DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Adaptive Resource-Aware Batch Scheduling for HPC systems

Mohak Chadha

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Adaptive Resource-Aware Batch Scheduling for HPC systems

Adaptive Ressourcen-bewusste Stapelplanung für HPC-Systeme

Author:	Mohak Chadha
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Jophin John
Submission Date:	17.02.2019

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 17.02.2019 Mohak Chadha

Acknowledgments

I would like to express my deep and sincere gratitude to my supervisor Prof. Dr. Michael Gerndt, for his guidance, support, and constant encouragement during this thesis work. Moreover, I am incredibly grateful to him for providing me the opportunity to work in the domain for HPC for almost the entire duration of my master's. First, in the domain of energy-efficiency as part of the READEX project and now in the domain of scheduling as part of the Invasive Computing project. My numerous discussions with him have helped me improve my knowledge of HPC and also the quality of this work.

I would also like to thank my advisor Jophin John for helping me understand and set up the current infrastructure. Lastly, I would like to thank my mom and dad for their unparalleled support, without which none of this would have been possible.

Abstract

Distributed memory modern HPC systems consist of millions of compute nodes interconnected via a high-performance network. These systems are shared among thousands of users for submitting and executing their applications. The main software component responsible for scheduling and allocating resources to the user's jobs is the middleware called Resource and Job Management System (RJMS). The main element of RJMS responsible for resource management and scheduling is the Batch Scheduler. The applications submitted by the users are typically large scale simulations related to astronomy, climate modeling, biology, fluid and molecular dynamics, etc. Moreover, the resource requirements of these applications can dynamically change during their runtime. Changing the resources of an application at runtime requires an adaptive parallel runtime system and a dynamic resource management system. No RJMS software currently supports the dynamic reconfiguration of running applications. Towards this, in this thesis, the scalable workload manager SLURM is extended to support the dynamic reconfiguration of resource-elastic queued applications written using the Invasive MPI adaptive library. Different SLURM binaries are extended to allow users to submit resource-elastic jobs in batch mode. The batch scheduler in SLURM is extended through a scheduling plugin to support the efficient combined scheduling of rigid and malleable applications. Moreover, multiple scheduling strategies for elastic applications are implemented and evaluated. Finally, the overhead for dynamic adaptation operations, i.e., expansion and reduction, is analyzed.

Contents

1 Introduction

Before the introduction of modern day PC's, computer technology has long been used for scientific purposes more specifically for numerically demanding calculations. The complexity or capability of the integrated circuits used in development of computer chips is influenced by *Moore's Law* [1], according to which the complexity of the integrated circuits doubles every 24 months. Increasing chip transistor counts and clock speeds, have led to the development of advanced techniques such as Pipelined functional units, Superscalar architectures and Out-of-order execution which have significantly influenced application performance. According to Dennard Scaling the dynamic (switching) power consumption of CMOS circuits is proportional to frequency. However, due to it's breakdown and the emergence of the *power-performance dilemma* (i.e. increasing clock speeds leads to increase in power dissipation), there has been an architectural shift from single core to multicore and manycore processors.

According to Flynn's taxonomy for parallel systems [2], Multiple Instruction Multiple Data (MIMD) systems can be classified into shared and distributed memory systems. However, due to the bottlenecks of Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) based shared memory systems such as scalability of Cache Coherency Protocol, all modern-day High Performance Computing (HPC) systems are based on distributed memory architecture. An HPC system or a supercomputer is a machine that can perform more than 10^{15} floating-point-operations per second (Flop/s). Large scale HPC systems offer petabytes of main memory and consist of thousands of multicore or manycore processors tightly integrated into streamlined blades or nodes. A typical HPC ecosystem consists of front-end nodes, compute nodes, system nodes, and a storage system. System nodes consist of Reliability, Availability and Serviceability components to increase the robustness of the HPC system. Storage is provided through a parallel file system which allows high performance read and write access to data. A compute node generally consists of two multicore processors called sockets organized in a distributed shared memory architecture. The compute nodes in an HPC system are connected via a high performance interconnect network, e.g., Infiniband organized in a particular network topology. The front-end nodes are identical to the compute nodes in terms of hardware and offer users, access to the system.

The Top500 [3] list which is published twice every year ranks the fastest 500 general purpose supercomputers in the world based on their performance on the LINPACK benchmark. Table 1.1 shows the type of architecture used in one node, total number of cores and the

HPC System	Architecture	Number of cores	Perf. (Pflop/s)
Summit	IBM Power9, Nvidia Volta	2,414,592	148.60
Sierra	IBM Power9, Nvidia Volta	1,572,480	94.64
Sunway TaihuLight	Sunway SW26010 260C	10,649,600	93.01
Tianhe-2A	Intel Ivy-Bridge	4,981,760	61.44
Frontera	Intel Cascade Lake	448,448	23.51

Table 1.1: Architecture, number of cores and performance for the top five HPC systems as published in the Top500 list on June 2019 [4].

LINPACK performance in petaflops for the top five supercomputers in the world as published in the Top 500 [4] list. It is believed that future HPC systems will be able to perform exaflops or 10^{18} Flop/s.

HPC systems (see Table 1.1) are generally used for large scale scientific simulations in all domains of science such as astronomy, biology, climate modeling, fluid, and molecular dynamics. However, due to the large investment and operating costs associated with such systems they are usually shared among a large number of users. Furthermore, due to the increasing complexity of scientific applications it is becoming difficult to maximize performance and efficiently exploit the rapidly changing underlying hardware in HPC systems. To efficiently distribute computing resources such as nodes, network, and storage among users of an HPC system and to efficiently map the submitted applications or jobs to the underlying hardware, HPC systems utilize a Resource and Job Management System (RJMS) [5].

RJMS is essentially a middleware and acts as an interface between the hardware and the application layer. It is one of the most critical components in an HPC system. It consists of three primary subsystems namely, Resource Mangement (RM), Job Management (JM) and Scheduling [5].

The primary function of the RM subsystem is to gather and collect all information regarding the available resources in an HPC system. The collected information is then given to the user to check the availability of the HPC system and to the scheduler to initiate job scheduling. It performs three main tasks Resource treatment, Job launching, and propagation, and Task placement. Resource treatment refers to the characterization of hierarchical resources in an HPC system (node/core/HW thread) for efficient utilization. Job launching and propagation involves initialization of the tasks on selected compute nodes through a specific propagation technique (ssh or daemons) and then the creation of the task. Task placement is responsible for the optimal binding of the generated tasks according to the underlying hardware and network topology.

The JM subsystem is responsible for declaration, control, and monitoring of jobs. Typical

RJMS softwares provide users a way to declare and characterize their jobs and to specify resources on which the jobs need to be run. In most cases, users also have an option between an interactive or a batch job. In an interactive job, the user is responsible for launching the application manually, while in a batch job, the application is automatically executed on the allocated compute nodes. Job control refers to certain functionalities which allow users to modify certain parameters of the job, once they have been submitted. The type of options offered varies across RJMS softwares. For example, some RJMS softwares allow users to modify the *wall clock time* of the submitted job when it is still in the queue. Furthermore, RJMS softwares monitor and log information such as total execution time, total energy consumed, etc for the submitted jobs. This information is reported to the user and can also be visualized using explicit tools.

An important part of all RJMS softwares is the Scheduler. The primary role of the job scheduler is to manage job queues and to assign computational resources to the user submitted jobs according to a particular scheduling policy or algorithm. Common scheduling algorithms used in RJMS softwares include First In First Out (FIFO), Back-filling, Gang Scheduling, TimeSharing, Fair-Share and Preemption. The combination of RM subsystem along with the Scheduler is referred to as the Batch System. Some examples of commercial and open-source resource and job management systems includes IBM LoadLeveler, LSF, Moab, Condor, SLURM [6], Torque and Maui etc.

However, due to the increasing complexity and size of modern-day supercomputers, current resource and job management systems face a large number of challenges. Since the primary function of RJMS is to efficiently distribute resources among a large number of users of an HPC system, the scheduler and job launcher needs to be highly scalable. Scientific computing applications such as Adaptive Mesh Refinement (AMR) [7] or Quadflow [8], which uses multiscale analysis are dynamic and change their resource requirements during execution due to varying computation phases. Most resource and job management systems today only support the execution of rigid jobs and lack the management facilities for executing dynamic applications. This means that the number of resources allocated remains fixed during the entire execution of the application, which leads to underutilization of resources or a decrease in performance and response times for the type of applications mentioned above. This problem is further increased due to the static and rigid nature of programming models, such as Message Passing Interface (MPI) [9].

Energy efficiency and energy conservation are some of the most crucial constraints for meeting the 20MW power envelope desired for exascale systems. The published data in the Top500 List [3] indicates that the performance of HPC systems has been continuously increasing along with increasing power consumption [10], [11]. Optimizing and evaluating the energy efficiency of HPC systems has been a significant area of research in the HPC community [12, 13, 14]. Since resource and job management systems know about the systems hardware and the user's application, they should be able to schedule and execute the

applications in a manner that improves the energy efficiency of the system. Thus, to increase the energy efficiency of the system and resource utilization and response times for dynamic jobs, current resource and job management systems need to be extended. Furthermore, parallel programming models also need to be extended to support dynamic resource changes at runtime of an application.

1.1 Invasive Computing

The Transregional Collaborative Research Center: Invasive Computing funded by the German Research Foundation (DFG) focuses on the design and development of a paradigm for resource-aware programming of current and future parallel systems [15]. It targets different hardware architectures such as Tightly-Coupled Processor Arrays (TCPAs), shared memory embedded multicore systems, and distributed memory HPC systems. TCPAs are processor arrays developed as part of the Invasive Computing project, which increases the performance of vector operations in an energy-efficient manner [16]. Furthermore, Invasive Computing supports the development of software such as languages, operating systems, compilers, and adaptive scientific applications for these architectures. This work primarily focuses on extending an existing resource and job management system to support efficient resource-aware scheduling of Invasive MPI based adaptive applications.

The ubiquitous standard for programming distributed memory HPC systems is the Message Passing Interface (MPI) [9]. MPI is an Application Programming Interface (API) specification first designed in 1992 with the first standard (MPI-1) published in 1994. The most recent version of the standard is MPI-3.1 [9] and provides bindings for the programming languages C and Fortran (up to Fortran 2008). The basic functionality includes routines for point to point communication, one-sided communication (Remote Memory Access (RMA)), collective communication and management of a group of processes.

According to a recent survey [17], conducted as part of the Exascale Computing Project (ECP), 96% of the people expect that the exascale versions of their applications will be based on MPI. Even though the fundamental principles of the MPI standard, i.e., negligible memory management and global state per process target scalability, it only supports the static allocation of resources during the runtime of the application. Although dynamic process support was added to the MPI standard in MPI 2.0 through the operations `MPI_COMM_SPAWN` and `MPI_COMM_SPAWN_MULTIPLE`, it is rarely used due to several limitations [16].

The added dynamic process operations are a form of blocking collective communication between the parent and the child processes. This leads to significant performance overhead for the parent process when a new child process is created due to initialization and startup

costs. Moreover, a change in the resources of a running MPI application can only be initiated by the application. This fact is counterproductive since the application, unlike a resource and job management system, does not have a holistic view about the pending jobs and free resources in an HPC system. Furthermore, since MPI is implemented as a library and is implementation-dependent most MPI implementations such as MPICH [18] and OpenMPI [19] support dynamic process creation only in pre-allocated resources to the application. This limits the benefits of the added spawn operations in the MPI standard. To overcome the limitations of dynamic process creation in MPI, the Invasive MPI Library (IMPI) was developed by adding four additional routines to the MPI standard (see Section 3.1). The added routines support both the creation and reduction of processes.

1.2 Adaptive Resource-Aware Batch Scheduling

An important aspect of dynamic process creation is that it requires the support of a batch system capable of dynamic resource management. A batch system that includes a resource manager along with a scheduler is responsible for running parallel jobs, maintaining high system utilization and throughput along with ensuring high response times and fairness among submitted jobs. However, current batch systems are not capable of handling dynamic resource changes for an application during its runtime. Dynamic resource changes for an application can either involve the addition of computing resources, i.e., an expand operation or reduction of computing resources, i.e., a shrink operation.

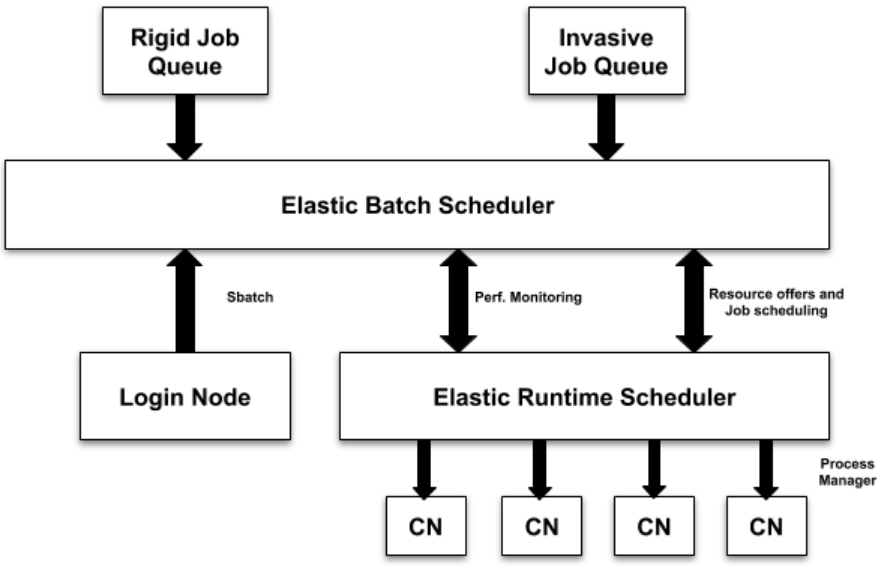


Figure 1.1: Overview of Interactions between Adaptive Runtime and Batch Scheduler.

Apart from supporting expand and shrink operations, a batch system should also be able to decide which applications to expand and which to shrink efficiently. Towards this, this work extends an existing batch system Simple Linux Utility for Resource Management (SLURM) [6] for efficiently scheduling, expanding and shrinking adaptive Invasive MPI applications.

Figure 1.1 shows an overview of the interactions between the implemented elastic batch scheduler and the elastic runtime scheduler. A user writes a job script and submits the job for execution using the SLURM utility `sbatch`. The utility *sbatch* is extended to allow users to specify options for considering a job as invasive. If the appropriate options are specified, and the application is programmed using Invasive MPI, it is added to the Invasive Job Queue in the elastic batch scheduler. Otherwise, the job is considered as rigid and added to the rigid job queue. Depending on the priority of the job and the available number of resources, it is scheduled and launched on the compute nodes by the elastic runtime scheduler with the help of the Process Manager. The process manager in SLURM is called the SLURM step daemon (`slurmstepd`) and is responsible for interacting with the processes of a parallel application running on the node via the Process Management Interface (PMI). The subsequent expand and shrink operations for the running invasive applications are decided depending on the job and malleability management policies. Furthermore, the elastic runtime scheduler continuously monitors and updates the system performance metrics such as system utilization, waiting, and response times of the jobs according to the current system state.

Towards efficient dynamic resource management and process creation, the main goals of this work are:

- Extending the SLURM batch system to support combined scheduling of adaptive and rigid jobs through an elastic batch scheduling plugin.
- Implementing and analyzing different job and malleability management policies for efficient expand/shrink operations, increased system utilization and response times.
- Demonstrating the benefits of malleable jobs.
- Analyzing the overhead for expand and shrink operations.

The rest of the thesis is structured as follows. Chapter 2 describes the existing techniques and frameworks for scheduling adaptive applications. In Chapter 3 Invasive MPI, the Invasive Runtime Scheduler and the Elastic Phase Oriented Programming Model (EPOP) are discussed. Chapter 4 describes the extensions to SLURM for initiating resource-elastic queued jobs. In Chapter 5, the scheduling infrastructure and strategies are described. Chapter 6 highlights some of the implementation details. In Chapter 7 experimental results are presented. Finally, Chapter 8 concludes the work and presents an outlook.

2 Related Work

The problem of scheduling and managing adaptive applications essentially requires three components an adaptive parallel runtime system, an adaptive job scheduler, and a dynamic runtime scheduler. Considerable research has been done in all three domains from the development of an adaptive runtime system, scheduling strategies for adaptive applications, and extending current batch systems for dynamic resource management. This section describes and discusses the relevant work in all three domains.

One of the earliest contributions towards supporting malleability in MPI applications was through Charm++ [20, 21, 22] and Adaptive MPI (AMPI) [23, 24, 25]. Adaptive MPI builts on top of Charm++ and utilizes over-subscription of virtual MPI processes to the same CPU core to provide malleability. A virtual MPI process (rank) in AMPI is a user-level thread encapsulated into a Charm++ object. The AMPI runtime provides automatic load balancing features and automated checkpoint and restart mechanisms for fault tolerance [26]. Automatic load balancing is achieved through routines that allow virtual MPI process migration without the need for custom pack and unpack routines [27]. It supports MPI up to version 2.2, and new features such as virtual topologies and non-blocking collectives introduced in MPI-3.1 [9] are not supported. AMPI and Charm++ achieve resource-elasticity with the help of an adaptive scheduler. The running applications periodically receive information about resource availability from the scheduler, based on which the application is adapted to a different number of virtual MPI processes using Charm++ object migration. In contrast to Charm++ and AMPI, the adaptive parallel runtime system Invasive MPI (IMPI) used in this work follows the current MPI execution model of processes with private address spaces and no oversubscription of virtual MPI processes [16, 28].

Other examples of adaptive runtime frameworks include ReSHAPE [29] and Flex-MPI [30]. Both of the frameworks provide performance-aware dynamic reconfiguration for iterative Single Program Multiple Data (SPMD) MPI applications. While ReSHAPE assumes identical iterations for all parallel applications in terms of computation and communication times, Flex-MPI supports irregular and regular communication patterns in parallel applications. Flex-MPI is a library that builds on top of MPICH [18] and utilizes a computational prediction model formulated using Performance Monitoring Counters (PMCs) and network performance data for selecting and evaluating application reconfiguration decisions. On the other hand, ReSHAPE employs a more costly exhaustive approach for reconfiguration decisions.

The efficient scheduling of parallel applications and utilization of resources has been a significant area of research in the field of HPC and computer science for decades [31, 32]. The throughput of the system, however, depends upon the type of jobs being scheduled and executed by the job scheduler. According to Feitelson and Rudolph [33], jobs can be primarily classified into four categories based on their flexibility. The first and the most common type of jobs are called rigid jobs, which require a fixed number of resources throughout their execution. The second category is called moldable jobs, whose resource requirements can be modified by the batch system before the application is executed. However, after the application starts execution, the allocated resources remain fixed during its runtime. The third type is evolving jobs. In this case, the application requests a resource allocation expansion or reduction during its runtime. Applications such as Adaptive Mesh Refinement [7], which tend to generate load imbalances due to varying computation phases, fall in this category. The final type of job is called malleable jobs. In this case, the resource changes, i.e., expansion or shrinkage, are triggered by the batch system. This work primarily deals with the scheduling of malleable and rigid jobs.

Executing malleable jobs on modern HPC systems can significantly improve system utilization and lead to reduced response and makespan times, as shown by Gupta et al. [34] and Hungershofer [35]. Towards this, several strategies have been proposed in the literature for efficiently scheduling malleable applications. Carroll et al. [36] propose an incentive compatible online scheduling method for scheduling malleable applications. The main aim of the approach is to assign resources to jobs in a way that reduces the response time. The users submit the jobs along with parameters such as arrival time and deadline. To ensure that users report truthful information about their jobs' parameters, incentives were given to users if their jobs' completed by the specified hard deadline. Sun et al. [37] propose an approach for fair and efficient scheduling of malleable applications on HPC systems. For fairness, they utilize the equipartitioning algorithm, which equally divides the available number processors among the running applications at any time. To achieve efficiency, they use a feedback-driven adaptive scheduler, which takes into account the history of the executing jobs. They show that there approach leads to better system utilization and response times as compared to an existing rigid scheduler. However, both of the approaches [36] and [37] deal with theoretical aspects of scheduling and are only evaluated using simulations.

In contrast to the above approaches, some prototypes which combine an adaptive batch system along with an adaptive job scheduler have been developed. Utrera et al. [38] propose a Job Scheduling Strategy (JSS) based on the principle of virtual malleability (VM). In VM, the original number of processes is preserved, and the job is allowed to adapt to changes in the number of CPUs at runtime. The concept of VM is implemented using MPI and the interposition mechanism. They propose a First Come First Serve (FCFS) malleable job scheduling policy according to which the jobs are allocated resources in the earliest-started-job-first-order. It is an event-driven algorithm, executed when a job arrives and exits. The VM library is dynamically linked and communicates with the job scheduler via TCP/IP socket

communication. The authors show that for a cluster composed of only malleable jobs, FCFS malleable job scheduling policy leads to a 31% improvement in average response time as compared to the widely used EASY backfilling strategy.

