National Technical University of Athens
School of Electrical and Computer Engineering
Computer Science Division
Computing Systems Laboratory

Flurm - Dynamic Flux Deployment Through Slurm Job Submission with Co-Scheduling Capabilities

Diploma Thesis

of

Konstantinos Katsikopoulos

Supervisor: George Goumas, Professor NTUA

Athens, October 2025

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Flurm - Δυναμική Εγκατάσταση του Flux μέσω Υποβολής Εργασιών Slurm με Δυνατότητες Συνεκτέλεσης Εφαρμογών

Διπλωματική εργασία

του

Κωνσταντίνου Κατσικόπουλου

Επιβλέπων: Γεώργιος Γκούμας, Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 30 Οκτωβρίου 2025.

.....	.....	.....
Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Οκτώβριος 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Computer Science Division
Computing Systems Laboratory

Υπεύθυνη Δήλωση Για Λογοκλοπή Και Για Κλοπή Πνευματικής Ιδιοκτησίας

Έχω διαβάσει και κατανοήσει τους κανόνες για τη λογοκλοπή και τον τρόπο σωστής αναφοράς των πηγών που περιέχονται στον οδηγό συγγραφής Διπλωματικών Εργασιών. Δηλώνω ότι, από όσα γνωρίζω, το περιεχόμενο της παρούσας Διπλωματικής Εργασίας είναι προϊόν δικής μου εργασίας και υπάρχουν αναφορές σε όλες τις πηγές που χρησιμοποίησα.
Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτή τη Διπλωματική εργασία είναι του συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών ή του Εθνικού Μετσόβιου Πολυτεχνείου.

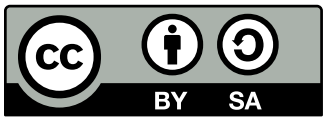
Κωνσταντίνος Κατσικόπουλος, Αθήνα 2025

Statement About Plagiarism And Intellectual Property Theft

I have read and understood the rules about plagiarism and the proper way of refrencing constained in the Diploma Thesis writing guide. I declare that, to the best of my knowldege, the content of this Diploma Thesis is the product of my own work and there are refrences to all sources that I have used.
The opinions and conclusions contained in this Diploma Thesis are of the author and should not be interpeted as representing the official views of the School of Electrical and Computer Engineering or the National Technical University of Athens.

Konstantinos Katsikopoulos, Athens 2025

.....
Κωνσταντίνος Κατσικόπουλος,
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ



Περίληψη

Τα συστήματα High Performance Computing (HPC) είναι απαραίτητα για την επίλυση σύνθετων υπολογιστικών προβλημάτων σε διάφορους επιστημονικούς και μηχανικούς τομείς. Η αποδοτική διαχείριση πόρων και η διαχείριση εργασιών είναι κρίσιμα στοιχεία που επηρεάζουν άμεσα την απόδοση και την αξιοποίηση αυτών των συστημάτων. Ενώ το Slurm είναι ένας ευρέως υιοθετημένος open-source διαχειριστής φόρτου εργασίας/πόρων που παρέχει βασικές δυνατότητες δρομολόγησης (scheduling), σε αρκετές περιπτώσεις του λείπει η ευελιξία που απαιτείται για την λεπτομερή κατανομή πόρων και την δυναμική δρομολόγηση εργασιών. Το Flux είναι ένα framework διαχείρισης πόρων επόμενης γενιάς που έχει σχεδιαστεί για να αντιμετωπίσει αυτούς τους περιορισμούς, επιτρέποντας πιο ευέλικτες και αποδοτικές στρατηγικές κατανομής πόρων και δρομολόγησης.

Αυτή η διπλωματική εργασία παρουσιάζει την ενσωμάτωση του Flux Framework σε ένα υπάρχον Slurm Cluster σε επίπεδο χρήστη, στο οποίο αναφερόμαστε ως Flurm, επιτρέποντας στο Flux να εκκινείται και να διαχειρίζεται μέσα σε ένα Slurm allocation. Επιπλέον, εξετάζει πώς το Flux μπορεί να προσαρμοστεί για να παρέχει λεπτομερή έλεγχο σε μια ποικιλία υλικών πόρων (π.χ. sockets, cores) σε συνδυασμό με τον graph-based δρομολογητή του Flux - Fluxion. Έτσι, το Flux τροποποιήθηκε για να υποστηρίζει προηγμένες λειτουργίες κατανομής πόρων όπως η συνεκτέλεση, η συντοποθέτηση και η ευέλικτη κοινή χρήση πόρων εντός του περιβάλλοντος Slurm. Τέλος, διεξάγεται μια σειρά πειραμάτων για την αξιολόγηση της τυχόν επιβάρυνσης (overhead) που εισάγεται από την ενσωμάτωση, καθώς και της κλιμακωσιμότητας της προτεινόμενης λύσης.

Λέξεις Κλειδιά

High Performance Computing (HPC), Slurm, Flux framework, Διαχείριση Πόρων, Δρομολόγηση Εργασιών, Συνεκτέλεση, Συντοποθέτηση, Συνδρομολόγηση

Abstract

High Performance Computing (HPC) systems are essential for solving complex computational problems across various scientific and engineering domains. Efficient resource management and job scheduling are critical components that directly impact the performance and utilization of these systems. While Slurm is a widely adopted open-source workload/resource manager that provides basic scheduling capabilities, it often lacks the flexibility required for fine-grained resource allocation and dynamic job scheduling. Flux is a next-generation resource management framework designed to address these limitations by enabling more flexible and efficient resource allocation strategies.

This thesis presents the integration of the Flux Framework into an existing Slurm cluster at the user level, which we refer to as Flurm, enabling Flux to be launched and managed from within a Slurm allocation. It further demonstrates how Flux can be customized to provide fine-grained control over a variety of hardware resources (e.g. sockets, cores) in accommodation with Flux's graph-based scheduler - Fluxion. Thereby, Flux was modified to support advanced resource allocation features such as co-execution, colocation, and flexible resource sharing within the Slurm environment. Finally, a series of experiments are conducted to assess any overhead introduced by the integration and the scalability of the proposed solution.

Keywords

High Performance Computing (HPC), Slurm, Flux framework, Resource Management, Job Scheduling, Co-execution, Colocation, Co-scheduling

Acknowledgements

First of all, as my journey as a student at NTUA comes to an end, I would like to thank my family and friends for their continuous support and encouragement throughout all the hard and also good times in my studies.

I would also like to thank my supervisor, Professor George Goumas, for giving me the opportunity to explore my interests and for his guidance and support during this work.

Last, but certainly not least, I would like to thank PhD candidate Nikolaos Triantafyllis for his invaluable help, support and insights throughout the development of this diploma thesis.

Konstantinos Katsikopoulos
October 2025

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation - Problem Statement

High Performance Computing (HPC) clusters are complex shared systems where efficient resource allocation and job scheduling are critical to maximizing throughput, utilization, and fairness. As HPC workloads grow in heterogeneity—mixing simulations, data analysis, and AI/ML components—traditional scheduling policies are sometimes insufficient to extract full performance from modern hardware. These concerns motivate the exploration of advanced scheduling techniques such as co-location and co-scheduling and the need for experimentation with them in real HPC environments.

However, real HPC clusters are often managed by established resource managers like Slurm, which may not natively support such advanced scheduling features or require configuration changes by system administrators. This creates a barrier for researchers and users who wish to experiment with and evaluate new scheduling strategies without disrupting existing workflows. To address this, one promising approach is to integrate another resource management framework within the context of an existing Slurm-managed cluster.

This thesis explores this approach by integrating the Flux framework into the real Slurm cluster ARIS at GRNET. This integration, referred to as Flurm, empowers end users of the cluster to experiment with advanced resource scheduling (co-scheduling, colocation, dynamic overlay) within their own allocations, without requiring system administrator changes, the foundation of which is presented in this work. Finally, a series of experiments are conducted to evaluate the overhead introduced of the proposed solution, to prove its correctness and demonstrate its scalability.

1.2 Goal

The goal of this thesis is to enable a workflow where users can:

- Develop scheduling algorithms locally using some simulation tool like ELiSE [1].
- Write these algorithms in Flux and test them in a controlled - dockerized environment.
- Utilize Flurm to seamlessly transition from local testing to real cluster execution for evaluation.

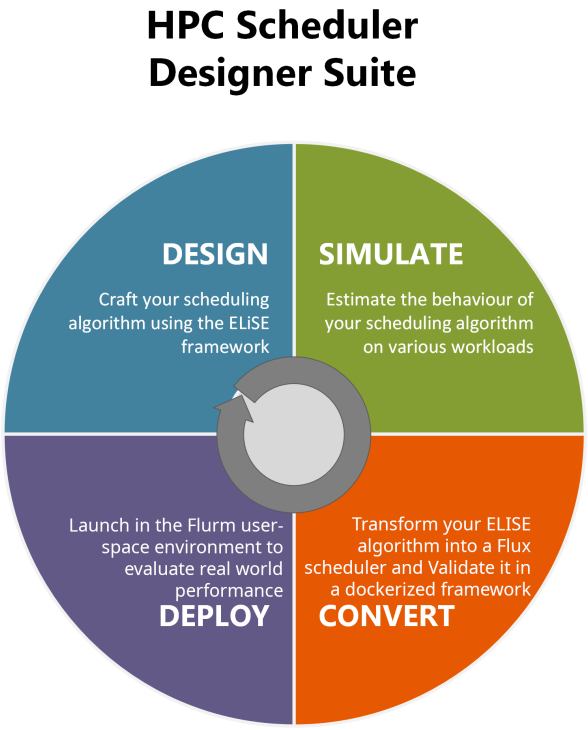


Figure 1.1: Flurm Workflow Overview

Chapter 2

High Performance Computing (HPC)

In this chapter we provide the background on HPC Systems.

2.1 Introduction

In just a few decades, supercomputing has evolved from primitive machines capable of performing thousands of operations per second into large-scale systems executing exaFLOP workloads (10^{18} floating-point operations per second). The world's most powerful supercomputers now routinely break that barrier—most recently, El Capitan achieved 1.742 exaFLOPS on the HPL benchmark [2].

Across the TOP500 [3] list alone, the aggregate processing power surged from about 5 exaFLOPS in mid-2023 to over 11 exaFLOPS by late 2024. This dramatic growth reflects continuous advancement across multiple fronts: processor architectures, memory systems, interconnects, accelerators, and system software. Moreover, modern supercomputers are no longer purely numerical workhorses—they must support increasingly heterogeneous workloads involving AI, data analytics, and simulation in tandem. Indeed, 83 of the top 100 systems now incorporate hardware accelerators such as GPUs.

In this landscape, High Performance Computing (HPC) has matured into a foundational tool for science, engineering, and data-driven discovery. It complements empirical experimentation and theoretical modeling by enabling large-scale simulations, real-time data analytics, and predictive algorithms at scales otherwise infeasible. In many scientific domains—from climate modeling and genomics to materials science and astrophysics—supercomputers are critical to exploring unseen regimes, validating theories, and interpreting massive datasets.

2.2 Flynn's Taxonomy

Flynn's taxonomy (1966) classifies computer architectures based on how many concurrent instruction streams and data streams they support.
The taxonomy defines four categories:

- **Single Instruction Single Data (SISD)**: This is the traditional von Neumann architecture where a single processor executes a single instruction stream on a single data stream.
- **Single Instruction Multiple Data (SIMD)**: In this architecture, a single instruction operates on multiple data points simultaneously. This is commonly used in vector

processors and GPUs.

- **Multiple Instruction Single Data (MISD)**: This is a less common architecture where multiple instructions operate on a single data stream. It is mostly theoretical and not widely implemented.
- **Multiple Instruction Multiple Data (MIMD)**: In this architecture, multiple processors execute different instructions on different data streams. This is the most flexible and widely used architecture in modern supercomputers.

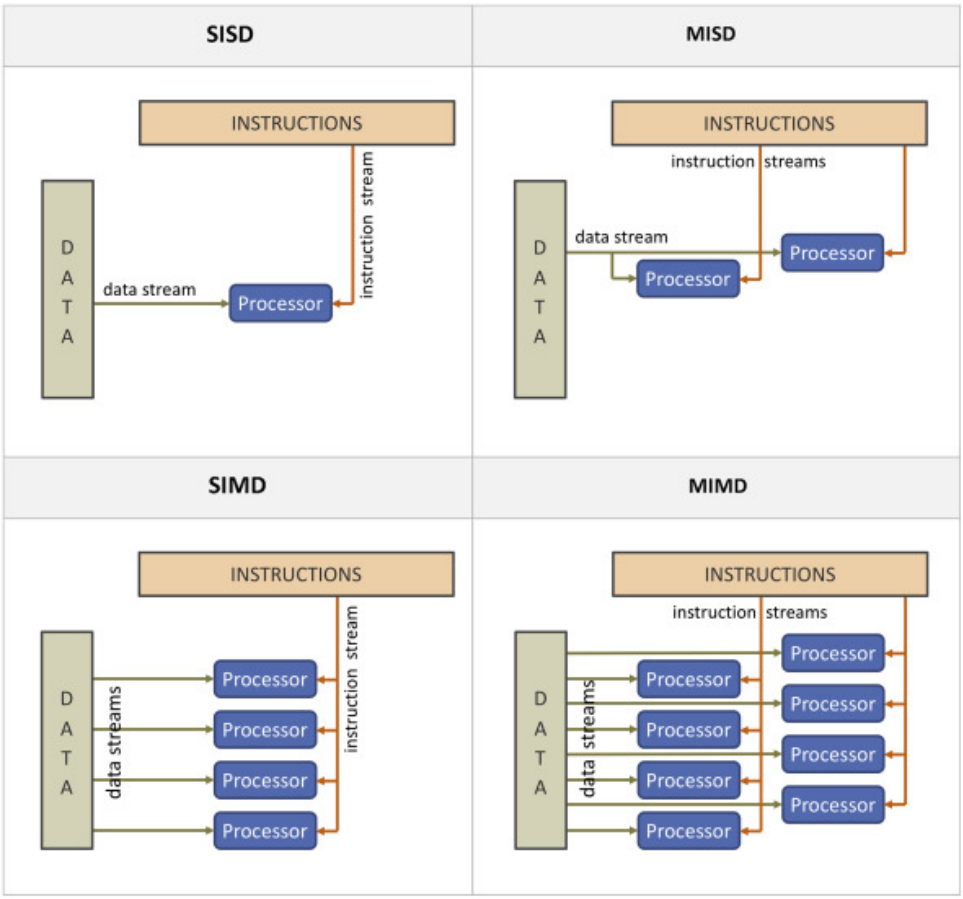


Figure 2.1: Flynn's Taxonomy [4]

2.3 Multi-processor Systems

The multiprocessor class of parallel computer is the dominant form of supercomputer today. Most broadly, it is any system comprising a set of individual self-controlled computers integrated by a communications network and coordinated to perform a single workload. By the Flynn taxonomy the multiprocessor is a MIMD-class machine. There are three mainstream configurations in use: Shared-Memory Multiprocessors, Massively Parallel Processors, and commodity clusters, which will be discussed in the following sections.

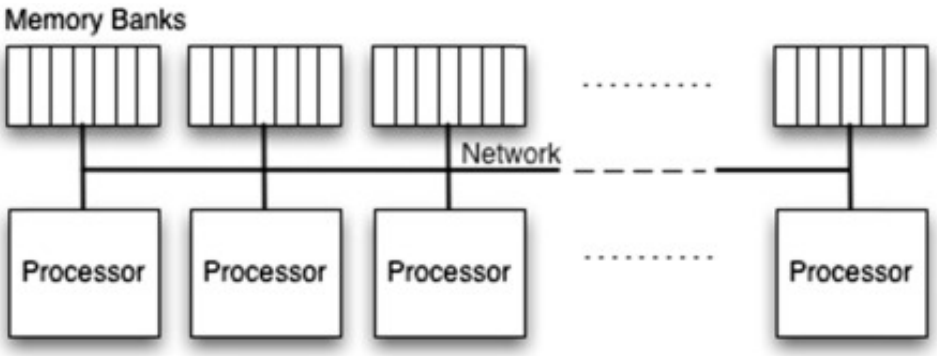


Figure 2.2: Shared-Memory Multiprocessor Architecture

2.3.1 Shared-Memory Multiprocessors

A shared-memory multiprocessor is an architecture consisting of a modest number of processors, all of which have direct (hardware) access to all the main memory in the system 2.2. This permits any of the system processors to access data that any of the other processors has created or will use. The key to this form of multiprocessor architecture is the interconnection network that directly connects all the processors to the memories. Shared-memory multiprocessors are also differentiated by the relative time to access the common memory blocks by their processors into two classes:

- **Uniform Memory Access (UMA) / Symmetric Multi-Processor (SMP)**: All the processors can access each memory block in the same amount of time. Access times can still vary, as contention between two or more processors for any single memory bank will delay access times of one or more processors. But all processors still have the same chance and equal access. UMA systems are typically easier to program because of their uniformity, but they can suffer from scalability issues as the number of processors increases.

- **Non-Uniform Memory Access (NUMA)**: Retain access by all processors to all the main memory blocks within the system 2.3. But this does not ensure equal access times to all memory blocks by all processors. NUMA architectures benefit from scaling, permitting more processor cores to be incorporated into a single shared-memory system than SMPs. However, because of the difference in memory access times, the programmer has to be conscious of the locality of data placement and use it to take best advantage of computing resources.

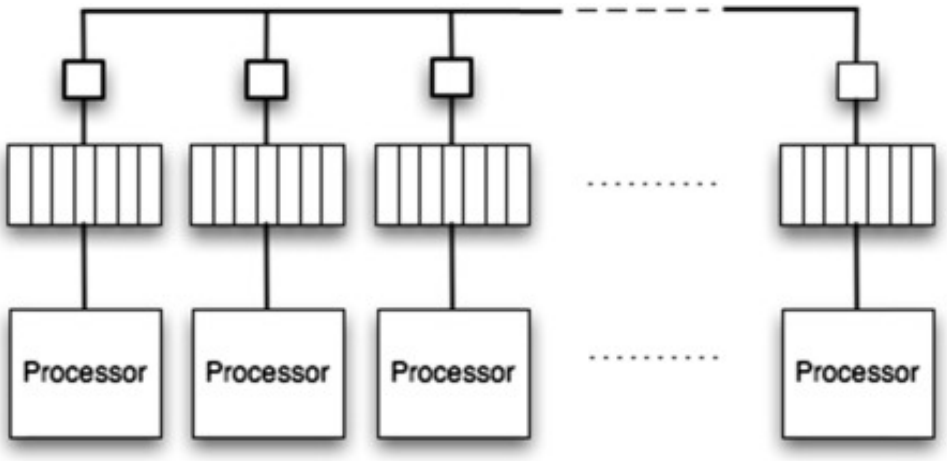


Figure 2.3: Non-Uniform Memory Access Architecture

2.3.2 Massively Parallel Processors

A massively parallel processor (MPP) is a computer system with many independent processors, each with its own local memory, interconnected by a high-speed communication network. The largest supercomputers today, comprising millions of processor cores, are of this class of multiprocessor. MPPs are (in most cases) not shared-memory architectures, but are distributed memory. In an MPP separate groups of processor cores are directly connected to their own local memory. Such groups are colloquially referred to as ”nodes”, and there is no sharing of memory between them; this simplifies design and eliminates inefficiencies that impede scalability. But in the absence of shared memories, a processor core in one group must employ a different method to exchange data and coordinate with cores of other processor groups.

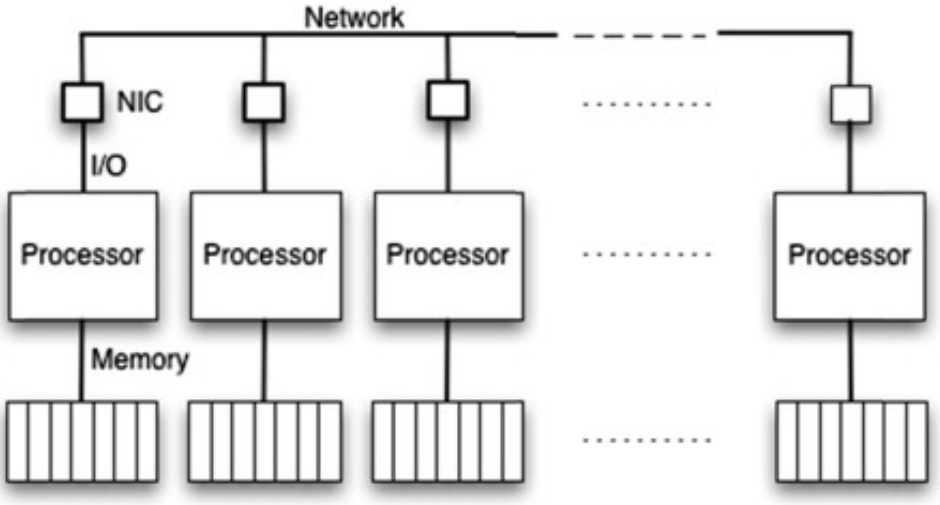


Figure 2.4: Massively Parallel Processor Architecture

2.3.3 Commodity Clusters

While all current generations of supercomputers exploit the economic advantages of incorporating VLSI microprocessors and DRAM main memory that are mass produced for commercial and consumer markets, the systems discussed thus far are still based on special-purpose designs to provide tight coupling among processor cores for superior performance. However, the dominant class of deployed supercomputers, the commodity clusters, take exploitation of mass-market economics one step further to introduce even an greater cost advantage. As the name suggests, such systems consist exclusively of commodity subsystems, sometimes referred to as COTS (commodity off-the-shelf). A cluster in which both the network and the compute nodes are commercial products available for procurement and independent application by organizations (end users or separate vendors) other than the original equipment manufacturer. The key idea is that a supercomputer can be made up of component subsystems, all of which can be procured by and are produced for a much larger user market than the deployed base of supercomputers, thus leveraging economy of scale for dramatic improvements of performance to cost. Clusters represent more than 80% of all the systems on the Top 500 list and a larger part of commercial scalable systems. While they do not drive the very peak performance in the field, they are the class of system most likely to be encountered in a typical machine room, even when such a data center may include a more tightly coupled and expensive massively parallel processor among its other computing resources.

