

Genisys: A Resource Management and Placement Mechanism for HPC and Datacenter Applications under Kubernetes

Georgios Zervas

Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Angelos Bilas*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

**Genisys, a Resource Management and Placement Mechanism for HPC
and Datacenter applications under Kubernetes**

Thesis submitted by
Georgios Zervas
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Georgios Zervas

Committee approvals: _____
Angelos Bilas
Professor, Thesis Supervisor

Konstantinos Magoutis
Associate Professor, Committee Member

Polyvios Pratikakis
Associate Professor, Committee Member

Departmental approval: _____
Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, July 2022

Contents

	i
	v
1 Introduction	9
2 Design Overview	13
2.1 Virtual Cluster Design	13
2.2 Single-point Resource allocation	13
2.3 Slurm-Kubernetes Interface	15
2.4 Placement Policies	16
3 Implementation	21
3.1 Preparing and Deploying Virtual Clusters	21
3.2 Scheduling Extensions to Support Hybrid Workloads	22
4 Experimental Methodology	25
4.1 Workload Generation	29
4.1.1 Pilot Workload	30
4.1.2 Small Task Size Workload	30
4.1.3 Large Task Size Workload	31
5 Experimental Evaluation	33
5.1 Genisys Max Loaded Policy vs Genisys Least Loaded Policy Performance Comparison	33
5.2 Genisys vs Unmanaged Cluster Performance Comparison	34
5.2.1 Pilot Workload Evaluation	35
5.2.2 Small Task Size Workload Evaluation	36
5.2.3 Large Task Size Workload Evaluation	37
5.3 Genisys vs Partitioned Cluster Performance Comparison	39
5.3.1 Pilot Workload Evaluation	39
5.3.2 Small Task Size Workload Evaluation	41
5.3.3 Large Task Size Workload Evaluation	42
5.4 Genisys vs Thread Partitioned Cluster Performance Comparison	43

5.4.1	Genisys vs Thread Partitioned 50% Comparison	44
5.4.2	Genisys vs Thread Partitioned 75% Comparison	45
6	Related Work	47
6.1	HPC-Cloud Convergence	47
6.2	Performance of Containers in HPC	47
6.3	Workload Scheduling	48
7	Conclusion and Future Work	49
8	Appendix	51
8.1	Pod Placement Controller	51
8.2	Resource Updater Controller	52
8.3	Slurm Connector	53
8.4	Kubernetes API Communicator	54
	Bibliography	55

List of Figures

2.1	Each container instance of a virtual cluster runs in a different physical machine, while multiple virtual clusters may run in parallel. The custom Slurm job placement plugin communicates with Genisys to perform job placement. These jobs are visible at the Kubernetes level as “dummy allocations”.	14
2.2	Genisys periodically monitors the performance of data center applications, to compare their current performance against their target, and adjusts the size and the number of pods according to the difference.	17
2.3	Genisys is an external pod scheduler, monitoring metrics using Prometheus and a custom metric server. The custom metric server inserts metrics into the metrics database of Prometheus.	18
5.1	The execution time of each workload between Genisys and Unmanaged when using the Pilot workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.	35
5.2	The execution time of each workload between Genisys and Unmanaged when using the small workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.	37

5.3	The execution time of each workload between Genisys and Unmanaged when using the large workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster's CPU utilization for each different scenario.	38
5.4	The execution time of each workload between Genisys and Partitioned when using the Pilot workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster's CPU utilization for each different scenario.	40
5.5	The execution time of each workload between Genisys and Partitioned when using the small workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster's CPU utilization for each different scenario.	41
5.6	The execution time of each workload between Genisys and Partitioned when using the large workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster's CPU utilization for each different scenario.	42
5.7	The execution time of each workload step in six different scenarios when using the Pilot workload. In the first graph we show the individual execution times of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster's CPU utilization for each different scenario. .	43

5.8	The execution time of each workload step in six different scenarios when using the small workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster's CPU utilization for each different scenario. .	44
5.9	The execution time of each workload step in six different scenarios when using the large workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster's CPU utilization for each different scenario. .	45

Abstract

Today, Cloud and HPC workloads tend to use different approaches for managing resources. However, as more and more applications require a mixture of both high-performance and data processing computation, convergence of Cloud and HPC resource management is becoming a necessity. Cloud-oriented resource management strives to share physical resources across applications to improve infrastructure efficiency. On the other hand, the HPC community prefers to rely on job queueing mechanisms to coordinate among tasks, favoring dedicated use of physical resources by each application.

In this work, we design a combined Slurm-Kubernetes system that is able to run unmodified HPC workloads under Kubernetes, alongside other, non-HPC applications. First, we containerize the whole HPC execution environment into a *virtual cluster*, giving each user a private HPC context, with common libraries and utilities built-in, like the Slurm job scheduler. Second, we design a custom Slurm-Kubernetes protocol that allows Slurm to dynamically request resources from Kubernetes. Essentially, in our system the Slurm controller delegates placement and scheduling decisions to Kubernetes, thus establishing a centralized resource management endpoint for all available resources. Third, our custom Kubernetes scheduler applies different placement policies depending on the workload type. We evaluate the performance of our system compared to statically partitioned Kubernetes and Slurm-based HPC clusters and demonstrate its ability to allow the joint execution of applications with seemingly conflicting requirements on the same infrastructure with minimal interference.

Acknowledgements

First of all, I would like to express my sincere appreciation to my advisor, Professor Angelos Bilas for his support, assistance and continuous guidance throughout both my undergraduate and postgraduate studies.

I would also like to acknowledge Dr. Anthony Chazapis for his invaluable support and advice which contributed to a great extend to shaping my thesis. Furthermore, I am also grateful to Yiannis Sfakianakis and Christos Kozanitis for their support on my first steps to the Computer Architecture and VLSI Systems Laboratory, training me and providing insightful advice, guidance and help.

My best regards to the members of the Computer Architecture and VLSI Systems Laboratory , Manos Pavlidakis and Stelios Mavridis for their valuable feedback on my thesis presentation.

Special thanks to the members of my committee, Prof. Kostas Magoutis and Prof. Polyvios Pratikakis for their valuable questions and comments during my defense.

Finally, I want to thank my family (my father Markos, my mother Anna and my sister Eugenia) and my friends for the support, encouragement and trust in me throughout the whole process of my thesis.

Chapter 1

Introduction

Cloud and HPC computing environments are mostly similar in hardware specifications, but differ largely in the software stack and how it manages available resources. Cloud providers use virtualization mechanisms to facilitate sharing, whereas in HPC clusters workloads are allocated resources, based on requirements given by the user when submitting the respective job.

As the complexity of modern applications increases, it is not uncommon for deployments to include parallel provisioning of backend services such as web servers and databases, as well as on-demand execution of data analytics pipelines and HPC codes. For such workloads, it is essential to accommodate both resource allocation schemes on the same hardware infrastructure, exploiting resource sharing, but also avoiding interference as much as possible.

We aim at combining the best features of HPC, Big Data and Cloud to create an infrastructure and ecosystem, where users deploy complex applications processing large amounts of data with the simplicity and efficiency provided by cloud infrastructures. It is not uncommon for an application workflow to rely on 5 different frameworks, such as Apache Spark [4], Dask [5], TensorFlow [8]/Keras [6], Kafka [2], and MPI [7]. In this context, we were faced with the challenge of mixing different types of execution frameworks as part of the same processing pipeline, as well as running multiple such pipelines on a shared HPC cluster.

In this work, we explore the convergence of Cloud and HPC in a common, container-based environment, backed by Kubernetes, the most prominent distributed container orchestration framework [9]. Containers are gaining ground as the preferred deployment method in the Cloud, as they implement a convenient packaging scheme for applications, they are lightweight when running, and provide isolation between instances for security purposes. Kubernetes provides abstractions for hardware resources and automatically scales service replicas to meet demand, while providing redundancies to cope with unadvertised failures.

To make application pipelines scalable and reproducible, we opted to containerize all individual components and use a high-level workflow orchestration framework. This proved to be a natural match with the Kubernetes environment and its microservices architecture, thus we decided early on to deploy Kubernetes on all cluster nodes.

The HPC world has cautiously been following the containerization trend, primarily utilizing containers as a portable method to bundle applications with associated library dependencies. These containers are then typically submitted as jobs using Slurm, a popular workload manager responsible for coordinating the allocation of resources throughout the cluster, via submission queues shared among multiple users.

To run HPC applications in Kubernetes, we introduce the concept of a *virtual cluster*, as a group of multiple container instances that function as a unified cluster environment from the user’s perspective.

Virtual clusters allow the seamless integration of MPI steps in workflows by offering an easily deployable, portable, and extensible HPC environment that is created on demand in containers. By using virtual clusters, whose Slurm setups delegate job placement to the central Kubernetes scheduler, we have been able to run unmodified HPC workloads in Kubernetes, making it possible to colocate them with other workloads and services over the same physical hardware. In addition Genisys maximizes the cluster’s efficiency when running multiple parallel workflow steps.

Each node in a virtual cluster embeds all necessary libraries and utilities, as well as a private Slurm deployment; the user working inside a virtual cluster can only view and manage jobs submitted from within the same context. In practice though, each such Slurm setup does not function independently. We extend the Slurm controller with a custom protocol, to communicate with the central Kubernetes scheduler when requiring resources, effectively placing Kubernetes in charge of resource allocations for the whole cluster. Moreover, we developed *Genisys*, a custom Kubernetes scheduler that distinguishes between “HPC” and “Data Center” type services (typical Kubernetes deployments that run in other containers), in order to apply different allocation policies and maximize overall usage. In cases where HPC workloads do not consume all resources in the nodes at which they spread out, Genisys places data center services where spare CPU cycles are available, while constantly satisfying their user-defined performance targets. Therefore, HPC and data center workloads execute transparently on the same infrastructure, achieving high levels of CPU utilization.

This integration has several benefits:

- **Compatibility:** Supporting Slurm inside the virtual clusters is crucial in order to keep compatibility with existing scripts written for Slurm.

- Colocation: By following the approach of containerizing the whole run-time environment and using Kubernetes as the substrate, we are able to run hybrid workloads on top of the same physical cluster, optimizing for high utilization and avoiding static cluster partitioning for HPC and data center tasks.
- Portability: The containerized environment offered with virtual clusters allows users to install different dependencies without polluting the bare metal infrastructure or requiring different versions of the same libraries. Each user can create personalized container images containing only the libraries required by the specified workload. It also makes the migration to a different physical cluster supporting Kubernetes possible, just by transferring the container images to the other system and deploying them using the same Kubernetes scripts.

Through the scope of this work we present a method to run HPC workloads in Kubernetes using portable and extensible containerized environments called virtual clusters. Virtual clusters include Slurm, so users can run existing scripts unmodified, and are deployed alongside other Kubernetes services on the same physical cluster. To avoid resource allocation conflicts, we integrate Slurm with Kubernetes, by extending the Slurm controller to delegate placement decisions to Genisys, our custom Kubernetes scheduler (Genisys). We compare our system’s performance to vanilla Slurm, an Unmanaged and a Partitioned cluster scenario running multiple workloads. Our results indicate that Genisys is able to perform to near unmodified Slurm levels while outperforms Unmanaged by 35% and Partitioned by 38% across all workloads.