Prabhakaran et al. [39] extend the existing Torque/Maui batch system for dynamic resource management and on-demand allocation of resources for evolving jobs. To ensure fairness among evolving and rigid jobs, they propose a novel, dynamic resource allocation strategy. Their results showed reduced turnaround and waiting times for applications while increasing system utilization and throughput. Furthermore, they built on this and extended the Torque/Maui batch system for handling expand and shrink operations for malleable jobs [40]. The malleability of applications is supported by using a communication protocol between the Torque/Maui batch system and the Charm++ runtime. They propose a Dependency-based Expand Shrink (DBES) scheduling algorithm which is capable of combined scheduling of rigid, evolving and malleable jobs. The algorithm uses dependency analysis and backfilling strategy for efficient scheduling across varying dynamics of the workload. For fairness, it also uses the equipartitioning strategy for distributing idle resources among jobs after the dependency analysis and backfilling stages. They evaluate and demonstrate their approach for the modified Effective System Performance Benchmark (ESP) [41] in terms of the metrics average response time and system utilization. They show that the proposed DBES scheduling strategy consistently outperforms scheduling strategies such as equipartitioning, earliest started first, earliest deadline first, and latest deadline first for varying number of malleable jobs in the workload. However, their scheduling strategy does not account for the performance of the running application for reconfiguration decisions. Similar to this work, this is the only complete batch system in the literature that combines a dynamic batch system with an adaptive parallel runtime.

Buisson et al. [42] propose a modified version of the equipartition strategy along with Favour Previously Started Malleable Applications (FPSMA) scheduling policy for scheduling malleable applications in GRID systems. They evaluate and demonstrate their approach for the KOALA multicluster grid scheduler. In this work, we extend these policies for scheduling malleable jobs with certain constraints on the number of required nodes. Furthermore, we adapt these policies to take performance aware dynamic reconfiguration decisions for running malleable applications and evaluate our approach on a distributed memory HPC system.

3 Background

This chapter describes in detail the adaptive parallel runtime system, i.e., Invasive MPI and the dynamic resource management system, .i.e. SLURM [6] used in this work for the development of a complete batch system capable of managing and scheduling both malleable and rigid applications. Furthermore, a programming model that eases the development of invasive applications for developers called Elastic Phase Oriented Programming Model (EPOP) is discussed.

3.1 Invasive MPI (IMPI)

One of the key features of the MPI standard [9] is that it is portable since the behavior of each MPI routine is clearly defined. Furthermore, it can be easily used for the development of parallel libraries [43] due to its modular design and support for communication contexts, virtual topologies, and datatypes. As a result of this, it is the most widely used standard for writing parallel applications on both homogeneous and heterogeneous distributed memory systems [17]. MPI is implemented as a library with the most common being MPICH [18], OpenMPI [19], and MVAPICH [44].

Dynamic process creation routines, i.e. `MPI_COMM_SPAWN` and `MPI_COMM_SPAWN_MULTIPLE` were added to the MPI standard from MPI-2.2. However, due to their limitations, such as high-performance overhead (see Section 1.1), the MPICH [18] library was extended by adding four routines to support dynamic process creation and reduction. The routines are an alternative for dynamic process support in the MPI standard [9] and are designed with latency hiding, minimal collective latency and ease of programming in mind [16]. They enable ease of efficient integration with resource mangers and allow efficient implementation in MPI communication libraries.

The workflow of an MPI program written using the IMPI routines is different from a standard MPI program. In the beginning, the MPI processes in an IMPI application are initialized by calling the proposed new initialization routine (see Section 3.1.1), instead of the standard `MPI_INIT` operation. Following this, the IMPI application continuously checks for adaptation instructions, i.e., expand or shrink by using the proposed new probe

operation (see Section 3.1.2). If an adaptation instruction is received, it starts only once the application reaches a safe location, i.e., at the beginning or end of a computation loop or phase. Adaptations are performed by creating adaptation windows by using the proposed new `MPI_COMM_ADAPT_BEGIN` (see Section 3.1.3) and `MPI_COMM_ADAPT_COMMIT` (see Section 3.1.4) operations. At the beginning of the adaptation phase, the application is provided with helper communicators which can be used for data redistribution. After the adaptation completes, the global communicator `MPI_COMM_WORLD` is modified permanently. Following this, the application resumes its computations.

There are several critical differences in semantics and behavior of the proposed new routines for dynamic process support [16] as compared to the routines in the MPI standard [9]. These differences are the reason why the proposed routines overcome the limitations of dynamic process support in the MPI standard. Firstly, the new and existing process groups are asynchronous to each other during expansion as compared to it being a blocking collective operation for the parent and child processes in the MPI standard. Secondly, the routines modify the global communicator `MPI_COMM_WORLD` and its process groups. Furthermore, an equal number of process groups are created for repeated adaptation operations. Thirdly, adaptation operations are initiated by the resource manager, i.e., malleable as compared to by the application, i.e., evolving in the MPI standard. This allows for efficient resource management and system utilization since the resource manager has information about the available resources in the system and the number of pending jobs. Finally, the resource manager can expand or shrink jobs with the proposed adaptation routines. Moreover, the resource manager can initiate shrink operations without group size restrictions.

3.1.1 IMPI Initialization Operation

The C interface for initializing MPI in adaptive mode is shown in Listing 3.1. It notifies the

```
int MPI_Init_adapt(int *argc, char **args, int *local_status);
```

Listing 3.1: IMPI initialization routine interface.

resource manger that the application will be adaptive and takes an additional `local_status` parameter as compared to the standard `MPI_Init` routine. The `local_status` parameter is used to distinguish between the processes which were created during the start of the application and processes which were created after an adaptation operation. It can have two values `MPI_ADAPT_STATUS_NEW` and `MPI_ADAPT_STATUS_JOINING`. The former implies that the process was created during the start of the application, and the latter implies that it was created due to an adaptation operation. Distinguishing between the processes is essential since the routine `MPI_COMM_ADAPT_BEGIN` (see Section 3.1.3) needs to be immediately called by the newly created process to take part in the adaptation window, and also by the preexisting

processes to continue the adaptation.

3.1.2 IMPI Probe Operation

Listing 3.2 shows the C interface which allows the preexisting processes to probe the resource

```
int MPI_Probe_adapt(int *pending_adaptation, int *local_status, MPI_Info *info);
```

Listing 3.2: IMPI probe routine interface.

manager for adaptation operations. The adaptation operations, i.e., expand or shrink, are always initiated by the resource manager, which is also responsible for setting the `pending_adaptation` argument. The parameter `pending_adaptation` can either return `MPI_ADAPT_TRUE` or `MPI_ADAPT_FALSE` to the preexisting process. If the value returned is `MPI_ADAPT_TRUE`, then it implies that the resource manager initiated an adaptation operation. The type of adaptation operation, i.e., expand or shrink, is decided by the value of the `local_status` parameter. It can have three values `MPI_ADAPT_STATUS_LEAVING`, `MPI_STATUS_STAYING` and `MPI_STATUS_JOINING`. `MPI_ADAPT_STATUS_LEAVING` implies a shrink operation while `MPI_STATUS_JOINING` implies an expand operation. The value of `local_status` is set by the IMPI initialization routine for newly created processes. The `MPI_Info` object is optional and can be be used to provide additional information from the resource manager.

3.1.3 Initiating Adaptation Operation

The C interface for the routine which initiates the adaptation window is shown in Listing 3.3.

```
int MPI_Comm_adapt_begin(MPI_Comm *intercomm, MPI Comm * new_comm_world, int *
    stayingcount, int *leavingcount, int *joiningcount);
```

Listing 3.3: IMPI adaptation window init routine interface.

In the case of an expand adaptation operation, `MPI_Comm_adapt_begin` is called immediately by the new processes after invoking the initialization routine. It is a blocking collective operation. Hence it needs to be called by all the new processes. Following this, the preexisting processes are notified about the adaptation operation. In the case of a shrink operation, it needs to be called by the preexisting processes. The proposed routine provides two communicators as outputs an intercommunicator and an intracommunicator. The intercommunicator is similar to the one produced in dynamic process creation in the MPI standard [9], while the intracommunicator (`new_comm_world`) provides an early view on the future `MPI_COMM_WORLD`

communicator. The newly created or leaving processes are added or removed from the intercommunicator depending upon the adaption operation, i.e., expand or shrink. The intracommunicator can be used by preexisting and the new processes to identify their future ranks. The proposed routine also provides counts for staying, leaving, and joining processes that can be utilized for repartitioning schemes. It also notifies the resource manager to prevent any new adaptation operations since nesting of multiple adaptation windows is not possible.

3.1.4 Finishing Adaptation Operation

Listing 3.4 shows the C interface for the routine which commits an adaptation window.

```
int MPI_Comm_adapt_commit( ) ;
```

Listing 3.4: IMPI adaptation window end routine interface.

It does not take any parameters and indicates the end of the adaption window. It modifies the `MPI_COMM_WORLD` communicator and sets it equal to the intracommunicator generated from the `MPI_Comm_adapt_begin` operation (see Section 3.1.3). The resource manager is also notified about the completion of the adaptation, so that new adapatation operations can be triggered.

3.2 Elastic Phase Oriented Programming Model (EPOP)

Writing applications that are resource-elastic for distributed memory systems has several challenges, two of the most crucial being implementation of data redistribution schemes and differentiating between preexisting and newly joining processes. To overcome these challenges and ease the process of software development for developers, EPOP was designed. EPOP is an abstraction over IMPI and provides a driver program (see Section 3.2.1) which automatically invokes the IMPI routines (see Section 3.1.1-3.1.4). The EPOP programming model allows developers to modularize their code and divide the computation into different phases [16, 45].

The EPOP programming model is based on three design principles Amdahl's law [46], division of a parallel application into logical phases and combination of an adaptive parallel runtime (IMPI) and an adaptive resource manager (SLURM) for making applications resource-elastic. According to Amdahl's law for strong scaling parallel applications, the speedup of the application is limited by its serial part. A typical parallel application has three phases initialization, computation, and finalization. The initialization and finalization phases are

typically sequential where the application does input-output (I/O) operations, distributes the data, and writes the results. The computation phase generally has the most potential for optimization and can benefit from resource changes. Therefore, for resource-elastic applications, all sequential phases should be identified and ignored for resource adjustments and performance analyses. Furthermore, resource-elastic applications have two control flows to account for dynamic resource changes at runtime. It is the responsibility of the programmer to structure the application into multiple phases logically, in order to handle the addition or removal of resources at runtime using the proposed IMPI routines (see Section 3.1). The EPOP programming model simplifies this process and makes the division of applications into phases easier. Moreover, it can be used for the development of both resource and non-resource elastic applications.

Using the EPOP programming model, the application developer can divide the application into different phases by using abstraction routines. The different phases supported by EPOP are init, elastic, branch, and rigid. The EPOP driver (see Section 3.2.1) provides helper routines that can be used by the developer to create a vector of phases at the beginning of the application [45]. During execution, the driver iterates through the vector in the order declared by the programmer.

The Init phase must always be the first phase in the vector of phases for all EPOP applications. The developer can use this phase for data allocation and initialization of the application. Every EPOP application should have only one init phase, which is called by the driver exactly once for all the processes in the application. It is also called by the EPOP driver for newly created processes in case of an expand operation.

The compute-intensive phase in an application that must support adaptation operations constitutes the elastic phase. In contrast to an initialization phase, an application can have multiple elastic phases. Each elastic phase must contain three components a computational code block, a variable specifying the iteration count for the computational code block, and an adaptation block [45]. The iteration count is necessary so that the application can exit from the current phase. Each type of process, i.e., preexisting, joining, or leaving, must have a separate adaptation block. If the resource manager triggers a shrink or an expand operation, then the EPOP driver (see Section 3.2.1) executes the appropriate adaptation block depending on the process. Moreover, the EPOP driver collects performance data of MPI calls in this phase and provides it to the resource manager to enable performance-aware scheduling decisions.

The developer can identify and mark the different phases of an application as rigid, which do not require the need to support adaptation operations and that cannot benefit from additional resources. A typical example of a rigid phase is when the application is finishing and writing the results. A developer can write a non-resource elastic application using EPOP by utilizing only the init and the rigid phases. An iteration count for the rigid phase can also be specified by the developer.

The final phase, supported by the EPOP programming model is the branch phase. This phase can be used to change the control flow of the EPOP driver, i.e., how it iterates across the vector of phases initialized by the programmer at the beginning. The branch phase can be used for both backward and forward jumps in the vector of phases. The next phase to be executed is determined by the return value at the end of the branch phase.

3.2.1 EPOP Driver

The main program responsible for executing the resource-elastic EPOP based application along with managing its control flow and data transfers between multiple phases (see Section 3.2) is the EPOP driver. The current implementation [45] provides several functionalities such as probing for resource changes, creation of adaptation windows, profiling and performance modeling of elastic applications, and pausing an application.

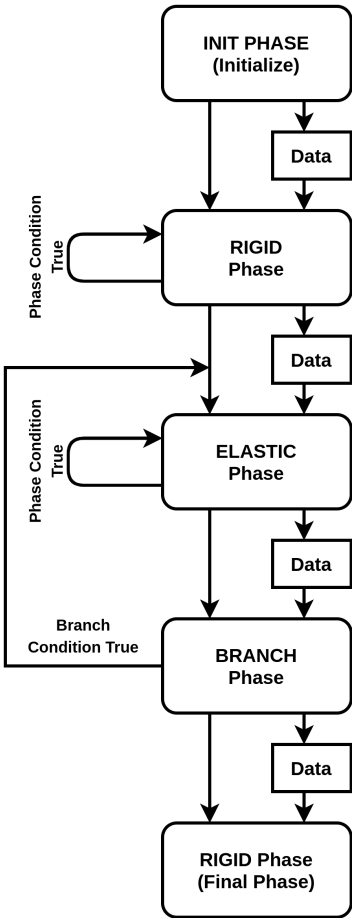


Figure 3.1: Data and Control Flow for an elastic EPOP application solving the 2-D heat equation [45].

The EPOP driver consists of two parts the application part and the EPOP master. The application portion is primarily responsible for the control flow of the EPOP program, i.e., execution of phases. On the other hand, the EPOP master is responsible for the performance profiling of the EPOP program and the communication between the EPOP driver and the invasive resource manager (IRM). Profiling of applications, specifically for MPI routines can be enabled by the user. If it is enabled, metrics such as the average MPI time of a phase, the time spent in each MPI call, the computation time of a phase, and the time of adaptive operations are computed and written to a log file by the EPOP driver.

An EPOP program is a logical collection of phases executed in a particular order. The vector of phases defined by the programmer in the beginning and the branch phase (see Section 3.2) in a program determines the order of execution of phases. Figure 3.1 shows the control and data flow of an elastic EPOP application solving the 2-D heat equation using the stencil based Jacobi Iteration. The init phase initializes the data required by the application. The first rigid phase creates a 2-D Cartesian grid and a virtual topology of MPI processes needed for solving the heat equation. The distribution of data among the MPI processes, along with computing the solution to the heat equation using the Jacobi Iteration method, is done in the elastic phase. The branch phase is used for recomputing the solution, and the final rigid phase writes the solution to a file. It is important to note that data and logical phases are entirely separated, as shown in Figure 3.1. All data required by the program is stored in a data block and passed between phases according to the control-flow of the program.

The MPI process with rank 0 is the EPOP master. It provides the IRM with performance data of the running applications to enable performance-aware adaptation decisions. The EPOP driver measures the total time spent by each MPI process in each phase across all iterations. Furthermore, the total time spent in MPI calls is also measured. This is done by adding routines to IMPI (see Section 3.1), which allow performance measurement of MPI calls. Each MPI process calculates its MPI to phase time ratio, which is then aggregated using a reduction operation and stored in the EPOP master. Following this, the average ratio across all processes is forwarded to the IRM.

The EPOP driver provides two strategies for sending the information to the IRM, namely polling and data push. In the polling strategy, the resource manager requests EPOP driver for performance data, during a scheduling pass. The data is computed and sent to the resource manager by the EPOP master before the next scheduling pass. This is the strategy used for most resource-elastic EPOP applications. On the other hand, in the data push strategy, the data is computed and sent to the resource manager, if a change in performance is detected. This reduces the decision-making time of the scheduler since a resource adaptation operation can be issued in the first scheduler pass. The EPOP master is also responsible for sending vital information such as job id and its IP address to the resource manager during the start of the EPOP application. The received data is stored in a pointer to the `job_record` structure in the resource manager.

3.3 Simple Linux Utility for Resource Management (SLURM)

The most critical component for sharing resources and scheduling applications on a shared memory system is the operating system. The operating system is responsible for managing the access to input/output (I/O) devices, network, memory, audio, and video devices etc. Operating systems have an in-built scheduler that is responsible for the placement of applications and is generally optimized for throughput, i.e., time-sharing. Distributed memory HPC systems consist of multiple compute nodes, i.e., shared memory systems interconnected by a high-performance network, with optimized communication latencies and bandwidth. The compute nodes typically run an operating system, mostly Linux, with the scheduler operating in time-sharing mode. Due to the absence of a production operating system that supports distributed memory, a Resource and Job Management System (RJMS) is a critical component in HPC systems.

The Simple Linux Utility for Resource Management (SLURM) [6] is an example of such a RJMS software and is used in several of the fastest general purpose HPC systems in the world [4]. SLURM is an open-source, scalable, and fault-tolerant RJMS software currently being developed by SchedMD. It is designed to handle several failures such as node crashes, termination of tasks without the running job being affected. It is not a comprehensive cluster administration and monitoring system and does not utilize the current state of the compute nodes. Furthermore, it only supports resource and workload management for a single cluster and is not a meta-batch system like Globus [47]. It provides a simple FCFS based batch scheduler by default, which operates in the space-sharing mode, i..e, submitted jobs by the users get exclusive access to resources.

Name	Functionality
`scancel`	Cancelling queued jobs and terminating running jobs, job steps.
`scontrol`	Provides capabilities for SLURM administration by the super user.
`squeue`	Current state of jobs submitted to SLURM, i.e., RUN, PENDING.
`sinfo`	Current state of SLURM partition and compute nodes, i.e., IDLE, DRAIN.
`sstat`	Status information of running jobs.
`sacct`	Accounting data of current jobs in the system.

Table 3.1: Command Line Utilities and their functionalities provided by SLURM [6].

SLURM consists of several binaries that provide different functionalities such as monitoring the machine status and partition info, submission and canceling of jobs, monitoring the job queues, etc. The binaries are designed to be highly scalable and have a threaded design. Furthermore, SLURM provides a generic plugin interface to extend functionalities such as scheduling, topology, MPI/PMI support, database support, etc. All binaries share a common configuration file called `slurm.conf`, which is utilized for setting several parameters in SLURM, such as the scheduling plugin to be used, scheduling plugin tick, nodes in the

partition, etc. The important SLURM binaries include the SLURM controller SLURMCTLD, node daemons SLURMD, step daemons SLURMSTEPD, the interactive parallel job launcher SRUN and the job launcher in batch mode SBATCH. Apart from this, SLURM provides several simple command-line utilities which allows users to interact with it. Their names and functionalities are shown in Table 3.1.

3.3.1 SLURM Controller SLURMCTLD

The SLURM controller or SLURMCTLD is the only centralized component in SLURM and contains most of the state information. It is a multi-threaded application and includes independent locks for reading and writing operations to enhance scalability. It reads the SLURM configuration file slurm.conf when it starts and periodically saves the current state information to disk for fault tolerance. It consists of three components, namely the node, partition, and the job manager.

The node manager is responsible for monitoring the state of each compute node in the system. The state information is received from the SLURM node daemon (see Section 3.3.2) asynchronously through periodic polling. The partition manager is responsible for grouping compute nodes into non-overlapping sets called as partitions. It is also responsible for allocating nodes to jobs depending on the user configurations and the partition state.

The job manager is responsible for accepting and storing user-submitted jobs in a priority-ordered job queue. It runs on a separate thread and is executed periodically or on specific events such as job completion, job submission, etc. If a simple FCFS based scheduling policy is used, then the earliest job in the job queue is considered for scheduling. All the other jobs in the queue wait until enough resources are available for the highest priority job to start execution. The job manager is also responsible for performing cleanup when a job completes.

3.3.2 SLURM Node Daemon SLURMD

The SLURM Node Daemon SLURMD is a multi-threaded daemon running on each compute node in the system. It periodically communicates with the SLURM controller and exchanges node and job status information. It must always be executed as a super user (root) since it is responsible for job initialization for other users. It consists of five different components, namely Machine and Job Status Services, Remote Execution, Stream Copy Service, and Job Control.