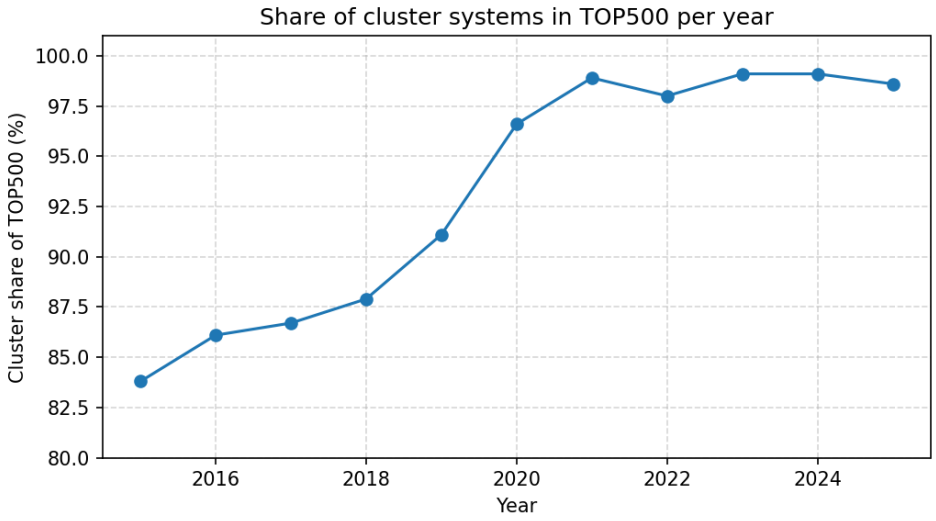


Figure 2.5: Percentage of Clusters in the TOP500 list over time

Chapter 3

Resource Management in HPC Systems

Supercomputer installation frequently represents a significant financial investment by the hosting institution. However, the expenses do not stop after the hardware acquisition and deployment is complete. The hosting data center needs to employ dedicated system administrators, pay for support contracts and/or a maintenance crew, and cover the cost of electricity used to power and cool the machine. Together these are referred to as ”cost of ownership”. The electricity cost is frequently overwhelming for large installations. A commonly quoted average is over US$1 million for each megawatt of power consumed per year in the United States; in many other countries this figure is much higher. It is not surprising that institutions pay close attention to how supercomputing resources are used and how to maximize their utilization.

3.1 Resource Management Systems

Addressing these concerns, resource management software plays a critical role in how supercomputing system resources are allocated to user applications. It not only helps to accommodate different workload sizes and durations, but also provides uniform interfaces across different machine types and their configurations, simplifying access to them and easing (at least some) portability concerns. Resource management tools are an inherent part of the high performance computing (HPC) software stack. They perform three principal functions: resource allocation, workload scheduling, and support for distributed workload execution and monitoring.

3.1.1 Resource Allocation

Resource allocation takes care of assigning physical hardware, which may span from a fraction of the machine to the entire system, to specific user tasks based on their requirements. Resource managers typically recognize the following resource types:

- **Compute nodes** Increasing the number of nodes assigned to a parallel application is the simplest way to scale the size of the dataset (such as the number of grid points in a simulation domain) on which the work is to be performed, or reduce the execution time for a fixed workload size. Node count is therefore one of the most important parameters requested when scheduling an application launched on a parallel machine. Even single physical computers may include various node types; for example differing in memory capacity, central processing unit (CPU) types and clock frequency, local storage

characteristics, available interconnects, etc. Properly configured resource managers permit selection of the right kind of node for the job, precluding assigning resources that will likely go unused.

- **Processing cores (processing units, processing elements)** Most modern supercomputer nodes feature one or more multicore processor sockets, providing local parallelism to applications that support it through multithreading or by accommodating several concurrent single-threaded processes. For that reason, resource managers provide the option of specifying shared or exclusive allocation of nodes to workloads. Shared nodes are useful in situations where already assigned workloads would leave some of the cores unoccupied. By coscheduling different processes on the remaining cores, better utilization may be achieved. However, this comes at a cost: all programs executing on the shared node will also share access to other physical components, such as memory, network interfaces, and input/output (I/O) buses. Users who perform careful benchmarking of their applications are frequently better off allocating the nodes in exclusive mode to minimize the intrusions and resulting degradation due to contention caused by unrelated programs. Exclusive allocation can also be used for programs that rely on the affinity of the executing code to specific cores to achieve good performance. For example, programs that rely on lowest communication latency may want to place the message sending and receiving threads on cores close to the PCI express bus connected to the related network card. This may not be possible when multiple applications enforce their own, possibly conflicting, affinity settings at the same time.

- **Interconnect** While many systems are built with only one network type, some installations explicitly include multiple networks or have been expanded or modernized to take advantage of different interconnect technologies, such as GigE and InfiniBand architecture in combination. Selection of the right configuration depends on the application characteristics and needs. For example, is the program execution more sensitive to communication latency, or does it need as much communication bandwidth as possible? Can it take advantage of channel bonding using different network interfaces? Often the answer may be imposed by the available version of the communication library with which the application has been linked. For example, it is common to see message-passing interface (MPI) [5] installations with separate libraries supporting InfiniBand and Ethernet if both such network types are available. Selecting a wrong network type will likely result in less efficient execution.

- **Permanent storage and I/O options** Many clusters rely on shared file systems that are exported to every node in the system. This is convenient, since storing a program compiled on the head node in such a file system will make it available to the compute nodes as well. Computations may also easily share a common dataset, with modifications visible to the relevant applications already during their runtime. However, not all installations provide efficient high-bandwidth file systems that are scalable to all machine resources and can accommodate concurrent access by multiple users. For programs performing a substantial amount of file I/O, localized storage such as local disks of individual nodes or burst buffers (fast solid-state device pools servicing I/O requests for predefined node groups) may be a better solution. Such local storage pools are typically mounted under a predefined directory path. The drawback is that the datasets generated this way will have to be explicitly moved to the front-end storage after job completion to permit general access (analysis, visualization, etc.). Since there is no single solution

available, users should consult local machine guides to determine the best option for their application and how it can be conveyed to the resource management software.

- **Accelerators** Heterogeneous architectures that employ accelerators (graphics processing units (GPUs), many integrated cores (MICs), field programmable gate array modules, etc.) in addition to main CPUs are a common way to increase the aggregate computational performance while minimizing power consumption. However, this complicates resource management, since the same machine may consist of some nodes that are populated with accelerators of one type, some nodes that are populated with accelerators of a different type, and some nodes that do not contain any accelerating hardware at all. Modern resource managers permit users to specify parameters of their jobs so that the appropriate node types are selected for the application. At the same time, codes that do not need accelerators may be confined to regular nodes as much as possible for best resource utilization over multiple jobs.

3.1.2 Workload Scheduling

Jobs

Resource managers allocate the available computing resources to jobs specified by users. A job is a self-contained work unit with associated input data that during its execution produces some output result data. The output may be as little as a line of text displayed on the console, or a multiterabyte dataset stored in multiple files, or a stream of information transmitted over the local or wide area network to another machine. Jobs may be executed interactively, involving user presence at the console to provide additional input at runtime as required, or use batch processing where all necessary parameters and inputs for job execution are specified before it is launched. Batch processing provides much greater flexibility to the resource manager, since it can decide to launch the job when it is optimal from the standpoint of HPC system utilization and is not hindered by the availability of a human operator, for example at night. For this reason, interactive jobs on many machines may be permitted to use only a limited set of resources. Jobs may be monolithic or subdivided into a number of smaller steps or tasks. Typically each task is associated with the launch of a specific application program. In general, individual steps do not have to be identical in terms of used resources or duration of execution. Jobs may also mix parallel application invocations with instantiations of single-threaded processes, dramatically changing the required resource footprint.

Queues

Pending computing jobs are stored in job queues. The job queue defines the order in which jobs are selected by the resource manager for execution. As the computer science definition of the word suggests, in most cases it is ”first in, first out” or ”FIFO”, although good job schedulers will relax this scheme to boost machine utilization, improve response time, or otherwise optimize some aspect of the system as indicated by the operator (user or system administrator). Most systems typically use multiple job queues, each with a specific purpose and set of scheduling constraints. Thus one may find an interactive queue solely for interactive jobs. Similarly, a debug queue may be employed that permits jobs to run in a restricted parallel environment that is big enough to expose problems when running on multiple nodes using the same configuration as the production queue, yet small enough that the pool of nodes for production jobs may remain substantially larger. Frequently there are multiple production queues available, each with a different maximum execution time imposed on jobs or total job size (short versus long, large versus small, etc.).

Scheduling

With hundreds to thousands of jobs with different properties pending in all queues of a typical large system, it is easy to see why scheduling algorithms are critical to achieving high job throughput. Common parameters that affect job scheduling include the following.

- **Availability** of execution and auxiliary resources is the primary factor that determines when a job can be launched.
- **Priority** permits more privileged jobs to execute sooner or even preempt currently running jobs of lower priority.
- **Resources** allocated to the user determines the long-term resource pool a specific user may consume while his or her account on the machine remains active.
- **Maximum number of jobs** that a user is permitted to execute simultaneously.
- **Requested execution time** estimated by the user for the job.
- **Elapsed execution time** may cause forced job termination or impact staging of pending jobs for upcoming execution.
- **Job dependencies** determine the launching order of multiple related jobs, especially in producer-consumer scenarios.
- **Event occurrence**, when the job start is postponed until a specific predefined event occurs.
- **Operator availability** impacts the launch of interactive applications.
- **Software license availability** if a job is requesting the launch of proprietary code.

In general, scheduling algorithms [6] are divided into two classes: time-sharing and space-sharing.
Time-sharing algorithms divide time on a processor into several discrete intervals or slots. These slots are then assigned to unique jobs.
Space-sharing algorithms give the requested resources to a single job until the job completes execution.
Most cluster schedulers operate in space-sharing mode.

3.1.3 Workload Execution and Monitoring

Resource managers are equipped with optimized mechanisms that enable efficient launching of thousands or more processes across a comparable number of nodes. Naïve approaches, such as repeated invocation of a remote shell, will not yield acceptable results at scale due to high contention when transferring multiple programs' executables to the target nodes.

Job launchers employ hierarchical mechanisms to alleviate the bandwidth requirements and exploit network topology to minimize the amount of data transferred and overall launch time. Resource managers must be able to terminate any job that exceeds its execution time or other resource limits, irrespective of its current processing status. Again, distributed termination should be efficient to release the allocated nodes to the pool of available nodes as quickly as possible. Finally, resource managers are responsible for monitoring application execution and keeping track of related resource usage. The actual resource utilization data is recorded to enable accounting and accurate charging of users for their cumulative system resource usage.

Chapter 4

Slurm Workload Manager

4.1 Introduction - Overview

Slurm [7] is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm requires no kernel modifications for its operation and is relatively self-contained. As a cluster workload manager, Slurm has three key functions. First, it allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work. Optional plugins can be used for accounting, advanced reservation, gang scheduling (time sharing for parallel jobs), backfill scheduling, topology optimized resource selection, resource limits by user or bank account, and sophisticated multifactor job prioritization algorithms.

4.2 Architecture

Slurm has a centralized manager, slurmctld, to monitor resources and work. There may also be a backup manager to assume those responsibilities in the event of failure. Each compute server (node) has a slurmd daemon, which can be compared to a remote shell: it waits for work, executes that work, returns status, and waits for more work. The slurmd daemons provide fault-tolerant hierarchical communications. There is an optional slurmdbd (Slurm DataBase Daemon) which can be used to record accounting information for multiple Slurm-managed clusters in a single database. There is an optional slurmrestd (Slurm REST API Daemon) which can be used to interact with Slurm through its REST API. User tools include srun to initiate jobs, scancel to terminate queued or running jobs, sinfo to report system status, squeue to report the status of jobs, and sacct to get information about jobs and job steps that are running or have completed. The sview commands graphically reports system and job status including network topology. There is an administrative tool scontrol available to monitor and/or modify configuration and state information on the cluster. The administrative tool used to manage the database is sacctmgr. It can be used to identify the clusters, valid users, valid bank accounts, etc. APIs are available for all functions.

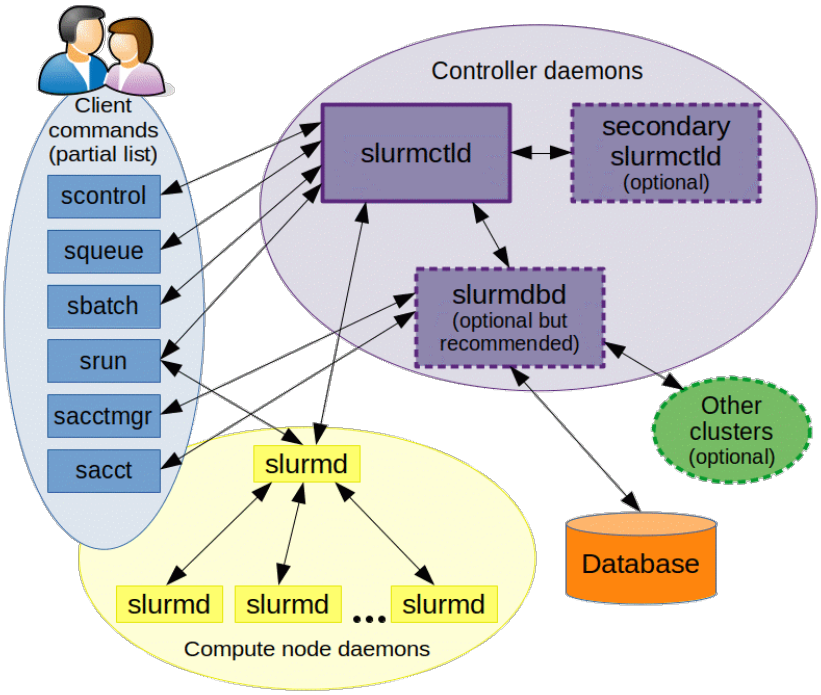


Figure 4.1: Slurm Architecture Overview

4.3 Workload Organization

The entities managed by these Slurm daemons, include nodes, the compute resource in Slurm, partitions, which group nodes into logical sets, jobs, or allocations of resources assigned to a user for a specified amount of time, and job steps, which are sets of (possibly parallel) tasks within a job. The partitions can be considered job queues, each of which has an assortment of constraints such as job size limit, job time limit, users permitted to use it, etc. Priority-ordered jobs are allocated nodes within a partition until the resources (nodes, processors, memory, etc.) within that partition are exhausted. Once a job is assigned a set of nodes, the user is able to initiate parallel work in the form of job steps in any configuration within the allocation. For instance, a single job step may be started that utilizes all nodes allocated to the job, or several job steps may independently use a portion of the allocation. Slurm provides resource management for the processors allocated to a job, so that multiple job steps can be simultaneously submitted and queued until there are available resources within the job's allocation.

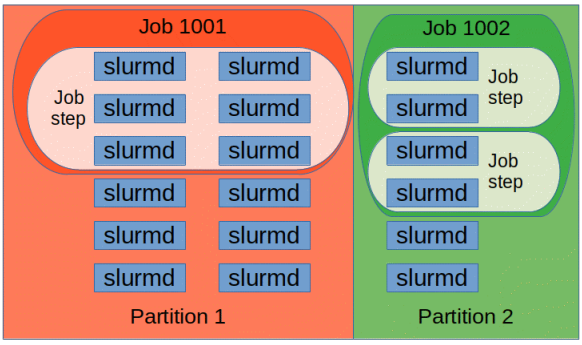


Figure 4.2: Example of Slurm Entities

4.4 Slurm Plugins

Slurm has a general-purpose plugin mechanism available to easily support various infrastructures. This permits a wide variety of Slurm configurations using a building block approach. These plugins presently include:

- **Accounting Storage:** Primarily Used to store historical data about jobs. When used with SlurmDBD (Slurm Database Daemon), it can also supply a limits based system along with historical system status.
- **Account Gather Energy:** Gather energy consumption data per job or nodes in the system. This plugin is integrated with the Accounting Storage and Job Account Gather plugins.
- **Authentication of communications:** Provides authentication mechanism between various components of Slurm.
- **Containers:** HPC workload container support and implementations.
- **Credential (Digital Signature Generation):** Mechanism used to generate a digital signature, which is used to validate that job step is authorized to execute on specific nodes. This is distinct from the plugin used for Authentication since the job step request is sent from the user's srun command rather than directly from the slurmctld daemon, which generates the job step credential and its digital signature.
- **Generic Resources:** Provide interface to control generic resources, including Graphical Processing Units (GPUs).
- **Job Submit:** Custom plugin to allow site specific control over job requirements at submission and update.
- **Job Accounting Gather:** Gather job step resource utilization data.
- **Job Completion Logging:** Log a job's termination data. This is typically a subset of data stored by an Accounting Storage Plugin.
- **Launchers:** Controls the mechanism used by the 'srun' command to launch the tasks.
- **MPI:** Provides different hooks for the various MPI implementations. For example, this can set MPI specific environment variables.
- **Preempt:** Determines which jobs can preempt other jobs and the preemption mechanism to be used.
- **Priority:** Assigns priorities to jobs upon submission and on an ongoing basis (e.g. as they age).
- **Process tracking (for signaling):** Provides a mechanism for identifying the processes associated with each job. Used for job accounting and signaling.
- **Scheduler:** Plugin determines how and when Slurm schedules jobs.
- **Node selection:** Plugin used to determine the resources used for a job allocation.

- **Site Factor (Priority):** Assigns a specific site_factor component of a job's multifactor priority to jobs upon submission and on an ongoing basis (e.g. as they age).
- **Switch or interconnect:** Plugin to interface with a switch or interconnect. For most systems (Ethernet or InfiniBand) this is not needed.
- **Task Affinity:** Provides mechanism to bind a job and its individual tasks to specific processors.
- **Network Topology:** Optimizes resource selection based upon the network topology. Used for both job allocations and advanced reservation.

4.5 Configurability

Node state monitored include: count of processors, size of real memory, size of temporary disk space, and state (UP, DOWN, etc.). Additional node information includes weight (preference in being allocated work) and features (arbitrary information such as processor speed or type). Nodes are grouped into partitions, which may contain overlapping nodes so they are best thought of as job queues. Partition information includes: name, list of associated nodes, state (UP or DOWN), maximum job time limit, maximum node count per job, group access list, priority (important if nodes are in multiple partitions) and shared node access policy with optional over-subscription level for gang scheduling (e.g. YES, NO or FORCE:2). Bit maps are used to represent nodes and scheduling decisions can be made by performing a small number of comparisons and a series of fast bit map manipulations.

Chapter 5

Flux

5.1 Introduction

Flux [8] is a next-generation resource and job management framework developed by Lawrence Livermore National Laboratory. It expands the scheduler's view beyond the single dimension of "nodes" combining hierarchical job management with graph-based scheduling. Instead of simply developing a replacement for SLURM and Moab, Flux offers a framework that enables new resource types, schedulers, and framework services to be deployed as data centers continue to evolve. Even though Flux is still under active development, it is currently being used in production on the following Top500 systems [9]:

- El Capitan (#1)
- Tuolumne (#12)
- El Dorado (#28)
- rzAdams (#64)
- Tioga (#257)
- Tenaya (#337)

5.2 Architecture

5.2.1 The Flux Instance

A Flux instance is a self-contained workload manager that consists of one or more Flux brokers communicating over a tree-based overlay network 5.1. Each broker provides message passing, event routing, and service discovery within the instance, forming the backbone of Flux's distributed architecture.

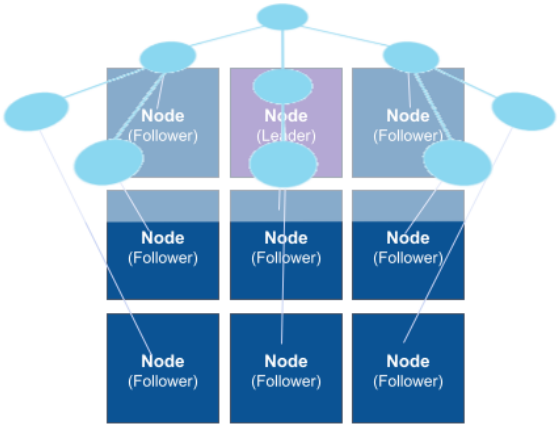


Figure 5.1: Flux Network Overview [10]

5.2.2 The Flux Broker

The flux broker [11],[12],[13] is a distributed message broker daemon that provides communications services within a Flux instance. Each broker is a program built on top of the ∅MQ (ZeroMQ) [14] networking library. The broker contains two main components.

