ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPA04

ORCHESTRATING MeerKAT's DISTRIBUTED SCIENCE DATA PROCESSING PIPELINES

A. F. Joubert*, B. Merry†, SKA SA, Cape Town, South Africa

Abstract

The 64-antenna MeerKAT radio telescope is a precursor to the Square Kilometre Array. The telescope's correlator beamformer streams data at 600 Gb/s to the science data processing pipeline that must consume it in real time. This requires significant compute resources, which are provided by a cluster of heterogeneous hardware nodes. Effective utilisation of the available resources is a critical design goal, made more challenging by requiring multiple, highly configurable pipelines. We initially used a static allocation of processes to hardware nodes, but this approach is insufficient as the project scales up. We describe recent improvements to our distributed container deployment, using Apache Mesos for orchestration. We also discuss how issues like non-uniform memory access (NUMA), network partitions, and fractional allocation of graphical processing units (GPUs) are addressed using a custom scheduler for Mesos.

INTRODUCTION

The MeerKAT radio telescope [1] is currently under construction in the Karoo region of South Africa. In total it will have 64 dish antennas when construction is completed in 2018. It is a precursor to the larger Square Kilometre Array project [2], and will be integrated into the mid-frequency array, SKA1 MID.

The focus of this work is the recent advances in the deployment process of the Science Data Processing pipelines, leveraging so-called "container orchestration" tools that have become common in micro services architectures [3]. We are using Docker containers [4]. In this context, *container orchestration* refers to automating the following: deciding which host a container should be run on and launching it, connecting containers together, and monitoring and reporting on the state of the containers and the services they provide.

This containerised, microservices architecture was selected and combined with continuous integration and deployment tools to achieve a number of benefits: quick deployment which minimises downtime for upgrades, simple and consistent deployment, higher availability as faulty hardware is easily switched out, and improved package management — some software packages are difficult to install due to dependency issues, but by confining each to a dedicated container this problem is simplified.

This paper is organised as follows. First, the MeerKAT system is summarised for context. Next, the Science Data Processor is discussed, as well as the motivation for the changes to the orchestration process. The Apache Mesos platform [5, 6] and the role of scheduler frameworks is then presented. Details of our custom scheduler framework follow, before concluding.

* ajoubert@ska.ac.za
† bmerry@ska.ac.za

MEERKAT SYSTEM OVERVIEW

This section provides only a brief overview — more details are available [7]. There are three major parts to the telescope data processing pipeline: antennas, correlator beamformer (CBF), and Science Data Processor (SDP).

Each antenna provides a digitised stream of data at a rate of approximately 34 Gb/s, depending on the frequency band, for a total of 2.2 Tb/s. This data is processed in real time by the CBF using banks of Field Programmable Gate Arrays (FPGAs). Tasks such as frequency channelisation, baseline correlation and beamforming are performed by the CBF.

The CBF output is ingested by the SDP which performs tasks such as imaging and calibration. The maximum data rate expected to be ingested by the SDP is approximately 600 Gb/s. This data is transmitted from the CBF in a number of streams, with up to 17 Gb/s in a single stream. While some streams can be split and distributed across nodes, others need to be processed in a single location. Thus, we need to extract the maximum performance from each node.

Overall management of all the telescope's subsystems, including antennas, CBF and SDP, is handled by the Control And Monitoring system (CAM).

SCIENCE DATA PROCESSOR

The physical hardware planned to implement the SDP cluster includes 10 Gb/s and 40 Gb/s Ethernet switches, approximately 50 dual-socket servers, most of which include multiple Graphics Processing Units (GPUs), and an array of spinning disk, solid state and tape drives for the archival of data. Other than noting that these nodes are heterogeneous, with different Central Processing Units (CPUs), GPUs, memory and networking capabilities, the details of each node are inconsequential for this paper and not discussed further.

The processing pipeline moves the data through various logical processes that are distributed over this compute cluster hardware. Depending on the science being performed, the configuration of the pipeline changes. This typically happens a few times per day. The design of the MeerKAT telescope requires multiple instances of the processing pipeline to be active simultaneously. Once the science workload for a pipeline is completed, the resources can be released.

While the SDP pipeline does consist of many small software processes each doing a part of the work, it is not a conventional micro-services architecture [8]. The control plane does not use Hypertext Transfer Protocol (HTTP) or a message bus. Instead it uses a simple remote procedure call protocol named KATCP — the Karoo Array Telescope

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPA04

Control Protocol [9]. The data plane uses multicast User Datagram Protocol (UDP) rather than the more common Transmission Control Protocol (TCP), due to the real-time constraints. Many of the processes in the pipeline have state, which must be maintained at least for the duration of the science observations being performed in a particular configuration.

