutate states (e.g., create jobs) or give read-only access to data (e.g., lookup job)
                - Manages state machines of all objects (machines, tasks, allocs, etc.)
                - Communicates with the Borglets
                - Offers a web UI as a backup to Sigma
            - A separate controller
        - Actually a single process, but being replicated 5 times. Each replica has an in-memory copy of most state in the cell
        - Elected master: for each cell

    - Serves as a Paxos (?) leader and a state mutator
    - Suggested by the Paxos when a cell is brought up and whenever the elected Paxos fails
    - Able to re-synchronize from a failed one to a new one
- Checkpoint: The Borgmaster's state at a point in time
    - A periodic snapshot + change log
    - Like git
- Fauxmaster: a high-fidelity (?) Borgmaster simulator
    - Simulate real Borgmaster for planning purposes
    - Useful for capacity planning

c. Scheduling

- When a job is submitted, it is recorded in the Paxos and added to the pending queue (done by Borgmaster)
- Scanned asynchronously by the scheduler – assign tasks (not jobs) to machines if there are enough resources
- Algorithm has two parts
    - Feasibility checking: find which machine to run on
    - Scoring: picks the most feasible one
- In general terms, the Borg system tries to give out resources to the ones with the highest priority, even kill the lower ones to allocate enough resources

d. Borglet

- The local Borg agent on every machine in a cell
- Tasks

    - Start and stop tasks
    - Restart if they fail
    - Manipulate local resources
    - Roll over debug logs
    - Report the state of the machine to the Borgmaster
- They are fault-tolerance
    - They are built to run independently even if the Borgmaster replica fails
    - Will carry on works so it guarantees concurrency

e. Scalability

- Don’t have a known limit – we manage to overcome it every time
- Ways to handle
    - Split the process so it can be operated in parallel with each Borgmaster function replicated for failure tolerance
    - A scheduler replica operates on a cached copy: retrieves changes, updates the local copy, assigns tasks, informs the elected master about these changes
    - The master will accept or reject the inappropriate
- Add separate threads to talk to the Borglets and respond to read-only RPCs to improve performance. Also considering sharding
- What makes the Borg scheduler more scalable
    - Score caching: Caches scoring until the properties of the machine change
    - Equivalence classes: Only do feasibility and scoring for one task per equivalence class

- Relaxed randomization: Examines machines in random order until it has found enough machines to score and then selects the best one within that set.

4. Availability
    - Failures on large-scale systems are the norm
    - However, Borg still tries to mitigate the impact of these events
    - A key design is that already-running tasks continue to run even if the Borgmaster or a task's Borglet goes down. Each cell is independent of the others
    - The final goal is not scalability limitation but rather an argument against larger cells
5. Utilization
    a. Evaluation methodology
        - Cell compaction
        - Given a workload, we try to remove machines until they cannot handle anymore
        - Use Fauxmaster to create simulations that can be used to obtain results
    b. Cell sharing
        - Borg helps better utilize the resources. Prod jobs usually have reserved overhead, which can be reclaimed using Borg; this helps to reduce the number of excessive resources
        - CPI (cycle per instruction) result
            - Positively correlated with two measurements over the same time interval
            - Experiments confirm that performance comparisons at the warehouse-scale are tricky
            - Sharing doesn't drastically increase the cost of running

- Sharing is still a win: CPU slowdown is outweighed by the decrease in machines required

c. Large cells

- Purposes: allow large computations and decrease resource fragmentation

d. Resource reclamation

- A job can have a resource limit
- Instead of wasting unused allocated resources, we can recollect and reappropriate them to other batch jobs
- Called resource reclamation
- However, the Borg system doesn't rely on the reclaimed part but rather allocates resources on its own.
- A task that exceeds the limit will be the first one to be killed. On the other hand, the CPU can be readily throttled to push through short-term spike

6. Isolation

a. Overview

- Sharing machines increases utilization, but it needs to ensure they don't interfere with each other.
- Has two types: Security and Performance Isolation

b. Security isolation

- Use Linux chroot jail as the primary security isolation mechanism
- Has automatically generated sshkey

c. Performance isolation

- All Borg tasks run in a Linux cgroup-based resource container

- The Borglet manipulates the container setting
- Borg tasks have an application class, or appclass for short
- The difference is the latency-sensitive appclasses and the rest (batch)
- There are two types: compressible (CPU cycles, disk bandwidth) and non-compressible resources. The Borglet can kill tasks in non-compressible; it will kill tasks to make space. However, for the compressible, it will throttle usage.
- Tasks are permitted to consume up to their limit, and even 5% more for compressible resources.

7. Related works
    - It is interesting to see how other technologies share the same principle framework with Borg, which was designed years ago, especially Kubernetes placement inside Docker containers.
    - VM is used a lot
    - Emphasizes automation and operator scale-out
8. Lessons learned
    a. Lessons learned: the bad
        - Jobs are restrictive as the only grouping mechanism for task
            - Borg cannot manage the entire multi-job services
            - Kubernetes rejects the job notion
        - One IP address per machine complicates things
            - All tasks on a machine use a single IP address, which shares the host's port space
            - Requires more manual work – see file
        - Optimizing for power users at the expense of casual ones

- Allows “power-user” to essentially fine-tune lots of details, which is great
- However, this makes things complicated

b. Lessons learned: the good

- Allocs are useful
  - Widely used log saver pattern
  - Periodically updates the data used by a web server
  - Allows services to be developed by separate terms
- Cluster management is more than task management
  - Not just managing the lifecycles of tasks and machines
  - Benefits from other services, including naming and load balancing
- Introspection is vital
  - Borg doesn’t hide debugging information
  - It makes it easier to debug, although it comes at the price of harder-to-change internal policies.
- The Borgmaster is the kernel of a distributed system

c. Conclusion

- All of Google’s cluster workloads have been switched to use Borg over the decade
- Lessons learned are used to develop Kubernetes

Answer to Questions

1. Briefly describe the difference between "priority" and "quota" and their relationships to scheduling and running jobs. State the difference in your own words.