Overlay Network

First, the broker implements Flux-specific networking abstractions over ∅MQ, such as remote-proceedure call (RPC) and publication-subscription (pub-sub). The broker session is interconnected using ∅MQ sockets to implement three persistent overlay network planes:

- a PGM publish-subscribe bus for events and synchronization heartbeats
- a TCP request-response tree for scalable RPCs, barriers, and reductions
- a secondary TCP request-response overlay with configurable topology for rank-addressed RPCs

The session wire-up is depicted in Figure 5.2. Each message plane implements reliable, in-order message delivery, and can self-heal when interior nodes fail. Although a binary RPC/reduction tree is pictured, the tree shape is configurable.

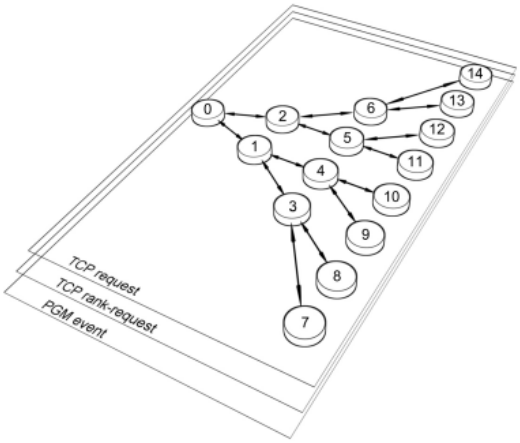


Figure 5.2: A Broker Session [12]

Core Services

Second, the broker contains several core services, such as PMI (for MPI support), run control support (for enabling automatic startup of other services), and, most importantly, broker module management. The remainder of a Flux broker's functionality comes from broker modules: specially designed services that the broker can deploy in independent OS threads 5.4.

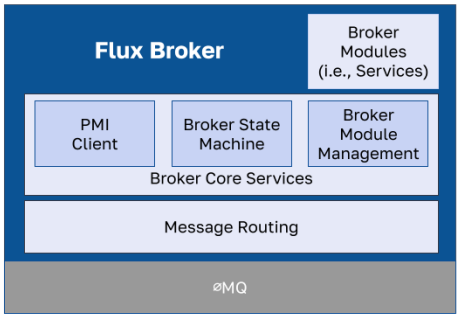


Figure 5.3: Flux Broker Overview

Broker Startup

When a Flux instance is started, one flux-broker process is launched on each allocated node. Each broker is assigned a rank from 0 to size - 1. The rank 0 node is the root of a tree-based overlay network. This network may be accessed by Flux commands and modules using Flux API services. A logging service aggregates Flux log messages across the instance and emits them to a configured destination on rank 0. After its overlay network has completed wire-up, flux broker starts the initial program on rank 0. Most of Flux's distributed systems and services that aren't directly associated with a running job are embedded in the flux-broker executable or its dynamically loaded plugins.

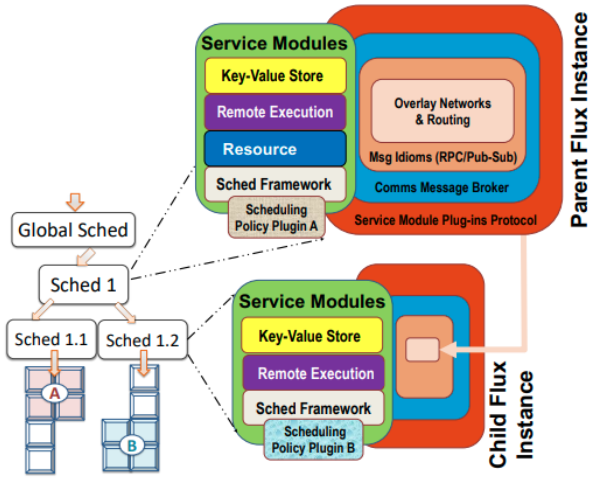


Figure 5.4: An Overview of the Modular and Hierarchical Nature of Flux

5.2.3 KVS

Key-Value Stores (KVS) have become ubiquitous building blocks in large-scale Internet services but have been underutilized in HPC. For Flux, however it provides one of the essential building blocks.

The Flux KVS is implemented as a core broker module that utilizes the request-response and event overlay networks. It provides a general purpose data store used by other Flux components, and supports the distributed caching and synchronization needed for parallel data exchanges, such as required for MPI bootstrap.

5.2.4 Flux Instance Modes

Single User Mode

Flux may be used in single-user mode, where a Flux instance is launched as a parallel job, and the instance owner (the user that submitted the parallel job) has control of, and exclusive access to, the Flux instance and its assigned resources. On a system running Flux natively, batch jobs and allocations are examples of single user Flux instances.

Multi User / System Mode

When Flux is deployed as the system instance, or native resource manager on a cluster, its brokers still run with the credentials of a non-privileged system user, typically flux. However, to support multiple users and to act as a long running service, it must be configured to behave differently:

- The Flux broker is started directly by systemd on each node instead of being launched as a process in a parallel job.
- The systemd unit file passes arguments to the broker that tell it to use system paths for various files, and to ingest TOML files from a system configuration directory.
- A single security certificate is used for the entire cluster instead of each broker generating one on the fly and exchanging public keys with PMI.
- The Flux overlay network endpoints are statically configured from files instead of being generated on on the fly and exchanged via PMI.
- The instance owner is a system account that does not correspond to an actual user.
- Users other than the instance owner (guests) are permitted to connect to the Flux broker, and are granted limited access to Flux services.
- Users connect to the Flux broker's AF_UNIX socket via a well known system URI if FLUX_URI is not set in the environment.
- Job processes (including the Flux job shell) are launched as the submitting user with the assistance of a setuid root helper on each node called the IMP.
- Job requests are signed with MUNGE, and this signature is verified by the IMP.
- The content of the Flux KVS, containing system state such as the set of drained nodes and the job queue, is preserved across a full Flux restart.
- The system instance functions with some nodes offline.
- The system instance has no initial program.

The same Flux executables are used in both single user and system modes, with operation differentiated only by configuration. This architectural consistency enables seamless transitions between user-managed and system-managed deployments, supporting both research and production use cases within the same software framework.

5.3 Flux Software Components

Flux was conceived as a resource manager toolkit rather than a monolithic project, with the idea to make components like the scheduler replaceable. In addition, several parts of flux can be extended with plugins. At this time the primary component types are:

- **broker modules** Each broker module runs in its own thread as part of the broker executable, communicating with other components using messages. Broker modules are dynamically loadable with the flux-module command. Core services like the KVS, job manager, and scheduler are implemented using broker modules.
- **jobtap plugins** The job manager orchestrates a job's life cycle. Jobtap plugins extend the job manager, arranging for callbacks at different points in the job life cycle. Jobtap plugins may be dynamically loaded with the flux-jobtap command. An example of a jobtap plugin is the Flux accounting multi-factor priority plugin, which updates a job's priority value when it enters the PRIORITY state.
- **shell plugins** When a job is started, the flux-shell is the process parent of job tasks on each node. Shell plugins extend the job environment and can be configured on a per-job basis using the –setopt option on job submission commands.
- **connectors** Flux commands open a connection to a particular Flux instance by specifying a URI. The scheme portion of the URI may refer to a native connection method such as local or ssh. Native connection methods are implemented as plugins called connectors.
- **URI resolver plugins** Other URI schemes must be resolved to a native form before they can be used. Resolvers for new schemes may be added as plugins. For example, the lsf resolver plugin enables LSF users to connect to Flux instances running as LSF jobs by specifying a lsf:JOBID URI.
- **validator plugins** Jobs may be rejected at ingest if their jobspec fails one of a set of configured validator plugins. The basic validator ensures the jobspec conforms to the jobspec specification. The feasibility plugin rejects job that the scheduler determines would be unable to run given the instance's resource set. The require-instance plugin rejects jobs that do not run in a new Flux instance.
- **frobnicator plugins** The frobnicator allows a set of configured plugins to modify jobspec at submission time. For example the defaults plugin sets configured default values for jobspec attributes such as duration and queue.

Broker modules 5.5 are the most fundamental component type, and the following sections describe the most important one, the Flux scheduler - Fluxion (flux-sched).

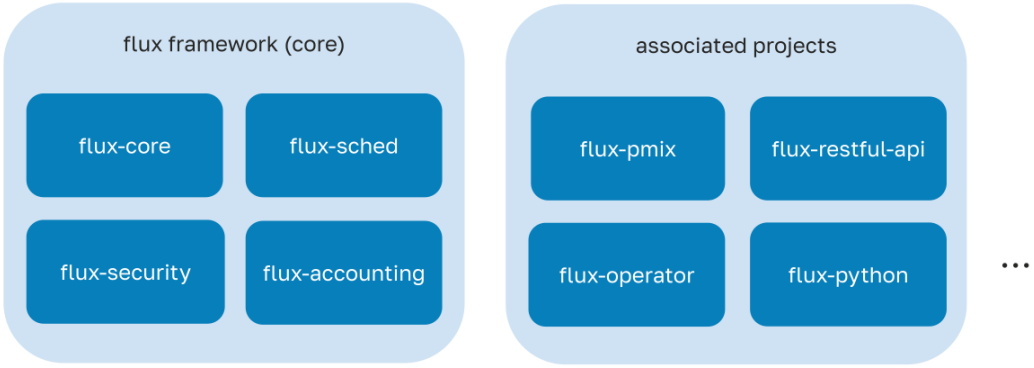


Figure 5.5: Flux Modules and Associated Projects Overview

5.3.1 Flux Scheduler - Fluxion

Graph-based Resource Representation

Current-generation workload management products are designed to manage static, homogeneous HPC systems of the past, and their representation of resources reflects this rigid thinking. Data management and storage structures designed for efficiently representing compute-node-centric hardware resources do not encode complex and changing relationships (e.g., power capping, network flows, location), which makes them incapable of representing important components of newer heterogeneous, dynamic systems. Flux overcomes the limitations of current products by basing its resource representation (a model for characterizing resources) on a directed graph - a powerful and expressive structure capable of dynamically defining arbitrary resource types.

A vertex can be a hardware resource (e.g., a CPU or compute node), and an edge can indicate containment (i.e., a server contains a CPU). 5.6 is a visual representation of resource vertices and edges in a system with multi-tiered disk storage that can be allocated as a global pool or with respect to the distance (measured in number of edges) from other resources (e.g., a core). Matching a resource request consists of descending into the graph and checking vertices for suitability. Specifying different vertex and edge structure allows for tremendous request flexibility: Selecting solid-state drives in 5.6 via a path through a rack (e.g., purple vertex rack0 to green vertex mtl1_0) versus through mt1l2_0 (orange vertex near the graph center) permits priority based on proximity which is extremely difficult for current-generation schedules to replicate. The ability to allocate resources in different ways based on paths is a unique capability of Flux, and one that is necessary for the upcoming El Capitan exascale system at LLNL.

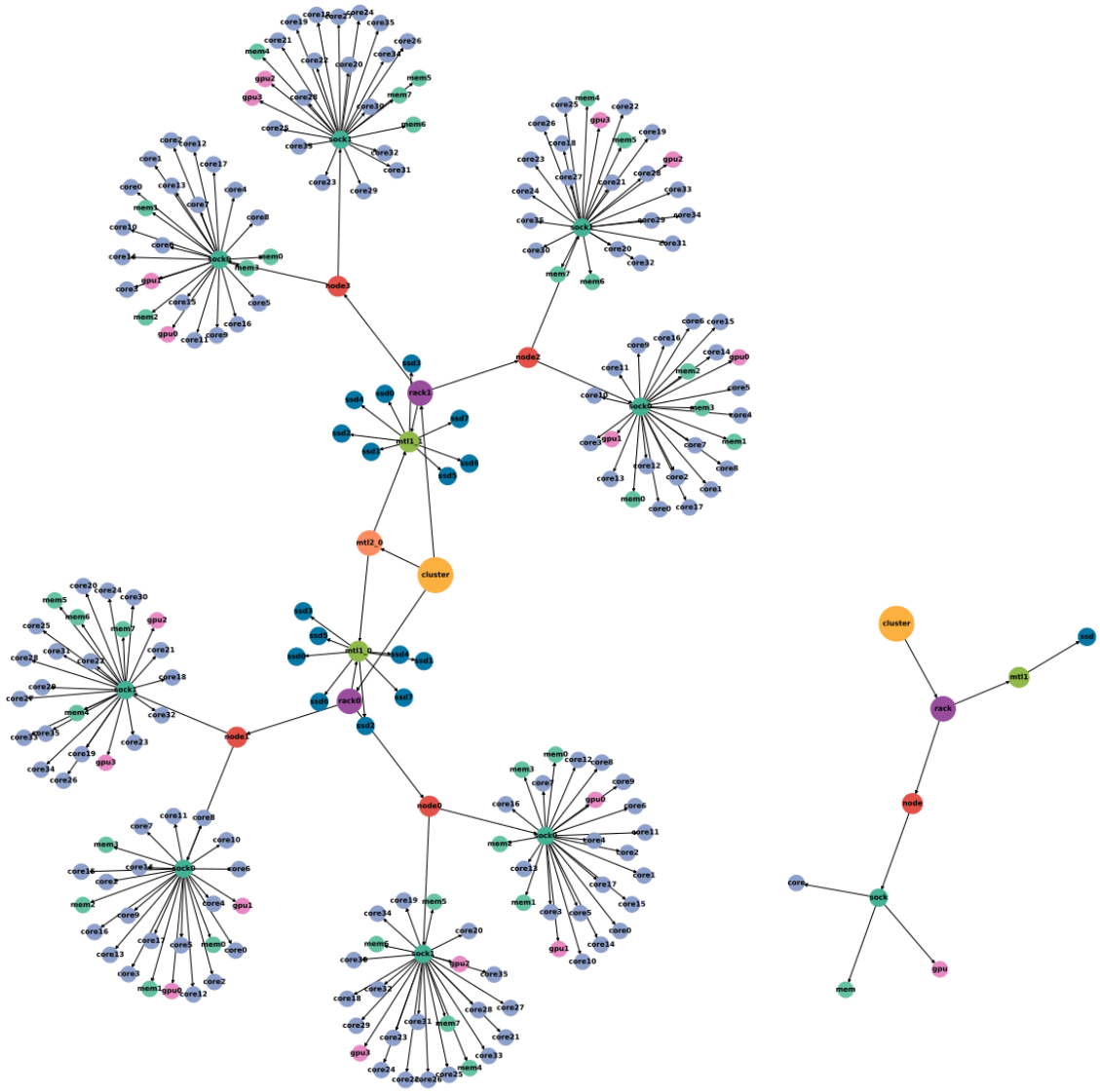


Figure 5.6: Resource Representation Example [15]

Using a directed graph as a foundation for resource representation provides Flux with several key capabilities. The abstract model facilitates tremendous flexibility: Any type of resources (e.g., hardware, software, power distribution units) can be a vertex, and relationships between vertices are well-defined. Hierarchical scheduling assumes an elegant form when based on a directed graph model. Each Flux instance manages and schedules a subgraph (subset of the vertices and edges) of the resource graph, where a child instance's purview is a subgraph of its parent. Furthermore, a tremendous number of algorithmic techniques and optimized software libraries exist for performing fast operations on directed graphs. By basing its resource model on a directed graph, Flux integrates the fruits of algorithmic development to perform many required operations: e.g., quickly checking resource states, scheduling allocations, adding/removing resources, and transforming representations.

5.3.2 Flux Job Model

A user job request is represented as a jobspec, which is a hierarchical data structure that describes the job's resource requirements and execution parameters.
The Jobspec has the following form:

- **Resources**:
  The resource request graph, which describes the resources required by the job. Resource types include clusters(there is multi-cluster support), racks, nodes, sockets, cores, gpus, memory. The only resource type that is non physical is the slot resource, which is an abstract resource that represents a schedulable place where a program process or processes will be spawned and contained. All resource vertices that are specified as children or descendants of a slot resource are considered to be the slot's resources. Each slot has a unique label that maps it to a specific task.

- **Tasks**:
  The set of task groups that comprise the job. Each task group is mapped to a slot label and command to be executed within that slot. This command may be a single program or a parallel job launcher such as mpirun or srun. The number of tasks or how it is distributed among slots for each command is specified as part of the task group.

- **Attributes**:
  The set of user and system attributes that encompass the environment in which the job will run. This includes environment variables, working directory, I/O redirection, time limit, and other parameters.

Here is an example jobspec in YAML format:

```yaml
version: 1
resources:
  - type: node
    count: 4
    with:
      - type: slot
        count: 1
        label: default
        with:
          - type: core
            count: 2
tasks:
  - command: [ "app" ]
    slot: default
    count:
      per_slot: 1
attributes:
  system:
    duration: 3600.
    cwd: "/home/flux"
    environment:
      HOME: "/home/flux"
```

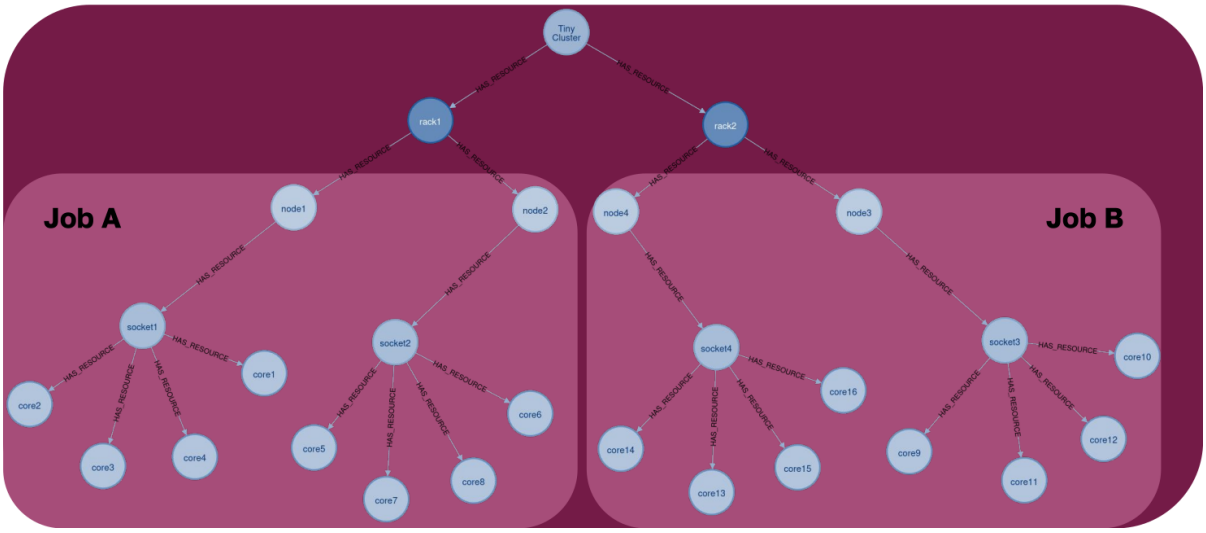


Figure 5.7: All jobs on a Flux cluster are child instances of the system instance

Graph-based Scheduling

Flux's scheduler component, called Fluxion [16], is represented in Figure 5.8. During Flux instance initialization, Fluxion first populates an in-memory resource graph store (A) comprising vertices that represent the HPC system's various compute resources and edges that represent the relationships among those resources. The initialization process also includes the selection of the graph resource's representation granularity and traversal type if users decided to use non-default. Once initialization is complete, Fluxion is ready to receive the jobs' resource requests from Flux's core framework. Flux first constructs a job's resource request in the form of an abstract resource request graph (B). The abstract request graph generally specifies the job's resource requirements in terms of both node-local resources (e.g., amount of compute cores and memory to be used) and higher level or even global resources (e.g., compute racks, network switches, power, parallel filesystem bandwidth). The abstract request graph then becomes the input for the selected graph traverser (A) to find its best- matching resource vertices and edges. The traverser "walks" the concrete resource graph store in this pre-defined walking order and matches the abstract request graph to the concrete resource graph. As shown at (C), the best-matching criteria is determined by the match policy within Flux's traverser. The policy is invoked every time the traverser visits a vertex. The policy then evaluates how well a given resource vertex matches with the abstract request graph and scores it accordingly. Flux's resource model must also efficiently keep track of the status changes of resources over time in order to support various queuing and backfilling policies common to HPC job scheduling (e.g., EASY and conservative backfilling policies). Thus, the model integrates a highly efficient resource-time state tracking and search mechanism into every resource vertex. This mechanism (and a simple abstraction) is called Planner (E).