CAM requests changes in the science pipeline configuration via the SDP's master controller. This master controller is a long-running process that manages the SDP pipelines at a high level. The master controller is responsible for starting or stopping the necessary Docker containers and reporting general SDP health.

DEPLOYMENT HISTORY AND MOTIVATION FOR CHANGE

The very earliest versions of the SDP were deployed by manually checking out source code on production servers, installing it, and restarting the affected services. While simple to do, this required manual intervention from developers, and was error prone.

As time allowed, the tooling improved. Provisioning of servers was automated using the Fully Automatic Installation (FAI) tool [10], and Ansible [11] scripts. Jenkins [12] is used for Continuous Integration — testing code and building Docker images. At runtime, monitoring and analysis is performed using Elasticsearch [13], LogStash [14], Kibana [15], Grafana [16], and Prometheus [17].

Static Allocation

The SDP master controller used to have a static configuration specifying which containers to run on which compute nodes. In this implementation, a single instance of the master controller runs on a predefined node, while the Docker daemon, *dockerd*, runs on all nodes.

This works well in the ideal case, but fails if a required node is down. It is also easy to overload one node while others are underutilised. Thus a more dynamic implementation that scales automatically was sought.

Dynamic Allocation

In order to perform dynamic allocation of the containers, an orchestration mechanism was needed. Such a mechanism would abstract the cluster of compute nodes into a single pool of resources and automatically place containers in an efficient, rule-based way. Monitoring the health of the nodes and running containers, and maintaining the required state is also important. Such software could be written from scratch, but it is far more effective to use an existing orchestration tool.

There are many to choose from, including Apache Mesos, Kubernetes [18], Docker Swarm [19] and Hashicorp Nomad [20]. We will not perform a detailed analysis of all of these. One of the main differences between them is the way the scheduling is performed. In this context, scheduling refers to deciding where to place the containers required to execute a particular job or perform a service. In other words, mapping application code to compute infrastructure.

There are three broad categories of schedulers [21]: monolithic (e.g., Swarm), two-level (e.g., Mesos) and shared state (e.g., Kubernetes and Nomad). The differences relate to the concurrency and optimism of the schedulers. The choice becomes especially important when going to very large scale, with the monolithic architecture scaling least well. In our case, our compute cluster is not on the order of thousands of nodes, and the rate of scheduling is very low — only a few times per day. Thus the performance of any of these would be adequate.

Of more importance to us was the ability to plug in a custom scheduler framework which could be managed by the SDP master controller. A few of the tools discussed allow custom schedulers in some form, although this may require recompilation [22, 23]. At the time this work was starting, we briefly investigated the capabilities of Kubernetes, Mesos and Docker Swarm. Mesos, which was specifically designed to handle multiple schedulers, including custom implementations, was selected as the most suitable. Note that ongoing development of all of these tools may mean that Mesos is not necessarily the optimal choice for new projects.

MESOS OVERVIEW

A brief introduction is given here to provide some context for the discussion of our custom scheduler in the next section.

Mesos Architecture

The architecture is shown in Fig. 1. Overall management of the cluster is handled by a *master*. For redundancy, there can be multiple masters running, with a single one elected as the leader, and the remainder on standby. The leader election is handled by a distributed key-value store called ZooKeeper, which can be run on multiple nodes for high availability. The worker nodes are known as *agents*.

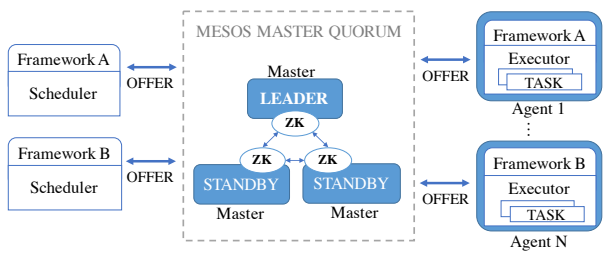

Figure 1: Example of a Mesos system architecture, where ZK refers to ZooKeeper. Image based on [24].

The remainder of the system is made up by one or more *frameworks*. Each framework consists of a *scheduler* and optionally a custom *executor*, which need to work together to perform tasks for the user.

A *scheduler* is responsible for scheduling, placement, replication and failover. There can be multiple schedulers, each customised for certain types of workloads, or

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPA04

even duplicate schedulers, if required for scaling or redundancy. There are many readily available schedulers, including Marathon [25] for long running services, Chronos [26] for batch processing and Spark [27] for data processing.