The main contributions of this work are:

- Virtual Clusters: Virtual Clusters offer a containerized run-time environment able to run HPC workloads utilizing Slurm and supporting RDMA operations while offering per user isolation. The concept Virtual Cluster is the backbone of this work as it offers portability and compatibility with existing scripts through Slurm.
- Genisys: Genisys is our custom Kubernetes scheduler able to apply custom resource allocation and placement policies for both HPC and data-center workloads. In contrast to the default Kubernetes scheduler, Genisys is crucial in this context as attempts to fit as many as possible HPC tasks to the underlying infrastructure while offering a Kubernetes - Slurm communication interface.
- Slurm Modification: We modified the Slurm controller in order to communicate with Genisys in order to place tasks and allocate resources.

This type of modification is needed in order to avoid resource overlapping between HPC and data-center tasks as by default there is no communication between Slurm and Kubernetes.

Chapter 2

Design Overview

2.1 Virtual Cluster Design

A *virtual cluster*, is a group of container instances that virtualizes an environment to run HPC workloads that use MPI and other software frameworks. From the perspective of applications, virtual clusters are indistinguishable from physical nodes that execute instances of MPI processes in parallel, as all physical processing cores, RAM, the low-latency InfiniBand network, and accelerators are available in each container context. Each virtual cluster spans all physical nodes and multiple virtual clusters can co-exist over the same set of physical nodes, as shown in Fig. 2.1, which presents the high-level concepts of the overall design.

Inside each virtual cluster, as part of the bundled software stack, we deploy a private Slurm context, so users can invoke existing scripts to run HPC workloads. One of the virtual cluster nodes acts as the Slurm controller, while all virtual cluster nodes run the Slurm agent and register with the controller. Configuration of the Slurm deployment is automatically done at virtual cluster initialization. Unmodified, the virtual-cluster-local Slurm would perform resource allocation and scheduling of Slurm jobs as if it were in control of the whole cluster. Each independent Slurm installation is isolated inside its own containers and does not account for the presence of other containers running and consuming computing resources; that being other virtual clusters or typical Kubernetes services.

2.2 Single-point Resource allocation

To schedule and place workloads across multiple virtual clusters and prevent the interference introduced by overlapping jobs, we have modified the Slurm controller's placement mechanism to delegate all respective decisions to the external Kubernetes scheduler. The Kubernetes scheduler in this scheme

is the central authority that has the full knowledge of the cluster’s current resource allocations and acts as a global coordinator for new requests. Moreover, Genisys, our custom Kubernetes scheduler implementation (described in 3.2) distinguishes between “HPC” and “Data Center” type workloads, in order to improve the overall utilization of available hardware. Data center services do not use virtual clusters, but are deployed in Kubernetes as deployments, jobs, or other API objects that execute containers alongside the ones used by virtual clusters.

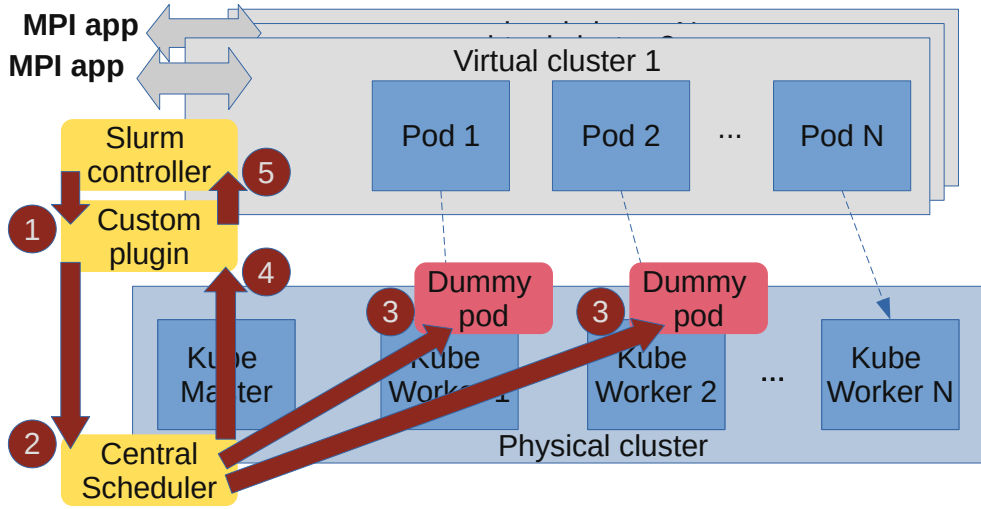


Figure 2.1: Each container instance of a virtual cluster runs in a different physical machine, while multiple virtual clusters may run in parallel. The custom Slurm job placement plugin communicates with Genisys to perform job placement. These jobs are visible at the Kubernetes level as “dummy allocations”.

Fig. 2.1 illustrates the steps involved in the communication between virtual clusters and the cluster-wide Kubernetes scheduler (HPC workloads), which works by the following description:

1. On job initialization Slurm sends an allocation request to the main Kubernetes controller via our custom plugin. In this request Slurm specifies the resources needed for the job (node count, CPU count, etc.).
2. Our modified version of Slurm uses a custom Kubernetes plugin written in Golang making use of the Kubernetes API in order to forward these allocations to Genisys. This plugin takes as input the Slurm job’s specifications and creates a Kubernetes allocation of placeholder “dummy pods”¹. We specify as “dummy pods” a Kubernetes deployment that

¹A *pod* is a set of one or more container instances running as a single entity in Kubernetes.

is directly mapped to an actual Slurm job and is responsible for allocating the resources needed by the corresponding Slurm job. Dummy allocations have specific CPU and memory requirements that match the requirements of the submitted Slurm job.

3. On receiving a dummy allocation for a Slurm job placement, Genisys iterates over the cluster nodes and checks if a set of nodes with enough resources for the job is available. If so, the custom controller schedules dummy containers allocating the resources from Kubernetes.
4. This allocation is communicated back to the plugin as a node list.
5. The plugin, in turn, forwards the response to the Slurm controller. The node list contains the selected nodes for the Slurm job deployment. If no suitable set of nodes is found over this period, the controller saves the allocation on a priority queue for later deployment.

Dummy containers are practically idle; they consume no resources themselves, but act as placeholders for the allocation of resources that will be used by the actual jobs inside the virtual cluster. Also, note that this design is not dependent on the container runtime used by Kubernetes, and should be compatible with future versions of Kubernetes that use containerd directly, skipping Docker, as well as modified deployments with Singularity containers.

2.3 Slurm-Kubernetes Interface

We have modified the decision making part of the Slurm controller that runs inside virtual clusters in order to override the default placement policy and delegate the respective decision to the Kubernetes scheduler. The Slurm controller uses a node bitmap in order to represent the reservation state of the available HPC nodes and find suitable nodes to place incoming jobs. For job placement, Slurm uses the `_job_test()` function, which is called by the controller when a new job arrives. `_job_test()` takes as input the job's resource requirements and a node bitmap containing the Slurm's node state. As a next step, it checks if a set of computing nodes is available, in order to place the job by calling `_select_nodes()`. The latter takes as input the job's resource requirements, and if an available node list for the job placement is found, it returns a list with the selected nodes and proceeds to mark them as allocated in the node bitmap. If the selection process returns an empty node list, then Slurm schedules the job for later placement, else Slurm proceeds to start the job on the nodes selected.

To delegate all job placement decisions to the Kubernetes scheduler, we have overridden Slurm's node selection process and forward the job placement request externally. First, we reset the node bitmap after the `_select_nodes()`

function returns, in order to keep it unmodified from the Slurm’s selection process. Second, we implement a custom plugin written in Golang that takes as an input the Slurm’s job resource requirements (CPU cores, node count and memory limit per process) and creates a mirror Kubernetes allocation of “dummy pods” using the same resources, to be scheduled by Genisys. After the dummy pods are placed by the scheduler, the custom plugin returns a node list with the selected nodes back to the Slurm controller for the specified job. On receiving the list, Slurm modifies the node bitmap and places the job.

The custom plugin, which implements the interfacing between Slurm and Kubernetes, runs next to each Slurm controller. On startup, it reads the Kubernetes cluster configuration, to be able to communicate with the Kubernetes API server, and records the virtual cluster’s namespace, to create placeholder allocations using the corresponding service account. Then, it is ready to create dummy pods representing Slurm jobs. The plugin constantly monitors the state of such deployments until all pods are scheduled successfully. When the dummy pods are ready, it fetches the list of nodes that the containers have been placed on and forwards it back to the Slurm controller, for the actual Slurm job to be deployed.

2.4 Placement Policies

Genisys ensures each virtual cluster does not share resources with other virtual clusters or data-center services. When a data-center service or HPC job is deployed, Genisys iterates over an internal free resource list for each node and attempts to find which nodes have enough free resources for the task to fit in. If *resource over subscription* is not enabled Genisys will always place tasks on nodes with enough free resources to fit in. Furthermore, the scheduler supports two different placement policies:

- The *Least Loaded Selection Policy* attempts to find a list with the least loaded nodes on the cluster in order to place the task. The main advantage of this policy is that by choosing the least loaded nodes we are able to fit more jobs on a given set of nodes running in parallel, achieving higher cluster resource utilization. On the other hand, if the total cluster workload size is small we are going to use a high number of nodes at low utilization, thus achieving low power efficiency.
- The *Max Loaded Selection Policy* attempts to find a list with the max loaded nodes that the task fits in. The main advantage of this policy is that in low to moderate cluster utilization Genisys is able to pack as many services using the least number of nodes, allowing for high cluster energy efficiency. The main drawback is that if the workload consists

of multiple multi-node tasks, we might not be able to fit as many tasks on the cluster in contrast to when spreading the load (as some "loaded" nodes will not have enough free resources).

In general, Genisys does not schedule "HPC" workloads for execution, when they require more resources to execute than what is available. Therefore, Slurm queues inside virtual clusters wait for running "HPC" workloads to free enough resources when they finish executing.

For data center workloads, Genisys always allows sharing of resources. Genisys estimates the aggregate resources that are required for the data center workload to achieve a user-defined performance objective (e.g. latency, throughput). It manages four types of resources: number of cores, memory size, I/O bandwidth, and network bandwidth. Genisys performs its estimations using a feedback control loop similar to Skynet. Afterwards, it decides on the size, the number, and the placement of containers to physical nodes using the max-load or least-loaded policy approach.