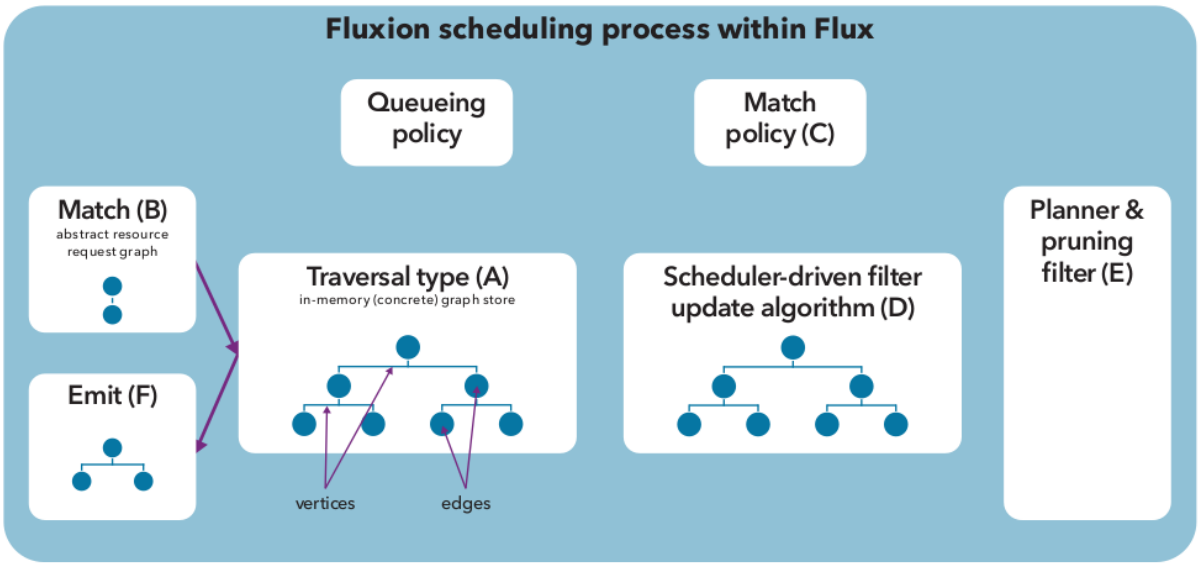


Figure 5.8: Fluxion Scheduler Overview

After judicious selection of the appropriate representation granularity for the concrete resource graph—striking a balance between performance and scheduling effectivenes - the resulting graph can still be quite large when modeling high- end systems. Thus, the Fluxion scheduler includes other scalability strategies in its model, such as pruning filters (E). For example, pruning filters can be installed at high-level resource vertices (e.g., compute racks) to track the amount of available lower-level resources (e.g., compute cores) in aggregate, which reside somewhere in the subgraph rooted at that vertex. Fluxion also introduces a novel scheduler- driven filter update algorithm (D) that updates and maintains these filters without incurring high performance overhead. This filter significantly improves performance by pruning the required graph search. Finally, once Fluxion determines the best matching resource subgraph, this is emitted as a selected resource set representation at (F). Flux's core framework can then make use of this resource set to contain, bind and execute the target program(s) within those resources.

Scheduling Policies

Fluxion supports a variety of resource matching and queuing policies. The resource matching policies are the following:

- **low** Select resources with low ID first (e.g., core0 is selected first before core1 is selected).
- **high** Select resources with high ID first (e.g., core15 is selected first before core14).
- **lonode** Select resources with lowest compute-node ID first; otherwise the low policy (e.g., for node-local resource types).
- **hinode** Select resources with highest compute-node ID first; otherwise the high policy (e.g., for node-local resource types).
- **lonodex** A node-exclusive scheduling whose behavior is identical to lonode except each compute node is exclusively allocated.

- **hinodex** A node-exclusive scheduling whose behavior is identical to hinode except each compute node is exclusively allocated.
- **first** Select the first matching resources and stop the search.
- **firstnodex** A node-exclusive scheduling whose behavior is identical to first except each compute node is exclusively allocated.

Match policies are initialized as collections of individual attributes that help the scheduler to select matches. For convenience, these attributes are exposed to users such that they can write custom policies. Below is a list of match attributes which can be selected by users.

- **policy** Allowed options are low or high. If only policy=low or policy=high is specified, the behavior of the match policy is the same as if match-policy=low or match-policy=high were selected, respectively.
- **node_centric** true or false are allowed options. Evaluate matches based on the ID of the compute node first.
- **node_exclusive** true or false are allowed options. Exclusively allocate compute nodes when a match is found.
- **set_stop_on_1_matches** true or false are allowed options. When a match is found, take it, without evaluating for potentially more optimal matches.

The queuing policies offered are the following:

- **fcfs**
  First come, first served policy if the priority of pending jobs are same: i.e., jobs are scheduled and run by their submission order. If pending jobs have different priorities, they are serviced by their priority order.
- **easy**
  EASY-backfilling policy: If the highest-priority pending job cannot be run with fcfs because its requested resources are currently unavailable, one or more next high priority jobs will be scheduled and run as far as this will not delay the start time of running the highest-priority job.
- **conservative**
  CONSERVATIVE-backfilling policy: Similarly to easy, pending jobs can run out of order when the highest-priority job cannot run because its requested resources are currently unavailable. However, this policy is more conservative as a lower priority job can only be backfilled and run if and only if this will not delay the start time of running any pending job whose priority is higher than the backfilling job.
- **hybrid**
  HYBRID-backfilling policy: This is an optimization of conservative where a lower priority job can only be backfilled and run if and only if this will not delay the start time of running N pending jobs whose priority is higher than the backfilling job. N can be configured by the policy-params.reservation-depth parameter: see policy-params.reservation-depth

5.4 Flux CLI Commands

This section will cover the most important and commonly used Flux commands.

flux start

Usage: [launcher] flux start [OPTIONS] [initial-program [args...]]
or: flux start –test-size=N [OPTIONS] [initial-program [args...]]
Description:
flux start assists with launching a new Flux instance, which consists of one or more flux-broker processes functioning as a distributed system. It is primarily useful in environments that don't run Flux natively, or when a standalone Flux instance is required for test, development, or post-mortem debugging of another Flux instance.
Useful options:
-S, –setattr=ATTR=VAL Set broker attribute ATTR to VAL. This is equivalent to -o,-SATTR=VAL
-o, –setopt=OPT=VAL Set broker option OPT to VAL. This is equivalent to -S, -oOPT=VAL
-c, –config-path=PATH Set the PATH for broker configuration.

flux alloc

Usage: flux alloc [OPTIONS] [COMMAND...]
Description:
runs a Flux subinstance with COMMAND as the initial program. Once resources are allocated, COMMAND executes on the first node of the allocation with any free arguments supplied as COMMAND arguments. When COMMAND exits, the Flux subinstance exits, resources are released to the enclosing Flux instance, and flux alloc returns. If no COMMAND is specified, an interactive shell is spawned as the initial program, and the subinstance runs until the shell is exited.
Useful options:
-n, –nslots=N Set the number of slots requested. This parameter is required unless –nodes is specified.
-c, –cores-per-slot=N Set the number of cores to assign each slot. The default is 1.
-g, –gpus-per-slot=N Set the number of GPU devices to assign to each slot (default none).
-N, –nodes=N Distribute allocated resource slots across N individual nodes.
-x, –exclusive With –nodes, allocate nodes exclusively.
-q, –queue=NAME Submit a job to a specific named queue. If a queue is not specified and queues are configured, then the jobspec will be modified at ingest to specify the default queue. If queues are not configured, then this option is ignored

flux submit

Usage: flux submit [OPTIONS] [–ntasks=N] COMMAND...
Description:
flux submit enqueues a job to run under Flux and prints its numerical jobid on standard output. The job consists of N copies of COMMAND launched together as a parallel job. If –ntasks is unspecified, a value of N=1 is assumed.
Useful options:

The most useful options are the same as for flux alloc.

flux job attach

Usage: flux job attach id
Description:
A job can be interactively attached to via flux job attach. This is typically used to watch stdout/stderr while a job is running or after it has completed.

flux run

Usage: flux run [OPTIONS] [–ntasks=N] COMMAND...
Description:
flux run submits a job to run interactively under Flux, blocking until the job has completed. It is equivalent to running flux submit followed by flux attach.

flux batch

Usage: flux batch [OPTIONS] SCRIPT ...
or: flux batch [OPTIONS] –wrap COMMAND ...
Description:
flux-batch submits SCRIPT to run as the initial program of a Flux subinstance. SCRIPT refers to a file that is copied at the time of submission. Once resources are allocated, SCRIPT executes on the first node of the allocation, with any remaining free arguments supplied as SCRIPT arguments. Once SCRIPT exits, the Flux subinstance exits and resources are released to the enclosing Flux instance. If there are no free arguments, the script is read from standard input. If the –wrap option is used, the script is created by wrapping the free arguments or standard input in a shell script prefixed with #!/bin/sh. If the job request is accepted, its jobid is printed on standard output and the command returns. The job runs when the Flux scheduler fulfills its resource allocation request. Flux commands that are run from the batch script refer to the subinstance. For example, flux-run would launch work there. A Flux command run from the script can be forced to refer to the enclosing instance by supplying the flux –parent option.
Useful options:
The most useful options are the same as for flux alloc.

flux jobs

Usage: flux jobs [OPTIONS] [JOBID...]
Description:
flux jobs is used to list jobs run under Flux. By default only pending and running jobs for the current user are listed.
Useful options:
-a List jobs in all states, including inactive jobs.
-A List jobs for all users.

flux cancel

Usage: flux cancel [OPTIONS] [JOBID...]
Description:
flux cancel cancels one or more jobs by raising a job exception of type=cancel. An optional message included with the cancel exception may be provided via the –message option. Canceled jobs are immediately sent SIGTERM followed by SIGKILL after a configurable timeout (default=5s). flux cancel can target multiple jobids by either taking them on the command line, or via the selection options –all, –user, or –states. It is an error to provide jobids on the command line and use one or more of the selection options.
Useful options:
-a, –all Cancel all jobs.
-u, –user=USER Cancel all jobs owned by USER.
-s, –states=STATES Set target job states (default: active). Valid states include depend, priority, sched, run, pending, running, active.

flux shutdown

Usage: flux shutdown [OPTIONS] [TARGET]
Description:
The flux shutdown command causes the default Flux instance, or the instance specified by TARGET, to exit RUN state and begin the process of shutting down. TARGET may be either a native Flux URI or a high level URI. Only the rank 0 broker in RUN state may be targeted for shutdown.

All available commands and their options can be found in the official flux-core documentation [17].
Another useful resource is the Flux Cheatsheet https://flux-framework.org/cheat-sheet/.

5.5 Flux and Slurm Comparison

Table 5.1: Flux and Slurm Feature Comparison [18]

Features	Flux	Slurm
In general		
Open Source	Yes	Yes
Multi-User Mode		
Multi-user workload management	Yes	Yes
Full hierarchical resource management	Yes	No
Graph-based advanced resource management	Yes	No
Scheduling specialization	Yes	No
Security: only a small isolated layer running in privileged mode for tighter security	Yes	No
Modern command-line interface (cli) design	Yes	Outdated
Application programming interface (APIs) for job management, job monitoring, resource monitoring, low-level messaging	Yes(4/4)	Some(3/4)
Language bindings	Yes (C, C++, Python, Lua, Rust, Julia, REST)	Some (C, REST)
Bulk job submission	Yes	No
High-speed streaming job submission	Yes	No
Single-User Mode		
User-level workload management instance	Yes	No
Support for nesting within foreign resource managers	Yes (Slurm, lsf, ...)	N/A
Fully hierarchical management of instances	Yes	N/A
Scheduler specialization for user level	Yes	N/A
Graph-based advanced scheduling for user level	Yes	N/A
Built-in facilities for inter-job communication and coordination	Yes	N/A
Modern command-line interface (cli) design	Yes	N/A
Support to launch message passing interface (MPI) jobs	Yes	N/A

Chapter 6

Flurm

In this Chapter the design and implementation of Flurm, the integration of the Flux framework into a Slurm managed cluster, is presented. The architecture of the system is described, along with the modifications made to Flux to enable its operation within a Slurm allocation. The chapter includes the challenges faced during the implementation and how they were addressed.

6.1 ARIS Cluster Overview

ARIS is the name of the Greek supercomputer, deployed and operated by GRNET S.A. (National Infrastructures for Research and Technology S.A.) in Athens [19]. When it was installed in 2015, it was included in the TOP500 list at position 468. ARIS consists of 532 computational nodes separated in four "islands" as listed here: All the nodes are connected via Infiniband

Table 6.1: ARIS Cluster Node Types [20]

Node Type	Count	Accelerator	Memory	Cores
THIN nodes	426	w/o	64 GB	20@2.8 GHz (two sockets)
GPU nodes	44	dual tesla k40m	64 GB	20@2.6 GHz + 2 x K40
PHI nodes	18	dual xeon phi 7120p	64 GB	20@2.6 GHz + 2 x 7120p
FAT nodes	44	w/o	512 GB	40@2.4 GHz (four sockets)
ML node	1	8 volta v100	512 GB	40@2.2 GHz (two sockets)

network and share 2PB GPFS storage. Access to the system is provided by two login nodes. There are two shared file systems on ARIS, HOME and WORKDIR . All login and compute nodes may access same data on shared file systems.
The cluster is managed by Slurm, which is used for job scheduling and resource management. Users can submit jobs to the cluster using the sbatch command, specifying the required resources and job parameters. Once a job is submitted, it is queued and scheduled for execution based on the available resources and the job's priority. For the purpose of this thesis, the user level integration of Flux into Slurm is implemented and tested on the THIN nodes of the cluster.

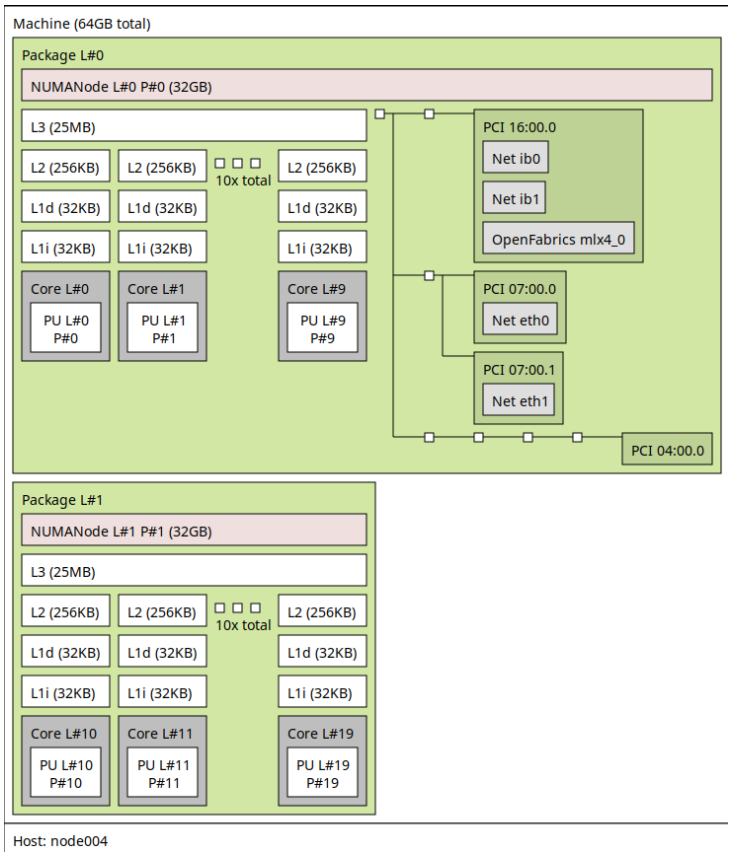


Figure 6.1: ARIS THIN Node Overview

6.2 Flux Installation

Installing Flux on ARIS with user level permissions is quite challenging due to the lack of dependencies and the age of the operating system (CentOS 7). To ensure a successful installation of Flux without interfering with the system's existing software, a specified package manager for user level installations in HPC systems named Spack [21] is used.

6.2.1 Spack Overview

Spack is a package manager for supercomputers, Linux, macOS, and Windows [21]. It makes installing scientific software easy. With Spack, you can build a package with multiple versions, configurations, and compilers, and all of these builds can coexist on the same machine. Spack isn't tied to a particular language; you can build a software stack consisting of Python or R packages, link to libraries written in C, C++, or Fortran, easily swap compilers, and target specific microarchitectures. Spack does not require administrator privileges to install packages. It can install software in any directory, making it easy to manage packages in the home directory or a shared project location without needing sudo access. Spack's core strength is creating highly customized, optimized software builds from source code. While it's primarily a from-source package manager, it also supports fast binary installations through build caches. It can be viewed as a virtual environment manager for scientific software, similar to venv with pip or conda for Python. Finally, Spack is written in Python, making it easy to extend and customize.

6.2.2 Flux Installation with Spack

To install Flux using Spack, the following steps are followed:

- Install Spack:
  The first step is to install Spack on the system. This can be done by cloning the Spack repository from GitHub.
- Load Modules for Running:
  Before using Spack, it's important to load the necessary modules for Spack to function correctly. The modules required for running Spack on ARIS are:
  - gnu/8
  - gnu/13.2.0
  - python/3.9.18
  - git

  Spack does not detect the loaded python module automatically, so it is specified with the environment variable SPACK_PYTHON. You can also set the environment variables SPACK_USER_CACHE_PATH, SPACK_USER_CONFIG_PATH to specify a custom location for Spack's cache and configuration files to prevent conflicts with other spack installations.
- Load Spack: Once the necessary modules are loaded, you need to load Spack. This can be done by sourcing the ‘spack‘ script in the ‘share‘ directory of your Spack installation.

- Prepare for Flux Installation:
  Before installing Flux, it's important to load the rust and zlib modules, which are dependencies for Flux. We also need to hint Spack to use those external modules we loaded, instead of trying to build them from source.

- Install Flux:
  Finally, we can install Flux using Spack. This can be done by running the command 'spack install flux-sched'. This command will download, build, and install flux-core, flux-sched and all their dependencies.

- Load Flux:
  After the installation is complete, we can load Flux by running the command 'spack load flux-sched'.

However, Spack in ARIS had a few more issues that needed to be addressed before the installation of Flux could be completed successfully.

- **Trouble fetching packages**:
  Due to ARIS using an old version of curl, Spack was unable to fetch packages from https sources. To resolve this, we disabled SSL verification by setting the spack configuration option 'verify_ssl false'.

- **Trouble recognizing rust**:
  Spack was unable to recognize the rust module, which resulted in Spack trying to build rust from source. To resolve this, we specified rust path found by spack as unbuildable.

- **Trouble with library paths of some modules, LDFLAGS and python forward compatibility**:
  Python and gnu libraries were not found by Spack during the installation of Flux. To resolve this, we specified the library paths of those modules for each executable run by Spack by changing the environment variable LD_LIBRARY_PATH in the executable.py file in spack's source code. We also set the environment variable LDFLAGS which was not set by Spack for some builds. Finally, we set the environment variable PYO3_USE_ABI3_FORWARD_COMPATIBILITY to 1 to resolve issues with building python packages required for building flux-sched.

- **flux-core and flux-sched latest versions**:
  The latest versions of flux-core and flux-sched were not available in Spack's package repository. To resolve this, we modified the package.py files for flux-core and flux-sched to include the latest versions from the official Flux GitHub repository. This was sufficient to install and build them successfully without any conflicts or issues.

6.3 Flurm Overview

6.3.1 Essential Workflow

The Flurm workflow in its essence consists of the following steps:

- A user submits a job to Slurm using the sbatch command, specifying exclusive access to a set of nodes that will be the nodes of the Flux managed cluster.

- Inside the Slurm job script, the user loads the Flux module using Spack.
- The user then starts a Flux instance using the flux start command within a srun command, which ensures that the Flux instance is started on the allocated nodes and will be terminated when the Slurm job ends, leaving no ghost processes. We need to specify –mpi=pmi2 to ensure that flux will use the PMI2 interface provided by Slurm for inter-process communication.
- flux start need to be run with a command as an argument in order to be executed non-interactively. The basic use case is to run flux start with flux run PROGRAM as the command argument.
- We need to ensure that the flux broker with rank 0 is purely a management node and does not run any tasks. This is achieved by using the ‘–requires=not hostlist <broker_node>‘ option of flux run.

In order for the above workflow to work, a few modifications to Flux are necessary in order to launch properly:

- **Flux certificate signature fails**:
  When starting a Flux instance with flux start, the flux broker fails to start and stalls without any error message. This is due to the fact that for the signing a certificate, Flux uses a library (libsodium) that relies on the /dev/random device for generating random numbers, which depends on the system's entropy pool. However, the thin nodes of ARIS have almost no entropy available, which causes the certificate signing process to block indefinitely. To resolve this, we modified the flux broker to use /dev/urandom instead of /dev/random for generating random numbers. The /dev/urandom device does not block when the entropy pool is low, making it more suitable for environments with limited entropy. A custom .so file was created to override the default behavior of the open system call to redirect any attempts to open /dev/random to /dev/urandom inside the slurm and flux processes.