Machine and Job Status Services are responsible for asynchronous communication of

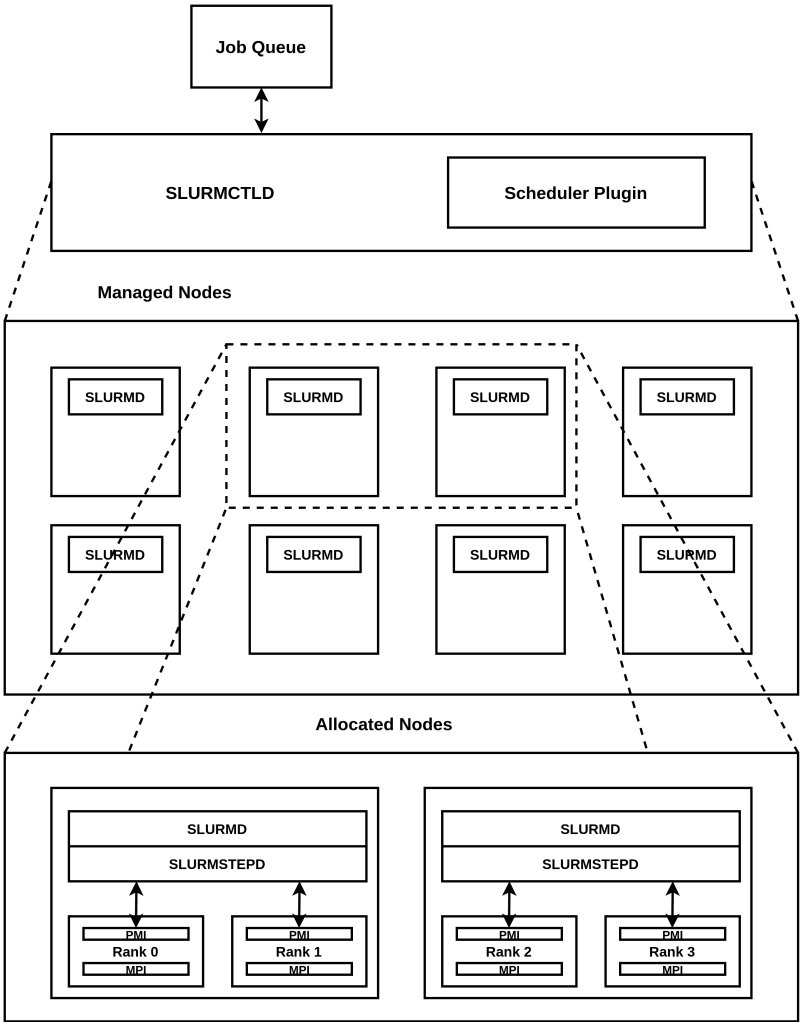


Figure 3.2: Overview of the interactions between the different SLURM binaries.

machine and job state information with the SLURM controller. Remote execution is responsible for start-up, management, and clean-up of parallel processes. This may include the execution of a prolog program, setting effective and real userids, allocation of interconnect resources, setting of environment variables and the working directory, initializing standard input-output, managing process groups, the execution of an epilog program, etc [6]. The `stderr`, `stdout` and `stdin` of remote tasks is handled by the Stream Copy Service. Finally, Job Control is responsible for propagating signals to a group of locally managed processes through asynchronous interaction with the remote execution environment.

Apart from this, `SLURMD` is also responsible for starting an additional daemon called the `SLURMSTEPD`. `SLURMSTEPD` is accountable for managing the subset of application processes running on that node.

3.3.3 SLURM SRUN

The binary SRUN is responsible for allocating resources, adding user jobs to the job queue, and initiating job steps. The user can also run a job interactively using the --interactive command-line option in SRUN. In this case, SRUN communicates with SLURMDs for job initiation, stdout and stderr information, and to respond to the signals from the user. The interactive mode of SRUN can also be used to create a bash shell with access to a set of allocated resources.

Figure 3.2 shows the interaction between the different binaries SLURMCTLD, SLURMD, SLURMSTEPD and SRUN in a distributed memory HPC system. It also shows the position of the different binaries in an HPC system and MPI and PMI libraries linked to the MPI processes. The allocated nodes are obtained by SRUN from the set of managed nodes as shown in Figure 3.2.

Apart from SRUN, SLURM also provides a binary called SBATCH which can be used by the users for submitting jobs defined in a batch script. The control flow of SBATCH is different from SRUN and is discussed in more detail in Section 4.1 .

3.3.4 Invasive Resource Manger (IRM)

The efficient scheduling of resource-elastic applications requires two main components, a resource-aware scheduling strategy, and an adaptive resource manager. Since an adaptive application can change its resources at runtime due to expand or shrink operations, the scheduling strategy should maximize performance metrics such as throughput, system utilization, and minimize metrics such as response and waiting times for HPC systems. Furthermore, the scheduling strategy should account for the performance of the running application when taking expand or shrink decisions. This can be accomplished using heuristic criterions such as MPI to compute time (MTCT) or MPI to phase time. After the scheduling strategy has decided an expand or shrink adaptation operation, the adaptive resource manager should support dynamic reconfiguration of the running system. Moreover, dynamic reconfiguration should account for consistency across the different distributed components in an HPC system and the running process groups. No resource managers supports these features for the MPI runtime today. Towards, this SLURM was extended to support dynamic reconfiguration of resource-elastic IMPI (see Section 3.1) or EPOP (see Section 3.2) based applications [16, 45]. SLURM was selected due to its maturity and widespread usage across general purpose HPC systems.

Since SLURM [6] by default only supports static resource allocations, a large number of changes were introduced to different SLURM binaries to support dynamic resource

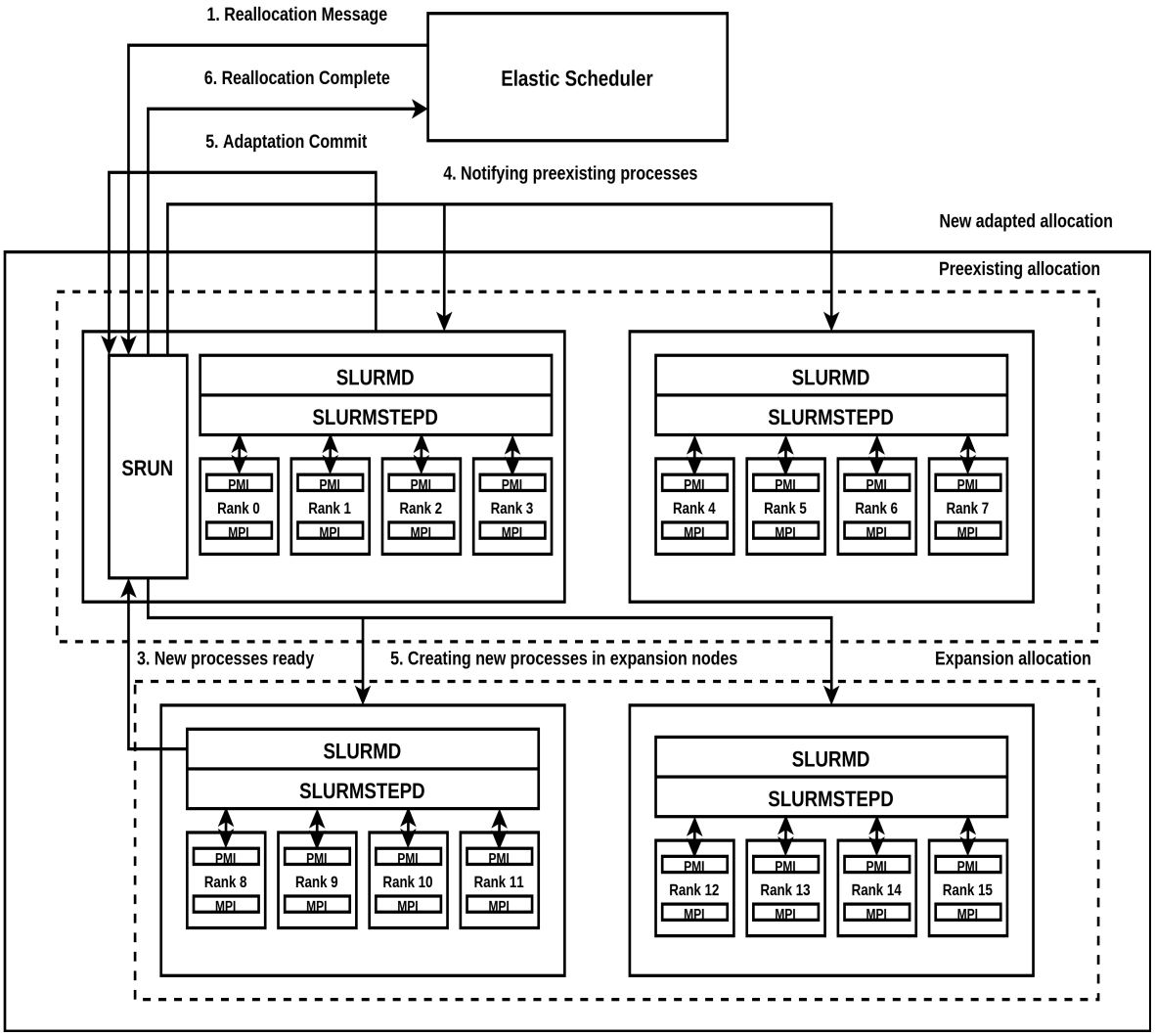


Figure 3.3: Overview of the interactions between SLURM and IMPI (MPICH).

configuration at runtime. Furthermore, extensions were added to support close integration with the MPI runtime and to support the proposed routines for dynamic process management (see Section 3.1). This is because the addition or removal of processes at runtime must be done in consideration with MPI.

Figure 3.3 shows the interactions between the different resource manager components and Invasive MPI. It shows the different SLURM components `Elastic Runtime Scheduler`, `SRUN`, `SLURMD` and `SLURMSTEPD` along with the two linked libraries, i.e., PMI and MPI inside MPI processes. In contrast to the default SLURM architecture as shown in Figure 3.2, the `SLURMCTLD` (see Section 3.3.1) is replaced by the `Elastic Runtime Scheduler` (`irtsched`).

To accommodate dynamic resource changes at runtime, a new SLURM message called `slurm_reallocation_message` was added. Furthermore, appropriate handlers were created and added to `SRUN` (see Section 3.3.3), since the creation of processes is only possible through the launcher. The message contains information such as the number of tasks to create or destroy, nodes to invade or retreat from, and other metadata required for reallocation. This message is created for a running application and sent by the `Elastic Runtime Scheduler` to `SRUN` depending upon the scheduling strategy used in the scheduling plugin.

The Node Daemon `SLURMD` (see Section 3.3.2) was modified to support the `MPI_Probe_adapt` (see Section 3.1.2) and `MPI_Comm_adapt_begin` (see Section 3.1.3) IMPI routines. The node local MPI processes are created and managed by the `SLURMSTEPD` daemon. Similar to the default SLURM architecture, each compute node has only one running instance of `SLURMD`. `SRUN` was extended to send and receive control specific information about compute nodes and MPI processes from `SLURMD`.

Dynamic reconfiguration of an application at runtime is a six-step process. It includes the generation of a reallocation message, the creation of new processes in the expansion nodes, notifying if the new processes are ready, notifying preexisting processes, committing an adaptation, and the generation of a reallocation complete message. The six different messages, along with their interaction with SLURM resource components and IMPI, are shown in Figure 3.3. In the first step, an adaptation decision is taken by the `Elastic Runtime Scheduler`. Following this, a reallocation message is generated and applied to the job step's `SRUN` instance. If the adaptation operation is an expansion, then a launch command is sent by `SRUN` to the `SLURMD` daemons of all expansion nodes. The `SLURMD` daemons in the expansion nodes notify `SRUN` after the newly created processes are ready at the `MPI_Comm_adapt_begin` (see Section 3.1.3) IMPI routine.

If the `Elastic Runtime Scheduler` decides to shrink an application, `SRUN` sends instructions to `SLURMD` daemons of nodes which are to be retreated from. Each `SLURMD` calls `MPI_Probe_adapt` (see Section 3.1.2) and updates the metadata for its local proceses. After an expand or shrink operation completes, `SRUN` is notified by the leader node. Following this, `SRUN` notifies the `Elastic Runtime Scheduler` that the adaptation was completed by sending a reallocation complete message. Finally, the `Elastic Runtime Scheduler` sends `SRUN` the updated job credentials. This design allows for simultaneous expansion and reduction for different running applications [16]. The only limitation is that the original node on which the `SRUN` instance is running must always be a part of the adapation operation and cannot be migrated.

4 Batch Scheduling and Resource Management Infrastructure

A typical ecosystem of a distributed memory HPC system consists of login nodes and several compute nodes. The compute nodes are interconnected via a high-performance network in a network topology such as a pruned fat tree, dragonfly [48], etc. The communication between nodes is optimized for high throughput and low latency using an external network interface card supporting a computer-networking communications standard such as Infiniband [49]. The users of the HPC system have access to it through login nodes, which usually have the same configuration and architecture as the compute nodes.

The users use the login nodes for compiling their applications and submitting them for execution. Since thousands of users share an HPC system, it also has a resource and job management system (RJMS). The main component of RJMS responsible for managing the compute resources among the users and executing user's applications is called the Batch Scheduler. This chapter describes the batch scheduling mechanism in SLURM [6], particularly the resource management aspect and its control flow in detail. Furthermore, the extensions to SLURM to support resource-aware batch scheduling are described.

4.1 Initiating Queued Jobs in SLURM

The Batch scheduler in SLURM, by default, operates in the space-sharing mode. This means that the jobs executing on the compute nodes are granted exclusive access to resources. Exclusive access to resources is required since minimizing interference between jobs is an essential criterion in HPC systems. SLURM provides a utility tool called `sbatch`, which can be used for submitting applications for execution in batch mode by the users. The user writes a job script specifying several job-specific options such as job name, wall clock time, partition, stderr and stdout directories and filenames, number of nodes, number of tasks per node, etc. The options supported by `sbatch` can be seen by using the `--help` flag.

The job script also contains a `srun` command for launching the application. `srun` is a binary provided by SLURM for launching the processes of an application on a set of allocated nodes

```
#SBATCH --job-name sample_batch_script
#SBATCH --output output.txt
#SBATCH --error error.txt
#SBATCH --time=00:15:00
#SBATCH --partition=test #Selecting the different queues
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=48
#SBATCH --contigous
#SBATCH --get-user-env

srun -n 2 test_application #job step 1
srun -n 2 test_hello_world #job step 2
```

Listing 4.1: Example of a Batch Script.

as described in Section 3.3.3. A job script can have multiple `srun` commands, each of which is called a job step. The job script is used as an input to the `sbatch` command-line utility. An example of a batch script is shown Listing 4.1. The order of executions of job steps depends upon the order specified in the batch script. In the example shown in 4.1 the second job step is only executed after the first completes. The `--contigous` `sbatch` option ensures that SLURM allocates continuous set of nodes for the job and the option `--get-user-env` loads the local user enviornment from the cluster.

After the user submits the job through `sbatch`, it is placed in a job queue and not immediately started. It is later executed without any interactions from the user. By default, SLURM considers all jobs as rigid, i.e., resource allocation remains the same during the entire execution of the application. This work extends `sbatch` to support the submission of resource-elastic jobs to the job queue (see Section 4.2). The starting time of the jobs is not guaranteed and depends upon several factors such as resource availability, priority, duration, requested partition, etc and the scheduling strategy used.

The scalable workload manager SLURM is a collection of binaries and provides several command-line utilities for user interaction. Some of the essential binaries and utilities are described in Section 3.3. For communication between daemons and command-line utilities, SLURM uses Remote Procedure Calls (RPCs). RPCs are an important technique for building scalable distributed, client-server based applications. It allows two or more processes communicating with each other to be on different systems, with a network interconnection amongst them. Furthermore, the called procedure according to the sent RPC request need not be on the same address space as the calling procedure. SLURM uses the TCP/IP protocol and network sockets for sending RPC requests between daemons and command-line utilities. The `slurm_reallocation_message` described in Section 3.3.4 is also a RPC request sent from the `Elastic Runtime Scheduler` to the application's `srun` instance.

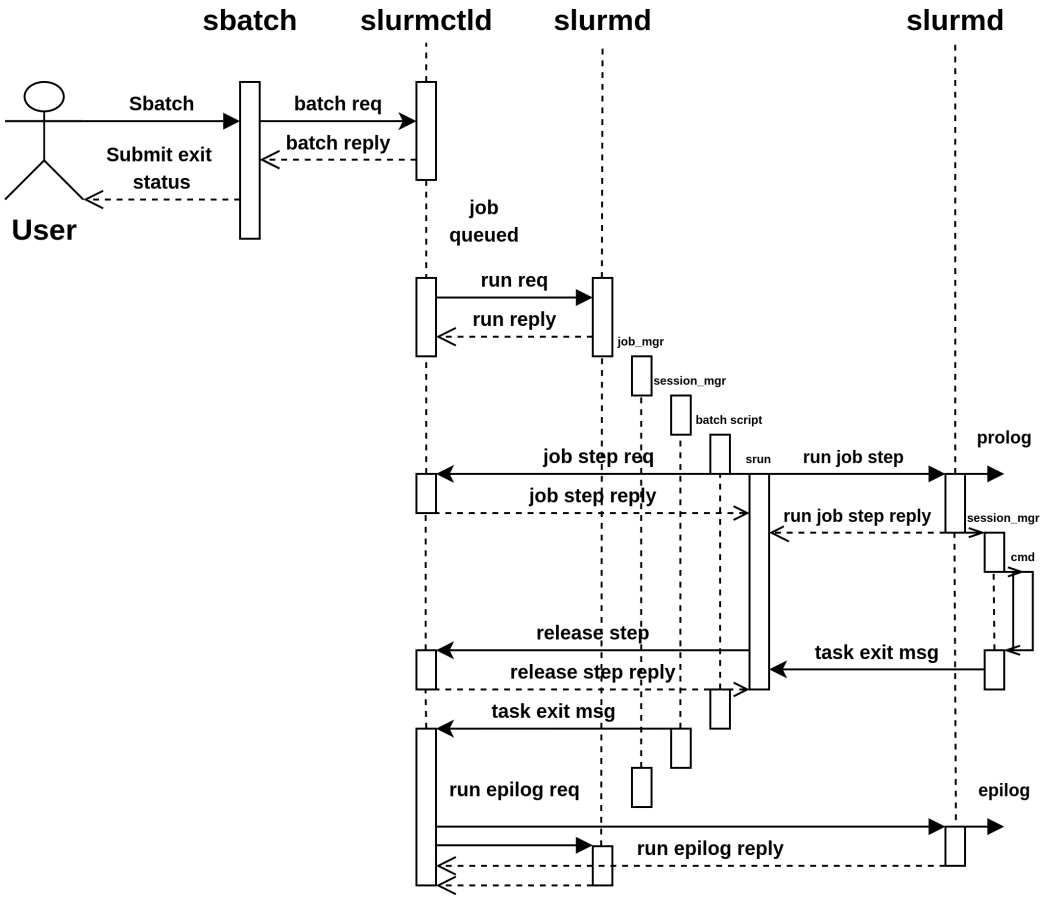


Figure 4.1: Control flow of queued job initiation using `sbatch` in SLURM [6].

In Figure 4.1, an interactive sequence diagram representing the control-flow and the communication between the different SLURM daemons and command-line utilities (see Section 3.3) is shown. Generally, `sbatch` is present on the login node and the `slurmctld` and `slurmd` node daemons are runnning on seperate compute nodes. Initially, the user writes the job script and submits it using `sbatch` on the login node, as described above. `sbatch` generates an RPC request message called `REQUEST_SUBMIT_BATCH_JOB` and sends it to the `slurmctld`. If the job was successfully submitted, `slurmctld` responds with the message `RESPONSE_SUBMIT_BATCH_JOB`. Otherwise, an error code is sent. The `slurmctld` identifies the stdin, stdout and stderr files, current working directory and the user environment for the job. On successful submission, `sbatch` exits after displaying the submitted job id.

The submitted job is added to the priority-ordered job queue maintained by the `slurmctld`. If the requested resources are available and the job has a high enough priority, it is granted resources by the `slurmctld`. Following this, the `slurmctld` generates an RPC request message called `REQUEST_BATCH_JOB_LAUNCH` and sends it to the `slurmd` node daemon of the first

allocated compute node. slurmd responds to the request message and initiates the job and the session managers and the submitted user script (see Listing 4.1).

The submitted user script can have one or multiple srun commands, i.e., job steps as shown in Listing 4.1. A srun is launched on all or some of the allocated nodes, depending upon the job step. However, it is important to note that in the case of queued job execution, the srun instance always runs on the first allocated compute node. After this, an RPC request message called REQUEST_JOB_ALLOCATION_INFO_LITE is generated by the running srun instance and sent to the slurmctld. The request message is used for retrieving specific information such as node count, node list, partition, etc. for the job step. On the successful response, srun generates the RPC request message REQUEST_JOB_STEP_CREATE and sends it to the controller.

After RESPONSE_JOB_STEP_CREATE is received by srun, the job step is initiated. For launching the associated tasks with the job step, srun generates the message REQUEST_LAUNCH_TASKS and sends it to the appropriate slurmd daemons. The slurmd daemons running on compute nodes initiate the slurmstepd daemon (see Section 3.3.2), which launches the node-local tasks. If the tasks are successfully initiated RESPONSE_LAUNCH_TASKS is send by the slurmd daemons to the running srun instance.