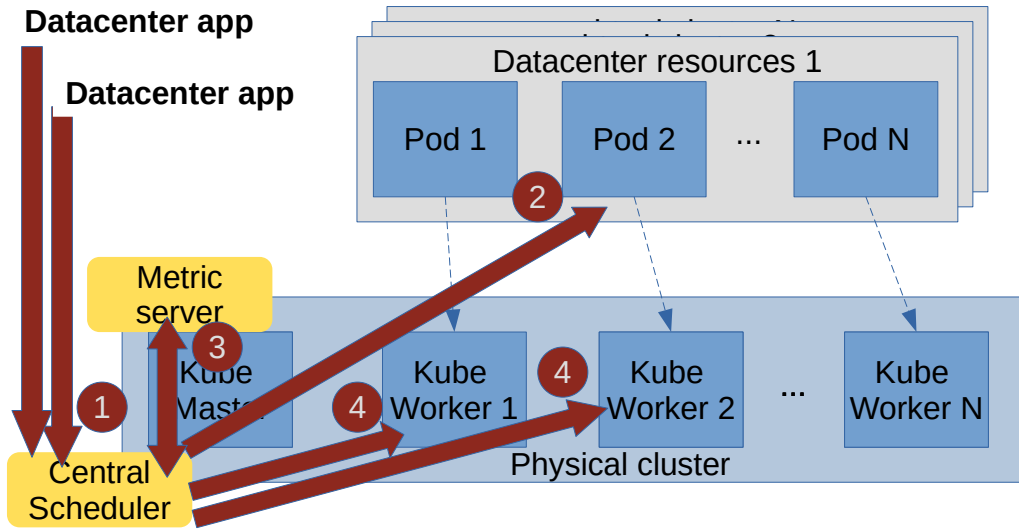


Figure 2.2: Genisys periodically monitors the performance of data center applications, to compare their current performance against their target, and adjusts the size and the number of pods according to the difference.

Fig. 2.2 illustrates the steps involved to properly size and place data center workloads in the underlying Kubernetes nodes.

1. Users issue a request for a new data center workload along with the required performance objective, i.e. a webserver with ten milliseconds latency.
2. For new applications, Genisys creates one pod with the following spec,

one CPU core, one gigabyte of memory, ten megabytes per second of I/O and network throughput, and places it in the best fitting physical node. For running applications, it adapts the size and number of pods to minimize the difference of the observed versus the current performance.

3. Periodically, Genisys gets feedback about the performance of the workload and if it differs significantly from the target, Genisys triggers the procedure to adjust the size and the number of pods accordingly.
4. Finally, Genisys places the newly created pods (if any), in the best fitting servers.

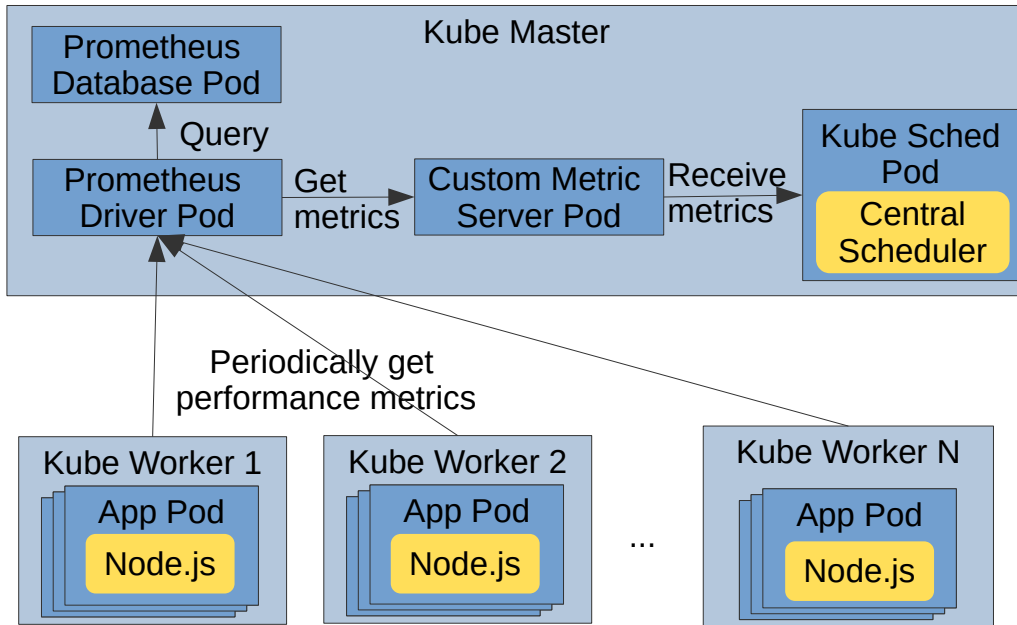


Figure 2.3: Genisys is an external pod scheduler, monitoring metrics using Prometheus and a custom metric server. The custom metric server inserts metrics into the metrics database of Prometheus.

Figure 2.3 illustrates the steps, executed once every second, to capture application metrics:

1. Each Node.js server periodically queries the application metrics and appends them in a file.
2. The Prometheus agent pod reads the new entries from the file and advertizes them to the Prometheus driver pod.
3. The Prometheus driver pod receives the metrics from all active agents and sends them to the Prometheus DB pod.

4. The Prometheus DB pod collects the metrics from the driver pod and stores them in its database.
5. The custom metric server pod, queries the application metrics from the Prometheus DB pod, and stores them locally.
6. Genisys requests the pod metrics from the custom metric server using the metric server API.

In cases where no performance metrics are available for a data center service, the user can define the resources need by the service in a static manner.

1. User issues a request for a new data center workload along with the required resources that are needed for execution (eg. CPU, memory, number of containers).
2. Genisys iterates over the current cluster's nodes in order to find a suitable set that the job fits into (according to the selected placement policy).
3. If a set of nodes is available then Genisys binds the job to the selected nodes.
4. If the cluster does not have enough free resources for the job to fit, then the job gets scheduled on the next selection cycle and remains in pending state.

Colocating HPC tasks with the data center services is configurable. The default behaviour allows tasks of both types to use the same nodes and share resources. The other option is to perform type-based placement, implicitly partitioning the nodes by placing HPC tasks on some nodes and data center tasks on others. This approach may minimize interference introduced by task colocation, but also reduces resource utilization efficiency.

Chapter 3

Implementation

In the following sections we describe how virtual clusters are prepared, deployed, and integrated with Kubernetes, starting with the containerization of the HPC context and then to the implementation details of the Genisys scheduler that allows the serial execution of HPC workloads, as well as efficiently colocating data center jobs.

3.1 Preparing and Deploying Virtual Clusters

Virtual cluster container images are prepared as “typical” Docker images, by starting from some base Linux distribution and adding layer after layer of development tools, libraries, and other software. Our reference images are based on CentOS and the Mellanox OpenFabrics Enterprise Distribution (OFED), which includes Open MPI with InfiniBand support as well as other libraries. In addition, we install several extra libraries and frameworks (i.e., CUDA, GROMACS, TensorFlow, Horovod, and others), the Slurm workload manager, as well as utilities to help in evaluating application performance. This base container recipe is available to our users, so they can tailor it to their needs, with different software versions or a completely diverse set of libraries and tools.

Upon instantiation, each virtual cluster container actually runs the SSH daemon as its primary process. The instance startup script first waits for all pods (nodes in the virtual cluster) to be ready and then creates all necessary configuration on the first pod: keys for password-less SSH connectivity, MPI hostfile, and Slurm configuration at `/etc/slurm.conf`. When done, it loops over all other pods and copies over the configuration. As the last step, it starts the Slurm controller (the first pod serves as the controller) and Slurm agents on all pods. Each virtual cluster is deployed using a Kubernetes DaemonSet, which assigns one pod per physical machine. As MPI developers usually assume similar capabilities and equal network-level distances across

nodes, placing a single pod in each node is more convenient and produces expected results.

3.2 Scheduling Extensions to Support Hybrid Workloads

The scheduling of dummy pods does not *require* a custom Kubernetes scheduler, however without special arrangements for HPC workloads, the default scheduler may place multiple HPC jobs on the same nodes, maximizing interference. To this end, we have extended our Genisys scheduler to support both “HPC” and “Data Center” workloads and enforce different types of placement policies.

To support HPC workloads, we label these types of tasks as “SLURM-JOB” in order to distinguish them from other workloads running on the same cluster. Allocations for Slurm applications happen in a static manner and Genisys can be configured to avoid colocating them with other jobs marked as “SLURM-JOB”, in order to keep performance optimal and avoid interference. This policy may be selected because of the lack of available metrics offered by MPI applications and their sensitivity due to synchronization barriers.

On the other hand, for data center workloads that include a user-defined performance objective, which Genisys must achieve during their execution. Genisys monitors periodically (every ten seconds) the performance of each running data center service to get feedback about the effectiveness of its current resource allocation. In case of a performance violation in a workload, Genisys increases its resource allocation according to the measured drop in performance. Correspondingly, in case of significant increase in performance, Genisys decreases the estimation according to feedback from runtime performance. For the new resource estimations, Genisys, considers also the history about previous performance measurements, which is affected mainly by the workload mix.

Genisys uses an extension of the Kubernetes API server, the custom metric server, for monitoring the performance objectives of data center workloads, and the Prometheus monitoring system [1] to collect application metrics. Prometheus includes a pod for the Prometheus database that stores the metrics, and a driver pod that collects application metrics from multiple application agent pods. Genisys requires applications to bundle a REST server that exposes all performance metrics of interest to an endpoint (e.g., /metrics) to Prometheus. We augment the containers of each application we use in our data center workloads to include a REST server written in Node.js. Genisys is implemented next to the default Kubernetes scheduler, which means that depending on the definition of each service, we can choose either the default scheduler or Genisys.

For data center workloads that do not support user-defined performance objectives, Genisys allocates resources in a static manner for each service instance. The user has to define the resources that the workload is going to use prior to submission.

Chapter 4

Experimental Methodology

We evaluate our system by running a mixture of MPI workloads and other services on the same cluster, and measuring the overall efficiency through the total runtime of all applications combined and the individual runtime per application. Typically, on a Slurm-only cluster, jobs request a specific amount of resources that are assigned in a static manner. In contrast, we colocate HPC with non-HPC tasks in the dynamic resource allocation environment provided by Genisys and investigate what are the performance advantages of running a Genisys-managed unified cluster compared to a (i) static partitioned and (ii) unmanaged cluster.

Our hardware setup consists of 5 identical servers, each with a single 32-core/64-thread AMD EPYC 7551P processor (running at 2.00GHz) and 128GB of memory, for 320 hyperthreads in total. All servers have SSDs installed as their primary storage devices, used by Docker for running containers, and are interconnected via 56Gb/s InfiniBand supporting RDMA operations. The software stack is based on Red Hat Enterprise Linux 7.6 and vanilla Kubernetes 1.19.7.