The *executor* knows how to run a particular task on a compute node. For example, there is a standard executor that can launch and monitor Docker containers, and shell scripts. While rarely necessary, a custom executor can be implemented, if the user needs non-standard behaviour.

Mesos Principle of Operation

The two-level scheduler in Mesos is implemented partly in the master and partly in the scheduler frameworks. The master acts as the central resource allocator. The agents inform the master of the resources they have available. The master's resource allocator considers the pool of available resources and decides to which scheduler to *offer* a portion of the resources. This would include details like the amount of CPU and memory being offered. The allocator attempts to achieve dominant resource fairness (DRF) [28] across all schedulers. While the offer is made, the resources are locked, and will not be offered to any other schedulers.

The scheduler can either accept or reject the offer. If it accepts, it provides a set of tasks that must be executed, each using a fraction of the resources. For example if the offer has 2 processor cores and 3 GB of memory, the scheduler may decide to run two tasks: one task using 1.5 cores and 1 GB of memory, and another using 0.25 cores and 2 GB of memory. Unused resources return to the allocator and may be offered again later. In this way, each scheduler can decide exactly where to place its tasks in the cluster, based on its own set of rules. If the scheduler has no work to do, it can suppress offers from the master. Later, when it has more work, it can revive offers.

The tasks to run are received by the master which then launches them using the appropriate executor on the agent. Status from the tasks is fed back to the scheduler. The scheduler may also decide to terminate a task before completion — such requests are made to the master, which passes them on to the executor.

SDP CUSTOM SCHEDULER

An existing Mesos framework could have been modified to work with the SDP's existing master controller, but as the master controller already included some scheduling code, it was decided to extended this into a new Mesos framework instead. Extending an existing framework may have worked. For example, Marathon has many benefits with regards to high availability, but we are not sure if we could have worked around some of the issues we found, specific to our requirements.

We did not need to create a custom executor as the Mesos Default Executor's ability to run Docker containers was sufficient.

Overview

When activating an SDP pipeline to process a set of science observations the following pseudocode illustrates the steps followed by the master controller. The italicised names refer to functions from the Mesos scheduler driver Application Programming Interface (API):

```
// receive configure pipeline
// request from CAM:
call reviveOffers;
while not sufficient resources do
  | receive resourceOffers;
end
call suppressOffers;
create details of tasks to be run;
while not all tasks started do
  | receive statusUpdate callbacks;
end
// pipeline ready
```

The program flow is very straightforward, with the most work required when creating the details of the tasks to be run. This is the point at which our custom scheduler must decide where to place each container and what command-line options to use.

NUMA Awareness and Core Affinity

We are using multiprocessor computers which have non-uniform memory access (NUMA) architectures. Each CPU has its own memory bank. One CPU can access the memory of another CPU, but this is over a slower bus and reduces performance. We benchmarked a small application that unpacks the type of data we expect to receive over the network, using memory only. By forcing the application to run on a single NUMA node, we saw performance increase by approximately 30%. These kind of performance deltas are reported in the literature [29, 30], but have not been addressed specifically for container orchestration tools.

Docker does not provide a direct mechanism to ensure a container runs on a single NUMA node, however, it does allow limiting to a set of CPU cores with the `-cpuset-cpus` option, if the correct CPU core numbers are known. The Linux scheduler initially allocates memory from the same NUMA node for these pinned processes. However, if the memory requirements exceed that available on the node, another node may be used — we need to keep our memory usage low enough to avoid this.

Our custom scheduler requires knowledge of which CPU cores are in each NUMA node. While Mesos does not provide any mechanisms for interrogating agent hardware, it allows arbitrary key-value attributes to be attached to each agent and made available to the scheduler. As part of deploying the agent, we use the *lstopo* tool [31] to determine the topology and store it in an attribute, e.g., `katsdpcontroller.numa=[[0, 2, 4,`

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPA04

6], [1, 3, 5, 7]] for a dual-socket, quad-core node. Mesos also allows for custom resources: we create a "range" resource to expose each core as a resource.

NUMA-sensitive tasks reserve a number of cores, and the scheduler restricts the allocation to be from a single NUMA node. This solution conflates NUMA awareness with core affinity, which is not ideal. In practice it works, as our high-performance tasks are invariably pinned to specific CPU cores to prevent the additional latency caused by level 1 cache invalidation.