After the tasks in a job step have completed their work, they exit and send the RPC request message MESSAGE_TASK_EXIT to the running srun instance. The requests are sometimes aggregated and sent together to srun. Once all tasks have exited, srun exits and notifies the slurmctld by generating and sending the RPC request message SRUN_JOB_COMPLETE. However, the submitted job script by the user continues execution. If the job script has more job steps, then another instance of srun is launched, and the same steps follow. If there are no more job steps present, then the thread executing the job script sends an exit task message to the slurmctld. The slurmctld sends an RPC request message called REQUEST_TERMINATE_JOB to all the allocated compute nodes and acknowledges job completion. Following this, job epilog is launched on all the allocated compute nodes. After successful completion of job epilog the slurmctld, is notified by the slurmd node daemons. Finally, the allocation is released by the slurmctld, and the nodes can be used for running other applications.

4.2 Initiating Resource-Elastic Queued Jobs in SLURM

The current implementation of the Invasive Resource Manager, as described in Section 3.3.4, supports the execution of resource-elastic applications only through launching interactive jobs using srun. The control flow of launching interactive applications using srun in SLURM as shown in Figure 4.2 has several differences as compared to the control flow of initiating a queued job as shown in Figure 4.1. The RPC requests used for communication between the

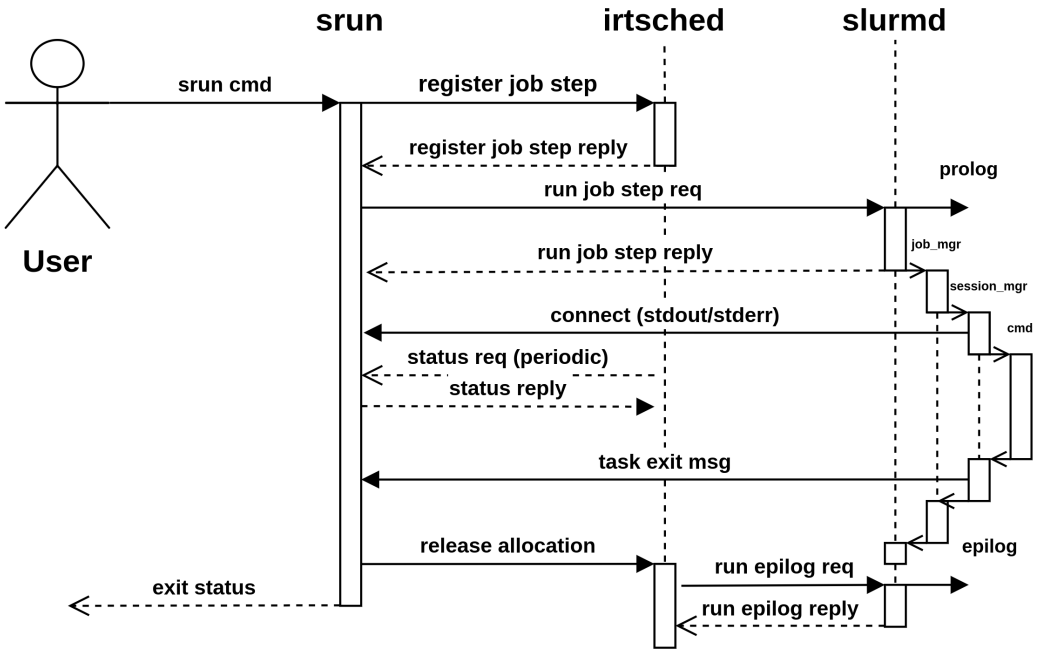


Figure 4.2: Control flow of interactive job initiation using `srun` in SLURM [6].

daemons and the command-line utilities are different. For instance, for resource allocation, `srun` sends a blocking RPC request message called `REQUEST_RESOURCE_ALLOCATION` to the `Elastic runtime scheduler` (irtsched). The `irtsched` is a modified version of `slurmctld` for supporting the execution of resource-elastic applications in SLURM (see Section 3.3.4). `srun` pauses its execution until the requested resources become available. Whenever the requested resources by the user are available, the `irtsched` responds to `srun` with the job credential, allocated list of nodes, CPUs per node, etc. Following this, `srun` continues its execution. Another important difference is that in the case of interactive job execution, the `srun` instance is always running on the login node. On the other hand, in the case of queued job execution, the `srun` instance always runs on the first allocated compute node as described in Section 4.1. This work extends the Invasive Resource Manager for queued initiation of resource-elastic applications using `sbatch`.

Figure 4.3 shows an overview of the different components and the interactions between them to enable the execution and scheduling of resource-elastic applications using `sbatch` in SLURM. The traditional `slurmctld` (see Section 3.3.1) is replaced by two different components the `Elastic Batch Sceduler` (EBS) and the `Elastic Runtime Scheduler` (ERS). The different functionalities of ERS, i.e., `irtsched` are described in Section 3.3.4. While the previous implementation of IRM did not maintain any queues, since the applications were launched using a blocking `srun` command as described above. In this work, the EBS maintains two separate priority-ordered queues, i.e., rigid and elastic, for storing the list of jobs submitted by the users.

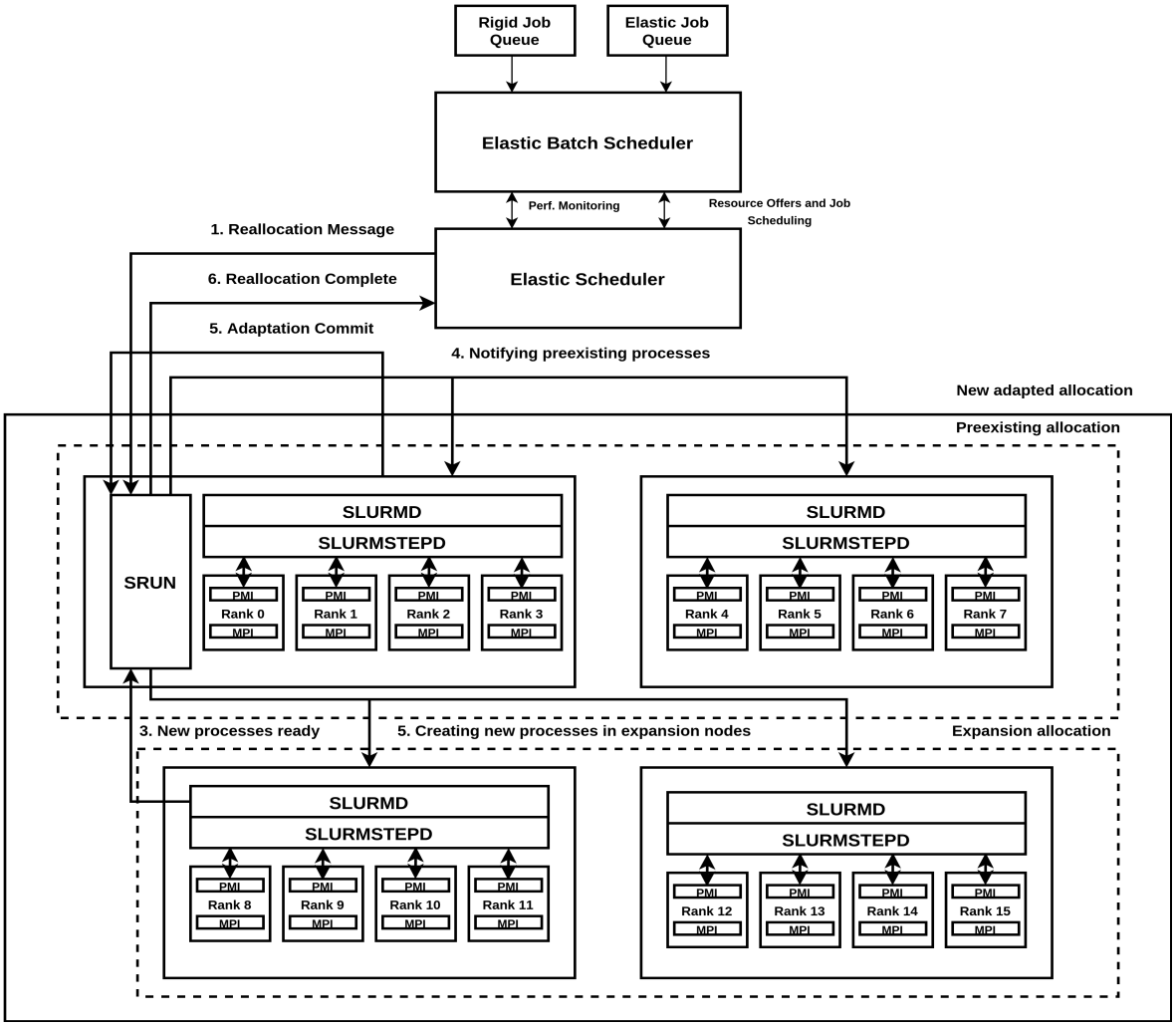


Figure 4.3: Overview of the interactions between the different SLURM binaries.

The EBS is primarily a binary which is dynamically loaded by the ERS during startup. It runs in a separate thread inside the ERS. The EBS and ERS closely interact with each other for efficient scheduling of applications. The EBS has a global knowledge about the available resources in the system at any given point of time and can take dynamic reconfiguration decisions at runtime for the currently running resource-elastic applications to maximize performance metrics such as throughput, system utilization etc. and minimize metrics such as waiting and response times, for jobs (see Section 7.2). The interaction between the EBS and the ERS depends on different situations and occurs on different events described below:

- On every execution of the scheduling loop. This depends upon the value of `SchedulerTick` variable which can be set in the `slurm.conf` file shared by all SLURM binaries (see Section 3.3).

- Whenever a new job is added to the global system priorty-ordered queues, i.e., rigid and elastic.

- Whenever a job completes its execution and terminates.

Whenever any of the above described events occur, the EBS takes decisions to expand or shrink currently running applications depending upon the jobs waiting in the queues and the used scheduling strategy (see Section 5.3.1). This implies that a decision to start a waiting job depends upon the evolution of resource allocations on the set of running resource-elastic applications. With two separate queues, the EBS is capable of scheduling and launching both rigid and malleable applications.

The scalable workload manager SLURM [6] was extended to support the communication between different SLURM components and resource-elastic applications written using the Invasive MPI library [16]. The IMPI library proposes a set of routines for dynamic process creation in MPI, as described in Section 3.1. The main difference between a resource-elastic application and a standard rigid MPI application is the usage of the initialization routine `MPI_Init_Adapt` (see Section 3.1.1) as compared to the standard MPI initialization operation `MPI_Init`. Every job in SLURM has a pointer to a `job_record` structure, which contains all the important information about a job such as the number of allocated nodes, start time, ports for communication with different daemons, partition-info, etc. When a resource-elastic application starts execution and calls `MPI_Init_Adapt`, a flag variable called `is_elastic` is set in the pointer to that job record's structure. This is how the invasive resource manager distinguishes between an elastic and a rigid application. However, the elastic batch scheduler needs to determine the type of the applications, i.e., rigid or elastic before its execution. This is required for proper placement of jobs in the priority-ordered queues and for dynamic adaptation decisions of running applications. Towards, this the `sbatch` binary was extended to allow the submission and initiation of elastic jobs.

Listing 4.2 shows an example of a batch script containing the added `sbatch` options for submitting elastic applications. Particularly, the options `--min-nodes-invasic`, `--max-nodes-invasic`, `--node-constraints` and `--consider-rigid` were added to `sbatch`. For submitting resource-elastic applications, the user must specify the minimum and maximum nodes required for the application. This is done because the users have the maximum knowledge about their applications and its requirements. If the above two options are specified, then the application is added to the elastic priority-ordered queue shown in Figure 4.3. Otherwise, the application is added to the rigid priority-ordered job queue. The priority of the submitted applications depends upon which job was submitted first, i.e., FCFS. Furthermore, any expansion adaptation operation will not allocate more than the specified number of maximum nodes to the job. Similarly, any shrink adaptation operation will not reduce the number of allocated nodes to less than the minimum. The current implementation of the extended invasive resource manager with the elastic batch scheduler done in this work only

```
#SBATCH --job-name simple_elastic_application
#SBATCH --output output.txt
#SBATCH --error error.txt
#SBATCH --time=00:15:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=48
#SBATCH --min-nodes-invasic=1
#SBATCH --max-nodes-invasic=5
#SBATCH --node-constraints="odd" #supported values pof2, even, odd, ncube
#SBATCH --consider-rigid=1

srun test_impi_application #job step 1
```

Listing 4.2: Extensions to `sbatch` for submission of resource-elastic queued jobs.

supports adaptation operations at a node level (see Section 4.4). It is important to note that all resource-elastic applications are always launched with the number of nodes specified, i.e. the value of `--nodes` parameter in the batch script (see Listing 4.2) and later expanded or reduced. The value of the `--nodes` parameter can be different from the minimum number of nodes that the job can be reduced to.

The users can specify constraints on the allocated nodes for dynamic reconfiguration decisions using the `--node-constraints` option. The supported options are power-of-two (`pof2`), `even`, `odd`, and cubic number of nodes (`ncube`). This is done to support the dynamic reconfiguration of applications that have different constraints on the number of processes (in our case the number of nodes) for their execution. For example, the application solving the 2-D heat equation using the Jacobi iteration (see Section 3.2.1) requires an even number of processes for execution. On the other hand, the LULESH [50] application requires a cubic number of processes for execution. If the user specifies `--node-constraints=pof2` in the submitted batch script, then it is taken into account for reconfiguration decisions by the elastic batch scheduler. On expansion, the application is allocated the highest power-of-two nodes less than or equal to the current idle nodes. If the required number of nodes are not available then it is not expanded. On a shrink operation, the number of nodes of the running application are reduced to the next lowest power-of-two of the currently allocated compute nodes. Other options for constraints, as shown in Listing 4.2, behave analogously.

The users also have the option of disabling any dynamic adaptation operations for resource-elastic applications by using the `--consider-rigid` option. If the consider-rigid flag is set in the submitted batch script by the user, then the EBS places the job in the priority-ordered rigid job queue. Furthermore, the application is never considered for any expand or shrink operations. This was done to increase user convenience and to allow the testing of different scheduling strategies. All the added options and their functionalities can be seen by using

the `--help` option in `sbatch`. To incorporate the changes to `sbatch`, the RPC request message `REQUEST_SUBMIT_BATCH_JOB`, sent by `sbatch` to the `irtsched` on the submission of a batch job (see Section 4.1) was modified. The current implementation of EBS supports the execution of resource-elastic applications submitted using `sbatch`, which have only one job step, i.e., single `srun` command. An example of this is shown in Listing 4.2.

4.3 Enabling Dynamic Resource Changes for Queued Elastic Jobs

One of the most important functions of the implemented Elastic Batch Scheduler (EBS) is to take dynamic reconfiguration decisions for running resource-elastic applications. After an expand or shrink decision has been taken by the EBS, the elastic runtime scheduler (ERS) needs to enforce the operation for the running application, while maintaining consistency in the system. A method to support dynamic reconfiguration of the system at runtime by using a RPC request message called `srun_realloc_msg` was introduced and implemented in [16].

In the method introduced in [16], the ERS decides to grow or shrink a running elastic application and then generate and send an RPC request message to the interactively running `srun` instance. In the case of an expansion operation, the message contains information such as the number of new tasks to create and the nodes to create them on. For a shrink operation, the message contains the number of preexisting tasks to destroy and the nodes to destroy them from. Furthermore, the message contains other relevant information such as updated job credentials, the updated process to node mappings, etc. If the message is received successfully by `srun`, a reallocation handler is executed. The reallocation handler is responsible for changing the job step context and launching new processes in the case of expansion and destroying the preexisting processes in the case of a shrink operation. It is important to note that the current implementation does not allow mixed adaptation operations, i.e., the RPC request message must either be a pure expansion operation or a pure shrink operation. The steps for dynamic reconfiguration of resources using the EBS shown in Figure 4.3 is similar to the steps shown in Figure 3.3. The extensions to the invasive resource manager for supporting `srun_realloc_msg` and the steps for dynamic reconfiguration are described in Section 3.3.4.

The communication between the running interactive `srun` instance and the ERS is done through a port for client communications. The `srun` instance launches a slurm allocation message thread on this port, which continuously runs in the background. The message thread contains essential callback functions such as the reallocation handler, job completion handler, timeout handler, etc.. This port is unique for each running job and is present in every job's pointer to the `job_record` structure. The value of this port is assigned during the allocation of resources after `srun` sends a blocking RPC request message to the ERS as described in Section

4.2. Since the control-flow for launcing interactive applications and queued applications is completely different in SLURM as shown in Figures 4.2 and 4.1, this port is never assigned when initiating applications using `sbatch`. This is extended in the current implementation of the invasive resource manager (IRM) to facilitate communication between the `srun` instance of the queued jobs and the ERS.

Another major point of difference between initiating interactive and queued jobs in SLURM is the location of the running `srun` instance. In the case of of interactive jobs `srun` is running mostly on the login node as shown in Figure 4.2. On the other hand, in the case of `sbatch`, `srun` is running on the first allocated computed node as shown in Figure 4.1. The value of the host responsible for `srun` communications is stored in the variable `resp_host` in every job's pointer to the `job_record` structure. However, this is not set when initiating queued jobs using `sbatch` due the differences in the control-flow. In the current implementation of IRM, this is extended and the value of `resp_host` is set to the srun host of the first job step specified in the batch script (see Listing 4.2). The above mentioned extensions are some of the changes implemented in the IRM for supporting queued job initiation through `sbatch`.

4.4 Resource to Process Mappings on Dynamic Reconfiguration Decisions

The elastic batch scheduler (EBS) (see Figure 4.3) can shrink a job to give resources to an application waiting in the priority-ordered job queue or can expand a job to improve the system utilization. These adaptation operations can occur multiple times during the runtime of an application. Since the scheduling entity supported by the EBS is a compute node, these operations can involve arbitrary node allocations and deallocations. The EBS identifies each compute node with a unique number called the node identifier which depends on the order in which the compute nodes are listed in the `Nodes` parameter of the `slurm.conf` file. The identifiers start from 0 and go till $num\_nodes - 1$.

For a particular expand or shrink operation, the EBS generates a resource vector (RSV). The invasive resource manager (see Section 3.3.4) utilizes the RSV for mapping the processes to the allocated nodes. The allocated nodes are arranged in an incremental order in the RSV. Furthermore, these RSVs are also used by the Invasive MPI Library (see Section 3.1) to order the rank of processes after an adaptation operation. The current implementation adds new ranks on an expansion operation and cuts higher ranks on a shrink operation [16]. The RSV has the format:

$$(vector, (identifier, count, processes), (identifier, count, processes), ...)$$

The `identifier` value refers to the node identifier from where the mapping should start. The

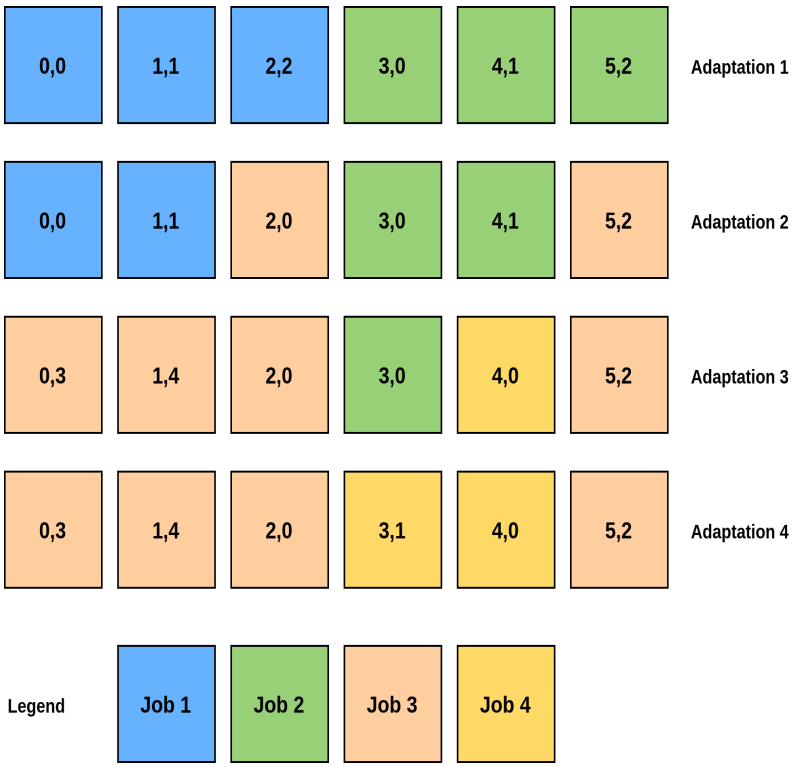


Figure 4.4: Sequence of adapataion operations leading to out of node identifiers for four jobs on six compute nodes [16].

`count` value specifies the number of nodes where the mapping should be applied, and the `processes` value indicates the number of processes to be created on a particular node. For a four node system, with each node containing 48 cores, a dense mapping for an application requiring 168 processes would have the resource vector:

$$(vector, (0, 3, 48), (3, 1, 24))$$

Every adaptation operation, i.e., expand or shrink, has multiple MPI process groups, as shown in Figure 4.3. For an expansion operation, it is the preexisting group, the expansion group, and the group created after the expansion has completed. In the case of a shrink operation, it is the preexisting group and the group of processes after adaptation. This implies that the IRM generates three RSVs for an expansion operation and two RSVs for a shrink operation.