In order to evaluate the performance of HPC tasks, we have created a multi step MPI workload using benchmarks from the NAS Parallel Benchmark Suite [11], a set of scientific benchmarks which have become widely accepted as a reliable indicator of MPI performance. The exact benchmarks used in the workload are presented in the table below with the configured parameters.:

HPC Benchmarks and Sizes					
Benchmark Name	Description	CPU Threads (Pilot work-load)	CPU Threads (Small work-load)	CPU Threads (Large work-load)	Benchmark Classes
CG	Conjugate Gradient, irregular memory access and communication	32-128	32	-	D
EP	Embarrassingly Parallel	16-64	16	128	D-E
SP	Scalar Penta-diagonal solver	64	16	64	D
FT	discrete 3D fast Fourier Transform, all-to-all communication	128	-	-	D
MG	Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive	-	-	128	E
LU	Lower-Upper Gauss-Seidel solver	32	32	128	C
BT	Block Tri-diagonal solver	64	16	64	D

The “Data Center” workload consists of Nginx [26] and memcached [16] deployments, as well as a series of Spark benchmarks from the Spark-Bench [20] performance suite. The “Data Center” applications and benchmarks are presented on the table below with the configured parameters.

Data Center Workloads and Sizes					
Benchmark Name	Description	CPU Threads (Pilot workload)	CPU Threads (Small workload)	CPU Threads (Large workload)	Workload Size
Memcached	A distributed memory object caching system, intended to speed up dynamic web applications by alleviating database query load. This application is memory intensive	6	2	12	(Pilot workload) 200 million (Small workload) 50 million (Large workload) 200 million YCSB operations
Nginx	A web server, load balancer and HTTP cache. This application is CPU intensive.	6	2	12	(Pilot workload) 200000 (Small workload) 200000 (Large workload) 200000 requests
Sparkpi	Computes an approximation to pi.	15	8	24	(Pilot workload) 200000 (Small workload) 50000 (Large workload) 200000 slices
LR	Runs Logistic Regression over the input dataset.	15	8	24	(Pilot workload) 200000 (Small workload) 50000 (Large workload) 200000 slices
KMeans	Runs the KMeans algorithm over the input dataset.	15	8	24	(Pilot workload) 200000 (Small workload) 50000 (Large workload) 200000 slices

We run these these workloads in 7 different configurations:

- **Genisys Least Loaded Policy:** We deploy both workloads at the same time, colocating them over the same physical resources. We use the Genisys scheduler to place the data center deployments, while inside virtual clusters we deploy the workloads using our modified version of the Slurm workload manager that communicates with Genisys for placement decisions. For this scenario, we use the Genisys’s *Least Loaded Policy*, that chooses the least loaded nodes in order to place the tasks.
- **Genisys Max Loaded Policy:** We deploy both workloads at the same time, colocating them over the same physical resources. We use the Genisys scheduler to place the data-center deployments, while inside virtual clusters we deploy the workloads using our modified version of the Slurm workload manager that communicates with Genisys for placement decisions. For this scenario we use the Genisys’s *Max Loaded Policy* that chooses the max loaded nodes that the tasks fit in for placement.
- **Unmanaged:** We deploy both HPC and data center workloads at the same time, colocating them over the same physical resources. We use the default Kubernetes scheduler and an unmodified Slurm workload manager in the virtual clusters.
- **Partitioned:** We deploy both HPC and data center workloads at the same time, however at different nodes, as we statically partition the cluster into a 2-node Kubernetes and a 3-node Slurm partition. The Kubernetes partition runs with an unmodified Kubernetes scheduler, while the Slurm partition uses the default Slurm controller. We deploy both HPC and data center workloads at the same time, colocating them over the same physical resources. As in the Unmanaged scenario, we use the default Kubernetes scheduler and an unmodified Slurm workload manager in virtual clusters.
- **Thread Partitioned 50%:** We partition the cluster’s nodes by giving the (50%) of each node’s CPU capacity to the Kubernetes cluster while the rest (50%) to the Slurm controller. In the cluster’s nodes we run both unmodified Kubernetes and Slurm managers, configured to each use half the CPU resources of each node. We deploy both HPC and data center workloads at the same time, colocating them over the same physical resources.
- **Thread Partitioned 75%:** We partition the cluster’s nodes by giving the (75%) of each node’s CPU capacity to the Kubernetes cluster and

(75%) to the Slurm controller. In the cluster’s nodes we run both unmodified Kubernetes and Slurm managers, configured to each use (75%) the CPU resources of each node. We deploy both HPC and data center workloads at the same time, colocating them over the same physical resources. In this scenario HPC and data center workloads partially overlap over the same physical resources as Slurm and Kubernetes see a total of (150%) of each node’s resources available.

- **Slurm-Kubernetes Dedicated (baseline):** We run the HPC workloads under a 5-node unmodified Slurm cluster. Afterwards we run the data center workload on a 5-node Kubernetes cluster. We use this scenario as a performance baseline for the results of the above scenarios, as this is the typical scenario for a Slurm and Kubernetes installation time-sharing the same resources.

Deployment Scenarios Description	
Deployment Scenario	Description
Genisys Least Loaded Policy	For this scenario, we use the Genisys’s <i>Least Loaded Policy</i> , that chooses the least loaded nodes in order to place the tasks.
Genisys Max Loaded Policy	For this scenario we use the Genisys’s <i>Max Loaded Policy</i> that chooses the max loaded nodes that the tasks fit in for placement.
Unmanaged	We deploy both HPC and data center workloads at the same time, colocating them over the same physical resources. We use the default Kubernetes scheduler and an unmodified Slurm workload manager in the virtual clusters.
Partitioned	We statically partition the cluster into a 2-node Kubernetes and a 3-node Slurm partition. The Kubernetes partition runs with an unmodified Kubernetes scheduler, while the Slurm partition uses the default Slurm controller. We deploy both HPC and data center workloads at the same time.
Thread Partitioned 50%	We partition the cluster’s nodes by giving the (50%) of each node’s CPU capacity to the Kubernetes cluster while the rest (50%) to the Slurm controller.
Thread Partitioned 75%	We partition the cluster’s nodes by giving the (75%) of each node’s CPU capacity to the Kubernetes cluster and (75%) to the Slurm controller. In this scenario HPC and data center workloads partially overlap over the same physical resources as Slurm and Kubernetes see a total of (150%) of each node’s resources available.
Slurm-Kubernetes Dedicated (baseline)	We run the HPC workloads under a 5-node unmodified Slurm cluster. Afterwards we run the data center workload on a 5-node Kubernetes cluster.

4.1 Workload Generation

In order to evaluate our system under different conditions, we create three workloads consisting of both HPC and data center tasks.

- **Pilot workload:** We choose realistic HPC task sizes and their distribution by following traces outlined in [28], which analyzes the HPC workloads run on the Lomonosov-2 supercomputer, categorized according to resource allocation sizes and CPU consumption.
- **Small task size workload:** We choose both HPC and data center tasks of small sizes. In theory small tasks will be able to match more efficiently the cluster’s resources and achieve higher node utilization.
- **Large task size workload:** We choose both HPC and data center tasks of large sizes. In theory large tasks will not be able to match the cluster’s resources efficiently and will achieve lower node utilization.

We describe these workloads below:

4.1.1 Pilot Workload

In order to generate the HPC workload we follow the same job size distribution as described in [28] by allocating (5%) of the workload’s CPU time to 16 thread, (20%) to 32 thread, (65%) 64 thread, and (10%) to 128 thread jobs. We classify these 4 job categories as small (16 threads), medium (32 threads), medium-large (64 threads) and large (128 threads).

As the Nginx workload we spawn a deployment consisting of 5 Nginx servers each one allocating 6 CPU threads. The Nginx servers act as web servers serving a static web page with images for each request. In order to load the servers we use the Apache Bench [3] utility with 200 thousand total requests. The higher the Nginx performance the faster the workload finishes.

As the memcached workload we spawn a deployment consisting of 5 memcached servers each one allocating 6 CPU threads. Such a memcached cluster may in practice implement a distributed memory object caching system, intended to speed up dynamic web applications by alleviating database query load. This application is memory intensive. In order to load the deployment we use the YCSB workload generator [13] with a 200 million operation workload. The higher the memcached performance the faster the workload finishes.

For each spark workload we spawn 5 Spark workers each one allocating 20 CPU threads as a Kubernetes job. When the spark benchmark finishes the Spark workers get deleted freeing up the allocated resources.

4.1.2 Small Task Size Workload

In order to generate the HPC workload we chose MPI tasks of small (16 threads) and medium (32 threads) categories.

As the Nginx workload we spawn a deployment consisting of 5 Nginx servers, each one allocating 2 CPU threads. The Nginx servers act as web servers serving a static web page with images for each request. In order to load the servers we use the Apache Bench [3] utility with 200,000 total requests. The higher the Nginx performance the faster the workload finishes.

As the memcached workload we spawn 3 identical deployments consisting of 5 memcached servers, each allocating 2 CPU threads. Such a memcached cluster may in practice implement a distributed memory object caching system, intended to speed up dynamic web applications by alleviating database query load. This application is memory intensive. In order to load the deployment we use the YCSB workload generator [13] with a 50 million operation workload. The higher the memcached performance the faster the workload finishes.

For each Spark workload we spawn 5 Spark workers, each allocating 8 CPU threads as a Kubernetes job. When the Spark benchmark finishes the Spark workers get deleted freeing up the allocated resources.

4.1.3 Large Task Size Workload

In order to generate the HPC workload we choose MPI tasks of large (64 and 128 threads) categories.

As the Nginx workload we spawn a deployment consisting of 5 Nginx servers, each allocating 12 CPU threads. The Nginx servers act as web servers serving a static web page with images for each request. In order to load the servers we use the Apache Bench [3] utility with 200,000 total requests. The higher the Nginx performance the faster the workload finishes.

As the memcached workload we spawn 3 identical deployments consisting of 5 memcached servers, each allocating 12 CPU threads. Such a memcached cluster may in practice implement a distributed memory object caching system, intended to speed up dynamic web applications by alleviating database query load. This application is memory intensive. In order to load the deployment we use the YCSB workload generator [13] with a 200 million operation workload. The higher the memcached performance the faster the workload finishes.

For each Spark workload we spawn 5 Spark workers each one allocating 24 CPU threads as a Kubernetes job. When the Spark benchmark finishes the Spark workers get deleted freeing up the allocated resources.

Chapter 5

Experimental Evaluation

The main purpose of this work is to integrate HPC workloads to the Kubernetes ecosystem. There multiple approaches that we compare Genisys against in order to measure the relative performance.

For the first approach we deploy both Kubernetes and Slurm over the same physical cluster, while this allows for both HPC and data center tasks to be executed over the same infrastructure, it introduces performance violations to both HPC and data center tasks. The lack of communication between Kubernetes and Slurm for placement decisions lead to resource overlapping between different kinds of jobs.

For the second approach we split the cluster into two partitions, one for running Slurm and another for data center tasks using Kubernetes. While this approach eliminates the interference introduced as data center and HPC tasks run different partitions, it can lead to underutilized resources. When both Kubernetes and Slurm are restricted to their respective partitions, if the one of the two partitions is underutilized, the second partition will not be able to leverage the spare resources leading to larger execution times and larger queues of waiting jobs.