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <stdarg.h>
#include <unistd.h>

int open(const char *pathname, int flags, ...) {
    static int (*real_open)(const char *, int, ...) = NULL;
    if (!real_open)
        real_open = dlsym(RTLD_NEXT, "open");

    if (strcmp(pathname, "/dev/random") == 0)
        pathname = "/dev/urandom";

    va_list args;
    va_start(args, flags);
    int fd = real_open(pathname, flags, args);
    va_end(args);
    return fd;
}
```

- **Temp file creation fails**:
  When a flux instance is setting up, a temporary file is created in order to cleanup any jobs or queues that may be lingering from a previous instance. The semantics of the function used by Flux to create the temporary file are not compatible with ARIS's shell, so flux fails with an error message. This temporary file creation is not essential for the operation of Flux under Slurm and it is only used when a environment variable ('FLUX_DISABLE_JOB_CLEANUP') is not set. We set this environment variable to 1 and flux finally starts without any issues.

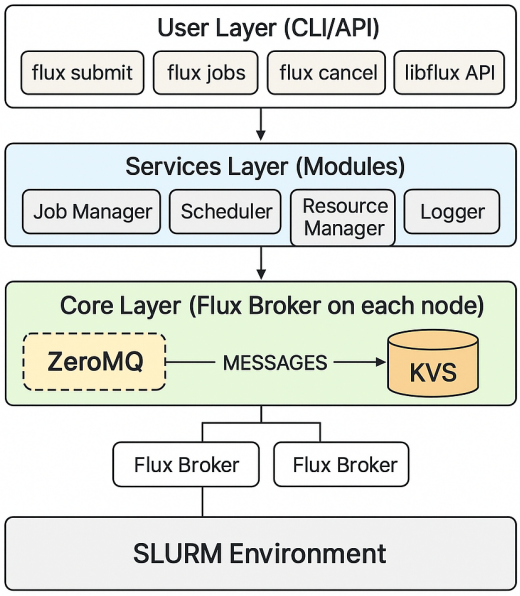


Figure 6.2: Flurm Architecture/Workflow Overview

6.3.2 Colocation and Co-Scheduling

Colocation

Colocation is the practice of running multiple applications or workloads on the same nodes. The resource manager should be able to assign different number of processors to each application leading to more nodes being used by applications than expected. This can be beneficial in scenarios where applications have complementary resource usage patterns, allowing for better overall resource utilization. For example, a CPU-intensive application can be colocated with a memory-intensive application, as they will not compete for the same resources. Colocation can also help reduce the overhead of context switching and improve cache utilization, leading to better performance for both applications.
The most relevant resource allocation techniques are the following:

- **Compact Allocation**:
  In this technique, the resource manager allocates resources in a way that minimizes the number of nodes used by the application. This can be achieved by filling up nodes completely before moving on to the next node.
- **Spread Allocation**:
  Spread allocation does compact allocation with half sockets. In other words, at each node, only half of the socket's cores are allocated before moving to the next one.
- **Job Striping**:
  When spread allocation is used, half of the cores of each socket remain unused. Job striping is when we allocate those unused cores to other applications.

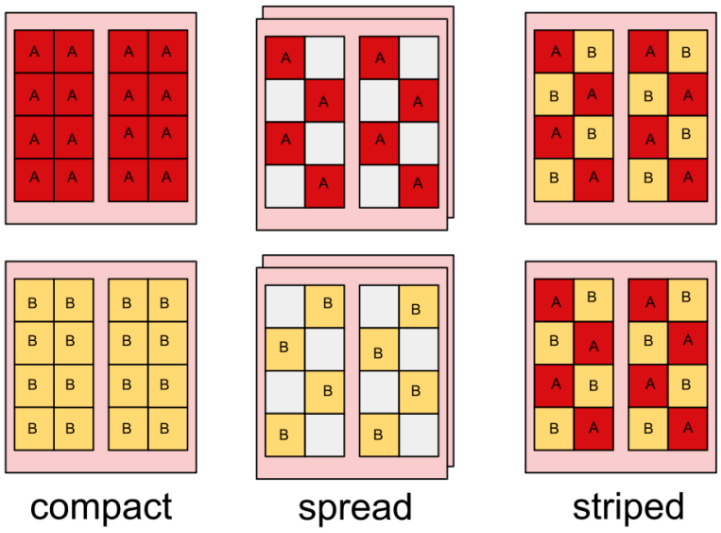


Figure 6.3: Resource Allocation Techniques [22]

Co-Scheduling

Co-scheduling is the practice of scheduling with colocation in mind. All co-scheduling algorithms essentially decide which applications to colocate together based on a variety of criteria. Some of them also decide on whether to colocate an application or not and which resource allocation technique to use. For the purpose of this thesis, we don't implement any co-scheduling

algorithms, so co-scheduling in Flurm currently is theoretically random. However, we provide the necessary infrastructure to support them in the future.

6.3.3 Colocation Support in Flux

Jobspec of Different Allocation types

First, we need to figure out the shape of the jobspec for each allocation type. **Compact Allocation** The jobspec for compact allocation is quite straightforward. For the resource shape, we request as many slots as the tasks the user wants to run, with each slot having one core.

```
resources:
  - type: slot
    count: NTASKS
    label: tasks
    with:
      - type: core
        count: 1
```

For the tasks, we request one task per slot.

```
tasks:
  - command: [ "app" ]
    slot: tasks
    count:
      per_slot: 1
```

We also need to specify that no two tasks can be placed on the same core during task assignment. This can be achieved by setting the attribute ‘cpu-affinity‘ to ‘per-task‘.

```
attributes:
  system:
    duration: 3600
    shell:
      options:
        cpu-affinity: "per-task"
```

Spread Allocation For spread allocation, the resource shape needs to request the number of sockets required to fit all the tasks, each socket having half the cores of a full socket. ARIS thin nodes, each socket has 10 cores, so we request $\lceil NTASKS/5 \rceil$ sockets, each with 5 cores.

```
resources:
  - type: socket
    count: CEIL(NTASKS/5)
    with:
      - type: slot
        count: 5
        label: tasks
        with:
          - type: core
            count: 1
```

Slots are placed right below sockets to make sure that sockets can be shared between multiple jobs, making Job Striping possible. For the tasks key, we cannot request one task per slot, as that would lead $\lceil NTASKS/5 \rceil \times 5$ tasks being created, which is more than the user requested in the general case. Instead, we request a total count of NTASKS tasks and let Flux assign them to the slots.

```
tasks:
    - command: [ "app" ]
      slot: tasks
      count:
        total: NTASKS
```

The attributes key is the same as in compact allocation.

Job Submission

In order to submit a custom jobspec, we can use the Python API provided by Flux. The Jobspec class can load a jobspec directly from a yaml file like so:

```
jobspec = flux.job.Jobspec()
jobspec.from_yaml_file("custom_jobspec.yaml")
```

We can submit the jobspec via flux.job.submit. We wait for the results via flux.job.wait. Finally we can print the results using the cli command flux job attach <jobid>. Python scripts containing flux api calls can be run via flux python script.py.

Challenges

Even though Flux provides the necessary infrastructure to support the workflow described above, there are still some challenges in order to make it work properly, due to flux still being under active development.

The flux-core included job-ingest module does not accept resource types other than slots, cores, gpus and nodes by default. We can disable the jobspec validation done by this module in the flux config file, which describes how the flux instance of Flurm is setup at launch.

However, there are still some issues with Fluxion not discovering resource types other than the default ones. To overcome this, the resource graph of our Flurm cluster should be specified to Fluxion for it to allocate non default resources properly. This can be done by generating a resource graph file in the JSON Graph Format (JGF) that fluxion can read to model the resources of the cluster. This resource graph file is an extension to the default resource specification file (R file) of flux-core that should also be specified in the flux config file. The R and JGF files are dependent on the allocation of the slurm user, so we generate them dynamically at the start of each slurm job some bash commands and a python script accordingly. We use the information provided by slurm about the hostnames of the allocated nodes.

6.3.4 Use cases for colocation and co-scheduling

Use case 1: Job-tagged based resource allocation

In this use case, the Flurm user wants to specify the allocation type to be spread or compact for a job based on a tag. This will provide a more user friendly interface for testing and using different allocation types.
Solution:
We wrote a flux cli plugin in python that will tranform the user's jobspec based on the tag specified.

- The user will submit a job specifying only the number of tasks (No node number) and a tag for the allocation type. This way the user request is well defined and the jobspec transformation will make sense.
- We introduce a new option to the flux submit family of commands named –alloc-type with possible values 'compact' and 'spread'.
- The plugin will be a part of the submission process and will transform the jobspec accordingly before submission if the option is specified.
- Introduce the plugin to flux via the environment variable FLUX_CLI_PLUGINPATH.

Example usage:
Inside the slurm job script, the user will run:
flux start –config=path/to/config flux run -n 42 –alloc-type=spread /path/to/app

Use case 2: Queue-tagged based resource allocation

This is a use case where the user specifies to run a job in a queue specified for co-scheduling. The queue will function exactly like a slurm partition, providing an environment for co-scheduled only jobs. This use case should be mutually exclusive with the previous one, as the user should not be able to specify both a tag and the co-scheduling queue.
Solution:
First, we need to create two queues in the flux config file, one for co-scheduled jobs and one for normal jobs. Each queue will be linked with a subset of nodes in the cluster using a label called 'properties', as mentioned in the flux documentation. The label of each partition will be the same as their queue name for simplicity. Additionally, a new flux cli plugin will be created just like in the previous use case.

- The user will submit a job specifying only the number of tasks (No node number) and a tag for the co-scheduling queue.
- A new option –cosched equivalent –queue=cosched for ease of use in the plugin.
- The plugin will do the jobspec transformation to spread allocation and set the queue to cosched if the –cosched/–queue=cosched option is specified.

Example usage:
flux start –config=path/to/config flux run -n 42 –cosched /path/to/app
or:
flux start –config=path/to/config flux run -n 42 –queue=cosched /path/to/app

Workflow Diagram

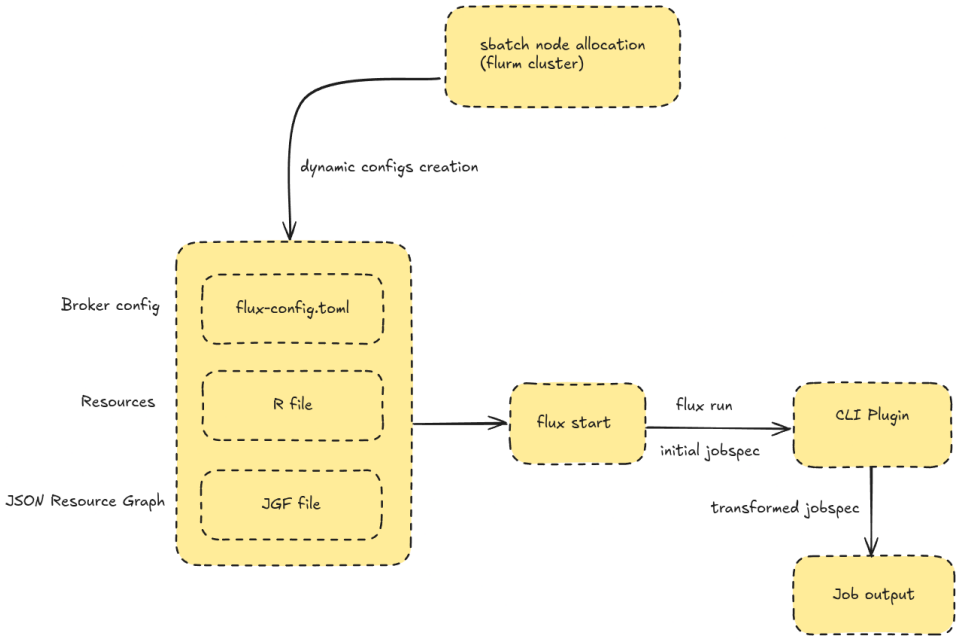


Figure 6.4: Flurm Workflow Diagram in both use cases

6.4 Dockerized Environment

It is important to mention that a dockerized environment was created to facilitate the early development and testing process of new features and modifications to Flux. The environment consists of a Slurm cluster with one controller node and three compute nodes, all running on Rocky Linux 8. The dockerized environment allows for quick iteration and testing of changes to Flux without the need to deploy them on the actual ARIS cluster or wait in a job queue. It includes the installation of Spack and Flux, as well as the necessary configurations to run Flux within a Slurm allocation. This same environment can be used for future development and testing of scheduling policies and other features in Flurm.

6.5 Source Code

The source code of Flurm can be found in the following GitHub repository: https://github.com/cslab-ntua/flurm The repository includes:

- A dockerized Slurm cluster for development and testing.
- Scripts for installing Spack and Flux in ARIS.
- The flux cli plugins for the use cases described above.
- Scripts for generating the R and JGF files dynamically.

Chapter 7

Evaluation

In this chapter, we present the evaluation methodology and results for the Flurm system integrated into the ARIS HPC cluster. We used flux-core version 0.78 and flux-sched version 0.47 for our experiments.

7.1 NAS Parallel Benchmarks (NPB)

The NAS Parallel Benchmarks (NPB) [23] are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications. Problem sizes in NPB are predefined and indicated as different classes. Reference implementations of NPB are available in commonly-used programming models like MPI and OpenMP. For the experiments in this thesis, we used the MPI implementation of NPB version 3.4.3.

Benchmark Specifications

The NPB suite consists of the following benchmarks:

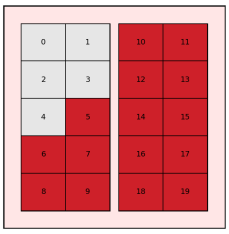
- Five kernels
  - IS - Integer Sort, random memory access
  - EP - Embarrassingly Parallel
  - CG - Conjugate Gradient, irregular memory access and communication
  - MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
  - FT - discrete 3D fast Fourier Transform, all-to-all communication
- Three pseudo applications
  - BT - Block Tri-diagonal solver
  - SP - Scalar Penta-diagonal solver
  - LU - Lower-Upper Gauss-Seidel solver

Benchmark Classes

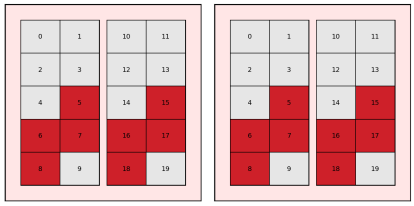
- Class S: small for quick test purposes
- Class W: workstation size (a 90's workstation; now likely too small)
- Classes A, B, C: standard test problems; 4X size increase going from one class to the next
- Classes D, E, F: large test problems; 16X size increase from each of the previous classes

7.2 Functionality Check

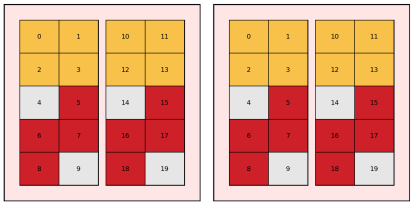
To verify the correct functionality of Flurm, we ran some NPB benchmarks for all allocation types supported by Flurm (compact, spread). We used the -o verbose=2 flag of flux to get detailed information about the cpu pinning of each task launched by flux.
As we can see from the following examples for the EP benchmark, the tasks are correctly pinned according to the allocation type:



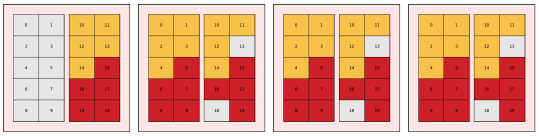
(a) Compact Allocation for 16 tasks



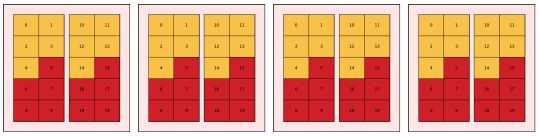
(b) Spread Allocation for 16 tasks



(c) Striped Allocation for 16 tasks per job



(d) Striped Allocation for 32 tasks per job



(e) Striped Allocation for 40 tasks per job

For Job Striping we submitted two same jobs simultaneously which is good enough to verify that tasks do not overlap on the same cores.

7.3 Performance Check

For the performance evaluation of Flurm, we need to compare the execution time of benchmarks running under Flurm with the execution time of the same benchmarks running directly under Slurm. We chose the following benchmarks:

- CG class D with 64 tasks, because of its irregularity in memory access and communication
- EP class E with 256 tasks, for a large task count and minimal communication
- FT class D with 256 tasks, for intense communication between a lot of tasks
- MG class E with 128 tasks, for its memory intensity

7.3.1 Spread Allocation Setup

First, we ran each benchmark alone in spread allocation mode under both Slurm and Flurm to get their baseline execution times.
Each benchmark ran repeatedly for 10 minutes, with any completed job being restarted. We recorded the median of the execution times for each benchmark and that was the time reported. The results are as follows:

Table 7.1: Slurm Spread Results

Benchmark	Tasks	Spread Time(s)
CG class D	64	149.70
EP class E	256	144.18
FT class D	256	37.68
MG class E	128	88.50

Table 7.2: Flurm Spread Results

Benchmark	Tasks	Spread Time(s)
CG class D	64	144.59
EP class E	256	144.13
FT class D	256	35.38
MG class E	128	88.05

As we can see, the execution times for each benchmark running alone under both Slurm and Flurm are very close to each other, with Flurm being slightly faster or equally fast in all cases.
A notable observation is that the FT benchmark shows a more significant performance improvement in Flurm compared to the other benchmarks, approximately 6% faster. However, to more accurately evaluate the performance difference between Flurm and Slurm, we need to run Flurm with the same task mapping as Slurm. To achieve this, we used the –taskmap option of flux to specify that we want 10 tasks per node except for the last node which will have the remaining tasks.

Table 7.3: Flurm Spread Taskmap Results

Benchmark	Tasks	Spread Time(s)
CG class D	64	146.10
EP class E	256	144.22
FT class D	256	36.64
MG class E	128	88.05

We can tell that the execution times are even closer, but still the FT benchmark shows a improvement over Slurm of around 2.7%. All clues point to Flurm having a minimal overhead, if any, when compared to Slurm.

7.3.2 Co-Execution Setup

In order to fully evaluate the performance of Flurm, we tried the co-execution scenarios of all pairs of the selected benchmarks. For each co-execution scenario, we paired the applications on the same nodes using spread allocation under both Slurm and Flurm. Just like before, each pair ran together for 10 minutes, with any completed job being restarted. We recorded the median execution times from these repeated runs as the co-execution time for the benchmark.

Slurm Results

The following table presents the results we obtained when running the benchmarks directly under Slurm:

Table 7.4: Slurm Co-Execution Results

Benchmark A	Tasks	Co-Execution Time(s)	Benchmark B	Tasks	Co-Execution Time(s)
CG class D	64	191.33	CG class D	64	191.66
CG class D	64	155.055	EP class E	256	149.595
CG class D	64	181.755	FT class D	256	42.305
CG class D	64	267.855	MG class E	128	105.77
EP class E	256	144.21	EP class E	256	144.205
EP class E	256	148.675	FT class D	256	38.77
EP class E	256	154.31	MG class E	128	96.475
FT class D	256	53.06	FT class D	256	53.01
FT class D	256	48.895	MG class E	128	107.835
MG class E	128	156.19	MG class E	128	156.49

Flurm Results

The results obtained when running the same benchmarks under Flurm (default task mapping):

Table 7.5: Flurm Co-Execution Results

Benchmark A	Tasks	Co-Execution Time(s)	Benchmark B	Tasks	Co-Execution Time(s)
CG class D	64	186.55	CG class D	64	186.58
CG class D	64	162.9	EP class E	256	145.515
CG class D	64	179.37	FT class D	256	39.31
CG class D	64	266.87	MG class E	128	105.54
EP class E	256	145.56	EP class E	256	144.11
EP class E	256	145.075	FT class D	256	36.46
EP class E	256	146.805	MG class E	128	97.29
FT class D	256	39.96	FT class D	256	39.95
FT class D	256	45.04	MG class E	128	108.015
MG class E	128	155.18	MG class E	128	157.78

And here are the results for Flurm with the same task mapping as Slurm:

Table 7.6: Flurm Co-Execution Taskmap Results

Benchmark A	Tasks	Co-Execution Time(s)	Benchmark B	Tasks	Co-Execution Time(s)
CG class D	64	188.70	CG class D	64	189.10
CG class D	64	168.69	EP class E	256	146.61
CG class D	64	176.39	FT class D	256	39.235
CG class D	64	264.54	MG class E	128	106.12
EP class E	256	145.515	EP class E	256	144.085
EP class E	256	147.34	FT class D	256	37.97
EP class E	256	148.145	MG class E	128	96.445
FT class D	256	42.615	FT class D	256	42.60
FT class D	256	47.085	MG class E	128	106.43
MG class E	128	155.54	MG class E	128	156.27

Results Analysis

In order to evaluate the time difference between the co-execution times we will use the Mean Absolute Percentage Error (MAPE) metric. MAPE is calculated using the formula:

$$MAPE = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{A_i - F_i}{A_i} \right| \times 100 \tag{7.1}$$

Where:

- A_i is the actual value (execution time under Slurm)
- F_i is the forecasted value (execution time under Flurm)
- n is the number of observations

This metric provides an average of the absolute percentage errors between the two sets of execution times, giving us a clear indication of the performance overhead introduced by Flurm. Calculating the MAPE for our results, we get a value of approximately 4.793% for the default taskmap and 3.907% for the taskmap that imitates Slurm.
That deviation is quite small but not insignificant. We need to further examine the individual benchmark time difference percentages. We choose to calculate the individual time difference percentages, we can call them Percentage Errors (PE), using the formula:

$$\text{Time Difference Percentage} = \left(\frac{F_i - A_i}{A_i}\right) \times 100 \tag{7.2}$$

So that we can see whether Flurm introduces a performance overhead (positive percentage) or a performance improvement (negative percentage) for each benchmark run.
The individual time difference percentages for each benchmark run are as follows:

Table 7.7: Individual Time Difference Percentages (Default Taskmap)

Benchmark A	Tasks	PE(%)	Benchmark B	Tasks	PE(%)
CG class D	64	-2.50	CG class D	64	-2.65
CG class D	64	5.06	EP class E	256	-2.73
CG class D	64	-1.31	FT class D	256	-7.08
CG class D	64	-0.37	MG class E	128	-0.22
EP class E	256	0.94	EP class E	256	-0.07
EP class E	256	-2.42	FT class D	256	-5.96
EP class E	256	-4.86	MG class E	128	0.84
FT class D	256	-24.69	FT class D	256	-24.64
FT class D	256	-7.88	MG class E	128	0.17
MG class E	128	-0.65	MG class E	128	0.82

Table 7.8: Individual Time Difference Percentages (Taskmap that imitates Slurm)

Benchmark A	Tasks	PE(%)	Benchmark B	Tasks	PE(%)
CG class D	64	-1.375	CG class D	64	-1.336
CG class D	64	8.794	EP class E	256	-2
CG class D	64	-2.952	FT class D	256	-7.257
CG class D	64	-1.238	MG class E	128	0.331
EP class E	256	0.905	EP class E	256	-0.083
EP class E	256	-0.9	FT class D	256	-2.063
EP class E	256	-4	MG class E	128	-0.031
FT class D	256	-19.69	FT class D	256	-19.638
FT class D	256	-3.702	MG class E	128	-1.303
MG class E	128	-0.416	MG class E	128	-0.141

From the above tables, we can see that in most cases, Flurm is really close the Slurm execution times in general. In some cases though, especially for the FT benchmark, Flurm deviates significantly from Slurm, showing a performance improvement of around 25% in default taskmap and 19.7% in the taskmap that imitates Slurm. A more detailed investigation is needed to understand the reasons behind these deviations, however it is clear that Flurm does a great job in keeping the performance overhead minimal and in some cases even improving the performance compared to Slurm.

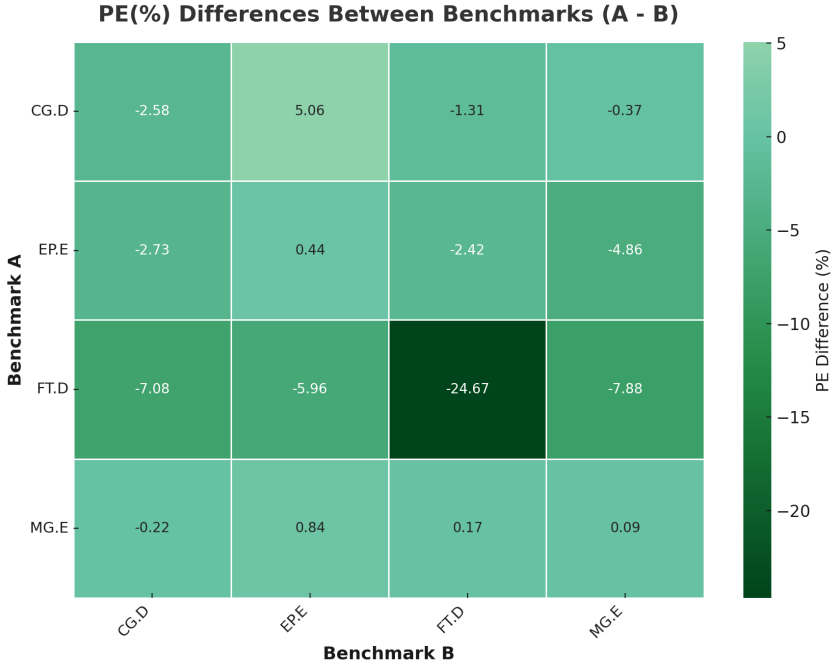


Figure 7.2: Co-Execution PE Heatmap (Default Taskmap)

Chapter 8

Summary and Future Work

8.1 Summary

This thesis presented Flurm, an integration of the Flux resource management framework within a Slurm-managed HPC cluster at the user level, which enables users to experiment and develop custom scheduling strategies without requiring system administrator changes. The primary focus of this work was to establish the foundation for Flurm, demonstrating how Flux can offer colocation and co-scheduling capabilities, meaning it supports resource sharing.

The first part of creating Flurm involved adapting Flux to operate within a Slurm allocation. This adaptation required configuring the system's environment and the package manager for Flux's proper installation and operation.

Next, Flux was customized to provide the fine-grained resource control that enables co-scheduling and colocation. This customization involved extending Flux's view of the system's hardware resources to include sockets. Additionally, a flux cli plugin was developed to allow for users to either specify a resource allocation type using a job tag or specify to submit a job tagged to use a queue assigned to co-schedule jobs.

Finally, a series of experiments were conducted to evaluate the correctness/functionality and performance of Flurm. It was shown that Flurm can successfully co-schedule and colocate jobs and that the performance overhead introduced by Flurm is negligible.

In conclusion, this work establishes Flurm as a viable solution for co-scheduling research and experimentation within a Slurm-managed HPC environment without disturbing existing workflows or requiring system administrator changes.

8.2 Future Work

While this thesis laid the groundwork for Flurm, several avenues for future work remain to be explored:

Implementing a Co-scheduling Policy