The method for ordering the rank of processes used by the IMPI library after an adaptation operation, as described above, assumes that the node identifiers are arranged in increasing order. However, an expansion operation issued by the EBS for a particular application might not lead to resource mappings with incremental node identifiers. Consider a sequence of adaptation operations shown in Figure 4.4.

The first number inside each square represents the global node identifier number, and the second number signifies the identifier local to the job. In the case of adaptation three, the third job is expanded and has the allocation sequence $0,3;1,4;2,0;5,2$. The local node identifiers for the job $3,1,0,2$ are not in incremental order. This would lead to incorrect resource to process mapping by the IMPI library. A similar case occurs when job four is expanded in the fourth adaption operation. To overcome these scenarios and support process mappings for arbitrary order of nodes, the `srun` infrastructure and the algorithm was extended in [16]. This work utilizes the extended framework for supporting dynamic reconfiguration decisions made by the EBS.

5 Scheduling Infrastructure

Modern HPC systems consist of millions of processor cores and offer petabytes of main memory. The published data indicates that the performance of such systems has been steadily increasing along with the increasing number of cores [10, 51]. Due to the absence of a distributed operating system, the resource and job management system (RJMS) is the most critical component for managing the systems resources and executing the user's applications. As described in previous sections, the Batch Scheduler is the main component of the RJMS. It consists of the resource manager and the job scheduler. These are two different software components required for fair and efficient utilization resources of an HPC system. The Elastic Batch Scheduler (EBS) described in Section 4.2 is responsible for both managing resources and efficient scheduling of applications. While the last chapter focused on the resource management side, this chapter describes the scheduling aspect in detail. First, an overview of scheduling in distributed memory HPC systems and the commonly used scheduling strategies are discussed. Following this, the scheduling plugin interface provided by SLURM and the scheduling strategy implemented by the EBS are described.

5.1 Scheduling in Distributed Memory HPC Systems

The scheduler in an RJMS middleware is responsible for maximizing system-wide performance metrics such as resource utilization, throughput, and minimizing metrics such as response and idle times for the submitted user applications. On a higher level, the scheduler takes as input the applications which are to be executed and the set of available compute resources in the system. Following this, the scheduler generates a schedule, which describes the order in which the submitted applications will be executed and their allocated compute resources. The generated schedules determine the efficiency of the scheduler.

The problem of scheduling has been studied significantly for both shared memory multicore [52, 53] as well as distributed memory HPC systems [40, 39]. It can be described as the problem of finding an ordered assignment of a given set of tasks to the available system resources that optimizes a set of objective functions. These objective functions can be system utilization, makespan, etc. The complexity of this optimization problem has also been studied significantly in literature [54, 55]. It is classified as a *NP-hard* problem, i.e., it cannot be

solved in polynomial time or a better complexity. As a result, most of the schedulers utilize approximation algorithms using heuristic criteria for generating schedules.

In the case of resource-static scheduling for distributed memory HPC systems, the tasks can be considered as the individual jobs submitted by the different users through batch scripts. These tasks are time bounded and have exclusive access to a fixed number of specified resources. The task set consists of different jobs submitted by multiple users of the HPC system. It has several properties such as the tasks can be submitted in any order, the number of tasks are unlimited, and the tasks must be removed after completion. Although, modern-day HPC systems are becoming more and more heterogeneous by using accelerators such as Nvidia GPUs inside compute nodes. In the case of the scheduling problem, a single compute node can be considered as a resource. The collection of several compute nodes constitutes the resource set. The resource set has properties, such as a finite number of available resources, homogeneous architecture across resources, and limited or no support for quality of service (QoS). The main differences in the scheduling problem for resource-elastic applications are that the submitted jobs in the task set can have a range of required resources, and the execution time of the job depends upon the number of allocated resources. The EBS implemented in this work supports resource-elastic scheduling of submitted user applications.

The important parameters required by the scheduler for generating the schedule, such as wall clock time, the necessary number of resources are present in the batch script. After submission, the jobs are placed in a priority-ordered job queue. The scheduler in HPC systems can use different criteria for sorting the jobs placed in the queue, such as :

- *arrival time*: This is the most commonly used strategy where the jobs are ordered on a First Come Firt Serve (FCFS) basis.

- *area*: In this approach, the jobs are ordered by the product of requested resources and the maximum execution time.

- *duration*: The jobs can be ordered either as Shortest Job First (SJF) or Longest Job First (LJF). This approach requires job runtime estimation and is considered to be not fair.

- *weight*: The submitted jobs can have weights associated with them, i.e., higher weights for important or urgent jobs.

The scheduler can employ different strategies for selecting jobs from the priority-ordered job queue and mapping them to the compute resources. These include:

- *front*: This is the most commonly used approach in which the first job from the priority-ordered queue is chosen for execution.

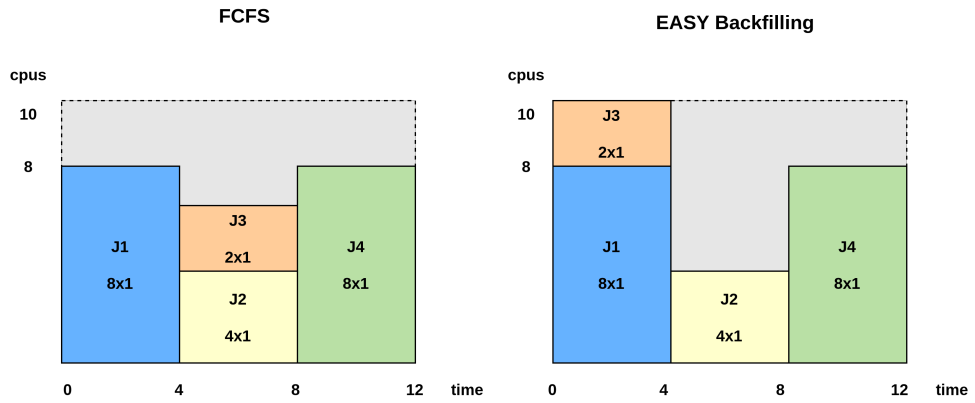


Figure 5.1: Comparing FCFS and EASY Backfilling scheduling techniques for a set of submitted user jobs.

- *first-fit*: The first job which meets the available resources is executed.
- *best-fit*: All the available jobs in the job queue are scanned and the job which optimizes a defined criterion is selected for execution.

Fairness amongst the submitted jobs, i.e., the order in which the applications are launched is the same as their arrival order is an essential criterion in scheduling. To achieve this, only the combination of sorting the priority-ordered queue based on *arrival* and picking the job based on `front` policy can be used. All other combinations do not generate fair schedules. However, the above-described approach can lead to fragmentation in the system as described in Section 5.1.1.

5.1.1 Commonly used Scheduling Strategies

The most common scheduling strategy used in batch schedulers for scheduling resource-static applications is First Come First Serve (FCFS). In this approach, the jobs are placed in the job queue according to their arrival time. The scheduler tries to start as many applications as possible at any given point in time. However, if the current most high priority job cannot be started, no other low priority job can be chosen for execution. This strategy leads to significant fragmentation in the system since the jobs submitted by the users with arbitrary resource requirements and arrival times may not pack perfectly.

Figure 5.1 shows an example of the FCFS scheduling policy for a system with ten CPUs. The x- and y- axis represent the number of CPUs and time, respectively. Each rectangle represents a different job and is labeled with the job id (e.g., J1) and the number of processes

per CPU. For example, 8x1 implies that the job requires eight CPUs with one process per CPU. The jobs arrive in the job queue in the following order J1, J2, J3, and J4, each with an execution time value of four. The execution of each job is delayed until enough resources are available, and the highest priority job cannot be started. Furthermore, when J1 is executing, there are two idle nodes in the system, and when J2 and J3 are executing, there are four idle nodes in the system, leading to fragmentation.

To overcome the problem of fragmentation in FCFS, most schedulers utilize the EASY Backfilling algorithm. In this approach, the scheduler first scans the priority-ordered job queue and tries to start as many jobs as possible. When no more jobs can be started, the scheduler takes the next highest priority job and makes a resource reservation for it at a time in the future when the requested number of resources by the job will be available. This time is known as shadow time. Following this, the scheduler searches the job queue for jobs, which can be started immediately without disturbing the resource reservation. Therefore, a lower priority job is launched, only if it does not delay the execution of the previously queued job, i.e., it finishes before the shadow time. A lower priority job can also be started if it requires resources that will not be used by the previous job.

Figure 5.1 also shows an example of the EASY Backfilling scheduling strategy. The job J3 is forwarded ahead of the queue and launched since it does not delay the execution of the previously queued job J2. This alleviates the problem of fragmentation when J1 and J3 are executing since there are no idle nodes. However, backfilling cannot always solve the fragmentation problem. The job J4 cannot be executed with the job J2 since it requires eight CPUs. As a result, six nodes remain idle when J2 is executing.

Several parameters of the backfilling scheduling algorithm can be tuned for different scenarios. The administrator can modify the parameter which decides the number of jobs that will be granted resource reservations. If the value of the parameter is one, then it is known as aggressive backfilling. Otherwise, it is called conservative backfilling. Aggressive backfilling can lead to delays in the execution of jobs waiting in the queue. Another variation of backfilling strategy is called flexible backfilling in which backfilling decisions are allowed to delay the execution of already reserved jobs up to a specific limit. While the most common approach of using backfilling is with an FCFS priority ordered job queue, different criteria such as weight, duration, etc. can also be utilized (see Section 5.1).

An optimization of the backfilling strategy is to consider a window of jobs rather than considering only one job at a time. Following this, a job to resource mapping, that leads to the maximum system utilization but preserves the resource reservations is selected. Another variation that can increase system utilization is called speculative backfilling. In this approach, the scheduler launches lower priority applications even if they interfere with the resource reservation of a previously queued higher priority job on the assumption that the launched application will terminate before the estimated time.

```
int init (void);
void fini (void);
int slurm_sched_p_reconfig (void);
int slurm_sched_p_schedule (void);
int slurm_sched_p_newalloc (struct job_record *job_ptr);
int slurm_sched_p_freealloc (struct job_record *job_ptr);
uint32_t slurm_sched_p_initial_priority ( uint32_t last_prio,
                                          struct job_record *job_ptr);
void slurm_sched_p_job_is_pending (void);
void slurm_sched_p_partition_change (void);
char *slurm_sched_p_get_conf (void);
```

Listing 5.1: SLURM scheduling plugin API.

5.2 SLURM Scheduling Plugin Interface

The SLURM [6] workload manager was designed with scalability and flexibility in mind. The various SLURM daemons and command-line utilities, as described in Section 3.3, are highly multi-threaded with separate read and write locks for the different data structures. Furthermore, SLURM is highly configurable and provides interfaces for writing plugins to provide different functionalities. A SLURM [6] plugin is a dynamically linked library which is loaded at runtime by the different SLURM components such as `slurmctld` (see Section 3.3.1), `slurmd` (see Section 3.3.2), etc..

SLURM provides a well-defined application programming interface (API) for writing plugins to support different functionalities such as user authentication, energy accounting, power management, different MPI versions, etc. Each plugin in SLURM is identified using a short character string of the format `<major>/<minor>`. The identifier `<major>` represents which well-defined SLURM plugin API is used by the plugin. For example, for a priority plugin, the major field will be `priority`, and for an MPI plugin, the major field can be `openmpi`, `pmi2`, etc. The `<minor>` field is used for distinguishing between plugins that use the same API. Since the SLURM is multi-threaded, the SLURM functions can be executed in a reentrant manner. Therefore, it is the responsibility of the programmer to use appropriate synchronization constructs to avoid the drawbacks of reentrant functions and make plugins thread-safe. Furthermore, if a plugin requires transactions with external systems such as databases, this should be done using a separate thread to ensure the high performance of the plugin.

SLURM plugins which implement the SLURM Scheduler Plugin API are called scheduler plugins. The `<major>` identifier for all scheduler plugins is `sched`. Each scheduling plugin must provide an implementation for all the functions shown in Listing 5.1.

```
int init(void)
{
    pthread_attr_t attr;
    pthread_mutex_lock(&thread_flag_mutex);
    if(elastic_thread)
    {
        pthread_mutex_unlock(&thread_flag_mutex);
        return SLURM_ERROR;
    }

    slurm_attr_init(&attr);
    /* since we do a join on this later we don't make it detached */
    if(pthread_create(&elastic_thread, &attr, batch_elastic_agent, NULL))
        error();
    pthread_mutex_unlock(&thread_flag_mutex);
    slurm_attr_destroy(&attr);

    return SLURM_SUCCESS;
}
```

Listing 5.2: The `init` function implementation for the EBS.

```
void fini(void)
{
    pthread_mutex_lock(&thread_flag_mutex);
    if(elastic_thread) {
        stop_batch_elastic_agent();
        pthread_join(elastic_thread, NULL);
        elastic_thread = 0;
    }
    pthread_mutex_unlock(&thread_flag_mutex);
}
```

Listing 5.3: The `fini` function implementation for the EBS.

The `init` function is called when the plugin is dynamically loaded and the `fini` is called when the plugin is removed. These are the two most important functions and their implementation for the elastic batch scheduler (EBS) is shown in Listings 5.2 and 5.3. The `init` function is responsible for launching the EBS scheduler thread `batch_elastic_agent`, while the `fini` function is responsible for stopping the execution of the scheduler thread.

The function `slurm_sched_p_reconfig` is used for rereading the `slurm.conf` file on changes. The scheduling pass is launched through the function `slurm_sched_p_schedule` for passive schedulers. The functions `slurm_sched_p_newalloc` and `slurm_sched_p_freealloc` are used for noting the allocation and deallocation of resources to a job. The function `slurm_sched_p_initial_priority` is used to provide priority to a newly submitted job and takes the priority of the last submitted job and pointer to the `job_record` structure as arguments. It returns the priority assigned to the new job. The other three functions shown in Listing 5.1 are used for noting that a job is pending execution, a change in state such as time or size limits of the partition and reporting scheduler specific information as an output to the `scontrol` command.

The SLURM workload manager by default uses the FCFS scheduling strategy as described in Section 5.1.1. However, it also provides an additional plugin called `sched/backfill` which implements the EASY backfilling algorithm with FCFS. The scheduling plugin to be used must be specified in the `slurm.conf` configuration file using the `SchedulerType` parameter. Furthermore, several scheduler specific parameters such as backfilling interval, resolution, window, etc. can also be specified in the configuration file [56].

5.3 The EBS Scheduling Algorithm

By default, the SLURM scalable workload manager only supports the execution of rigid applications, i.e., the number of resources allocated remains the same for the entire duration of the job. SLURM was extended by Ureña [16] to support dynamic reconfiguration, i.e., expansion and reduction of resource-elastic applications implemented using the Invasive MPI (IMPI) (see Section 3.1) library. This was done by modifying and replacing the traditional `SLURMCTLD` with a `Elastic Runtime Scheduler` (ERS) and extending other SLURM binaries to support dynamic reconfiguration as described in Section 3.3.4. However, the system allowed the execution of resource-elastic applications only through an interactive `srun` instance and no system queues were maintained (see Section 4.2).

To support the execution and combined scheduling of both rigid and resource-elastic applications in batch mode, in this work, an Elastic Batch Scheduler (EBS) is implemented. It maintains two seperate queues for rigid and elastic applications as described in Section 4.2. The EBS is primarily a scheduling plugin (see Section 5.2) and extends the default SLURM scheduler. It is a binary which is loaded by the `irtsched` (ERS) on startup and runs on a separate thread in the background. The overview of the scheduling iteration of the EBS is shown in Algorithm 1.

Initially, the EBS obtains information about the current resources in the system, such as the

total number of compute nodes, the total number of idle nodes, the total number of nodes in completing state, etc (Line 1). This information is obtained through the usage of `bitmaps`, which are global variables maintained by SLURM.

Algorithm 1: The Elastic Batch Scheduler (EBS) Iteration.

```
1  while TRUE do
2      Obtain resource information from SLURM
3      Obtain workload information from SLURM
4      Schedule both rigid and elastic jobs in priority order with job start if requested
         resources are available
5      Update scheduler metrics
6      if scheduling event then
7          Update resource information
8          for each job in job list do
9              Update waiting elastic and rigid job list
10             Update running elastic and rigid job list
11             if no performance data available for running elastic job then
12                 request_perf_data(job);
13             end
14             Check if any running elastic job is adapting
15         end
16         if no elastic job is adapting and number of running elastic jobs > 0 then
17             Generate resource vectors (RSV) for expanding or shrinking running elastic
                 jobs depending on the job scheduling strategy used
18             Update reduction job list
19             Update expansion job list
20             for each job in reduction job list do
21                 shrink_job(job);
22             end
23             for each job in expansion job list do
24                 expand_job(job);
25             end
26         end
27     end
28 end
```

The `bitmaps` are modified by the `node_select` plugin [57], which runs alongside the EBS and is responsible for allocating compute resources to the queued jobs. The `bitmaps` are also modified by the EBS after expand or shrink reconfiguration decisions. The plugin also obtains information about the current user-submitted jobs, such as the total number of jobs, and the required compute resources, etc. (Line 2). The jobs are added to the elastic job queue

only if the user specifies the fields `min-nodes-invasic` and `max-nodes-invasic` in the job script. Otherwise, they are added to the rigid queue (see Section 4.2). The submitted jobs are prioritized according to their arrival time, i.e., First Come First Serve (FCFS).

The EBS tries to schedule and start as many rigid and resource-elastic applications as possible, depending upon the resource requirements and priorities (Line 4). If a high priority application cannot be scheduled, due to the non-availability of required resources, then no lower-priority application is chosen for execution. The user-submitted resource-elastic applications are always executed with the resources specified in the `--nodes` field in the batch script (see Listing 4.2). The applications can then be expanded or reduced depending on the minimum and maximum number of nodes specified in the `min-nodes-invasic` and `max-nodes-invasic` fields respectively (see Section 4.2). Following this, the scheduler executes a function that updates the scheduler performance metrics (Line 5), depending upon the current state of the system such as system utilization, job waiting and response times, etc. (see Section 7.2).

The main scheduling loop in the EBS, which is responsible for dynamic reconfiguration decisions for running resource-elastic applications, is event-triggered and runs on three scheduling events (see Section 4.2). Firstly, on every scheduler tick which is specified in the `slurm.conf` file and read in by the EBS. Secondly, whenever a new application is added to the system job queues, i.e., rigid and elastic. Thirdly, whenever an application finishes execution (Line 6). If a scheduling event occurs, then the EBS gets the updated resource information of the system, which might have changed due to previously scheduled jobs (Line 7).

The EBS iterates across all jobs in the system and creates four separate job lists (Line 9-10). The lists created are running elastic and rigid jobs and currently waiting elastic and rigid jobs. These lists are later used for dynamic reconfiguration decisions depending upon the used malleability and job management policy (see Section 5.3.1). Following this, it iterates across all currently running elastic jobs and checks if the job has performance data associated with it. This is done by checking a flag variable in the job's pointer to the `job_record` structure. If no performance data is available for the job, then it is requested by the EBS (Line 11-13). Currently, only resource-elastic applications written using the Elastic Phase Oriented Programming Model (EPOP) (see Section 3.2) support this feature.

On request, the EPOP driver (see Section 3.2.1) returns the MPI time to compute time ratio (MTCT) for a particular phase associated with the job. The MTCT metric is used as a heuristic criterion for performance-aware dynamic reconfiguration decisions. The EBS also checks whether any job is currently adapting, i.e., expanding or shrinking (Line 14). This is done by checking the job state variable which is modified to `IS_ADAPTING`, whenever the EBS takes a reconfiguration decision for a particular job. Whether any jobs in the system are currently adapting or not is checked to ensure consistency among compute nodes, and to assure that new reconfiguration decisions are generated after all adaptations have completed.

If no jobs are currently adapting and there are running resource-elastic jobs in the system, then the EBS takes expand or shrink decisions depending upon the used job management and elastic scheduling strategy (Line 16). The elastic scheduling strategies implemented in the EBS are described in Section 5.3.1. The decision to expand or shrink a running application is taken by analyzing the resource vector (RSV). An RSV is an array with a size equal to the running malleable jobs. A scheduling strategy analyzes the currently running jobs in the system and returns an RSV. Each index of the RSV represents the number of nodes that a currently running application must have. The jobs are ordered wrt their arrival time.

The EBS iterates through the RSV and decides to expand or shrink a job depending upon the following criteria:

- *expansion*: If the nodes specified by the RSV for the current job index are greater than the current number of nodes allocated to the job.

- *reduction*: If the nodes specified by the RSV for the current job index are less than the current number of nodes allocated to the job.

For example, if an elastic job is currently running on two compute nodes, and the scheduling strategy decides to expand the job to four compute nodes, the RSV value at that job index will be four.

The EBS creates two separate lists representing jobs to be expanded and shrunk and adds appropriate jobs to each of them. The EBS first iterates through the reduction job list and creates an RPC message request of type `slurm_reallocation_message` for each job (Line 20-22). This message contains important information such as the number of preexisting tasks to destroy, nodes to retreat from, etc. The generated message is then sent to the Elastic Runtime Scheduler `irtsched`. A similar procedure is followed for expanding the jobs in the expansion job list (Line 23-25). However, in this case, the `slurm_reallocation_message` contains information such as new tasks to create and the nodes on which to create them. The process of dynamic reconfiguration for batch jobs is described in detail in Section 4.3. The EBS iteration continues to run until the `irtsched` is terminated.