5.1 Genisys Max Loaded Policy vs Genisys Least Loaded Policy Performance Comparison

In this section we compare the performance of the three workloads between Genisys's Max Loaded versus Least Loaded policies. In general the Least Loaded scenario is able to run more tasks in parallel and achieve higher cluster utilization, as spreading the tasks to the least loaded nodes allows for more efficient fitting of the tasks when compared to choosing the most loaded nodes. In the case of the Max Loaded scenario, filling the most loaded nodes first often leads to situations where tasks that request a specific number of nodes cannot fit into the cluster. Some nodes of the cluster are fully loaded

and the number of nodes with enough space is smaller than the requested number of nodes.

In the first graphs of Fig. 5.7, Fig. 5.8 and Fig. 5.9 figures we see the execution time for individual task, as well as the total execution time for each workload. Genisys’s Least Loaded Policy is represented by the blue bars, while Genisys’s Max Loaded Policy is represented by the green bars. On average, across all the three workloads, the Least Loaded policy achieved (14%) lower total execution times when compared to Max Loaded, as it was able to fit the tasks better to the cluster resources and run more tasks in parallel. There was also a (4%) higher individual task performance when using the Least Loaded policy. We assume that this performance benefit is observed due to the spread of the workloads across more nodes, resulting in better utilization of the cluster’s resources. Across all workloads the Least Loaded policy was able to achieve (17%) higher total CPU utilization in the cluster.

After this comparison we conclude that the Least Loaded Policy is better for both resource utilization and performance. In the next sections we use the Least Loaded Policy in order to evaluate Genisys in other configurations.

5.2 Genisys vs Unmanaged Cluster Performance Comparison

One approach to colocate both HPC and Datacenter tasks under the same physical infrastructure would be to deploy both Kubernetes and Slurm over the same cluster. While this approach allows users to run both HPC and data center tasks as both Kubernetes and Slurm would be available, it introduces high interference between the tasks as Kubernetes and Slurm are not able to coordinate task placement. This lack of coordination may lead to resource over-subscription as both Kubernetes and Slurm see each individual node’s resources as (100%) available. Especially in the case of MPI tasks, this resource over-subscription may have catastrophic results in their performance as the threads running in congested nodes will finish slower than the rest of the threads, leading in slower execution times for the whole job. MPI threads running on other nodes usually spin until the slower threads finish the execution, which leads in high waste of the cluster resources.

On the other hand, Genisys achieves coordination between Kubernetes and Slurm regarding task placement and always guarantees that each task will have enough resources on the nodes that it is going to be placed.

In order to evaluate and prove that the above assumption is correct, we run the three workloads using Genisys’s Least Loaded Policy, Genisys’s Max Loaded Policy and Unmanaged scenarios and show the performance of each individual task while also the total execution time of each workload.

In Figures Fig. 5.7, Fig. 5.8 and Fig. 5.9 we can see the execution performance and CPU utilization for each workload across all scenarios. These figures although not used in this section are probably useful as they represent the total system performance across all scenarios.

5.2.1 Pilot Workload Evaluation

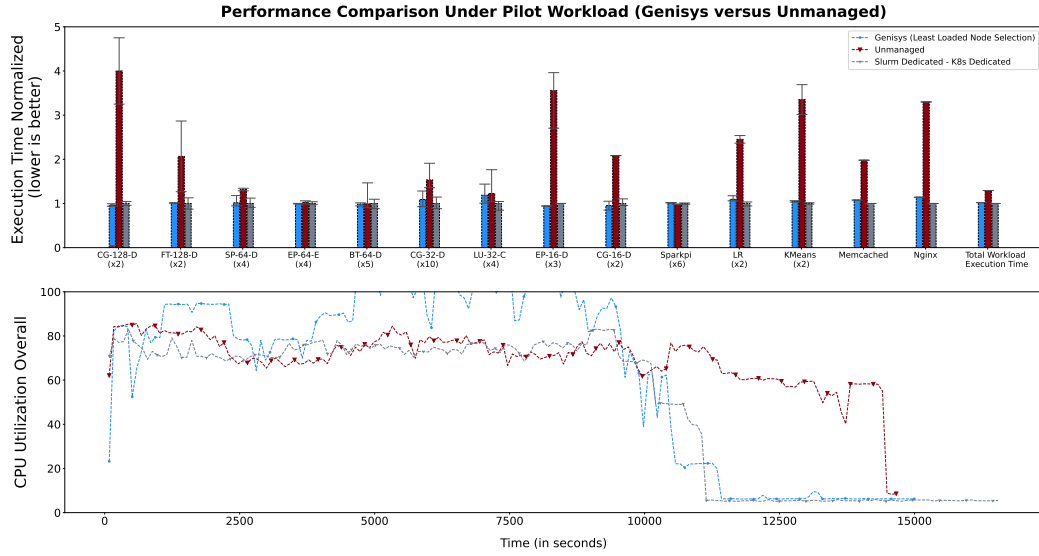


Figure 5.1: The execution time of each workload between Genisys and Unmanaged when using the Pilot workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.

We first compare the performance between the Genisys and Unmanaged scenarios and investigate if there is a speedup for the total and individual execution times of the Pilot workload when Genisys is used. The main goal of this experiment is to show that the interference introduced under the Unmanaged Scenario is not viable and that Genisys is able to improve the execution times of tasks to near native levels.

In the first graph of Fig. 5.1 we see the individual execution times of the MPI and data center tasks, while also the total execution time for each scenario. Genisys’s Least Loaded Policy is represented by the blue label, while the Unmanaged scenario by the red label. The total execution time needed by Genisys’s Least Loaded Policy to complete the combined workload

is 11200 seconds, which is (25%) faster when compared to Unmanaged (14633 seconds).

Because of the resource overlapping between the data center and MPI tasks, the individual task performance suffers under the Unmanaged scenario. On average the individual execution times are (59%) faster when using Genisys compared to Unmanaged.

In the second graph of Fig. 5.1 we see the CPU utilization for the duration of each scenario. Genisys achieves higher average CPU utilization (90%) compared to (71%) of the Unmanaged. In the case of the Unmanaged scenario, because of the lack of coordination on task placement, both Kubernetes and Slurm place tasks on oversubscribed nodes, while leaving the some of the cluster's nodes underutilized.

In the case of Slurm, the node selection is done serially by default, so when a job comes for placement it is placed in the first N cluster's nodes that it fits into. This kind of selection can leave underutilized nodes. For example if the users specify the node count that their jobs need to run, at first the jobs will fill serially the cluster nodes and the last jobs submitted may wait in the queue even if there are enough resources for them to run. In this case there may be underutilized nodes at the end of the partition.

In the case of Kubernetes, the default scheduler was used, which places tasks in a round-robin fashion.

Due to the lack of coordination, the default scheduler spreads the tasks across all nodes while Slurm places the tasks in the first four nodes. This leads to low resource usage of the fifth node, as only data center tasks are placed there.

5.2.2 Small Task Size Workload Evaluation

Next we compare the performance between the Genisys and Unmanaged scenarios when using the Small Task Size Workload. Earlier we assumed that choosing smaller-sized tasks for the workload should lead to higher cluster utilization. In this case though we observe lower CPU utilization when compared to the Pilot workload. This behaviour is justified as the smaller size for the data center tasks leaves resource gaps, as do not run as efficiently in the cluster's nodes as in the Pilot case.

In total, the time needed for the Genisys-managed scenario to complete the small workload is 6100 seconds, (63%) faster when compared to Unmanaged (11700 seconds). In the first graph of Fig. 5.2 we see the individual execution times of the MPI and data center tasks while also the total execution time for each scenario.

Genisys's Least Loaded Policy is represented by the blue label while the Unmanaged scenario by the red label. Due to interference between the MPI

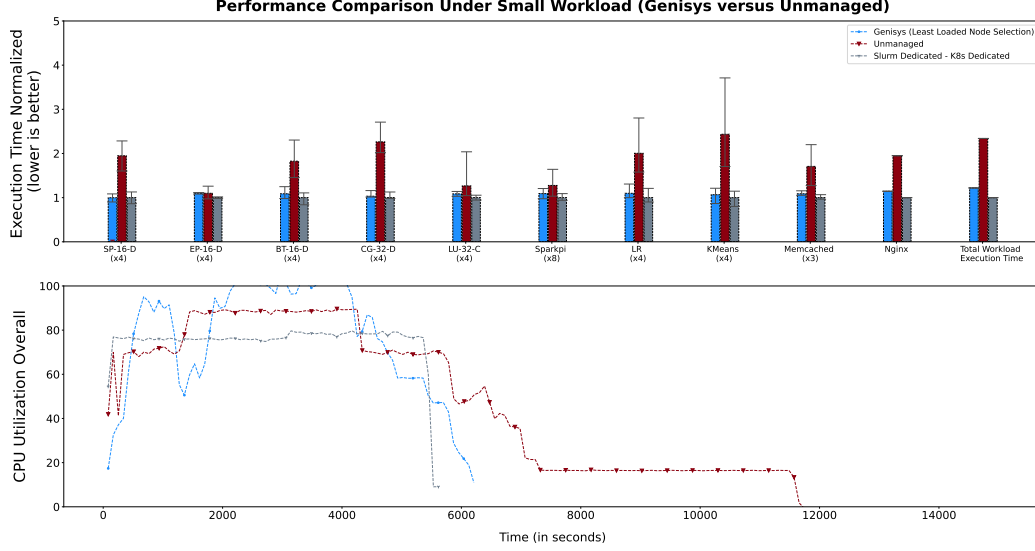


Figure 5.2: The execution time of each workload between Genisys and Unmanaged when using the small workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.

and data center tasks in the Unmanaged scenario, the average individual task completion time is (64%) faster when using Genisys.

The average CPU utilization is (80%) when using Genisys, (22%) higher compared to Unmanaged (58%). Similarly to the Pilot workload, Unmanaged Slurm places the tasks on the first 4 nodes, while Kubernetes spreads tasks using its default scheduler, leading to the underutilization of the fifth node.

5.2.3 Large Task Size Workload Evaluation

We now compare the performance between the Genisys and Unmanaged scenarios when using the Large Task Size Workload. We have assumed that by choosing large-sized tasks for the workload the total cluster utilization will be lower compared to the other two workloads, as the fitting of the large jobs will not be as efficient compared to the smaller ones.

In total, the time needed for the Genisys-managed scenario to complete the large workload is 9700 seconds, (18%) faster when compared to Unmanaged (11700 seconds). In the first graph of Fig. 5.3 we see the individual



Figure 5.3: The execution time of each workload between Genisys and Unmanaged when using the large workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.

execution times of the MPI and data center tasks while also the total execution time for each scenario.

Genisys’s Least Loaded Policy is represented by the blue label while the Unmanaged scenario by the red label. Due to interference between the MPI and data center tasks in the Unmanaged scenario, the average individual task completion time is (55%) faster when using Genisys.

The average CPU utilization is (76%) when using Genisys, (4%) lower compared to Unmanaged (82%). In contrast to the previous cases the Unmanaged scenario achieves higher CPU utilization, due to lower task-fitting efficiency under Genisys as the task sizes are larger. In the Unmanaged scenario both Kubernetes and Slurm clusters have access to 100% of each node’s resources leading to over-subscription and higher total utilization. In spite of this phenomenon, Genisys achieves lower total and individual execution times due to reduced interference and less MPI spinning.