- Priority: An index that illustrates a job's urgency to run. They are shown as a positive integer, which represents the importance of the job that needs to run. Higher-priority jobs take more resources to run compared to lower ones. This is significant as the Borg can kill less-priority jobs to make enough resources if needed. However, we don't allow higher-priority jobs to kill each other. Preempted jobs are then rescheduled to run at a different time.
- Quota is used to describe which job Borg should admit for scheduling to avoid scheduling jobs that the current cluster cannot meet its resource requirement. It is expressed as a vector of resource quantities, which is the maximum amount of resources that a job can request at a given priority at a given time. Besides, it is important to note that quota is a part of the admission control, not job scheduling. Jobs that have insufficient quota are immediately rejected upon submission.
- Since users typically request more resources than they actually need, it leads to resources being wasted. With that, Google charges a higher cost for higher-priority quotas rather than the lower ones. Google also offers priority quota zero (the lowest one) for free to encourage users to buy the amount of resources they really need.

2. In your own words, describe the core ideas and differences between "worst" and "best" fit approaches, along with Borg's approach to scheduling tasks. Note that the Borg scheduling approach isn't explained in detail, so for this part, you'll need to think about what the paper is saying and try to decipher the core idea and why it is an improvement over the other two approaches.
    - Both "Worst fit" and "Best fit" try to find the most feasible machine – the machine that meets the resource constraints – to run the tasks. This process is also called scoring.

There are several factors to yield the best score, and one of them is the ability to spread the task so that high-priority tasks can be expanded in a spike if needed.

- The “Worst fit” case uses E-VPM, a Google Borg scoring algorithm, to generate a single cost value across all heterogeneous resources. This results in a minimized change in cost when placing a task. One signature thing about E-VPM is that it spreads the load across all machines. This allows each machine to have a headroom reserved for spiking. However, one disadvantage of this method is increased fragmentation.
- In the “Best fit” case, it tries to fill machines as tight as possible. This essentially results in empty machines that have no jobs, which are reserved for large tasks. This resolves the fragmentation issue but comes at the cost of easy misestimation. However, this is still bad for applications with short-interval loads and batch jobs with low CPU needs since it has a high overhead and low throughput.
- The Borg’s approach is a hybrid one since it reduces the amount of stranded resources. Stranded resources mean machine resources can’t be used because another resource on the machine is fully allocated. I find this one a better idea than the “Best fit” one and, of course, the “Worst fit” one. My interpretation of this idea is that, with the “Best fit” one, since it allocates all of the resources of empty machines to run the task, some of the system resources may still be available while others run out. This leads to the available resources being wasted.
- Therefore, since Borg kills less-priority tasks to make space for higher ones, it makes resources well-spent. The killed (preemptive) tasks are then put into a queue and wait to be rescheduled. Furthermore, Borg tries to run tasks on machines that have enough packages and dependencies installed first so that resources are better spent. Eventually, it

solves the problems of the two approaches above. Machines are better allocated dynamically depending on the job constraints, so they don't have to be fragmented while maintaining the throughput.

3. In your own words, describe the interaction between the borgmaster (and replicas) and borglets, and how the interaction helps enable fault tolerance in Borg.
    - Borg is essentially a job scheduler. Google uses Borg on nearly all of its services nowadays. Borg schedules tasks on a set of clusters. Each cluster (cell) creates five replicas, which elect one replica, called Borgmaster, to be the leader. Borgmaster has two processes: the main Borgmaster process handles clients' requests, manages the machines' state, and communicates with the Borglets for health monitoring and metrics, and the second is a separate controller. If the Borgmaster fails, the other four replicas will elect another one to be the new Borgmaster.
    - Since there are multiple machines in one cell, each machine will have a Borglet, which acts like a worker for the Borgmaster. The Borglet is responsible for starting and stopping tasks, restarting a task if it fails, manipulating local resources, rolling over debug logs, and reporting the machine state back to the Borgmaster. The Borgmaster will pull data from every Borglet periodically. A Borglet will be marked as "dead" if it fails to respond. The Borgmaster will then send out a request to kill the task.
    - What is nice about the Borg system is that it always has a backup of everything. In case a Borglet fails and is then commanded to be killed by the Borgmaster, the Borglet will still carry out its works. On the other hand, if the Borgmaster fails and a new leader is subsequently elected, Borglets will also carry on its task to ensure concurrency. When the

connection is resumed, the Borgmaster can send out a command to kill the Borglet if there is a duplicate to avoid wasting resources.

Part 3: Observations

1. For the first paper about Google Cluster, it is actually interesting to see how Google facilitates its cluster system to power all its services. I am impressed by how Google conducts its search query function to yield a significant performance. I also did not expect the use of commodity PCs in Google's data centers. I previously thought that the clusters would use high-end machines, but they use a much cheaper version. I found it counterintuitive at first, but it makes sense in the end since more jobs can be run at the same time, which boosts overall performance.
2. The second one about the Borg system was completely fascinating when I realized that Borg is actually what Kubernetes and Docker are developed from. I have used these systems before and liked how they are leveraged in reality. Borg is also brilliant since it can dynamically allocate and manage resources available in the systems. Think about how many clusters are interconnected in one given data center, and yet somehow, Borg can manage and schedule jobs efficiently, even nowadays. I totally agree on how revolutionary these two systems are, even though they are still related nowadays.