5.3.1 Elastic Scheduling Strategies

In this work, different scheduling strategies for executing and scheduling malleable applications are implemented and evaluated. An elastic scheduling strategy can be considered to be a combination of two policies a job management policy and a malleability management policy. The job management policy is primarily responsible for deciding

Algorithm 2: Adapted Favor Previously Started Malleable Applications First (FPSMA) job scheduling strategy [42].

```
1  Function FPSMA_GROW(RSV, elasticJobList, jobCount):
2      elasticJobList ⟵ jobs sorted in increasing order of their start time
3      remainingIdleNodes ⟵ idle nodes in the system
4      RSVIndex ⟵ 0
5      for each job in elasticJobList do
6          jobRemainingTime ⟵ remaining time for the job
7          currentNodeCount ⟵ current allocated node count for the job
8          RSV[RSVIndex] ⟵ currentNodeCount
9          if currentNodeCount ≠ jobMaxNodes & remainingIdleNodes > 0 &
             jobRemainingTime > 60.0 then
10             Check job constraints wrt remainingIdleNodes
11             Calculate appropriate growValue
12             RSV[RSVIndex] ⟵ growValue
13             remainingIdleNodes ⟵ remainingIdleNodes − growValue
14         end
15         RSVIndex++
16     end
17 return
18 Function FPSMA_SHRINK(RSV, elasticJobList, waitingJob, jobCount):
19     elasticJobList ⟵ jobs sorted in decreasing order of their start time
20     remainingIdleNodes ⟵ idle nodes in the system
21     nodesRequired ⟵ waitingJobNodes − remainingIdleNodes
22     RSVIndex ⟵ 0
23     for each job in elasticJobList do
24         jobRemainingTime ⟵ remaining time for the job
25         currentNodeCount ⟵ current allocated node count for the job
26         RSV[RSVIndex] ⟵ currentNodeCount
27         if currentNodeCount ≠ jobMinNodes & nodesRequired > 0 &
             jobRemainingTime > 60.0 then
28             Check job constraints and calculate shrinkValue
29             RSV[RSVIndex] ⟵ shrinkValue
30             nodesRequired ⟵ nodesRequired − shrinkValue
31         end
32         RSVIndex++
33     end
34     if nodesRequired > 0 then
35         Don't shrink any job.
36     end
37 return
```

whether to expand or shrink a running resource-elastic application. On the other hand, the malleability management policy is responsible for deciding how to expand or shrink an application.

In this work the following two job management policies are considered:

- Priority to running malleable applications (PRMA): In this approach, if compute resources in the system become available, then a malleability management policy is used to expand an already running resource-elastic application. In this policy, no shrink operations are performed. However, if the resource requirements of the highest priority job waiting in the queue match the number of available resources, then it is executed.

- Priority to waiting malleable applications (PWMA): In this approach, when the next highest priority job in the job queue cannot be launched, a malleability management policy is used to shrink running malleable applications so as to provide resources for the waiting job. These shrink operations are compulsory. However, if enough additional processors cannot be obtained to launch the waiting application, a malleability management policy is considered to expand an already running job.

Buisson et al. [42] propose two malleability management policies for scheduling malleable applications for multicluster grid systems. In this work the policies, i.e., Favor Previously Started Malleable Applications First (FPSMA) and Equi-Grow Shrink (EGS) are adapted and extended for distributed memory HPC systems. Each of the two policies have procedures for growing and shrinking a malleable application. Algorithm 2 shows the grow and shrink procedures for the FPSMA malleability management policy. The grow and shrink procedures for the EGS policy are shown in Algorithm 3.

The `FPSMA_GROW` procedure shown in Algorithm 2 takes as arguments the elastic job list and the current job count. It returns the filled resource vector (RSV) which is used by the elastic batch scheduler (EBS) for expanding the running malleable jobs (see Section 5.3). First the procedure sorts the elastic jobs in the increasing order of their start time (Line 1) and calculates the current remaining idle nodes in the system (Line 2). The idle nodes are computed by using the idle node bitmap (see Section 5.3). Following this, it iterates across all jobs in the sorted elastic job list and computes the remaining time and current number of allocated nodes for each job (Line 6-7). A job is considered for expansion if all of the following three conditions are met (Line 9):

1. If the current number of nodes of the job are not equal to the parameter `--max-nodes-invasic` specified by the user in the job script (see Section 4.2).

2. There are still some idle nodes remaining in the system.

3. The job still has 60.0 seconds of execution time remaining.

Otherwise, there are no changes in the compute resources of the job (Line 8).

In the next step (Line 10) the job constraint specified by the `--node-constraints` parameter in the user batch script is considered (see Section 4.2). For example, if the value of the parameter is `--node-constraints='pof2'` and the current number of nodes allocated to the job is two and the idle nodes remaining are 10. The procedure calculates the next suitable *growValue*, which is eight in this case (Line 11). The job is then listed for expansion with the calculated *growValue* (Line 12). The number of idle nodes remaining in the system is updated (Line 13). The procedure continues until all jobs in the job list have been iterated over.

The `FPSMA_SHRINK` procedure shown in Algorithm 2 takes the same arguments as the `FPSMA_GROW` procedure with an additional *waitingJob* argument. The additional argument is a pointer to the next highest priority waiting job in the queue. It can be both a rigid or elastic job. This is done for the implementation of the PWMA job management policy. In contrast to the `FPSMA_GROW` procedure, the elastic job list is sorted in decreasing order of the job start times (Line 20). The number of idle nodes remaining in the system and the nodes required by the waiting job is calculated by the procedure (Line 21-22). In the next step, the procedure iterates across all jobs in the job list and calculates the remaining job time and the current number of nodes allocated to the job. A running job is considered for a shrink operation if the following three conditions are met:

1. If the current number of nodes of the job are not equal to the parameter `--min-nodes-invasic` specified by the user in the job script (see Section 4.2).

2. There are still some nodes required by the waiting job.

3. The job still has 60.0 seconds of execution time remaining.

Similar to the `FPSMA_GROW` procedure, the job constraints specified in the batch script are checked, and an appropriate *shrinkValue* is calculated. Taking the same example, if the value of the parameter is `--node-constraints='pof2'` and the current number of nodes allocated to the job is four. The procedure calculates the next suitable *shrinkValue*, which is 2 in this case (Line 29). The job is listed for reduction with the calculated *shrinkValue* (Line 30), and the value of the remaining nodes still required by the waiting job is updated (Line 31). If, after iterating through all the jobs in the elastic job list, the number of nodes required by the waiting job is still greater than zero, i.e., it was not possible to obtain enough nodes from shrinking the running jobs then no running job is shrunk (Line 34-36). The `FPSMA_SHRINK` procedure returns the value of the remaining nodes to the EBS. If the returned value is greater than zero, then a running application is chosen for expansion using the `FPSMA_GROW`

procedure.

Apart from the two grow and shrink procedures shown in Algorithm 2, a heuristic-based version of the two procedures is also implemented in the EBS. The heuristic criterion MPI time to compute time (MTCT) is used for making dynamic reconfiguration decisions. In the case of expansion, a running resource elastic application is only granted resources if the `MTCT` ratio is below a certain threshold. For reduction, the resources are only taken away if the `MTCT` ratio is greater than a certain threshold. The upper and lower ratios for the metric `MTCT` can be set in the `slurm.conf` file and are read by the EBS during startup. An implementation of the two functions is shown in Section 6.2.

The EGS malleability management policy shown in Algorithm 3 is similar to the equipartition scheduling strategy commonly used for the scheduling of malleable applications. However, there are some differences. In the equipartition scheduling strategy, all the resources are equally distributed among the running jobs, while in EGS, only the currently available resources are equally distributed. This implies that in equipartitioning, all running elastic applications must have the same number of resources, while this is not the case for EGS. Moreover, equipartitioning can generate a mixture of expand and shrink operations, while EGS only expands or shrinks at any particular time.

The `EQUI_GROW` and `EQUI_SHRINK` procedures described in Algorithm 3 are similar to the `FPSMA_GROW` and `FPSMA_SHRINK` procedures described in Algorithm 2 with a few differences. First, the current implementation of EGS procedures in the EBS does not support constraints on the number of nodes that can be allocated to a particular job. This means that the parameter `--node-constraints` is ignored since the aim of EGS is to divide the available processors among the running applications equally.

In the case of `EQUI_GROW`, the number of idle nodes in the system are equally distributed amongst all the running jobs by calculating the appropriate *growValue* (Line 3). If the number of nodes to be distributed are not exactly divisible by the number of running jobs, then a *growRemainder* value is calculated (Line 4). The remainder number of nodes are distributed equally among the running elastic jobs in the increasing order of their start time as a bonus (Line 11-14).

In the case of the `EQUI_SHRINK` procedure, the number of nodes required by the highest priority waiting application are equally reclaimed from the running applications. This is done by calculating the appropriate *shrinkValue* (Line 22). If the number of nodes required are not exactly divisible by the currently running applications, then a *shrinkRemainder* is computed (Line 23). The remainder number of nodes are reclaimed from the running applications in the decreasing order of their start time (Line 31-34).

The value of the remaining number of nodes required is also updated appropriately (Line 30,

34). Similar to the FPSMA_SHRINK procedure, if the remaining number of nodes required by the highest priority waiting application are still greater than zero, then no running application is reduced (Line 39-41). Following this, the EQUI_GROW procedure is used to grow the appropriate running resource-elastic applications.

Algorithm 3: Adapted Equi-Grow Shrink (EGS) job scheduling strategy [42].

```
1  Function EQUI_GROW(RSV, elasticJobList, jobCount):
2      elasticJobList ⟵ jobs sorted in increasing order of their start time
3      growValue ⟵ ceil(remainingIdleNodes / jobCount)
4      growRemainder ⟵ remainingIdleNodes%jobCount
5      RSVIndex ⟵ 0
6      for each job in elasticJobList do
7          jobRemainingTime ⟵ remaining time for the job
8          RSV[RSVIndex] ⟵ current allocated node count for the job
9          if jobRemainingTime > 60.0 & currentNodeCount ≠ jobMaxNodes then
10             RSV[RSVIndex] ⟵ RSV[RSVIndex] + growValue
11             if growRemainder > 0 then
12                 RSV[RSVIndex] ⟵ RSV[RSVIndex] + 1
13                 growRemainder ⟵ growRemainder − 1
14             end
15         end
16         RSVIndex + +
17     end
18 return
19 Function EQUI_SHRINK(RSV, elasticJobList, waitingJob, jobCount):
20     elasticJobList ⟵ jobs sorted in decreasing order of their start time
21     nodesRequired ⟵ waitingJobNodes − remainingIdleNodes
22     shrinkValue ⟵ ceil(nodesRequired / jobCount)
23     shrinkRemainder ⟵ nodesRequired%jobCount
24     RSVIndex ⟵ 0
25     for each job in elasticJobList do
26         jobRemainingTime ⟵ remaining time for the job
27         RSV[RSVIndex] ⟵ current allocated node count for the job
28         if jobRemainingTime > 60.0 & currentNodeCount ≠ jobMinNodes then
29             RSV[RSVIndex] ⟵ RSV[RSVIndex] − shrinkValue
30             nodesRequired ⟵ nodesRequired − shrinkValue
31             if shrinkRemainder > 0 then
32                 RSV[RSVIndex] ⟵ RSV[RSVIndex] − 1
33                 shrinkRemainder ⟵ shrinkRemainder − 1
34                 nodesRequired − −
35             end
36         end
37         RSVIndex + +
38     end
39     if nodesRequired > 0 then
40         Don't shrink any job.
41     end
42 return
```

6 Implementation

The scalable workload manager SLURM [6] is an open-source software written entirely using the C programming language. It consists of more than 300,000 lines of code and has a highly distributed architecture. It has several multithreaded resource components such as `slurmctld`, `slurmd`, `slurmstepd`, etc. and several command-line utilities such as `sbatch`, `srun`, `sinfo`, etc (see Section 3.3). The resource components and command-line utilities interact with each through a SLURM communication protocol using Remote Procedure Call (RPC) requests. Furthermore, SLURM provides a well-defined API for writing SLURM plugins to support different functionalities such as scheduling, energy and power management, etc. SLURM plugins are binaries that are dynamically loaded by different components at startup.

This work extends SLURM's [16] scheduling infrastructure to support the initiation and efficient scheduling of rigid and resource-elastic applications in batch mode. This is done by extending SLURM to support dynamic reconfiguration of queued batch jobs and the implementation of the Elastic Batch Scheduler (EBS), which implements the SLURM's scheduling plugin API. In this chapter, some of the implementation details to support dynamic reconfiguration of queued batch jobs are described. Several strategies to support the scheduling of resource-elastic applications are implemented in the EBS (see Section 5.3.1). This chapter describes the implementation of the grow procedure for one of those strategies.

6.1 Enabling Dynamic Reconfiguration for Elastic Batch Jobs

SLURM provides a command-line utility called `sbatch` for submitting jobs in batch mode. The users write a batch script describing their job, which is used as an input to `sbatch`. The extensions to `sbatch` for supporting initiation of queued resource-elastic applications and a sample batch script are described in Section 4.2 and Listing 4.2 respectively.

Every job submitted in SLURM has a pointer to the `job_record` structure. Some of the important fields of this structure are shown in Listing 6.1. The fields `min_nodes_invasic` and `max_nodes_invasic` are the minimum, and maximum compute nodes required by the submitted elastic job and are set to the values specified by the user in the batch script (see Section 4.2). Similarly, the value of the variable `node_constraints` is set to the constraint specified by the

```
struct job_record {
    /*other fields*/
    uint16_t other_port; /* port for client communications */
    char *resp_host; /* host for srun communications */
    int is_elastic_batch;
    int min_nodes_invasic;
    int max_nodes_invasic;
    char* node_constraints;
    /*other fields*/
};
```

Listing 6.1: The `job_record` structure in SLURM

```
void expand_job(struct job_record* job_ptr, slurm_cred_t* slurm_cred,
          int max_nodes, uint32_t* updated_node_ids, int curr_node_val,
          bitstr_t *new_nodes_to_invade_bitfield,
          bitstr_t *preexisting_nodes_to_retreat_from_bitfield,
          bitstr_t *nodes_to_set_ids);

void shrink_job(struct job_record* job_ptr, slurm_cred_t* slurm_cred,
          int max_nodes, uint32_t* updated_node_ids, int curr_node_val,
          bitstr_t* new_nodes_to_invade_bitfield,
          bitstr_t* preexisting_nodes_to_retreat_from_bitfield,
          bitstr_t *reserved_nodes,
          bitstr_t *nodes_to_set_ids);
```

Listing 6.2: Interface for expanding and shrinking a job.

user. This is done by modifying the RPC request message called `REQUEST_SUBMIT_BATCH_JOB`, which is generated and sent to the elastic runtime scheduler (ERS) on submission of a batch job by `sbatch`. If the minimum and maximum nodes required by the job are non-zero, then the field `is_elastic_batch` is set to one for that particular job. Otherwise, it is set to zero. This is required for separating the user-submitted rigid and malleable jobs into separate queues.

Dynamic reconfiguration decisions for running resource-elastic applications are taken by the elastic batch scheduler (EBS) (see Algorithm 1). Listing 6.2 shows the interfaces for expanding and shrinking running malleable applications. The functions take important information such as the updated SLURM credentials for the job and `bitmaps` representing nodes to create tasks on and nodes from which preexisting tasks must be destroyed. The functions generate the RPC request message of type `SRUN_REALLOC_MSG`, which is sent to the `itrsched` (ERS) by the EBS (see Section 4.3). The current implementation does not support mixed adaptation

```
resource_allocation_response_msg_t * existing_allocation(void)
{
    slurm_allocation_callbacks_t callbacks;
    callbacks.ping = _ping_handler;
    callbacks.timeout = _timeout_handler;
    callbacks.job_complete = _job_complete_handler;
    callbacks.realloc = _realloc_handler;

    allocated_port_sbatch = xmalloc(sizeof(uint16_t));

    /* create message thread to handle pings and such from itrsched */
    msg_thr = slurm_allocation_msg_thr_create(allocated_port_sbatch,
                                                        &callbacks);
    /*Other initialization*/
}
```

Listing 6.3: Assigning the port for communication with the reallocation handler for queued jobs.

operations, i.e., `SRUN_REALLOC_MSG` should either be an expansion or reduction. Towards this, if the function `expand_job` is called, then only the fields new nodes to invade and the number of new tasks to create are set in the RPC request message, while the variables preexisting tasks to destroy and preexisting nodes to retreat from are set to zero. The function `shrink_job` behaves analogously.

```
/*Other initialization*/
if(job_ptr->resp_host == NULL)
{
    ListIterator step_iterator;
    step_iterator = list_iterator_create(job_ptr->step_list);
    struct step_record *step_ptr;
    while ((step_ptr = (struct step_record *) list_next(step_iterator)))
    {
        char *IPbuffer;
        struct hostent *host_entry;
        host_entry = gethostbyname(step_ptr->host);
        IPbuffer = inet_ntoa(*((struct in_addr*)host_entry->h_addr_list[0]));
        job_ptr->resp_host = IPbuffer;
    }
}
```

Listing 6.4: Assigning the host for srun communications.

On receiving the SRUN_REALLOC_MSG, the irtsched, is responsible for dynamically reconfiguring the running queued job by communicating with the running srun instance for that particular job. For this, two things are required, a port for communication with the thread on which the reallocation_handler is running and the hostname of the compute node with the currently running srun instance. The above mentioned variables, i.e., other_port and resp_host are stored in each job's pointer to the job_record structure and are shown in Listing 6.1. There are several differences between launching elastic applications with an interactive srun instance and initiating queued elastic applications. The differences are shown in Figures 4.2 and 4.1 and described in Section 4.3. As a result the variables other_port and resp_host are not set when launching queued jobs.

To overcome this, the invasive resource manager was extended in this work to support the dynamic reconfiguration of queued elastic applications. Listing 6.3 shows a code snippet in which the SLURM allocation message thread, which contains the callback to the reallocation_handler is created on a particular port. The value of the port is assigned by the routine slurm_allocation_msg_thr_create to a suitable port supported by srun. After an srun instance starts on the first allocated compute node for a particular batch job, it sends an RPC request message of type REQUEST_ALLOCATION_INFO_LITE to the irtsched (see Section 4.1). This RPC request message was modified to send the previously allocated port number to the ERS. After the message is successfully received by the ERS, the port value is added to the job's pointer to the job_record structure.

When queued jobs are initiated, the srun instance always runs on the first allocated compute node. This is different for interactive job initiation in which the srun instance runs on the login node. A batch job can have multiple srun commands (see Section 4.1) and each srun command is called a job step. The current implementation of the EBS only supports the execution of batch jobs with one job step. Listing 6.4 shows a code snippet in which the job's resp_host variable is set to the hostname of the compute node where the srun instance of the first job step is running.

6.2 Enabling Performance-aware Dynamic Reconfiguration Decisions for Elastic Batch Jobs

The EBS implements several malleability management policies for growing and shrinking resource-elastic applications as described in Section 5.3.1. These include Favour Previously Started Malleable Applications (FPSMA) and Equi-Grow Shrink (EGS). The algorithms for the grow and shrink procedures for these two policies are shown in Algorithm 2 and Algorithm 3.