5.3 Genisys vs Partitioned Cluster Performance Comparison

As shown from the Unmanaged approach, if performance is a concern then it is not viable to run both Kubernetes and Slurm on top of the same physical nodes without a synchronization mechanism.

One other approach would be static partitioning of cluster nodes into a Kubernetes and a Slurm cluster. As in the Unmanaged case, the Partitioned approach allows users to run both data center and HPC tasks on top of the Cluster, while ensuring optimal performance for both kinds of tasks as there is no resource overlapping. While this approach eliminates the interference introduced by resource overlapping, it may lead to low cluster utilization, in cases of workload imbalances between the two partitions. For example, if the Slurm partition experiences high load, while the Kubernetes partition is underutilized, Slurm will never be able to utilize resources from the Kubernetes partition.

In the Genisys case we would not observe the above problem as we avoid cluster partitioning and Genisys keeps the same resource accounting for both HPC and data center tasks.

In order to evaluate the above assumption, we run the three workloads using Genisys's Least Loaded Policy, Genisys's Max Loaded Policy, and Partitioned scenarios and show the performance of each individual task while also the total execution time of each workload.

In Figures Fig. 5.7, Fig. 5.8 and Fig. 5.9 we can see the execution performance and CPU utilization for each workload across all scenarios. These figures although not used in this section are probably useful as they represent the total system performance across all scenarios.

5.3.1 Pilot Workload Evaluation

We first compare the performance between the Genisys and Partitioned scenarios and investigate if there is a speedup for the total and individual execution times for the Pilot workload when Genisys is used compared to Partitioned.

The main goal of this experiment is to show that when using Genisys we have better resource utilization as both HPC and data center workloads are not restricted to their respective partitions and when one partition is underutilized the other one will be able to leverage the free resources.

In the first graph of Fig. 5.4 we see the individual execution times of the MPI and data center tasks while also the total execution time for each scenario.

Genisys's Least Loaded Policy is represented by the blue label while the

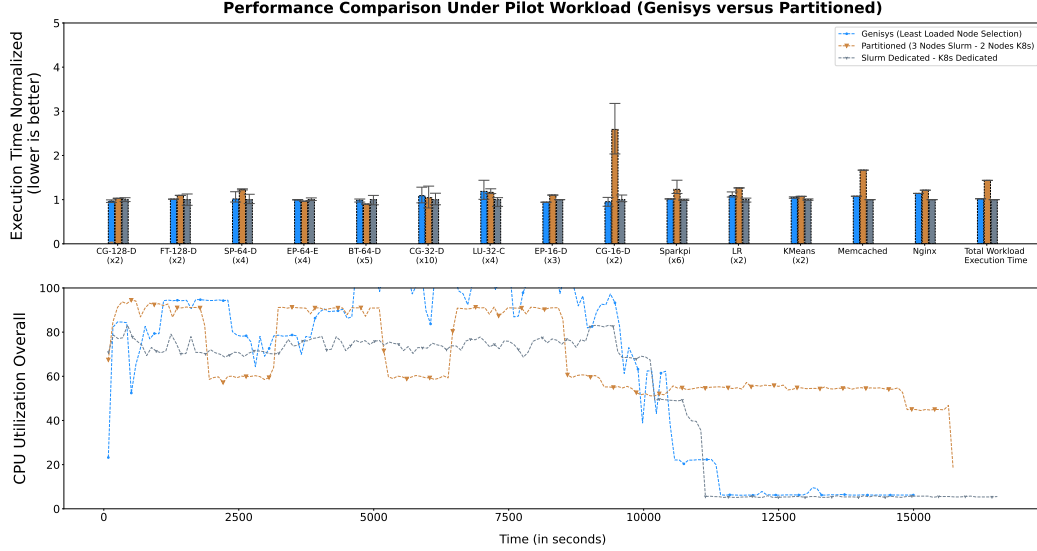


Figure 5.4: The execution time of each workload between Genisys and Partitioned when using the Pilot workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.

Partitioned scenario by the orange label. In the case of the Partitioned scenario, due to restriction of the workloads to their corresponding partition, Slurm can not leverage resources from the Kubernetes cluster even when the execution of the data center tasks finishes. This results in a higher total workload execution time of 15800 seconds, (34%) slower when compared to Genisys (11200 seconds).

Between the Genisys and Partitioned, the average individual task completion time is (4%) lower in the Genisys case, we assume that this is because Genisys spreads the tasks across all the cluster nodes and is able to better utilize the RDMA network of the cluster.

In the second graph of Fig. 5.4 we see the CPU utilization for the duration of each scenario. Genisys achieves higher average CPU utilization (90%) compared to (75%) of Partitioned, as in the Partitioned case when the data center workload finishes Slurm is not able to utilize the Kubernetes nodes.

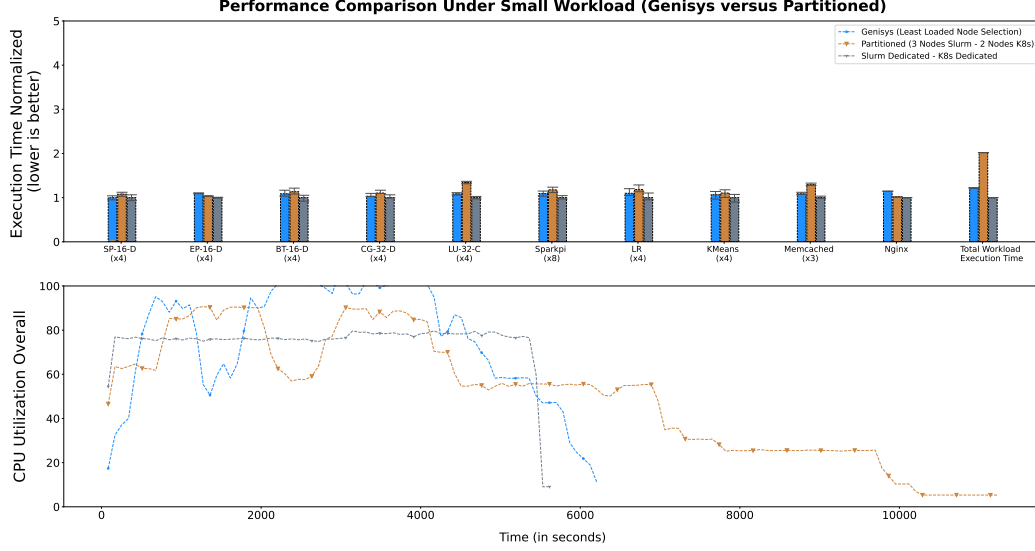


Figure 5.5: The execution time of each workload between Genisys and Partitioned when using the small workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.

5.3.2 Small Task Size Workload Evaluation

As a next step we compare the performance between Genisys and Partitioned under the Small Task Size Workload. The total time needed for the completion of the Genisys scenario is 6100 seconds, (52%) faster when compared to Partitioned (10400 seconds).

Similarly to the Pilot workload, because workloads are restricted to their allocated partitions in the Partitioned scenario, the HPC workload is not able to leverage resources from the Kubernetes partition even when there are idle resources under Kubernetes. This leads to higher completion times of the Partitioned workload when compared to Genisys.

In the first graph of Fig. 5.5 we see the individual execution times of the MPI and data center tasks while also the total execution time for each scenario. Genisys’s Least Loaded Policy is represented by the blue label while the Partitioned scenario by the orange label. In the second graph we see the CPU utilization of the cluster under each scenario.

The average CPU utilization of the cluster when using Genisys is (92%), compared to (62%) when Partitioned, as in the latter case the Slurm workload

runs for a prolonged period of time due to partition related restrictions.

5.3.3 Large Task Size Workload Evaluation

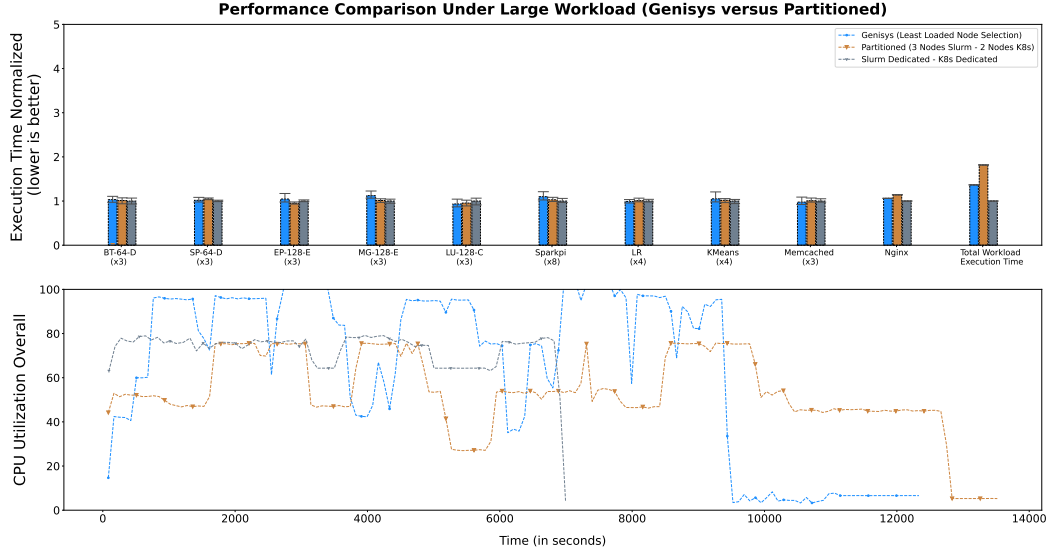


Figure 5.6: The execution time of each workload between Genisys and Partitioned when using the large workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.

As a next step we compare the performance between Genisys and Partitioned under the Large Task Size Workload. The total time needed for the completion of the Genisys scenario is 9700 seconds, (28%) faster when compared to Partitioned (12900 seconds).

Similarly to the Pilot workload, because workloads are restricted to their allocated partitions in the Partitioned scenario, the HPC workload is not able to leverage resources from the Kubernetes partition even when there are idle resources under Kubernetes. This leads to higher completion times of the Partitioned workload when compared to Genisys.

In the first graph of Fig. 5.6 we see the individual execution times of the MPI and data center tasks while also the total execution time for each scenario. Genisys’s Least Loaded Policy is represented by the blue label while the Partitioned scenario by the orange label. In the second graph we see the CPU utilization of the cluster under each scenario.

The average CPU utilization of the cluster when using Genisys is (78%), compared to (64%) when Partitioned, as in the latter case the Slurm workload runs for a prolonged period of time due to partition related restrictions.