Fractional GPU Allocation

Mesos has basic support for specifying GPUs as resources, but only in integer increments. The type of GPU, amount of memory, and the NUMA node is not available. We added custom resources and attributes with this information, as well as the ability to allocate fractions of a GPU's compute capacity and memory to a task. We also need to access device paths like `/dev/nvidia0` from inside the container. These are passed in via Docker arguments.

Unlike CPU assignments, our custom mechanism provides no enforcement of limits between different processes using the same GPU, so we must be careful not to oversubscribe the resources.

Network Interface Card Affinity

As with the CPUs and GPUs, the NUMA node that a particular network interface card (NIC) is connected to is important for optimal processing. Our network also has different segments for data and control, so we must ensure the selected NIC will have access to the correct segment.

We added custom resources for the input and output bandwidths; and custom attributes like interface name, device endpoint, network name, NUMA node, and Infiniband capabilities. Similar to the GPU solution, there is no enforcement of these limits by the operating system.

Multi-stage Launch

While existing orchestration systems often have the concept of a group of tasks that are launched together (e.g., "application groups" in Marathon), our requirements are slightly more complex. Each container provides metadata about its outputs to downstream consumers. Thus, the containers in a pipeline need to be started in front-to-back order, waiting for each one to come up before starting the next one.

While this could be achieved by running the pseudocode above separately for each container, this can lead to suboptimal scheduling decisions, because the first container is run as soon as we have somewhere that it fits, without considering the needs of other containers in the pipeline. It can also lead to a pipeline being half-launched, only to find that there are insufficient resources left in the cluster to continue.

We solve this problem by waiting until we have enough resources to launch all the containers, and launch them all together. However, each container with dependencies runs a wrapper script. This script makes an HTTP request back to the scheduler, which replies only when the real work of the container should begin.

Implementation

The bulk of the MeerKAT telescope is developed using the Python programming language [32], so we chose this for our scheduler too. We used the PyMesos package [33] to simplify interfacing with the Mesos master.

Our master controller, including the scheduler, consists of approximately 5 k lines of code. Mesos provided the bulk of the solution, with their 130 k lines.

We use Ansible to deploy the Mesos agents. As part of this deployment, all the custom resources and attributes are written into configuration files in the `/etc/mesos-slave/resources` and `/etc/mesos-slave/attributes` directories, accessible to the agents and subsequently advertised to the master. Where possible, these files are populated by querying the operating system.

Discussion

While our development was aided by a number of positive aspects of the tools and approach selected, it was not without challenges. We discuss both sides of this.

Writing our own scheduler gave us a lot of flexibility that an off-the-shelf scheduler such as Marathon is unlikely to provide. For example, describing tasks in Python allowed us to specify arbitrarily complex placement policies and adapt the task to the chosen placement (for example, choosing a Docker image that is pre-tuned for the type of GPU found). It also made it possible to handle requirements such as NUMA awareness. However, our scheduler is not fault-tolerant, and it would require significant further work to achieve the high availability of a framework like Marathon.

The Mesos project is mature, active and widely used, and thus includes many developer-friendly features. For example, Mesos is robust and does proper cleanup of old containers after framework crashes, which is sure to happen when developing a new framework. Despite the robustness, we still managed to crash agents and the master, although the latter did recover, and the developers are responsive to bug reports. This has not been an issue in production.

The Mesos HTTP API has wrappers written in many languages, including Python. The PyMesos package was a big time saver, but we still had to reinvent the wheel for simple things like aggregating resource offers from an agent and iterating over a range resource.

The web-based user interface (UI) is good and helps with debugging. However, it has no support for custom resources, so there is no visibility of resources such as the network bandwidth in use. Another issue is that changing an agent's attributes or resources requires a manual recovery step to discard its check-pointed state. This also means killing all containers currently running on that agent.

The ability to define custom resources and attributes was critical to our solution. On the downside, attribute values are limited to the characters `A-Za-z0-9_/.-` so our actual

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPA04

values must be base64 encoded, and are thus not human readable. This extra layer of indirection is cumbersome.

Making minor changes to the source code for a Docker process is tedious — the whole container has to be rebuilt, pushed to the registry and pulled again. One workaround we use in development is to look for and automatically apply a patch file when starting our Docker processes. This is a benefit of using Python — no recompilation of our applications is required.

FUTURE WORK

There are a number of areas where improvements could be made. The first is high availability of our master controller and its custom scheduler. As mentioned earlier, using or learning from the Marathon framework would be beneficial. Many of our components are stateful, and significant work is required to move all the state to a distributed store. Much of our state is kept in a Redis in-memory data store [34], which could be changed to a cluster-based implementation, or we could consider using ZooKeeper. There are also issues with lost data while switching over to new nodes.