A major drawback of the policies mentioned above is that they don't take into account

```
1  void fpsma_grow_heuristic(int **RSV, List elastic_job_list, int job_count, int max_nodes)
2  {
3  //Initialize variables RRV, RSV_order, elastic_iterator, original_list_iterator, metric_average, trend_mtct_epop
4  int idle_node_count = bit_set_count(idle_node_bitmap);
5  int remaining_idle_nodes = idle_node_count;
6  List original_job_list = elastic_job_list;
7  sort_job_queue_grow(elastic_job_list);
8  int RRV_index = 0;
9  elastic_job_iterator = list_iterator_create(elastic_job_list);
10
11 while ((job_ptr = (struct job_record *) list_next(elastic_job_iterator)))
12 {
13     remaining_time = difftime(job_ptr->end_time, current_time);
14     RSV_order[RRV_index] = job_ptr->job_id;
15     if(job_ptr->is_epop)
16     {
17         int is_perf = get_is_perf_data_available(job_ptr->job_id);
18         if (!is_perf_data_available)
19         {
20             RRV[RRV_index] = job_ptr->node_cnt;
21             RRV_index++;
22             continue;
23         }
24         metric_average = get_previous_mtct_time(phase_index,job_ptr->job_id);
25         trend_mtct_epop = get_current_mtct_time(phase_index,job_ptr->job_id);
26         reset_is_perf_data_available(job_ptr->job_id);
27         RRV[RRV_index] = job_ptr->node_cnt;
28         if(job_ptr->node_cnt != job_ptr->max_nodes_invasic && remaining_idle_nodes > 0 &&
               remaining_time > 60.0)
29         {
30             if(metric_average < MTCT_lower_threshold || trend_mtct_epop < MTCT_lower_threshold)
31             {
32                 if(job_ptr->node_constraints == NULL)
33                 {
34                     if(remaining_idle_nodes >= job_ptr->max_nodes_invasic)
35                     {
36                         RRV[RRV_index] = job_ptr->max_nodes_invasic;
37                         remaining_idle_nodes -= job_ptr->max_nodes_invasic;
38                     }
39                     else
40                     {
41                         RRV[RRV_index] = job_ptr->node_cnt + remaining_idle_nodes;
42                         remaining_idle_nodes = 0;
43                     }
44                 }
45                 if (strcasecmp(job_ptr->node_constraints, "pof2") == 0)
46                 {
47                     int curr_node_val = job_ptr->node_cnt;
48                     int p_of_two_lte_idle = get_val(curr_node_val, remaining_idle_nodes);
49
50                     RRV[RRV_index] = p_of_two_lte_idle;
51                     if(curr_node_val != p_of_two_lte_idle)
```

```
52                          remaining_idle_nodes −= (p_of_two_lte_idle−curr_node_val);
53
54                  }
55                  else if (strcasecmp(job_ptr−>node_constraints, "even") == 0)
56                  {
57                      int curr_node_val = job_ptr−>node_cnt;
58                      int even_lte_idle = get_val(curr_node_val, remaining_idle_nodes);
59
60                      RRV[RRV_index] = even_lte_idle;
61                      if(curr_node_val != even_lte_idle)
62                          remaining_idle_nodes −= (even_lte_idle − curr_node_val);
63
64                  }
65                  else if (strcasecmp(job_ptr−>node_constraints, "odd") == 0)
66                  {
67                      int curr_node_val = job_ptr−>node_cnt;
68                      int odd_lte_idle = get_val(curr_node_val, remaining_idle_nodes);
69
70                      RRV[RRV_index] = odd_lte_idle;
71                      if(curr_node_val != odd_lte_idle)
72                          remaining_idle_nodes −= (odd_lte_idle − curr_node_val);
73
74                  }
75                  else
76                  {
77                      //ncube case, ex LULESH
78                      int curr_node_val = job_ptr−>node_cnt;
79                      int next_cube_root_lte_idle = get_val(curr_node_val, remaining_idle_nodes);
80
81                      RRV[RRV_index] = next_cube_root_lte_idle;
82                      if(curr_node_val != next_cube_root_lte_idle)
83                          remaining_idle_nodes −= (next_cube_root_lte_idle−curr_node_val);
84
85                  }
86      }
87          }
88      }
89      RRV_index++;
90  }
91  //Fill RSV according to order of jobs in the original elastic job list before sorting
92  original_list_iterator = list_iterator_create(original_job_list);
93  int RSV_index = 0;
94  while ((job_ptr = (struct job_record ∗) list_next(original_list_iterator)))
95  {
96      int job_id = job_ptr−>job_id;
97      for(int i = 0; i < job_count; i++)
98      {
99          if(RSV_order[i] == job_id)
100         {
101             (∗RSV)[RSV_index] = RRV[i];
102             break;
103         }
```

```
104         }
105         RSV_index++;
106     }
107 }
```

Listing 6.5: The implementation of heuristic based FPSMA grow procedure.

the performance of the application when deciding to expand or shrink an application. The applications are expanded in the increasing order of their start time and reduced in the decreasing order of their start time (see Section 5.3.1). Towards this, heuristic based versions of the `FPSMA_GROW` and the `FPSMA_SHRINK` procedures are implemented in the EBS to support performance-aware dynamic reconfiguration decisions for running resource-elastic applications. The control flow of the implemented versions is similar to the one described in Algorithm 2.

Listing 6.5 shows the implementation of the heuristic-based `FPSMA_GROW` procedure. The EBS currently only supports performance-aware dynamic reconfiguration decisions for applications that are programmed using the EPOP programming model (see Section 3.2). The EBS requests performance data for all applications which are written using EPOP (see Algorithm 1). The integration of IRM and the EPOP driver [45] provides helper functions to request performance metrics for a running application and also to identify if whether an application is programmed using EPOP.

The heuristic criterion used for performance aware decisions is the MPI time to compute time (MTCT) ratio. The application is considered for expansion only if the MTCT ratio is below a certain threshold specified in the `slurm.conf` file (Line 30). On the other hand, the application is considered for reduction only if the MTCT ratio is greater than a certain threshold. In both grow and shrink procedures, the elastic job list is initially sorted in-place as described in Section 5.3.1. The code in Lines 94-108 is important to ensure that the expansion and reduction operations are done wrt to the original order of jobs in the job list.

7 Experimental Results

In this work, the batch system of the scalable workload manager SLURM was extended to support the scheduling of elastic and rigid applications through the addition of the elastic batch scheduler (EBS). Several scheduling strategies for elastic applications are implemented in the EBS, as described in Section 5.3.1. In this chapter first, the experimental platform used for evaluating the batch system is described. Following this, the different implemented scheduling strategies are compared and assessed wrt the performance metrics average waiting and response times, system utilization, and makespan. Furthermore, the benefits of malleable applications and an adaptive resource manager are demonstrated. Finally, the overhead of dynamic adaptation operations, i.e., expansion and reduction, are discussed.

7.1 Experimental Setup

For analyzing the performance of the implemented batch system, the next-generation HPC system called SuperMUC-NG [58] located at the Leibniz Supercomputing Center in Germany is used. SuperMUC-NG consists of eight islands comprising a total of 6480 compute nodes. The experiments in this work are performed on the thin nodes of SuperMUC-NG, which consists of 6336 compute nodes based on the Intel Skylake-SP architecture. Table 7.1 provides a summary of a compute node on SuperMUC-NG. Each compute node has two sockets, comprising of two Intel Xeon Platinum 8174 processors, with 24 cores each and a total of 96GB main memory. Hyperthreading is enabled on each compute node, with the support of two threads per core. The nominal operating core frequency for each core is 3.10 GHz. The sustained memory bandwidth of each compute node obtained through the STREAM benchmark is 95.78 GB/s.

For all the experiments, a virtual cluster on 18 compute nodes of the SuperMUC-NG system is simulated. The extended SLURM is used as the Resource and Job Management Software (RJMS) on this cluster. One compute node is used as a login node for submitting the applications used for evaluation, and another compute node runs the elastic runtime scheduler (`irtsched`). As a result, 16 compute nodes are available for running the applications on the virtual cluster. The SLURM version extended and used in this work is 15.08.

Cluster	SuperMUC-NG
Processor	Intel Xeon Platinum 8174
Architecture	Intel Skylake-SP
Interconnect	Intel OmniPath
Cores	24
Sockets	2
Base core clock freq. [GHz]	3.10
L3 cache/chip [MB]	33
Memory size [GB]	96
Peak DP [GFLOPs]	3533
STREAM [GB/s]	95.78

Table 7.1: Architectural overview of a SuperMUC-NG compute node.

7.2 Performance Metrics

To quantify the performance of the implemented elastic job scheduling strategies in the elastic batch scheduler (EBS), we use a variety of performance metrics. These include user-centric metrics: average waiting time (AWT) and average response time (ART) and system-centric metrics: average system utilization and makespan.

The average waiting time refers to the waiting time for all jobs in the system, the formula for which is shown in Equation 7.1. In Equation 7.1, N refers to the total number of jobs in the system, t_i refers to the time when a job starts its execution and t_a refers to the time when the job was submitted. The subtraction of $t_i^s - t_i^a$ gives the waiting time for a particular application.

$$AWT = \frac{1}{N}\sum_{i=1}^{N}(t_i^s - t_i^a) \tag{7.1}$$

The average response time describes when the results of a particular job are available. The formula for calculating average response time is shown in Equation 7.2. N refers to the total number of jobs in the system, t_i^w refers to the waiting time of the application as described above and t_i^r refers to the total execution time of the application.

$$ART = \frac{1}{N}\sum_{i=1}^{N}(t_i^w + t_i^r) \tag{7.2}$$

The metric average system utilization describes the percentage of system resources which were utilized during a specific period of time. Makespan refers to the time when the last job in a workload completes its execution. It can also be interpreted as the total execution time of a set of jobs in a workload. The EBS updates the performance metrics of the scheduler on every scheduler iteration as shown in Algorithm 1 and described in Section 5.3.

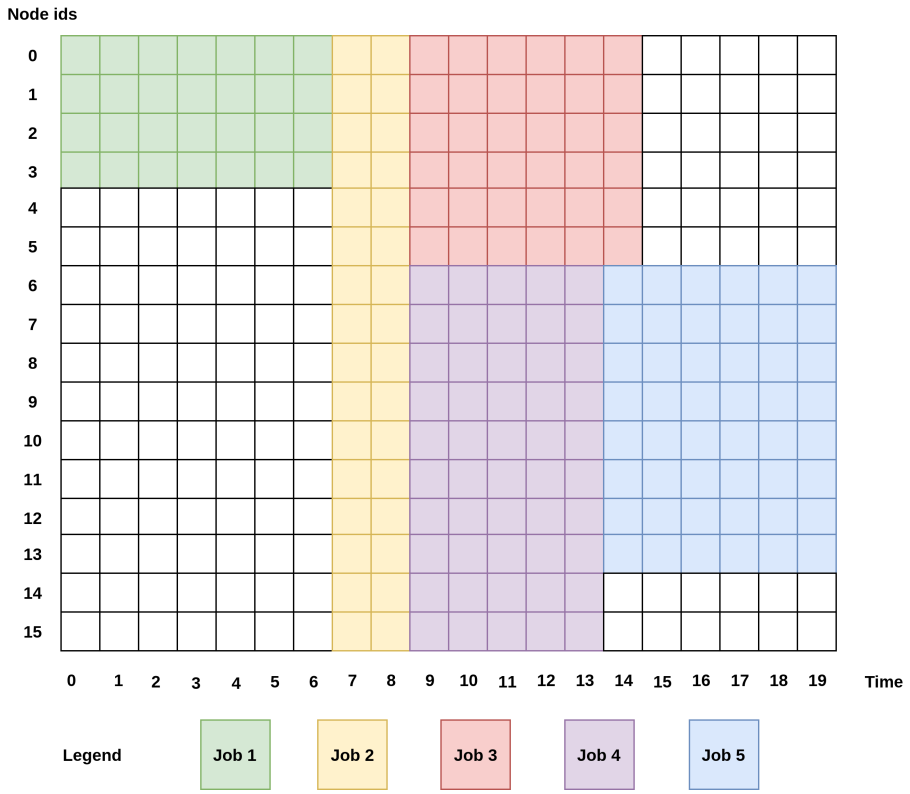
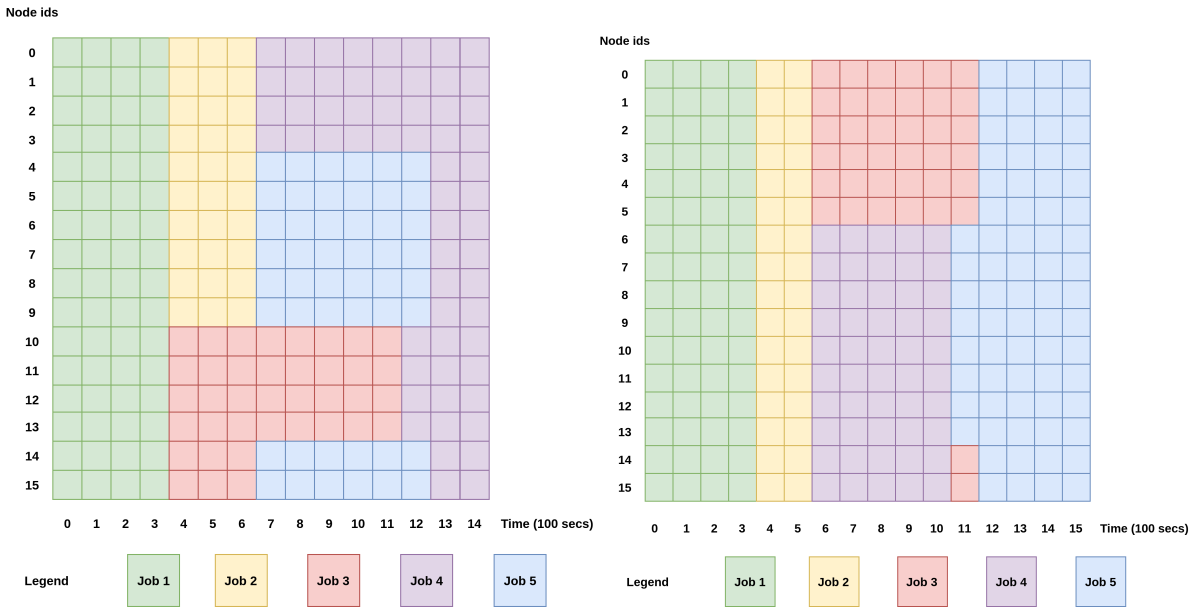


Figure 7.1: Allocation of nodes over time for the default FCFS scheduling strategy.

7.3 Evaluating the Advantages of Malleable Applications

The performance of a batch scheduler depends upon not only the used scheduling strategy but also on the type of applications that are being executed. Malleable or resource-elastic jobs that can dynamically expand or shrink at runtime can significantly improve system utilization, response, and waiting times as compared to traditional resource-static applications. In this work, the Elastic Batch Scheduler (EBS) was implemented in SLURM to support the dynamic reconfiguration of resource-elastic queued applications written using the IMPI library (see Section 3.1). This section demonstrates the benefits of executing malleable jobs on distributed memory HPC systems as compared to rigid jobs.

To compare and demonstrate the benefits, two scenarios are considered. In Scenario 1, a set of rigid jobs are submitted to the EBS, and in Scenario 2, a collection of malleable jobs are submitted to the EBS. For simplicity, the number of jobs is set to five in both cases, and the same type of application is used for all jobs. The inter-arrival time between the submission of each job is 100 seconds. For both scenarios, 18 compute nodes of SuperMUC-NG [58] are utilized. One compute node acts as a login node from where jobs are submitted to the EBS,



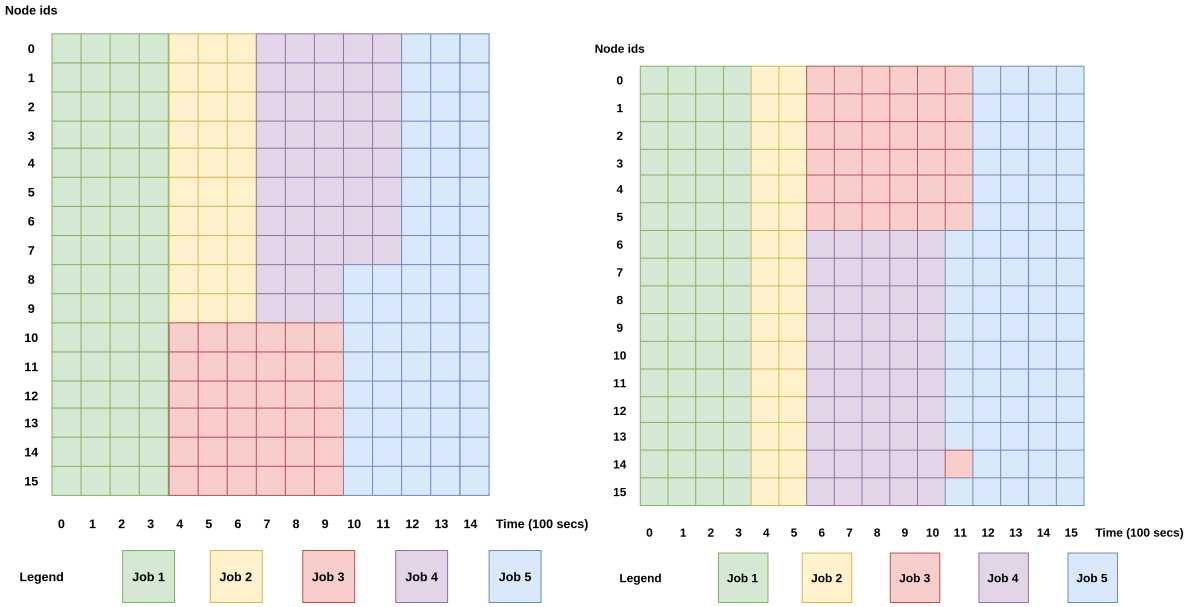
(a) Priority to Waiting Malleable Applications (PWMA).(b) Priority to Running Malleable Applications (PRMA).

Figure 7.2: Allocation of nodes overtime for the FPSMA (see Section 5.3.1) scheduling strategy and different job management policies.

and another runs the ERS (see Section 7.1).

The program used is a simple synthetic application implemented using the dynamic process management routines provided by the Invasive MPI library (see Section 3.1). For the synthetic malleable application, it is assumed that the addition of compute resources decreases the execution time, while a reduction in compute resources increases the execution time. The execution time of the application is modified by changing the number of iterations that need to be performed on the addition or removal of resources. Each job has a different static execution time, which is done by modifying the number of iterations that need to be performed.

The default scheduling strategy used in SLURM for scheduling applications is FCFS (see Section 5.1.1). The submitted jobs are added to a queue and assigned priorities wrt their arrival time. This scheduling strategy is used for evaluation in Scenario 1. The jobs are submitted to the EBS by specifying the `--consider-rigid` option in the job batch script (see Section 4.2). This disables all expand and shrink operations for the application, and it is treated as resource-static. Figure 7.1 shows the node allocation map over time for Scenario 1. Each submitted job has a different number of resource requirements. Job 1 requires four nodes, job 2 requires 16 nodes, job 3 requires six nodes, job 4 requires 10 nodes and job 5 requires eight nodes respectively.



(a) Priority to Waiting Malleable Applications (PWMA).(b) Priority to Running Malleable Applications (PRMA).

Figure 7.3: Allocation of nodes overtime for the EGS (see Section 5.3.1) scheduling strategy and different job management policies.

The *y*- axis in Figure 7.1 represents the individual node ids and the *x*- axis represents time in 100 seconds. On submission job 4, starts executing on four compute nodes. After job 2 arrives at t=100 seconds, it cannot start execution due to the non-availability of resources. Moreover, on arrival at t=200 seconds, job 3 cannot start execution even though the sufficient number of resources are available due to the FCFS scheduling policy. As a result, 12 compute nodes remain idle till job 1 completes execution at t=700 seconds. The idle noes are shown as white grid in Figure 7.1. This reduces the system utilization to 25%. After the completion of job 2, both jobs 3 and 4 are started since the required number of resources are available. Job 5 is started after job 3 completes execution. Overall, the five jobs take approximately 2000 seconds to complete, and the average system utilization is 59%.

The Elastic Batch Scheduler (EBS) supports different scheduling strategies for scheduling resource-elastic applications. An elastic job scheduling strategy (JSS) is a combination of job management and a malleability management policy. As described in Section 5.3.1, two management policies priority to running malleable applications (PRMA) and priority to waiting malleable applications (PWMA) are considered in this work. The malleability management policies supported by the EBS are favor previously started malleable applications (FPSMA) (see Algorithm 2), and Equi-grow shrink (EGS) (see Algorithm 3). For Scenario 2, the performance of all the above-mentioned policies is evaluated. In all cases the five jobs are submitted to the EBS with the `--min-nodes-invasic` (see Section 4.2) field in the batch script set to 4 nodes and the `--max-nodes-invasic` field for all jobs is set to 16 nodes. This

implies each job can expand to a maximum of 16 nodes and shrink to a minimum of 4 nodes. The number of nodes required for initiating the job, i.e., `--nodes parameter` is the same as in Scenario 1 and are set to 4, 16, 6, 10, 8 respectively. Furthermore, no constraints on the number of nodes that can be allocated after an expand or shrink operation for a particular job are assumed, i.e., the field `--node-constraints` is set to `NULL` in the batch script.

Figure 7.2a shows the allocation of nodes overtime for the FPSMA malleability management and the PWMA job management policies. On arrival of job 1 at t=0, it is expanded by the EBS to 16 nodes since no other job is currently waiting in the job queue. When job 2 arrives at t=100 seconds, job 1 is not shrunk since the maximum number of nodes that can be reclaimed from job 1 is 12, and job 2 requires 16 nodes for initiation. After job 3 arrives at t=200 seconds, job 1 is not shrunk since job 2 has higher priority than 3. The FPMSA algorithm along with the PWMA management policy only tries to shrink the currently running applications so as to launch the next highest priority job. If sufficient number of nodes cannot be reclaimed then the running applications are considered for expansion (see Algorithm 2).

Job 1 completes execution at $t = 400$ seconds and job 2 starts execution, Since job 3 is waiting in the job queue, the EBS decides to shrink job 2 to 10 nodes and start job 3. This significantly reduces the waiting time for job 2 and job 3 as compared to Scenario 1, in which the jobs start executing 600 and 700 seconds after their arrival, respectively. The response times for the jobs are also reduced. After job 2 completes, job 4 is initiated in 10 nodes. However, since job 5 is waiting in the queue, the EBS reclaims 2 and 6 nodes from jobs 3 and 4 respectively to start job 5. Job 4 is expanded to 8 nodes after job 3 completes at t=1300 seconds since it started earlier than job 5. The five jobs take approximately 1500 seconds to complete which also reduces the makespan as compared to Scenario 1.