5.4 Genisys vs Thread Partitioned Cluster Performance Comparison

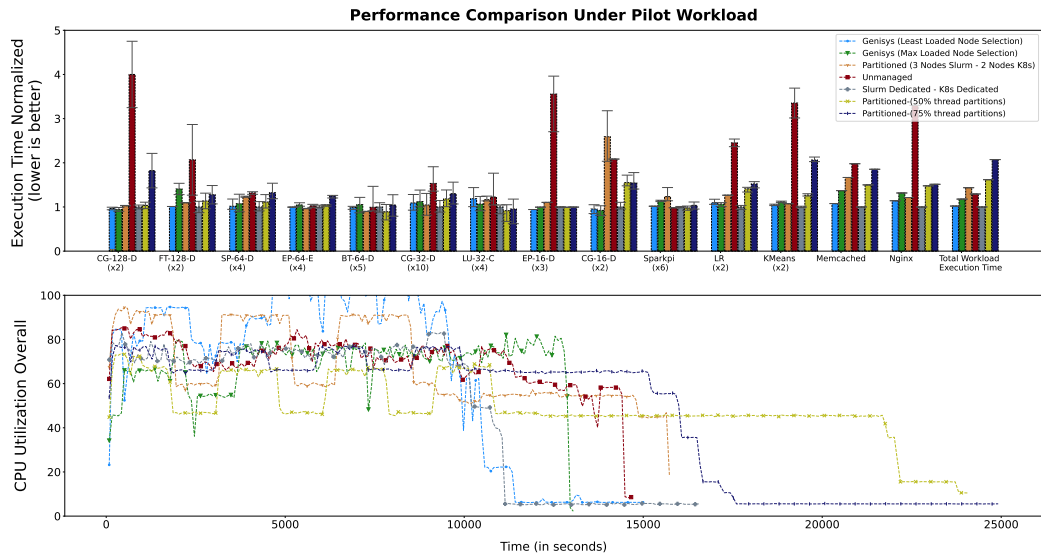


Figure 5.7: The execution time of each workload step in six different scenarios when using the Pilot workload. In the first graph we show the individual execution times of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.

Another approach to run both HPC and data center tasks under the same infrastructure would be to statically partition the cluster’s nodes by giving (50%) and (75%) of each node’s CPU capacity to the Kubernetes cluster while the rest (50%) and (75%) respectively to the Slurm controller. While this is an uncommon approach it would be interesting to compare it to Genisys, as Genisys allows resource sharing between data center and HPC tasks over the same physical nodes in a similar manner.

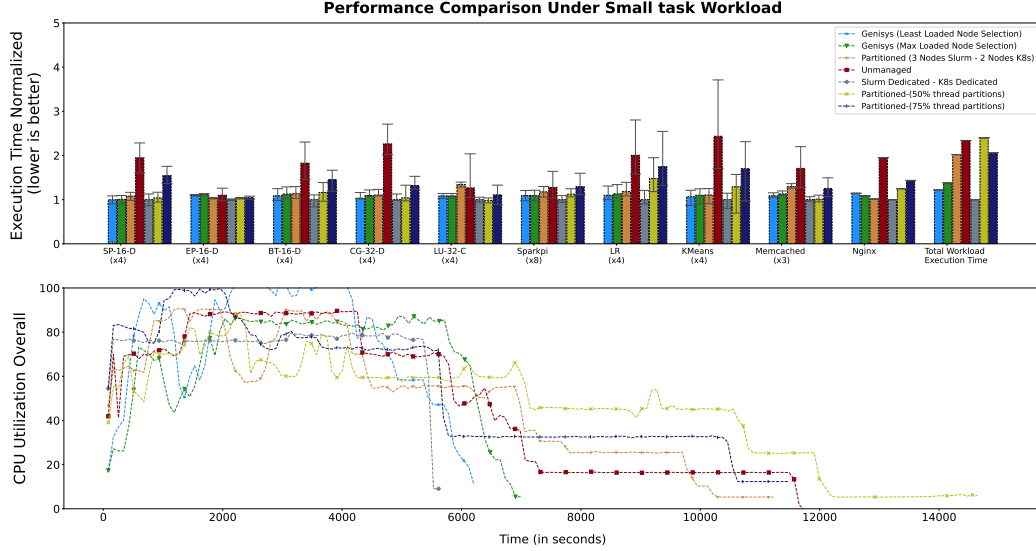


Figure 5.8: The execution time of each workload step in six different scenarios when using the small workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.

5.4.1 Genisys vs Thread Partitioned 50% Comparison

We first compare the performance between the Genisys and Thread Partitioned 50% scenarios and investigate if there is a speedup for the total and individual execution times across all three workloads when Genisys is used compared to Thread Partitioned 50%.

The main goal of this experiment is to show that under Genisys we are going to have better resource utilization as both HPC and data center workloads are not restricted to their respective partitions and when one partition is underutilized the other one will be able to leverage the free resources.

In the first graph of Fig. 5.7, Fig. 5.8 and Fig. 5.9 figures we see the individual execution times of the MPI and data center tasks while also the total execution time for each scenario.

Genisys’s Least Loaded Policy is represented by the blue label while the Thread Partitioned 50% scenario by the yellow label. In the case of the Thread Partitioned 50% scenario, due to restriction of the workloads to their corresponding partition, Slurm can not leverage resources from the Kubernetes side even when the execution of the data center tasks finishes. This

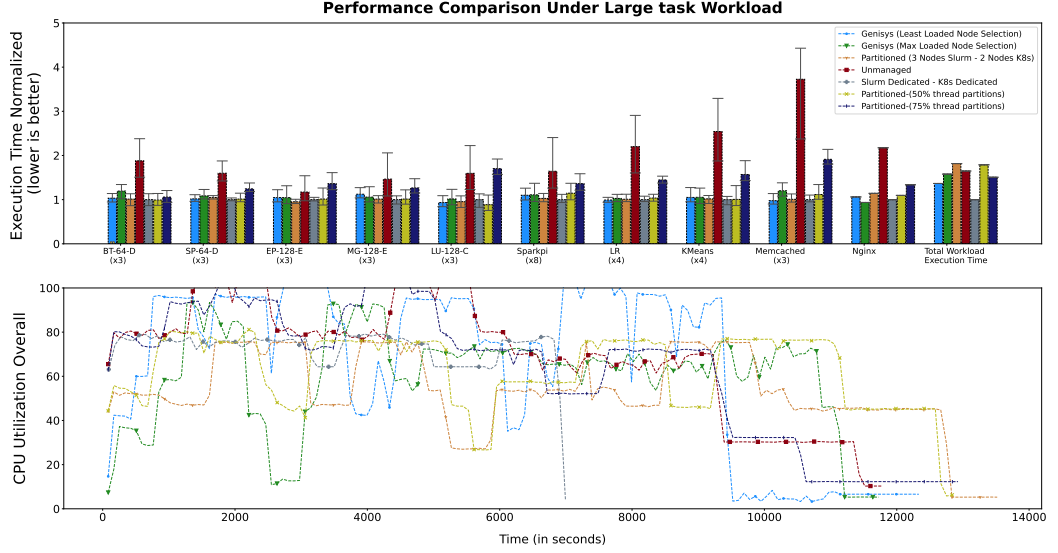


Figure 5.9: The execution time of each workload step in six different scenarios when using the large workload. In the first graph we show the individual execution time of each workload step. The execution time is normalized to the execution of the workloads when running in dedicated Slurm and Kubernetes installations. The last bar group shows the execution time of the whole workload. In the second graph we show the cluster’s CPU utilization for each different scenario.

results in higher total workload execution time, (44%) slower when compared to Genisys.

Between Genisys and Thread Partitioned 50%, the average individual task completion time is (4.5%) lower in the Genisys case. We assume that this happens because in the Thread Partitioned 50% case a higher number of data center and HPC tasks are colocated over the same time periods and the interference is higher than with Genisys.

On average, across all three workloads, Genisys achieves higher average CPU utilization (82%), compared to (56%) of the Thread Partitioned 50%. In the latter case when the data center workload finishes, Slurm is not able to utilize the Kubernetes nodes. Also due to the smaller number of threads available to both Slurm and Kubernetes, the tasks can not fit as efficiently as when running with Genisys.

5.4.2 Genisys vs Thread Partitioned 75% Comparison

In the case Thread Partitioned 75% scenario, both Kubernetes and Slurm cluster have access to 75% of each node’s resources, in total this approach

allows resource overlapping as combined the two platforms can load each node to up 150%.

Genisys's Least Loaded Policy is represented by the blue label while the Thread Partitioned 75% scenario by the dark blue label. In the case of the Thread Partitioned 75% scenario, resource overlapping of the workloads running on the same nodes results in higher total workload execution time, (47%) slower when compared to Genisys. Another important factor that contributes to the slower execution is the partitioning of the platform as when one of the data center or HPC workloads finishes, the workload that is still running is restricted to its respective partition.

Between Genisys and Thread Partitioned 75%, the average individual task completion time is (28%) lower in the Genisys case. In the Thread Partitioned 75% case, the resource overlapping on the congested nodes leads to interference and higher execution times.

On average, across all the three workloads, Genisys achieves higher average CPU utilization (82%), compared to (68%) of the Thread Partitioned 75%. In the latter case when the data center workload finishes, Slurm is not able to utilize the Kubernetes nodes. Also due to the smaller number of threads available to both Slurm and Kubernetes, the tasks can not fit as efficiently as when running with Genisys.

Chapter 6

Related Work

6.1 HPC-Cloud Convergence

The integration of HPC job management in the context of Kubernetes has been addressed in the past. In [21], the authors use a utility called *hpc-connector* that acts as an HPC job proxy: Users submit respective Kubernetes jobs with the appropriate settings, and *hpc-connector* forwards them to the HPC cluster, monitors their execution, and collects their results. This solution can probably be used with containers to address portability issues. On the other hand, the main focus is on HPC job management with a Kubernetes-compatible interface; the HPC and cloud clusters are treated as two separate environments making it impossible to monitor and place cloud and HPC workloads over the same physical cluster.

A similar approach is presented in [30], where a Kubernetes installation is interfaced to a Torque-based HPC cluster, using a custom tool called Torque-Operator. Although this study offers the flexibility of running containerized cloud and HPC jobs over the same front end interface through the WLM operator, it again distinguishes the cloud and HPC runtime environments by using two different clusters. On the contrary, our approach aims to completely integrate the runtime and management environment of an HPC system in the Kubernetes framework, using a single hardware setup.

6.2 Performance of Containers in HPC

One critical aspect of the containerization of HPC workloads is runtime performance when compared to an actual physical cluster. Related work has measured the network performance of containerized HPC codes, and findings suggest that there is little to no performance overhead when an InfiniBand network is used [12]. I/O performance of the HPC nodes is also very important as the process of checkpointing when running MPI applications at large scales is an I/O intensive

task that has the potential to dominate the execution time [22]. In [10], it is shown that Docker containers do not suffer from any performance overhead under synthetic I/O workloads. In general, Docker containers do not introduce significant performance overheads, while in some cases they can provide better QoS due to the usage of cgroups resource limiting mechanisms when compared with a bare metal runtime environment [17, 18, 15].