The NUMA memory modelling should be improved, and we need better isolation of reserved CPU cores, GPUs and network bandwidth. Many of these issues need to be tackled in the Docker runtime, or even the Linux kernel level, which makes this very challenging.

The multi-stage launch scheme is a good candidate for a custom executor; we chose the current approach because there does not appear to be an easy way to extend, rather than replace, the default executor.

CONCLUSION

We have provided a basic overview of the MeerKAT radio telescope and its Science Data Processor. The history of the SDP deployment was discussed as was the major motivating factor for moving from a static deployment process to a dynamic one. We discussed container orchestration tools with a focus on Mesos before delving into the details of our scheduler.

By adding custom resources and attributes to agents we showed that it is possible to apply Mesos to heterogeneous NUMA compute clusters and still maintain optimal performance. The Docker runtime helps to isolate CPU and memory resources. Our work on fine-grained allocation of GPU resources, and awareness of network segments and NIC capabilities is also beneficial, while not completely supported by Docker.

Our implementation is effective, but can be improved by making it more fault tolerant, and thus providing higher availability. Some of the limitations could only be overcome by changing the Docker runtime and Linux kernel.

REFERENCES

[1] MeerKAT Fact Sheet, SKA SA, http://www.ska.ac.za/wp-content/uploads/2016/07/meerkat-fact-sheet-2016.pdf

[2] Square Kilometre Array, http://skatelescope.org

[3] Automation & Orchestration With Docker & Containers — The New Stack eBook Series, https://thenewstack.io/ebooks/docker-and-containers/automation-orchestration-docker-containers

[4] Docker, https://www.docker.com

[5] Apache Mesos, http://mesos.apache.org

[6] B. Hindman *et al.*, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", *NSDI'11*, p. 295, 2011.

[7] L. R. S. Brederode and L. van den Heever, "MeerKAT: Project Status Report", presented at ICALEPCS'17, Barcelona, Spain, October 2017, paper THPHA066, this conference.

[8] M. Fowler, Microservices, https://martinfowler.com/articles/microservices.html

[9] KATCP Guidelines, http://pythonhosted.org/katcp/_downloads/NRF-KAT7-6.0-IFCE-002-Rev5.pdf

[10] FAI — Fully Automatic Installation, https://fai-project.org

[11] Ansible, https://www.ansible.com

[12] Jenkins, https://jenkins.io

[13] Elasticsearch, https://www.elastic.co/products/elasticsearch

[14] Logstash, https://www.elastic.co/products/logstash

[15] Kibana, https://www.elastic.co/products/kibana

[16] Grafana, https://grafana.com

[17] Prometheus, https://prometheus.io

[18] Kubernetes, https://kubernetes.io

[19] Docker Swarm, https://docs.docker.com/engine/swarm

[20] HashiCorp Nomad, https://www.nomadproject.io

[21] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. "Omega: flexible, scalable schedulers for large compute clusters", in *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, Prague, Czech Republic, Apr. 2013, pp. 351–364.

[22] Advanced Scheduling in Kubernetes, http://blog.kubernetes.io/2017/03/advanced-scheduling-in-kubernetes.html

[23] Docker Swarm scheduler strategies, https://github.com/docker/swarm/tree/master/scheduler/strategy

[24] An Introduction to Mesosphere, https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere

[25] Marathon — A container orchestration platform for Mesos and DC/OS, https://mesosphere.github.io/marathon

[26] Chronos — A fault tolerant job scheduler for Mesos, https://mesos.github.io/chronos

[27] Running Spark on Mesos, https://spark.apache.org/docs/latest/running-on-mesos.html

[28] A. Ghodsi *et al.*, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types", *NSDI'11*, p. 323, 2011.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPA04

[29] A. Muddukrishna, P. A. Jonsson, and M. Brorsson, "Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors", *Scientific Programming*, vol. 2015, Article ID 981759, 16 pages, 2015.

[30] P. Petrides, and P. Trancoso, "Heterogeneous-and NUMA-aware scheduling for many-core architectures", in *Proc. of the 10th ACM International Systems and Storage Conference (SYSTOR'17)*, Haifa, Israel, May. 2017, Article No. 2.

[31] Portable Hardware Locality (hwloc), https://www.open-mpi.org/projects/hwloc/lstopo

[32] Python Programming Language, https://www.python.org

[33] A pure Python implementation of Mesos scheduler and executor, https://github.com/douban/pymesos

[34] Redis in-memory data structure store, https://redis.io