The current implementation of Flurm allows for resource sharing but does not implement a specific (non-random) co-scheduling policy. Future work could involve developing and integrating a co-scheduling algorithm within Fluxion. As part of the Flux exploration work associated with this diploma, we investigated how could one implement a scheduling policy in Fluxion. The most viable approach we found was to modify the Fluxion source code to include a new scheduling policy. Under the flux-sched/qmanager/policies directory, there are several existing scheduling policies implemented as C++ classes. The base class with comments describing the required methods to implement a new policy is in the flux-sched/qmanager/policies/base directory. The new policy will need to be presented to the Fluxion scheduler by a factory class located in the flux-sched/qmanager/policies/queue_policy_factory.hpp file, which maps policy names to their corresponding classes.

Exploration of Different Resource Allocation Policies

The current experiments were limited to the default resource allocation policy provided by Fluxion, which is ”first”, meaning that the first resource match fitting for a job request is allocated for it. An interesting direction for future work would be to explore how different resource allocation policies impact the performance of co-scheduled and colocated jobs. Fluxion supports several resource allocation policies, such high node id first and low node id first. Limited customization is also possible with combinations of different criteria, as the ones mentioned above. Even more interesting would be to implement a custom resource allocation policy that takes into account the specific requirements of co-scheduled and colocated jobs. The development process would be similar to the one described in the previous subsection for implementing a co-scheduling policy.

Preemption in Flux/Flurm

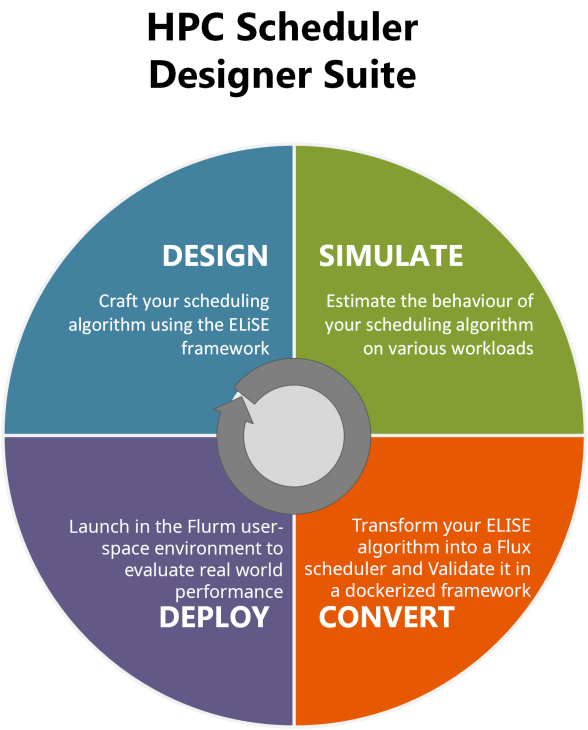
Preemption is a feature that allows higher priority jobs to interrupt and take over resources from lower priority jobs. Implementing preemption in Flux/Flurm and evaluating its impact on scheduling performance and resource utilization would be a valuable area for future research.

Εκτεταμένη Ελληνική Περίληψη

1 Εισαγωγή

Τα clusters υψηλής απόδοσης (HPC) είναι σύνθετα κοινόχρηστα συστήματα όπου η αποτελεσματική κατανομή πόρων και ο προγραμματισμός εργασιών είναι κρίσιμης σημασίας για τη μεγιστοποίηση της απόδοσης, της αξιοποίησης και της ισότητας. Καθώς τα φορτία εργασίας HPC γίνονται όλο και πιο ετερογενή — συνδυάζοντας προσομοιώσεις, ανάλυση δεδομένων και στοιχεία AI/ML — οι παραδοσιακές πολιτικές προγραμματισμού είναι μερικές φορές ανεπαρκείς για την πλήρη αξιοποίηση της απόδοσης του σύγχρονου υλικού. Αυτές οι ανησυχίες παρακινούν την εξερεύνηση προηγμένων τεχνικών προγραμματισμού, όπως η συν-τοποθέτηση και ο συν-προγραμματισμός, καθώς και την ανάγκη για πειραματισμό με αυτές σε πραγματικά περιβάλλοντα HPC. Ωστόσο, τα πραγματικά clusters HPC συχνά διαχειρίζονται από καθιερωμένους διαχειριστές πόρων όπως το Slurm, το οποίο ενδέχεται να μην υποστηρίζει εγγενώς τέτοιες προηγμένες λειτουργίες προγραμματισμού ή να απαιτεί αλλαγές διαμόρφωσης από τους διαχειριστές συστημάτων. Αυτό δημιουργεί ένα εμπόδιο για τους ερευνητές και τους χρήστες που επιθυμούν να πειραματιστούν και να αξιολογήσουν νέες στρατηγικές προγραμματισμού χωρίς να διαταράξουν τις υπάρχουσες ροές εργασίας. Για να αντιμετωπιστεί αυτό, μια πολλά υποσχόμενη προσέγγιση είναι η ενσωμάτωση ενός άλλου πλαισίου διαχείρισης πόρων στο πλαίσιο ενός υπάρχοντος cluster που διαχειρίζεται από το Slurm. Αυτή η διπλωματική εργασία εξερευνά αυτή την προσέγγιση ενσωματώνοντας το πλαίσιο Flux στο πραγματικό cluster Slurm ARIS στο GRNET. Αυτή η ενσωμάτωση, που αναφέρεται ως Flurm, δίνει τη δυνατότητα στους τελικούς χρήστες του cluster να πειραματιστούν με προηγμένο προγραμματισμό πόρων (συν-προγραμματισμός, συν-τοποθέτηση, δυναμικό overlay) εντός των δικών τους κατανομών, χωρίς να απαιτούνται αλλαγές από τον διαχειριστή συστήματος, το θεμέλιο της οποίας παρουσιάζεται σε αυτή τη δουλειά. Τέλος, διεξάγεται μια σειρά πειραμάτων για την αξιολόγηση της επιβάρυνσης που εισάγεται από την προτεινόμενη λύση, για να αποδειχθεί η ορθότητά της και να επιδειχθεί η κλιμακωσιμότητά της. Ο στόχος αυτής της διπλωματικής είναι να καταστήσει δυνατή μια ροή εργασίας όπου οι χρήστες μπορούν:

- Να αναπτύξουν αλγόριθμους προγραμματισμού τοπικά χρησιμοποιώντας κάποιο εργαλείο προσομοίωσης όπως το ELiSE [1].
- Να γράψουν αυτούς τους αλγόριθμους στο Flux και να τους δοκιμάσουν σε ένα ελεγχόμενο περιβάλλον dockerized.
- Να χρησιμοποιήσουν το Flurm για να μεταβούν απρόσκοπτα από τις τοπικές δοκιμές στην πραγματική εκτέλεση του cluster για αξιολόγηση.



Σχήμα 8.1: Επισκόπηση Ροής Εργασίας Flurm

2 Υπολογιστικά Συστήματα Υψηλής Απόδοσης (HPC)

Η υπολογιστική υψηλής απόδοσης (HPC) έχει εξελιχθεί από τις πρώτες, περιορισμένες μηχανές σε συστήματα μεγάλης κλίμακας ικανά να εκτελούν φορτία εργασίας exascale κλίμακας που υπερβαίνουν τα 10^{18} πράξεις κινητής υποδιαστολής ανά δευτερόλεπτο. Οι σύγχρονοι υπερυπολογιστές όπως ο El Capitan ξεπερνούν τα 1,7 exaFLOPS, και η συνδυασμένη απόδοση των συστημάτων TOP500 έχει υπερδιπλασιαστεί — από 5 exaFLOPS το 2023 σε πάνω από 11 exaFLOPS μέχρι το 2024. Αυτή η ανάπτυξη προέρχεται από την πρόοδο στους επεξεργαστές, τη μνήμη, τις διασυνδέσεις, τους επιταχυντές και το λογισμικό, με 83 από τα 100 κορυφαία συστήματα να ενσωματώνουν πλέον GPU ή παρόμοιους επιταχυντές.

Το HPC έχει γίνει ένα θεμελιώδες εργαλείο γιά τις φυσικές επιστήμες, τους μηχανικούς και τις ανακαλύψεις βασισμένες σε δεδομένα. Συμπληρώνει τα πειράματα και τη θεωρητική μοντελοποίηση, επιτρέποντας μεγάλες προσομοιώσεις, ανάλυση δεδομένων σε πραγματικό χρόνο και αλγορίθμους πρόβλεψης σε κλίμακες που διαφορετικά θα ήταν ανέφικτες.

Αρχιτεκτονικά, τα συστήματα HPC οργανώνονται σύμφωνα με την ταξινόμηση του Flynn, η οποία κατατάσσει τους υπολογιστές με βάση τον παραλληλισμό εντολών και δεδομένων στις κατηγορίες SISD, SIMD, MISD και MIMD, με την τελευταία να κυριαρχεί στους σύγχρονους υπερυπολογιστές. Η κυρίαρχη μορφή HPC σήμερα είναι το σύστημα πολυεπεξεργαστών, ένα δίκτυο επεξεργαστών που συνεργάζονται σε ένα ενιαίο φόρτο εργασίας. Αυτά μπορούν να έχουν τρεις βασικές διαμορφώσεις:

- **Πολυεπεξεργαστές Κοινής Μνήμης**: όλοι οι επεξεργαστές έχουν πρόσβαση σε μια κοινή μνήμη (είτε Uniform Memory Access — UMA/SMP είτε Non-Uniform Memory Access — NUMA).

- **Επεξεργαστές Μαζικής Παράλληλης Επεξεργασίας (MPP)**: χρησιμοποιούν κατανεμημένη μνήμη σε πολλούς κόμβους για επεκτασιμότητα και αποδοτικότητα.
- **Συστοιχίες υπολογιστών (Clusters / Commodity clusters)**: αποτελούνται εξ ολοκλήρου από έτοιμα προς χρήση εξαρτήματα, προσφέρουν οικονομικά αποδοτική επεκτασιμότητα και αντιπροσωπεύουν σήμερα πάνω από το 80 % όλων των συστημάτων της λίστας TOP500.

Συνολικά, αυτές οι αρχιτεκτονικές απεικονίζουν την εξέλιξη του HPC από εξειδικευμένες μηχανές σε ποικίλες, επεκτάσιμες πλατφόρμες που υποστηρίζουν τη σύγχρονη υπολογιστική επιστήμη και μηχανική.

3 Διαχείριση πόρων σε συστήματα HPC

Η βέλτιστη χρησιμοποίηση των πόρων σε συστήματα HPC αποτελεί κρίσιμο παράγοντα για την επίτευξη υψηλής απόδοσης. Οι διαχειριστές πόρων παίζουν κεντρικό ρόλο για το πως κατανέμονται οι υπολογιστικοί πόροι σε εφαρμογές χρηστών.

3.1 Κατανομή Πόρων

Η κατανομή αυτή αποτελεί την αντιστοίχιση υποσυνόλου των διαθέσιμων πόρων του συστήματος σε συγκεκριμένες εργασίες χρηστών βασισμένη στις απαιτήσεις τους. Τα βασικά είδη πόρων που αναγνωρίζουν οι διαχειριστές πόρων περιλαμβάνουν:

- **Υπολογιστικοί Κόμβοι (Compute Nodes)**
- **Επεξεργαστικοί Πυρήνες (Processing Cores)**
- **Διασύνδεση (Interconnect)**
- **Μέσα αποθήκευσης (Permanent Storage)**
- **Επιταχυντές (Accelerators)**

3.2 Δρομολόγηση Φόρτου Εργασίας (Workload Scheduling)

Εργασίες (Jobs)
Μια εργασία αποτελεί ένα αυτόνομο μερίδιο δουλειάς το οποίο σχετίζεται με μία είσοδο και κατά την εκτέλεσή της παράγει μία έξοδο. Οι εργασίες μπορούν να εκτελούνται διαδραστικά (interactively), κατά την οποία είναι δυνατή η συμμετοχή του χρήστη μέσω της κονσόλας, για την παροχή επιπλέον εισόδου στην εφαρμογή κατά την εκτέλεση, ή να επεξεργάζονται μαζικά (batch processing) όπου όλη η απαραίτητη πληροφορία για την ολοκλήρωση της εργασίας είναι διαθέσιμη κατά την υποβολή της. Το batch processing παρέχει μεγαλύτερη ευελιξία στα συστήματα διαχείρισης πόρων, καθώς μπορεί να αποφασίσει πότε είναι η ιδανική στιγμή να ξεκινήσει η κάθε εργασία σύμφωνα με την κατάσταση του cluster. Μία εργασία μπορεί να διαιρείται σε μικρότερα κομμάτια, τα tasks. Συνήθως, κάθε task σχετίζεται με την εκτέλεση ενός συγκεκριμένου προγράμματος. Σε γενικές γραμμές, δεν είναι απαραίτητο τα επιμέρους tasks μιας εργασίας να έχουν κοινά χαρακτηριστικά όσον αφορά χρησιμοποιούμενους πόρους και διάρκεια εκτέλεσης.
Ουρές εργασιών (Queues)

Οι εκκρεμείς εργασίες υπολογιστών αποθηκεύονται σε ουρές εργασιών που καθορίζουν τη σειρά με την οποία οι εργασίες επιλέγονται από τον διαχειριστή πόρων για εκτέλεση. Στα περισσότερα συστήματα διαχείρισης πόρων, η επιλογή γίνεται με την FIFO (First In First Out) πολιτική, όμως χαλαρώνουν αυτό το σχήμα για να αυξήσουν τη χρήση του μηχανήματος, να βελτιώσουν τον χρόνο απόκρισης ή να βελτιστοποιήσουν με άλλο τρόπο κάποια πτυχή του συστήματος, όπως υποδεικνύεται από τον χειριστή. Τα περισσότερα συστήματα χρησιμοποιούν συνήθως πολλαπλές ουρές εργασιών, καθεμία με συγκεκριμένο σκοπό και σύνολο περιορισμών δρομολόγησης.

Δρομολόγηση

Οι κοινές παράμετροι που επηρεάζουν την δρομολόγηση εργασιών περιλαμβάνουν τα εξής:

- **Διαθεσιμότητα** των πόρων εκτέλεσης και των βοηθητικών πόρων είναι ο πρωταρχικός παράγοντας που καθορίζει πότε μπορεί να ξεκινήσει μια εργασία.
- **Προτεραιότητα** επιτρέπει στις εργασίες με περισσότερα προνόμια να εκτελούνται νωρίτερα ή ακόμη και να προλαμβάνουν τις τρέχουσες εργασίες με χαμηλότερη προτεραιότητα.
- **Πόροι** που κατανέμονται στον χρήστη καθορίζουν το μακροπρόθεσμο σύνολο πόρων που μπορεί να καταναλώσει ένας συγκεκριμένος χρήστης ενώ ο λογαριασμός του στον υπολογιστή παραμένει ενεργός.
- **Μέγιστος αριθμός εργασιών** που επιτρέπεται να εκτελέσει ταυτόχρονα ένας χρήστης.
- **Ζητούμενος χρόνος εκτέλεσης** που εκτιμά ο χρήστης για την εργασία.
- **Χρόνος εκτέλεσης** μπορεί να προκαλέσει αναγκαστικό τερματισμό της εργασίας ή να επηρεάσει τη δρομολόγηση εκκρεμών εργασιών για μελλοντική εκτέλεση.
- **Εξαρτήσεις εργασιών** καθορίζουν τη σειρά εκκίνησης πολλαπλών σχετικών εργασιών, ειδικά σε σενάρια παραγωγού-καταναλωτή.
- **Εμφάνιση συμβάντος**, όταν η έναρξη της εργασίας αναβάλλεται μέχρι να συμβεί ένα συγκεκριμένο προκαθορισμένο συμβάν.
- **Διαθεσιμότητα του χειριστή** επηρεάζει την εκκίνηση διαδραστικών εφαρμογών.
- **Διαθεσιμότητα άδειας χρήσης λογισμικού**, εάν μια εργασία ζητά την εκκίνηση ιδιόκτητου κώδικα.

Σε γενικές γραμμές, οι αλγόριθμοι δρομολόγησης χωρίζονται σε δύο κατηγορίες:

- **Αλγόριθμοι διαμοιρασμού χρόνου (Time-sharing algorithms)**: διαχωρίζουν το χρόνο ενός επεξεργαστή σε διάφορα διακριτά διαστήματα ή χρονοθυρίδες (slots). Αυτές οι χρονοθυρίδες (slots) στη συνέχεια εκχωρούνται σε μοναδικές εργασίες.
- **Αλγόριθμοι διαμοιρασμού χώρου (Space-sharing algorithms)**: δίνουν τους ζητούμενους πόρους σε μία μόνο εργασία μέχρι να ολοκληρωθεί η εκτέλεσή της. Οι δρομολογητές σε clusters λειτουργούν σε λειτουργία space-sharing.

3.3 Εκτέλεση και Παρακολούθηση Φόρτου Εργασίας (Workload Execution and Monitoring)

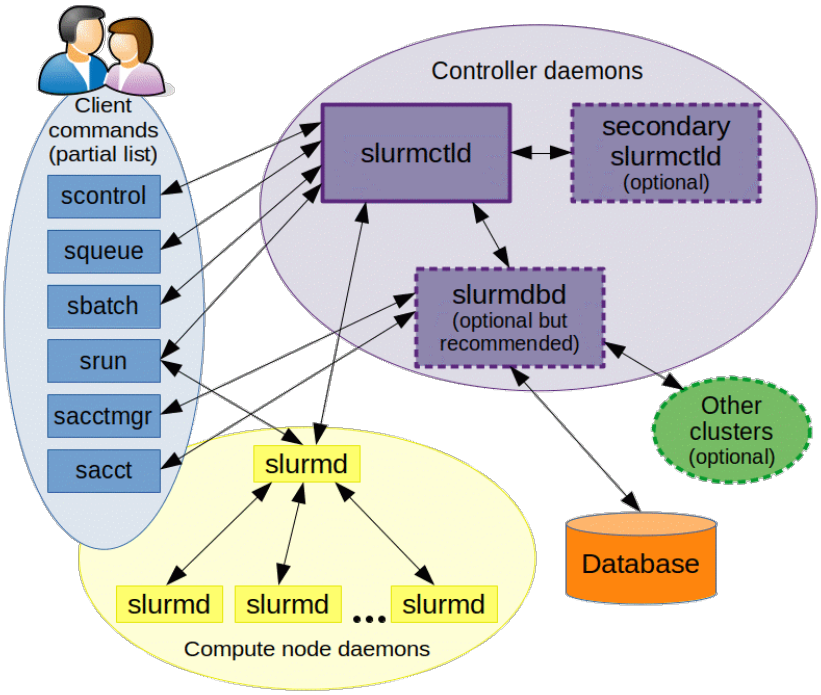
Οι διαχειριστές πόρων εκκινούν και διαχειρίζονται αποτελεσματικά εργασίες μεγάλης κλίμακας χρησιμοποιώντας ιεραρχικούς μηχανισμούς που ελαχιστοποιούν τη μεταφορά δεδομένων και αξιοποιούν την τοπολογία του δικτύου, αποφεύγοντας αναποτελεσματικές μεθόδους όπως οι επαναλαμβανόμενες απομακρυσμένες κλήσεις κελύφους. Μπορούν να τερματίσουν γρήγορα εργασίες που υπερβαίνουν τα όρια πόρων ή χρόνου, ώστε να ελευθερώσουν άμεσα κόμβους. Επιπλέον, παρακολουθούν τις εφαρμογές και καταγράφουν τη χρήση των πόρων για accounting και χρέωση.

4 Slurm Workload Manager

Το Slurm είναι ένας διαχειριστής φόρτου εργασίας ανοιχτού κώδικα, ανθεκτικός σε σφάλματα και εξαιρετικά επεκτάσιμος για συστάδες Linux, ο οποίος λειτουργεί χωρίς τροποποιήσεις στον πυρήνα. Κατανέμει τους υπολογιστικούς πόρους στους χρήστες, διαχειρίζεται την εκτέλεση και την παρακολούθηση των εργασιών και χειρίζεται την δρομολόγηση των εργασιών μέσω ενός συστήματος ουρών. Το Slurm υποστηρίζει επίσης προαιρετικά πρόσθετα για λειτουργίες όπως accounting, προχωρημένες κρατήσεις, backfill και gang scheduling, κατανομή πόρων με γνώση της τοπολογίας και προτεραιοποίηση εργασιών με πολλαπλούς παράγοντες.

4.1 Αρχιτεκτονική Slurm

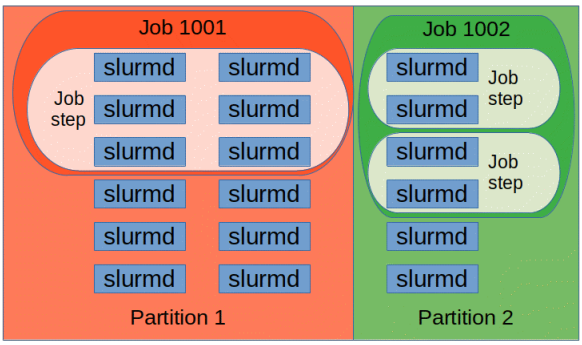
Το Slurm χρησιμοποιεί έναν κεντρικό διαχειριστή, το slurmctld, για την επίβλεψη των πόρων και των εργασιών, με προαιρετική δημιουργία αντιγράφων ασφαλείας για ανακατεύθυνση σε περίπτωση βλάβης. Κάθε υπολογιστικός κόμβος εκτελεί ένα δαίμονα slurmd που εκτελεί εργασίες και υποστηρίζει επικοινωνία ανεκτική σε σφάλματα. Τα προαιρετικά στοιχεία περιλαμβάνουν το slurmdbd για κεντροποιημένο accounting σε όλα τα clusters και το slurmrestd για πρόσβαση στο REST API. Τα εργαλεία χρήστη περιλαμβάνουν τα srun, scancel, sinfo, squeue και sacct για τον έλεγχο και την παρακολούθηση εργασιών, ενώ το sview παρέχει μια γραφική απεικόνιση. Οι διαχειριστές χρησιμοποιούν το scontrol για τη διαχείριση των clusters και το sacctmgr για τη διαμόρφωση (configuration) της βάσης δεδομένων και των χρηστών/λογαριασμών. API είναι διαθέσιμα για όλες τις λειτουργίες.



Σχήμα 8.2: Επισκόπηση Αρχιτεκτονικής Slurm

4.2 Οργάνωση Φόρτου Εργασίας Slurm

Το Slurm διαχειρίζεται διάφορες βασικές οντότητες: κόμβους (υπολογιστικούς πόρους), διαμερίσεις (partitions, ομάδες κόμβων που λειτουργούν και ως ουρές εργασιών), εργασίες (κατανομή πόρων για χρήστες) και βήματα εργασιών (σύνολα tasks εντός μιας εργασίας). Κάθε διαμέριση έχει περιορισμούς, όπως χρονικά όρια, μέγεθος εργασίας και δικαιώματα χρήστη. Οι εργασίες δρομολογούνται κατά προτεραιότητα μέχρι να γεμίσουν οι πόροι σε μία διαμέριση. Μόλις κατανεμηθούν, οι χρήστες μπορούν να εκτελέσουν ένα ή περισσότερα βήματα εργασίας σε οποιαδήποτε διαμόρφωση εντός των κόμβων που τους έχουν εκχωρηθεί. Το Slurm διαχειρίζεται επίσης τους επεξεργαστικούς πόρους εντός μιας εργασίας, επιτρέποντας πολλαπλά βήματα εργασίας να εκτελούνται ή να δρομολογούνται αποτελεσματικά εντός των κοινών εκχωρημένων πόρων.



Σχήμα 8.3: Παράδειγμα Οντοτήτων Slurm

5 Flux Framework

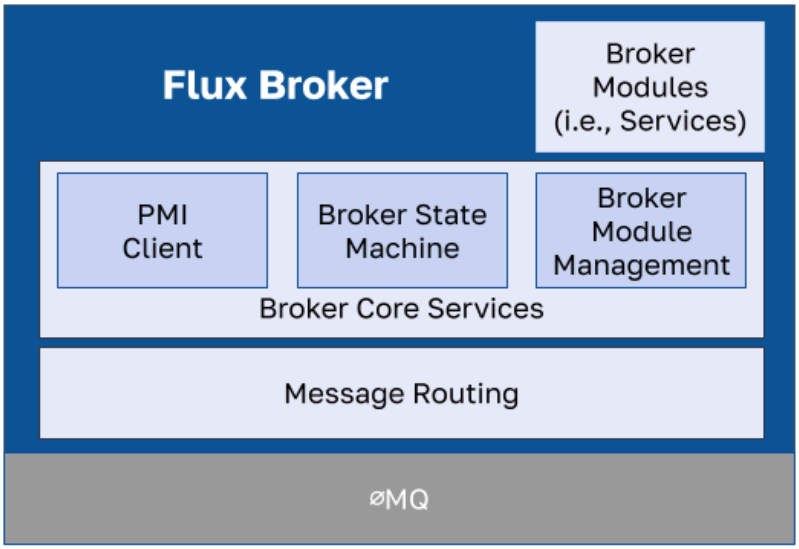
Το Flux [8] είναι ένα πλαίσιο διαχείρισης πόρων και εργασιών νέας γενιάς που αναπτύχθηκε από το Lawrence Livermore National Laboratory. Επεκτείνει την ορατότητα του δρομολογητή πέρα από τη μονοδιάστατη προσέγγιση των «κόμβων», συνδυάζοντας την ιεραρχική διαχείριση εργασιών με την δρομολόγηση βάσει γραφημάτων. Δεν έχει αναπτυχθεί απλώς ως ένα υποκατάστατο των SLURM και Moab. Το Flux προσφέρει ένα πλαίσιο που επιτρέπει την ανάπτυξη νέων τύπων πόρων, δρομολογητών και υπηρεσιών του, καθώς τα κέντρα δεδομένων συνεχίζουν να εξελίσσονται. Παρόλο που το Flux βρίσκεται ακόμη σε ενεργή ανάπτυξη (active development), χρησιμοποιείται επί του παρόντος στην παραγωγή στα ακόλουθα συστήματα Top500 [9]:

- El Capitan (#1)
- Tuolumne (#12)
- El Dorado (#28)
- rzAdams (#64)
- Tioga (#257)
- Tenaya (#337)

5.1 Αρχιτεκτονική Flux

Flux Instance
Ένα Flux Instance είναι ένας αυτόνομος διαχειριστής φόρτου εργασίας που αποτελείται από έναν ή περισσότερους Flux brokers συνδεδεμένους σε ένα δενδρικής μορφής δίκτυο επικάλυψης (tree-based overlay network). Αυτοί οι brokers χειρίζονται τη διαβίβαση μηνυμάτων, τη δρομολόγηση συμβάντων και την ανακάλυψη υπηρεσιών, σχηματίζοντας τον κατανεμημένο κορμό της αρχιτεκτονικής Flux. **Flux Broker**
Ο Flux Broker είναι ένας κατανεμημένος δαίμονας (daemon) που βασίζεται στο ZeroMQ (∅MQ) και παρέχει επικοινωνία εντός ενός Flux Instance. Έχει δύο κύρια συστατικά μέρη:

- **Δίκτυο επικάλυψης**: Εφαρμόζει αφαιρέσεις RPC και pub-sub χρησιμοποιώντας επίμονα επίπεδα δικτύου ZeroMQ για αξιόπιστη, τακτική και αυτοεπιδιορθούμενη επικοινωνία.
- **Βασικές υπηρεσίες**: Παρέχει βασικές λειτουργίες όπως διαχείριση διεργασιών MPI (PMI), έλεγχο εκτέλεσης και διαχείριση μονάδων μεσολάβησης. Πρόσθετες υπηρεσίες υλοποιούνται ως μονάδες που φορτώνονται δυναμικά.



Σχήμα 8.4: Επισκόπηση Flux Broker

KVS (Key-Value Store)
Το Flux KVS είναι ένα broker module που παρέχει ένα κατανεμημένο, συγχρονισμένο αποθηκευτικό χώρο δεδομένων που χρησιμοποιείται από άλλα στοιχεία του Flux. Υποστηρίζει παράλληλη ανταλλαγή δεδομένων και προσωρινή αποθήκευση — κρίσιμα για λειτουργίες όπως το MPI bootstrap και ο συντονισμός σε επίπεδο συστήματος. **Λειτουργίες (modes) Flux Instance**

- **Λειτουργία ενός χρήστη (Single-user mode)**: Ένα Flux Instance εκτελείται ως παράλληλη εργασία με αποκλειστική πρόσβαση στους εκχωρημένους πόρους από τον ιδιοκτήτη της εργασίας.
- **Λειτουργία πολλαπλών χρηστών/συστήματος (Multi-user/system mode)**: Το Flux λειτουργεί ως διαχειριστής πόρων σε επίπεδο συστήματος. Οι brokers εκτελούνται υπό έναν χρήστη συστήματος (flux), διαχειρίζονται από το systemd και υποστηρίζουν πολλαπλούς χρήστες με ασφαλή πιστοποίηση.

Τα ίδια εκτελέσιμα αρχεία χρησιμοποιούνται και στις δύο λειτουργίες, με μόνη διαφορά τη διαμόρφωση.

5.2 Στοιχεία λογισμικού Flux

5.3 Flux Scheduler – Fluxion

Αναπαράσταση πόρων με βάση γραφήματα

Το Flux χρησιμοποιεί ένα μοντέλο κατευθυνόμενου γραφήματος για την αναπαράσταση ετερογενών πόρων συστήματος, επιτρέποντας δυναμική και ιεραρχική δρομολόγηση. Κάθε κορυφή αντιπροσωπεύει έναν πόρο και οι ακμές ορίζουν τις σχέσεις. Αυτό το μοντέλο υποστηρίζει ευέλικτη, ευαίσθητη στην τοπολογία κατανομή και δρομολόγηση υπογραφημάτων.

Δρομολόγηση με βάση γράφους

Το Fluxion δημιουργεί έναν γράφο πόρων στη μνήμη, αντιστοιχίζει αιτήσεις εργασιών με διαθέσιμους πόρους χρησιμοποιώντας πολιτικές διάσχισης του γράφου του συστήματος και αντιστοίχισής του με το γράφο της εργασίας. Επίσης βελτιστοποιεί την δρομολόγηση μέσω φίλτρων κλαδέματος και παρακολούθησης της κατάστασης των πόρων σε σχέση με τον χρόνο, εξισορροπόντας την απόδοση με την αποτελεσματικότητα της δρομολόγησης σε συστήματα μεγάλης κλίμακας.

Πολιτικές Δρομολόγησης