6.3 Workload Scheduling

There is a plethora of papers that handle the scheduling of workloads with the main goal of increasing the utilization in the infrastructure. Sparrow [25] and Eagle [14], handle the scheduling of tasks of applications in clusters. Sparrow is a fully distributed scheduler that randomly selects a set of candidate servers to execute each incoming task; it assigns tasks to the fastest server. With its simplistic scheduling policy, Sparrow can schedule hundred thousands tasks in a few milliseconds. However, it is not appropriate for scheduling workloads with conflicting goals as in our case, i.e., resource sharing for some applications and resource dedication for others. Eagle implements a hybrid scheduler with a centralized component that handles the long-running applications and a distributed component for the short-running applications. The centralized component, performs careful placement, whereas the distributed component emphasizes on quick placement of tasks. Our approach is a hybrid scheduler like Eagle, however, with different goals.

Ursa [19] is a task scheduler for spark-monotasks [24] and Rhythm [29] is a data center scheduler that ensures the latency of latency-critical applications. Both works colocate “compatible” tasks to increase the utilization in the underlying infrastructure. The main limitation of these systems is that they do not guard against interference, in case of a scheduling error. This results in a significant and sudden drop in performance if interference occurs at a selected node. In contrast, our approach constantly monitors application performance and adjusts container placement and resource allocations at runtime to achieve a user-defined performance target.

Control loops for dynamically adjusting resources based on runtime performance have been used in systems such as SLAOrchestrator [23] and Sky-net [27]. The former tries to optimize cost of services when running in the Cloud, while the latter optimizes hardware efficiency by colocating more applications on the same nodes, as long as they achieve their user-defined performance targets. Genisys’s handling of data center tasks is based on Skynet, but it has also been extended to allocate HPC tasks with different, placement-based constraints.

Chapter 7

Conclusion and Future Work

This work presents a method to run HPC workloads in Kubernetes using portable and extensible containerized environments called *virtual clusters*. Virtual clusters include Slurm, so users can run existing scripts unmodified, and are deployed alongside other Kubernetes services on the same physical cluster. Without any additional changes, the resulting hybrid resource allocation environment would have individual Slurm instances operating within their virtual cluster constraints, unaware of what is happening at the overall cluster-level, where decisions are made by Kubernetes. To avoid resource allocation conflicts, we integrate Slurm with Kubernetes, by extending the Slurm controller to delegate placement decisions to Genisys, our custom Kubernetes scheduler.

Our evaluation results indicate that it is not only possible to colocate data center tasks with HPC jobs when remaining CPU cycles are available, but with appropriate scheduling it can be beneficial to overall performance. Overall Genisys + *virtual clusters* are able to integrate Slurm into the Kubernetes ecosystem with minimal performance overhead across all the task categories. The evaluation shows that with the use of Genisys it is possible to reduce the execution time of combined workloads compared to unmanaged and partitioned approaches.

At this point the main resource limiting mechanism that Genisys uses is Linux’s completely fair scheduler (CFS) in order to allocate container resources for both HPC and data center jobs. While this approach is convenient as CFS is the main resource limiting method for containers in Kubernetes, it can lead to lower performance when multiple jobs are colocated over the same physical nodes. We believe there are benefits in using affinity based placement for HPC jobs using the cpuset Linux’s cgroup mechanism and place them on dedicated CPU cores in order to reduce interference even further. Through previous experimentation with both CPU limiting mechanisms we found out that it is possible to use both of them on parallel each one for different kind of jobs.

Chapter 8

Appendix

In this section we are going to describe in detail each Genisys Kubernetes component and its main methods.

8.1 Pod Placement Controller

In order for Genisys to schedule Kubernetes Pods, we implemented a custom Placement Controller in Golang.

The main function of this component is the **Pod_Scheduler()** method. The “Pod_Scheduler()” method iterates over the Kubernetes Pods that are in “Pending” state. If a Pod is in “Pending” state it means that it has not yet been scheduled to an appropriate node. When “Pod_Scheduler()” chooses a Pod for scheduling, it checks first if the Pod’s Deployment object has multiple replicas of this Pod to be scheduled. In the case that the Pod’s deployment has multiple replicas then the **Pod_To_Node_Finder(Pod *v1.Pod)** method is called. This method checks if a set of Nodes is available for the deployment’s Pods to fit. At this stage the node selection is done according to the selected placement policy of the controller. If the Deployment fits in the cluster’s resources then we mark the Deployment to be scheduled by increasing its priority, which guarantees that the deployment is going to be scheduled and that there is going to be set of nodes with enough available resources for the deployment to fit. As a next step, the “Schedule_Pod(Pod *v1.Pod)” function is called for each Pod of the deployment, which binds the Pod to the appropriate node.

8.2 Resource Updater Controller

The Resource Updater Controller Genisys component is responsible for updating the cluster's deployment and node resources. For example, if a deployment gets deleted or finishes its tasks then Genisys should be able to update the resource's state and free the unused resources.

The **Resource_Bandwidth_Updater_Nodes()** function iterates over all the running Pods of the cluster and for each Pod it finds the Node that it is running in. After this step, it updates the Node by adding the Pod's allocated resources to the Node allocation. When the execution of this method finishes, each Node allocation is updated. In order to update the cluster's resource state this method is called every 10 seconds. During this method's execution we pause the scheduling of Deployments.

The **Resources_Bandwidth_Updater_Deployments()** function iterates over all the running "Data Center" type Pods of the cluster and for each Pod it enforces the CPU, Network, I/O, Memory allocations by using the cgroups resource limiting kernel mechanism. This function ensures that each running Pod is not going to exceed its resource allocation.

For each Kubernetes Virtual Cluster user, the **Resources_Limiter_Virtual_Clusters()** function iterates over the generated dummy resource reservation Pods, and calculates the resource allocation for each user's Virtual Cluster Nodes. As a next step for each Virtual Cluster Node, the function uses the cgroups resource limiting kernel mechanism in order to enforce resource limits.

8.3 Slurm Connector

In order for Genisys to free resources allocated by finished, deleted or failed Slurm jobs, we need a mechanism to communicate with each Virtual Cluster. The Slurm Connector component communicates with each Virtual Cluster and gets information about the status of the Slurm jobs running on it.

The **Get_Pods_Slurm_Job_Id_Per_Namespace(Slurm_Job_ID string, User string)** function gets as input the Slurm job ID and the User that the Virtual Slurm cluster belongs to and returns if the specified Slurm Job still exists on the cluster. If the Job is still running then the function returns “Valid”, in any other case it returns “Invalid”.

The **Slurm_Job_Deallocator()** function iterates over the generated dummy resource reservation Pods for each Slurm user and checks if the Slurm job for the specified dummy pods still exists. If the Slurm Job is no longer “Valid” then the “Slurm_Job_Deallocator()” deletes all the dummy pods mapped to the job thus freeing the allocated resources. This function is called periodically every 100 milliseconds.

8.4 Kubernetes API Communicator

Kubernetes API Communicator component contains the methods needed for Genisys to communicate and fetch objects from the Kubernetes API server using the Golang client.

The **Get_Dummy_Slurm_Pods(User string)** function fetches and returns the Pod objects referring to the Slurm dummy allocations for the specified user ID.

The **Get_Dummy_Slurm_Pods_All_Users()** function fetches and returns the Pod objects referring to the Slurm dummy allocations for the whole cluster.

The **Get_Virtual_Cluster_Pods(User string)** function fetches and returns the Pod objects referring to the Virtual Cluster Nodes for the specified user ID.

The **Get_Pod_By_Name(Pod_Name string)** function gets as input the name of a Pod, fetches and returns the Pod object from the API server.

The **Get_Deployment_Of_Pod(Pod *v1.Pod)** function gets as input a Pod object and returns the Deployment object of the specified Pod.

The **Get_Pod_User(Pod *v1.Pod)** gets as input a Pod object and returns the User's ID for the specified Pod.

The **Get_Pods.all_Users** fetches and returns the whole Pod object list of the cluster's Pods.

The **Get_Pods_Of_User(User string)** function fetches and returns all the Pod objects referring to the specified user ID.

The **Get_Nodes()** function fetches and returns all the Node objects of the cluster.

Bibliography

- [1] An open-source monitoring solution. <https://prometheus.io/>.
- [2] Apache kafka.
- [3] The apache software foundation. apache http server benchmarking tool. <http://httpd.apache.org/docs/2.2/programs/ab.html>.
- [4] Apache spark.
- [5] Dask: Scalable analytics in python.
- [6] Keras: The python deep learning api.
- [7] Mpi forum.
- [8] Tensorflow.
- [9] VMware: The State of Kubernetes 2020. <https://k8s.vmware.com/state-of-kubernetes-2020/>.
- [10] Manish Abhishek. High performance computing using containers in cloud. *International Journal of Advanced Trends in Computer Science and Engineering*, 9:5686, 09 2020.
- [11] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The nas parallel benchmarks. *Int. J. High Perform. Comput. Appl.*, 5(3):63b–73, September 1991.
- [12] Angel M. Beltré, Pankaj Saha, Madhusudhan Govindaraju, Andrew Younge, and Ryan E. Grant. Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 11–20, 2019.
- [13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*, page 143b–154, New York, NY, USA, 2010. Association for Computing Machinery.

- [14] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16*, pages 497–509, New York, NY, USA, 2016. ACM.
- [15] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, 2015.
- [16] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, August 2004.
- [17] Stephen Herbein, Ayush Dusia, Aaron Landwehr, Sean Mcdaniel, Jose Manuel Monsalve Diaz, Yang Yang, Seetharami Seelam, and Michela Taufer. Resource management for running hpc applications in container clouds. pages 261–278, 06 2016.
- [18] Joshua Higgins, Violeta Holmes, and Colin Venters. Orchestrating docker containers in the hpc environment. pages 506–513, 07 2015.
- [19] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [20] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Spark-bench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF '15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [21] Sergio López-Huguet, J. Damià Segrelles, Marek Kasztelnik, Marian Bubak, and Ignacio Blanquer. Seamlessly managing hpc workloads through kubernetes. In Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 310–320, Cham, 2020. Springer International Publishing.
- [22] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.
- [23] Jennifer Ortiz, Brendan Lee, Magdalena Balazinska, Johannes Gehrke, and Joseph L. Hellerstein. Slaorchestrator: Reducing the cost of performance

slas for cloud data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 547–560, Boston, MA, July 2018. USENIX Association.

- [24] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 184–200, 2017.
- [25] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.
- [26] Will Reese. Nginx: The high-performance web server and reverse proxy. *Linux J.*, 2008(173), sep 2008.
- [27] Yannis Sfakianakis, Manolis Marazakis, and Angelos Bilas. Skynet: Performance-driven resource management for dynamic workloads. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021.
- [28] Pavel Shvets, Vadim Voevodin, and Dmitry Nikitenko. *Approach to Workload Analysis of Large HPC Centers*, pages 16–30. 07 2020.
- [29] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.
- [30] Naweiluo Zhou, Yiannis Georgiou, Li Zhong, Huan Zhou, and Marcin Pospieszny. Container orchestration on hpc systems. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 34–36, 2020.