Το Fluxion υποστηρίζει προσαρμόσιμες πολιτικές αντιστοίχισης πόρων (π.χ. low/high ID, node-exclusive) και πολιτικές ουράς αναμονής (π.χ. FCFS, EASY, conservative, hybrid backfilling). Αυτές μπορούν να προσαρμοστούν μέσω χαρακτηριστικών που ορίζονται από τον χρήστη.

5.4 Μοντέλο εργασιών Flux

Μια εργασία αναπαρίσταται ως ένα σύνολο προδιαγραφών που λέγεται jobspec, το οποίο περιγράφει πόρους, tasks και χαρακτηριστικά περιβάλλοντος εκτέλεσής της.

- **Πόροι (Resources)**: Ορίζουν το γράφημα των φυσικών και λογικών οντοτήτων της εργασίας (π.χ. κόμβοι, πυρήνες, υποδοχές).
- **Tasks**: Εντολές που αντιστοιχίζονται σε υποδοχές.
- **Χαρακτηριστικά (Attributes)**: Περιβάλλον εκτέλεσης (π.χ. χρονικό όριο, κατάλογος εργασίας - working directory).

Flux Εντολές CLI

- **flux start** – εκκίνηση νέου Flux instance
- **flux alloc** – εκχώρηση πόρων και εκκίνηση sub-instance
- **flux submit** – υποβολή εργασίας για εκτέλεση
- **flux run** – εκτελεί μια εργασία διαδραστικά
- **flux batch** – υποβάλλει batch scripts για εκτέλεση
- **flux jobs** – εμφανίζει την λίστα εργασιών που έχουν υποβληθεί ή εκτελούνται
- **flux cancel** – ακυρώνει εργασίες
- **flux shutdown** – σταματά ένα Flux instance

6 Flurm

Αυτό το κεφάλαιο παρουσιάζει το **Flurm**, την ενσωμάτωση του πλαισίου Flux σε ένα σύμπλεγμα HPC που διαχειρίζεται το Slurm. Περιγράφει την αρχιτεκτονική, τις τροποποιήσεις στο Flux και τις προκλήσεις υλοποίησης που συναντήθηκαν κατά την ενσωμάτωση.

Επισκόπηση του cluster ARIS

Ο ARIS είναι ένας υπερυπολογιστής τον οποίο διαχειρίζεται το Slurm και λειτουργεί από το GRNET στην Αθήνα. Περιλαμβάνει 532 κόμβους (THIN, GPU, PHI, FAT και ML) που συνδέονται μέσω Infiniband και μοιράζονται 2 PB αποθηκευτικού χώρου GPFS. Η εργασία αυτή επικεντρώνεται στην ενσωμάτωση του Flux σε επίπεδο χρήστη στους κόμβους THIN, χρησιμοποιώντας το Slurm για την αρχική δρομολόγηση και την κατανομή πόρων.

Εγκατάσταση Flux

Η εγκατάσταση του Flux στον ARIS απαιτούσε ανάπτυξη σε επίπεδο χρήστη λόγω περιορισμών του συστήματος. Η χρήση του **Spack** επέτρεψε μια αυτόνομη κατασκευή(custom build) χωρίς προνόμια διαχειριστή, επιλύοντας προβλήματα συμβατότητας με το CentOS 7 και ελλείπουσες εξαρτήσεις.

Το Spack είναι ένας ευέλικτος διαχειριστής πακέτων σε επίπεδο χρήστη για συστήματα HPC, που επιτρέπει προσαρμοσμένες, βελτιστοποιημένες κατασκευές σε διάφορους μεταγλωττιστές και αρχιτεκτονικές χωρίς δικαιώματα root. Υποστηρίζει διαχείριση εξαρτήσεων, προσωρινής αποθήκευσης binaries και εύκολη επεκτασιμότητα καθώς είναι βασισμένο σε Python.

Εγκατάσταση Flux με Spack

Το Flux εγκαταστάθηκε μέσω Spack χρησιμοποιώντας modules όπως τα gnu, `python` και `git`. Προβλήματα όπως το `curl` παλιάς έκδοσης, μη αναγνωρισμένα modules μεταγλωττιστών και βιβλιοθήκες επιλύθηκαν μέσω τροποποιήσεων στη διαμόρφωση, τροποποίησης των μεταβλητών περιβάλλοντος και επιδιορθώσεων στα αρχεία πακέτων του Spack για να συμπεριληφθούν οι τελευταίες εκδόσεις του Flux.

Επισκόπηση Flurm

Το Flurm εκτελεί το Flux εντός μιας κατανομής Slurm, εκκινώντας ένα Flux instance μέσω του `srun`. Η ροή εργασίας (workflow) διασφαλίζει ότι οι brokers ξεκινούν στους κατανεμημένους κόμβους και τερματίζονται με καθαρό τρόπο στο τέλος της εκτέλεσης της εργασίας. Οι βασικές προσαρμογές περιλαμβάνουν:

- Ανακατεύθυνση του `/dev/random` στο `/dev/urandom` για παράκαμψη των διακοπών εκκίνησης που σχετίζονται με την εντροπία των thin nodes.
- Ρύθμιση του `FLUX_DISABLE_JOB_CLEANUP=1` για αποφυγή σφαλμάτων προσωρινών αρχείων.

Συντοποθέτηση και συν-δρομολόγηση

Συντοποθέτηση

Η συντοποθέτηση επιτρέπει σε πολλαπλές εφαρμογές να μοιράζονται κόμβους για την πιο αποδοτική αξιοποίηση τους. Οι τεχνικές περιλαμβάνουν:

- **Συμπαγής κατανομή (Compact allocation):** γέμισμα κάθε κόμβου πριν από τη μετάβαση σε επόμενο.
- **Διάσπαρτη κατανομή (Spread allocation):** συμπαγής κατανομή εργασιών σε μισά sockets.

- **Job Striping:** διάσπαρτη κατανομή σε πυρήνες που είναι αδρανείς (idle) και διαμοιρασμός sockets με άλλες εργασίες.

Συν-δρομολόγηση

Ο συν-δρομολόγηση αποφασίζει ποιες εργασίες θα τοποθετηθούν μαζί. Αν και το Flurm δεν εφαρμόζει ακόμη συγκεκριμένους αλγόριθμους (τυχαία συν-δρομολόγηση), το σύστημα υποστηρίζει μελλοντικές επεκτάσεις συν-δρομολόγησης.

Υποστήριξη συντοπισμού στο Flux

Το Flux υποστηρίζει τόσο συμπαγείς όσο και κατανεμημένες κατανομές μέσω προσαρμοσμένων jobspecs. Οι εργασίες υποβάλλονται μέσω του Flux Python API χρησιμοποιώντας jobspecs YAML που δημιουργούνται δυναμικά. Οι προκλήσεις περιλάμβαναν περιορισμένους τύπους προεπιλεγμένων πόρων και την ανάγκη δημιουργίας γραφημάτων πόρων (αρχεία R και JGF) δυναμικά ανά εργασία Slurm.

Περιπτώσεις χρήσης για συντοποθέτηση και συν-δρομολόγηση

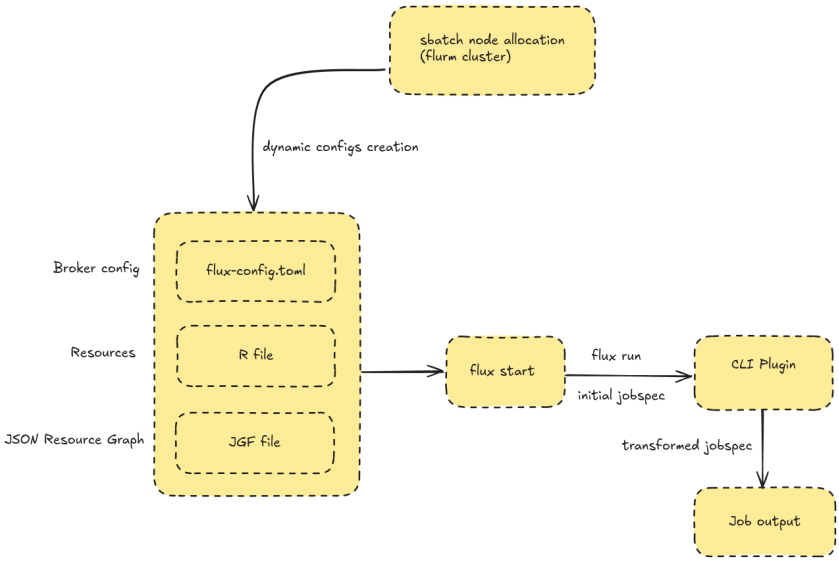
Αναπτύχθηκαν δύο Flux CLI plugins για την υποστήριξη των ακολούθων περιπτώσεων χρήσης:

Κατανομή με ετικέτες εργασιών

Ένα plugin Flux CLI προσθέτει την επιλογή `--alloc-type={compact,spread}` και την χρησιμοποιεί για την αυτόματη προσαρμογή του jobspec ανάλογα με την επιλεγμένη μέθοδο κατανομής πριν την υποβολή της εργασίας.

Κατανομή με ετικέτα ουράς

Δύο ουρές Flux (`normal` και `cosched`) επιτρέπουν τον διαχωρισμό (partitioning) των εργασιών που βρίσκονται στον ίδιο χώρο. Ένα plugin αντιστοιχίζει την επιλογή (option) `--cosched` ή `--queue=cosched` στην ουρά που προορίζεται για συν-δρομολόγηση και προσαρμόζει το jobspec σε spread allocation πριν από την υποβολή της εργασίας.



Σχήμα 8.5: Ροή εργασίας Flurm CLI Plugins

Dockerized Περιβάλλον

Ένα Dockerized Slurm cluster (ένας κόμβος ελέγχου, τρεις υπολογιστικοί κόμβοι) δημιουργήθηκε στο Rocky Linux 8 για γρήγορες δοκιμές. Περιλαμβάνει Spack, Flux και scripts διαμόρφωσης για την αναπαραγωγή της συμπεριφοράς του ARIS, υποστηρίζοντας την ανάπτυξη του Flurm.

8.3 Πηγαίος κώδικας

Ο πηγαίος κώδικας του Flurm είναι διαθέσιμος στη διεύθυνση:

https://github.com/cslab-ntua/flurm

Το αποθετήριο περιλαμβάνει:

- Ένα περιβάλλον δοκιμών Slurm με Docker.
- Ένα κατάλογο με τα ακόλουθα σχετικά αρχεία για το Flurm στον ARIS:
  - Scripts εγκατάστασης Spack και Flux.
  - CLI Plugins για κατανομή και συν-προγραμματισμό.
  - Scripts για δυναμική δημιουργία γραφημάτων πόρων R και JGF.

7 Πειραματική Αξιολόγηση

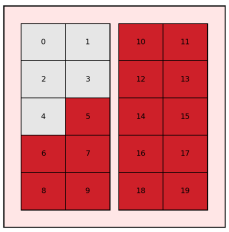
Αυτό το κεφάλαιο παρουσιάζει την αξιολόγηση του **Flurm**, της ενσωμάτωσης Flux–Slurm, στο ARIS HPC Cluster χρησιμοποιώντας το `flux-core v0.78` και το `flux-sched v0.47`. Ο στόχος ήταν να επαληθευτεί η λειτουργικότητα του Flurm και να αξιολογηθεί η απόδοσή του σε σύγκριση με την εγγενή εκτέλεση του Slurm.

NAS Parallel Benchmarks (NPB)

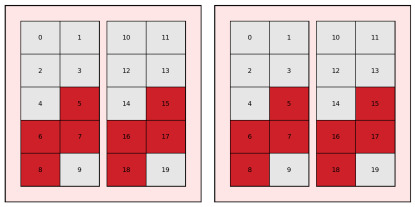
Η αξιολόγηση χρησιμοποίησε τη σουίτα NAS Parallel Benchmarks (NPB) (έκδοση 3.4.3, MPI υλοποίηση), που προέρχεται από computational fluid dynamics εφαρμογές (CFD). Περιλαμβάνει πέντε πυρήνες (kernels) (IS, EP, CG, MG, FT) και τρεις ψευδοεφαρμογές (BT, SP, LU) σε προκαθορισμένες κατηγορίες προβλημάτων (S–F). Επιλέχθηκαν συγκριτικές αξιολογήσεις των κατηγοριών D και E που αντιπροσωπεύουν ρεαλιστικά, απαιτητικά υπολογιστικά φορτία.

Έλεγχος λειτουργικότητας

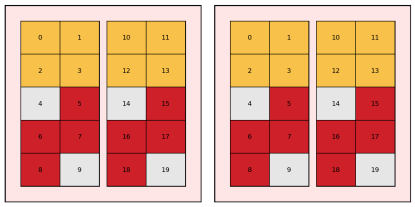
Για την επαλήθευση της ορθότητας, το EP benchmark εκτελέστηκε χρησιμοποιώντας τις κατανομές **compact**, **spread** και **striped** του Flurm. Η ανάθεση task σε πυρήνες (CPU pinning) ελέγχθηκε με τη σημαία `-o verbose=2`. Τα αποτελέσματα επιβεβαίωσαν ότι όλες οι εργασίες στερεώθηκαν στους αναμενόμενους πυρήνες για κάθε τύπο κατανομής, αποδεικνύοντας τη σωστή κατανομή εργασιών και κοινή χρήση κόμβων.



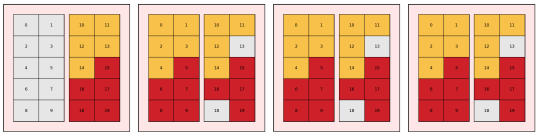
(α΄) Συμπαγής Κατανομή για 16 εργασίες



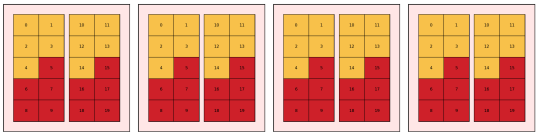
(β΄) Διάσπαρτη Κατανομή για 16 εργασίες



(γ΄) Striped Κατανομή για 16 εργασίες ανά εργασία



(δ΄) Striped Κατανομή για 32 εργασίες ανά εργασία



(ε΄) Striped Κατανομή για 40 εργασίες ανά εργασία

Έλεγχος απόδοσης

Η απόδοση αξιολογήθηκε συγκρίνοντας τους χρόνους εκτέλεσης των NPB τόσο στο Slurm όσο και στο Flurm. Οι επιλεγμένοι δείκτες αναφοράς ήταν:

- CG (Κατηγορία D, 64 εργασίες) – ακανόνιστη μνήμη και επικοινωνία
- EP (Κατηγορία E, 256 εργασίες) – παράλληλη επικοινωνία
- FT (Κατηγορία D, 256 εργασίες) – εντατική επικοινωνία όλων με όλους
- MG (Κατηγορία E, 128 εργασίες) – εντατική χρήση μνήμης

Διάσπαρτη κατανομή
Κάθε benchmark εκτελέστηκε για 10 λεπτά τόσο στο Slurm όσο και στο Flurm. Οι χρόνοι εκτέλεσης μεταξύ των δύο συστημάτων ήταν σχεδόν πανομοιότυποι, με το Flurm να παρουσιάζει ελάχιστο ή καθόλου επιπλέον φόρτο. Το FT benchmark είχε ακόμη και έως και **6% ταχύτερη** απόδοση στο Flurm.

Πίνακας 8.1: Slurm Spread Results

Benchmark	Tasks	Χρόνοι Spread(s)
CG class D	64	149.70
EP class E	256	144.18
FT class D	256	37.68
MG class E	128	88.50

Κατά την αντιστοίχιση του task mapping του Flux με αυτή του Slurm, τα αποτελέσματα παρέμειναν συνεπή με απόκλιση μικρότερη από **3%**.

Πίνακας 8.2: Flurm Spread Results

Benchmark	Tasks	Χρόνοι Spread(s)
CG class D	64	144.59
EP class E	256	144.13
FT class D	256	35.38
MG class E	128	88.05

Πίνακας 8.3: Flurm Spread Taskmap Results

Benchmark	Tasks	Χρόνοι Spread(s)
CG class D	64	146.10
EP class E	256	144.22
FT class D	256	36.64
MG class E	128	88.05

Δοκιμές συν-εκτέλεσης (Colocation)

Τα benchmarks συνδυάστηκαν και εκτελέστηκαν ταυτόχρονα και στα δύο συστήματα χρησιμοποιώντας κατανομή spread. Για τη σύγκριση χρησιμοποιήθηκαν οι ενδιάμεσοι (median) χρόνοι συν-εκτέλεσης.

Πίνακας 8.4: Slurm Co-Execution Results

Benchmark A	Tasks	Χρόνοι Συνεκτέλεσης(s)	Benchmark B	Tasks	Χρόνοι Συνεκτέλεσης(s)
CG class D	64	191.33	CG class D	64	191.66
CG class D	64	155.055	EP class E	256	149.595
CG class D	64	181.755	FT class D	256	42.305
CG class D	64	267.855	MG class E	128	105.77
EP class E	256	144.21	EP class E	256	144.205
EP class E	256	148.675	FT class D	256	38.77
EP class E	256	154.31	MG class E	128	96.475
FT class D	256	53.06	FT class D	256	53.01
FT class D	256	48.895	MG class E	128	107.835
MG class E	128	156.19	MG class E	128	156.49

Πίνακας 8.5: Flurm Co-Execution Results

Benchmark A	Tasks	Χρόνοι Συνεκτέλεσης(s)	Benchmark B	Tasks	Χρόνοι Συνεκτέλεσης(s)
CG class D	64	186.55	CG class D	64	186.58
CG class D	64	162.9	EP class E	256	145.515
CG class D	64	179.37	FT class D	256	39.31
CG class D	64	266.87	MG class E	128	105.54
EP class E	256	145.56	EP class E	256	144.11
EP class E	256	145.075	FT class D	256	36.46
EP class E	256	146.805	MG class E	128	97.29
FT class D	256	39.96	FT class D	256	39.95
FT class D	256	45.04	MG class E	128	108.015
MG class E	128	155.18	MG class E	128	157.78

Πίνακας 8.6: Flurm Co-Execution Taskmap Results

Benchmark A	Tasks	Χρόνοι Συνεκτέλεσης(s)	Benchmark B	Tasks	Χρόνοι Συνεκτέλεσης(s)
CG class D	64	188.70	CG class D	64	189.10
CG class D	64	168.69	EP class E	256	146.61
CG class D	64	176.39	FT class D	256	39.235
CG class D	64	264.54	MG class E	128	106.12
EP class E	256	145.515	EP class E	256	144.085
EP class E	256	147.34	FT class D	256	37.97
EP class E	256	148.145	MG class E	128	96.445
FT class D	256	42.615	FT class D	256	42.60
FT class D	256	47.085	MG class E	128	106.43
MG class E	128	155.54	MG class E	128	156.27

Ανάλυση αποτελεσμάτων

Η διαφορά απόδοσης ποσοτικοποιήθηκε χρησιμοποιώντας το **Μέσο απόλυτο ποσοστό σφάλματος (MAPE)**:

$$MAPE = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{A_i - F_i}{A_i} \right| \times 100$$

Το Flurm πέτυχε MAPE **4,793%** (προεπιλεγμένη αντιστοίχιση) και **3,907%** (αντιστοίχιση τύπου Slurm). Η ανάλυση μεμονωμένων benchmarks έδειξε ότι το Flurm ταιριάζει σε μεγάλο βαθμό με τη συνολική απόδοση του Slurm, με αξιοσημείωτες βελτιώσεις σε φόρτους εργασίας με έντονη επικοινωνία, όπως το FT (έως και **25% ταχύτερο**):

Πίνακας 8.7: Μεμονωμένες Διαφορές Ποσοστών (Προεπιλεγμένη Αντιστοίχιση)

Benchmark A	Tasks	PE(%)	Benchmark B	Tasks	PE(%)
CG class D	64	-2.50	CG class D	64	-2.65
CG class D	64	5.06	EP class E	256	-2.73
CG class D	64	-1.31	FT class D	256	-7.08
CG class D	64	-0.37	MG class E	128	-0.22
EP class E	256	0.94	EP class E	256	-0.07
EP class E	256	-2.42	FT class D	256	-5.96
EP class E	256	-4.86	MG class E	128	0.84
FT class D	256	-24.69	FT class D	256	-24.64
FT class D	256	-7.88	MG class E	128	0.17
MG class E	128	-0.65	MG class E	128	0.82

Πίνακας 8.8: Μεμονωμένες Διαφορές Ποσοστών (Αντιστοίχιση τύπου Slurm)

Benchmark A	Tasks	PE(%)	Benchmark B	Tasks	PE(%)
CG class D	64	-1.375	CG class D	64	-1.336
CG class D	64	8.794	EP class E	256	-2
CG class D	64	-2.952	FT class D	256	-7.257
CG class D	64	-1.238	MG class E	128	0.331
EP class E	256	0.905	EP class E	256	-0.083
EP class E	256	-0.9	FT class D	256	-2.063
EP class E	256	-4	MG class E	128	-0.031
FT class D	256	-19.69	FT class D	256	-19.638
FT class D	256	-3.702	MG class E	128	-1.303
MG class E	128	-0.416	MG class E	128	-0.141

8 Συμπεράσματα και Μελλοντικές Κατευθύνσεις

Συνολικά, το Flurm επιδεικνύει:

- Σωστή λειτουργικότητα και ακριβή τοποθέτηση εργασιών σε όλους τους τύπους κατανομής.
- Απόδοση συγκρίσιμη ή καλύτερη από το Slurm με αμελητέο επιπλέον κόστος.
- Περιστασιακή βελτίωση της απόδοσης σε φόρτους εργασίας με έντονη επικοινωνία.

Αυτά τα αποτελέσματα επιβεβαιώνουν ότι η ενσωμάτωση του Flux στο Slurm μέσω του Flurm είναι τόσο αποτελεσματική όσο και πρακτική, διατηρώντας την απόδοση του Slurm και επιτρέποντας ταυτόχρονα προηγμένες λειτουργίες του Flux, όπως η συν-δρομολόγηση. Μερικές ενδιαφέρουσες κατευθύνσεις για μελλοντική έρευνα περιλαμβάνουν:

Υλοποίηση πολιτικής συν-δρομολόγησης

Επί του παρόντος, το Flurm υποστηρίζει την κοινή χρήση πόρων, αλλά δεν διαθέτει καθορισμένο αλγόριθμο συν-δρομολόγησης. Μελλοντικές εργασίες θα μπορούσαν να περιλαμβάνουν την ανάπτυξη και ενσωμάτωση μιας προσαρμοσμένης πολιτικής συν-δρομολόγησης απευθείας στο `Fluxion`.

Εξερεύνηση διαφορετικών πολιτικών κατανομής πόρων

Τα πειράματα χρησιμοποίησαν την προεπιλεγμένη πολιτική κατανομής πόρων «πρώτης αντιστοίχισης» (first match) του Fluxion. Μελλοντική έρευνα θα μπορούσε να συγκρίνει διαφορετικές ενσωματωμένες πολιτικές (π.χ. χαμηλό/υψηλό αναγνωριστικό κόμβου πρώτα) ή και να σχεδιάσει προσαρμοσμένες πολιτικές βελτιστοποιημένες για συν-δρομολογημένα φορτία εργασίας.

Preemption στο Flux/Flurm

Η εισαγωγή του **preemption** — που επιτρέπει σε εργασίες υψηλής προτεραιότητας να ανακτούν πόρους από εργασίες χαμηλότερης προτεραιότητας — θα ενίσχυε την ευελιξία της δρομολόγησης. Η αξιολόγηση του αντίκτυπού της στην απόδοση και τη χρήση των πόρων αποτελεί μια άλλη πολλά υποσχόμενη κατεύθυνση έρευνας.

Bibliography

[1] E. Karapanagiotis, N. Triantafyllis, A. Tsoukleidis-Karydakis, G. Goumas, and N. Koziris, *Elise: A tool to support algorithmic design for hpc co-scheduling*, 2025.

[2] *Hpl - high performance linpack*. [Online]. Available: https://www.netlib.org/benchmark/hpl/.

[3] TOP500, *Top500 supercomputer site*. [Online]. Available: https://www.top500.org/.

[4] T. Sterling and et al., *High Performance Computing*. Morgan Kaufmann, 2018, pp. 43–82, 142–145. DOI: 10.1016/B978-0-12-420158-3.00002-2.

[5] *Message passing interface (mpi) standard*. [Online]. Available: https://en.wikipedia.org/wiki/Message_Passing_Interface.

[6] D. Liang, P.-J. Ho, and B. Liu, "Scheduling in distributed systems", *Department of Computer Science and Engineering University of California, San Diego*, 2000.

[7] S. LLC, *Slurm workload manager*. [Online]. Available: https://slurm.schedmd.com/overview.html.

[8] T. F. Team, *Flux*. [Online]. Available: https://flux-framework.readthedocs.io/en/latest/.

[9] T. F. Team, *Flux isc aws 2025 presentation*. [Online]. Available: https://github.com/flux-framework/Tutorials/blob/master/2025/ISC-AWS/Flux-ISC-2025.pdf.

[10] T. F. Team, *Flux components*. [Online]. Available: https://flux-framework.readthedocs.io/en/latest/guides/components.html.

[11] D. H. Ahn, J. Garlick, M. Grondona, and D. Lipari, "Vision and plan for a next generation resource manager", Technical Report, LLNL-TR-636552-DRAFT. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[12] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A next-generation resource management framework for large hpc centers", in *2014 43rd International Conference on Parallel Processing Workshops*, 2014, pp. 9–17. DOI: 10.1109/ICPPW.2014.15.

[13] D. H. Ahn *et al.*, "Flux: Overcoming scheduling challenges for exascale workflows", *Future Generation Computer Systems*, vol. 110, pp. 202–213, 2020, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2020.04.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19317169.

[14] P. Hintjens, *Zeromq: The guide*. [Online]. Available: https://zguide.zeromq.org.

[15] T. F. Team, *Flux learning guide*. [Online]. Available: https://flux-framework.readthedocs.io/en/latest/guides/learning_guide.html.

[16] T. Patki *et al.*, "Fluxion: A scalable graph-based resource model for hpc scheduling challenges", in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23, Denver, CO, USA: Association for Computing Machinery, 2023, pp. 2077–2088, ISBN: 9798400707858. DOI: 10.1145/3624062.3624286. [Online]. Available: https://doi.org/10.1145/3624062.3624286.

[17] T. F. Team, *Flux documentation*. [Online]. Available: https://flux-framework.readthedocs.io/projects/flux-core/en/latest/man1/index.html.

[18] T. F. Team, *Flux compared to other resource managers*. [Online]. Available: https://flux-framework.readthedocs.io/en/latest/tables/comparison-table.html.

[19] GRNET, *Introduction to high performance computing systems and aris system*. [Online]. Available: https://www.hpc.grnet.gr/supercomputer/.

[20] GRNET, *Aris documentation*. [Online]. Available: https://doc.aris.grnet.gr/.

[21] S. Team, *Spack package manager*. [Online]. Available: https://spack.io/.

[22] A. D. Breslow *et al.*, "The case for colocation of high performance computing workloads", *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 232–251, 2016.

[23] D. Bailey *et al.*, "The nas parallel benchmarks", *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991. DOI: 10.1177/109434209100500306. eprint: https://doi.org/10.1177/109434209100500306. [Online]. Available: https://doi.org/10.1177/109434209100500306.