The node allocation map over time for the FPSMA algorithm and PRMA job management policy is shown in Figure 7.2b. In contrast to PWMA, in PRMA the running resource-elastic applications are only considered for expansion. There are no shrink operations generated by the EBS. Similar to Figure 7.2a, job 1 is expanded to 16 nodes on arrival. However, job 2 is not reduced to start job 3, instead job 2 continues executing on 16 nodes. After job 4 finishes execution at t=1200 seconds, job 5 is started since 8 compute nodes are available. Job 3 is expanded to 8 compute nodes since it started before job 5. In this case, the execution of five jobs took approximately 1600 seconds, reducing the makespan as compared to in Scenario 1.

Figure 7.3a shows the node allocation map over time for the EGS malleability management and the PWMA job management policies. In contrast to the JSS, FPSMA and PWMA shown in Figure 7.2a, job 3 and job 4 are not reduced to run job 5 since it is not possible to equally reduce jobs 3 and 4 (see Algorithm 3). Job 5 starts executing at t=1100 seconds after job 4 is reduced to 8 compute nodes. Similar to Figure 7.2a, the five jobs take approximately 1500 seconds to complete execution. Hence, reducing the makespan as compared to the Scenario 1. The node allocation map over time for the EGS algorithm and PRMA job management policy

is shown in Figure 7.3b. The node allocation map is similar to the one shown in Figure 7.2b with one important point of difference. After job 4 completes its execution at t=1100 seconds, job 5 starts execution on 8 compute nodes in the system. At this point there are two idle nodes in the system which are equally divided between jobs 3 and 4.

It is important to note that every dynamic reconfiguration decision, i.e., expansion or reduction has a latency involved with it. This is not shown in Figures 7.2 and 7.3 and is discussed in more detail in Section 7.5. Taking overhead and latency into account, the average system utilization for the four cases is around 90%. Therefore, having malleable applications in a workload improves the system utilization, makespan, waiting and response times of the jobs as compared to a workload consisting only of resource-static jobs. For Scenarios 1 and 2 random jobs are selected as malleable.

7.4 Comparing the Performance of different scheduling strategies

Different strategies to support the combined scheduling of rigid and resource-elastic applications are implemented in the EBS (see Section 5.3.1). In this section, the performance of different strategies is analyzed and compared with respect to throughput and system utilization.

To compare the performance of the different implemented strategies, a modified version of the Effective System Performance (ESP) [41] benchmark is used. The original version of the ESP benchmark consists of 230 jobs with different job types running the same synthetic application. It was not possible to run 230 jobs on the current system (EBS + ERS) due to stability reasons. As a result, a workload consisting of 10 jobs of different job types is used. The application used in each job is a simple matrix multiplication which multiplies two matrices of size 1024 a particular number of times. The number of repetitions is changed to fit the job type's execution time. For the synthetic malleable application, it is assumed that the addition of compute resources decreases the execution time, while a reduction in compute resources increases the execution time. The type, number of nodes required, count and execution time for each job type is shown in Table 7.2.

The jobs are submitted to the EBS in a random order generated using a pseudo-random generator. However, the second job submitted is always of job type Z because, according to ESP benchmark, a job of type Z should always be submitted after 10% jobs. The inter-arrival time between each job is 60 seconds. In order to evaluate the performance of different strategies, three scenarios are considered. Scenario 1, in which 10% of the jobs are malleable, Scenario 2, in which 50% of the jobs are malleable and Scenario 3, in which 100% of the jobs are malleable. For each malleable job the `min-nodes-invasic` and `max-nodes-invasic`

Job Type	Number of nodes	Count	Static Execution Time [secs]
C	8	2	1204
D	4	2	513
G	2	3	696
J	1	2	522
Z	16	1	458

Table 7.2: Subset of job types and counts from the ESP Benchmark [41] used for testing.

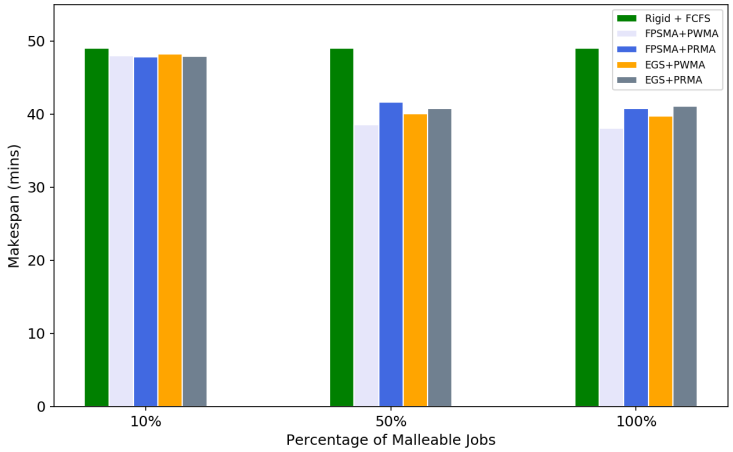


Figure 7.4: Makespan for varying amount of rigid and malleable jobs.

parameters are set to one and 16 respectively. The constraints on node allocations are ignored. The malleable jobs for Scenario 2 are chosen randomly. For all scenarios, the order of job submission is the same.

Five different strategies are evaluated. These include default scheduling strategy in SLURM, i.e., FCFS, favour previously started malleable applications first (FPSMA) and priority to waiting malleable applications (PWMA), FPSMA and priority to running malleable applications (PRMA), Equi-grow shrink (EGS) and PWMA, and EGS and PRMA (see Section 5.1.1). For comparing the different strategies four metrics, i.e., makespan, average system utilization, average waiting and response times are used (see Section 7.2).

Figure 7.4 shows the values of the system-centric metric makepspan for Scenarios 1, 2 and 3. For Scenario 1, when there is only one malleable job in the system, the makespan for the elastic scheduling strategies is similar to the FCFS scheduling policy with a difference of about one minute. For Scenarios 2 and 3, the different elastic scheduling strategies significantly outperform the FCFS rigid policy. The minimum makespan is achieved by the strategy FPSMA and PWMA of 38 minutes for Scenario 3 as shown in Figure 7.4. This signifies an increase in

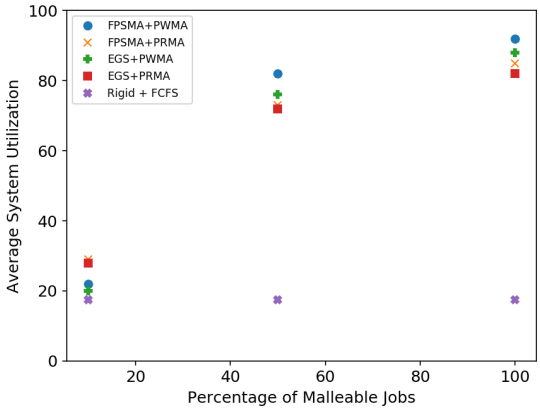
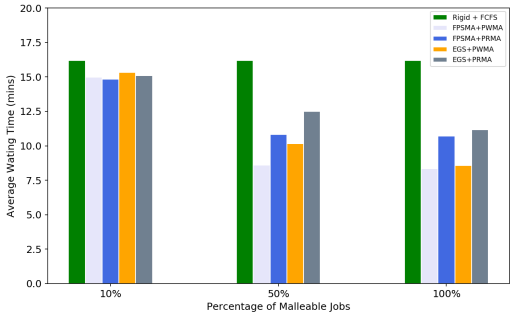
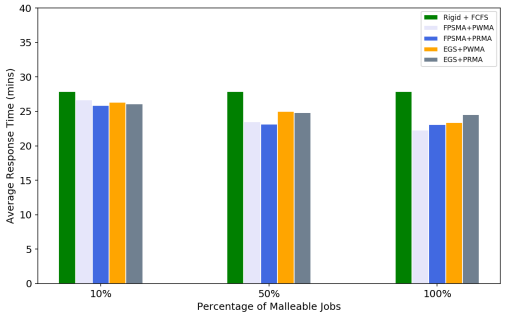


Figure 7.5: Average system utilization for varying amount of rigid and malleable jobs.



(a) Average waiting times for different scenarios.



(b) Average response times for different scenarios.

Figure 7.6: User-centric metrics for varying amount of rigid and malleable jobs.

throughput by 22% as compared to the rigid policy. However, there is no significant difference between the makespan values for Scenarios 2 and 3. This can be attributed to the usage of limited number of benchmarks and overhead due to repeated expansion and reduction operations (see Section 7.5).

The average system utilization for the three scenarios is shown in Figure 7.5. The average system utilization for rigid scheduling policy is 17.5%. For Scenario 1, maximum utilization of 29% is achieved by the FPSMA + PRMA scheduling strategy. For Scenarios 2 and 3, maximum utilization is achieved by the FPSMA + PRMA scheduling policy of 82% and 92% respectively. This highlights one of the major benefits of malleable applications that can expand and shrink depending on the current system state. The job management policy PRMA leads to less system utilization as compared to the PWMA, since it does not allow shrink operations and the number of expansion per job are limited to two. This is done to ensure fairness among

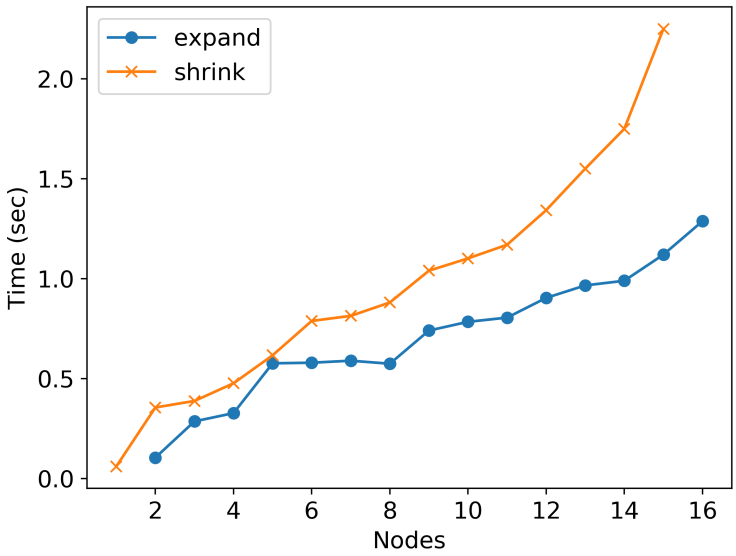


Figure 7.7: Latency for expand and shrink operations for a job running initially on 1 and 16 nodes respectively.

the jobs, because both FPSMA and EGS policies sort the applications in increasing order of their start times for expansion operations.

Figure 7.6 shows the average waiting and response times for the three scenarios. For Scenario 1, the average waiting and response times for the different elastic scheduling strategies is similar to the rigid scheduling policy. For Scenarios, 2, and 3, the minimum average waiting time of around 8 minutes is achieved by the FPSMA and PWMA scheduling strategy, as shown in Figure 7.6a. This indicates a performance improvement of 48.5% as compared to the rigid scheduling policy. For Scenario 2, FPSMA and PRMA policy achieves the average response time of 23.1 minutes, while for Scenario 3, FPSMA and PWMA policy achieves the minimum response time of 22.25 minutes respectively. This signifies a performance improvement of 20% in average response time for Scenario 3 as compared to the rigid policy. The scheduling strategy FPSMA and PWMA constantly outperforms the other strategies for all the three scenarios.

7.5 Analyzing the overhead of expand and shrink operations

Dynamic reconfiguration decisions for running resource-elastic applications are taken by the elastic batch scheduler (EBS). It is essentially a six step process as described in Section 4.2.

From the perspective of the resource manager, the EBS generates a Remote Procedure Call (RPC) request message called `SRUN_REALLOC_MSG`, which is sent to the elastic runtime scheduler (ERS). The ERS then sends the message to the running `srun` instance for the particular batch job, which then invokes the reallocation handler (see Section 4.3). The reallocation handler is responsible for updating the job's metadata and creating new processes or destroying preexisting processes.

From the application's perspective, dynamic reconfiguration of resources is supported through the routines provided by the Invasive MPI Library (see Section 3.1) and the creation of adaptation windows. Essentially it is the usage of three routines namely `MPI_Probe_adapt`, `MPI_Comm_adapt_begin` and `MPI_Comm_adapt_commit`. The `MPI_Probe_adapt` operation is primarily responsible for probing the resource manager for adaptation instructions. After the EBS tries to shrink or expand an application the `pending_argument` of the routine (see Listing 3.1.2) is set to `MPI_ADAPT_TRUE`. The application can then start an adaptation window by calling `MPI_Comm_adapt_begin` for an expansion or a reduction operation and can finish the adaptation by calling `MPI_Comm_adapt_commit`. The commit routine updates the communicator.

To quantify the overhead for expansion and reduction operations, the time for when the application starts an adaptation window and finally commits it is measured. A simple synthetic IMPI application is used for measurements. For calculating time, the standard MPI function `MPI_Wtime` is used. Figure 7.7 shows the overhead of expand and shrink operations. For expansion, the job initially starts from one compute node and is grown at every `SchedulerTick` by one node, up to 16 nodes. In the case of reduction, the job initially starts on 16 compute nodes and is shrunk at `SchedulerTick` by one node, until it runs only on one compute node. As described in Section 7.1, each compute node on SuperMUC-NG has 48 cores. This implies each expansion and reduction operation involves the creation and deletion of 48 processes. The values are obtained after running the expansion and reduction procedure five times and averaging the results.

The total time required for expansion operation increases with the increasing number of nodes, as shown in Figure 7.7. This can be attributed to latency in launching the new processes and communication with a greater number of nodes. The maximum latency observed is 1.29 seconds when the application is expanded from 15 to 16 compute nodes, and the minimum observed is 0.11 seconds when it is expanded from 1 to 2 compute nodes. Similar to expansion, the latency for shrink operations also increases with higher number of nodes. The maximum latency observed is 2.25 seconds when the application is shrunk from 16 to 15 compute nodes, and the minimum observed is 0.05 seconds when it is shrunk from 2 to 1 compute node. However, the observed latency for shrink operations is higher as compared to expansion operations. This is because the latency of a shrink operation depends upon the time required for preexisting tasks on preexisting nodes to complete. Every task must send a notification to the ERS after completion.

8 Conclusion and Future Work

As we move towards the era of exascale computing and the next generation of HPC systems, the importance of resource-elastic or malleable applications is increasing. Malleable applications can dynamically change their resources at runtime through expand and shrink operations. They can significantly improve system utilization, makespan, and reduce average waiting and response times. For supporting malleable applications on current and future HPC systems, three components are essential an adaptive job scheduler, an adaptive resource manager, and an adaptive runtime system.

The task of writing malleable applications is considerably challenging. The defacto standard for programming distributed memory HPC systems is the Message Passing Interface (MPI) standard. However, MPI, by default, is rigid and cannot be used for writing resource-elastic applications. Several adaptive parallel programming paradigms such as Charm++, Adaptive MPI have been introduced, which offer better support for malleability. However, they are not widely used. Moreover, the current batch systems, i.e., the combination of resource manager and the job scheduler, do not support the execution and dynamic reconfiguration of malleable applications and hence cannot utilize their potential. Towards this, in this work the scalable workload manager SLURM was extended to support the dynamic reconfiguration of queued malleable applications using the Invasive MPI (IMPI) library.

In this thesis, an Elastic Batch Scheduler (EBS) component for SLURM was developed. The EBS is primarily a scheduling plugin that implements the well-defined SLURM scheduling plugin API. It supports the combined scheduling of both rigid and malleable applications submitted by the users. It maintains two separate queues one for resource-static and the other for resource-elastic applications. The command-line tool `sbatch` provided by SLURM for submitting queued batch jobs was extended to submit jobs to each queue. Several options to support the constrained execution of malleable applications were added to `sbatch`. These include minimum and the maximum number of nodes that can be allocated to a malleable job and constraints on the number of nodes to be allocated after an expand or shrink operation. The EBS closely interacts with the Elastic Runtime Scheduler (ERS) to support dynamic reconfiguration of resources for running resource-elastic applications. This is done through the usage of Remote Procedure Call (RPC) request messages. Several of the RPC messages were modified to support expand and shrink operations for malleable applications submitted in batch mode.

Different job scheduling strategies (JSS) were implemented in the EBS and evaluated. In particular, two job management policies, namely priority to waiting malleable applications (PWMA) and priority to running malleable applications (PRMA) and two malleability management policies, namely favor previously started malleable applications first (FPSMA) and equi-grow shrink (EGS), were implemented. The job management policies are responsible for deciding when to expand or shrink an application. In contrast, malleability management policies are responsible for determining how to expand or shrink an application. Furthermore, to support performance-aware dynamic reconfiguration decisions for running malleable applications, a heuristic-based version of the FPSMA algorithm was implemented. The algorithm utilizes the criterion MPI time to compute time (MTCT) ratio for expand or shrink decisions. If the ratio is less than a particular threshold, then the application is expanded, otherwise, it is shrunk.

The performance of the different scheduling strategies was evaluated on the next generation SuperMUC-NG HPC system. All experiments were performed on 18 compute nodes with one compute node acting as a login node for submitting batch jobs and another running the ERS. In total, 16 compute nodes were available for running jobs. To demonstrate the benefits of malleable applications as compared to traditional jobs, two scenarios were considered. In Scenario 1, a set of five rigid jobs were submitted to the EBS with the FCFS scheduling strategy. In Scenario 2, five malleable jobs were submitted to the EBS. For scenario 2, the performance of all the different scheduling strategies was evaluated. The case study demonstrated that the presence of malleable applications in a workload significantly improves system utilization to 90% as compared to 59% in Scenario 1. Furthermore, the response and waiting times of the jobs and makespan of the workload are also considerably reduced.

The performance of the different scheduling strategies was compared by using a set of 10 jobs of different job types from the Effective System Performance (ESP) Benchmark. Three different scenarios with varying number of malleable and rigid jobs were considered. The job scheduling strategy FPSMA and PWMA constantly outperformed the other scheduling strategies. For a workload consisting of 100% malleable jobs, FPSMA and PWMA achieved a average system utilization of 92% and an improvement in makespan, average waiting and response times of 22%, 48.5% and 20% as compared to the default job scheduling strategy in SLURM.

The overhead of expand and shrink operations was also analyzed. The latency for periodically expanding a synthetic IMPI application running on one node to 16 nodes was measured. Similarly, the latency for periodically shrinking an application from 16 to 1 compute node was measured. The latency for expansion increases with an increasing number of nodes, and a maximum overhead of 1.29 seconds was observed. The same is true for shrink operations, and a maximum latency of 2.25 seconds was observed when shrinking from 16 to 15 compute nodes. The overhead observed for shrink operations was higher as compared to expansion operations since the latency depends upon the time taken for preexisting tasks to exit from

preexisting nodes.

8.1 Future Work

Adaptive resource management and scheduling systems are considered essential for the next generation of HPC systems. While this work only supports the scheduling and dynamic reconfiguration of malleable jobs, another form of adaptive jobs called evolving are significantly important for HPC systems. In contrast to malleable applications, the expansion and reduction of resources for the evolving application are requested by the application. The main principle behind evolving jobs is that the application requests for additional resources during its runtime, and if the request cannot be fulfilled, then the application must exit. While it is not possible to write evolving applications using IMPI, it can be done using the Elastic Phase Oriented Programming (EPOP) Model. The next development goal for this work is to extend the current EBS algorithm to support the combined scheduling and management of rigid, malleable and evolving jobs.

Another important point of research is the HPC community is the energy efficiency of HPC systems. Typical HPC centers consume a significant amount of power and have contracts with energy providers to maintain the power consumption of the system within a certain lower and upper bound, i.e., power-corridor. Violating this contract can lead to extra payments. The current implemented batch system can be extended to support power-aware dynamic reconfiguration decisions for resource-elastic applications. Particularly, the EBS can detect the power consumption of each running application and can give or reduce resources from applications that consume less or more power, respectively. This will ensure that the power corridor of the system is always maintained.

As the size of the current supercomputers continues to grow with increasing number of components the reliability of such systems in terms of mean time between failures (MTBF) continues to decrease. According to Schroeder et al. [59] future exascale systems are likely to fail every 3-26 minutes with the most common fault in large systems being single node failures. Therefore, there is a significant need for fault tolerance in HPC systems. The current batch system can be extended to support dynamic replacement of nodes during compute node failiures. Fault-tolerance can also be achieved through the development of an invasive checkpint-restart plugin in SLURM.

This work primarily uses the scalable workload manager SLURM for supporting dynamic reconfiguration of resource-elastic applications. Despite its benefits, SLURM was not designed for executing adaptive applications. Recently a new resource and management software (RJMS) called Flux [60] has been introduced in the literature. Flux is a highly scalable and

easily extensible RJMS and has been designed to support dynamic workload types, i.e., rigid, moldable, and malleable. A good point of research would be to evaluate the possibility of dynamic resource changes for resource-elastic applications written using IMPI on Flux.

List of Figures

List of Tables

Listings

Bibliography

[1] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.5 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860.

[2] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[3] H. W. Meuer. "The TOP500 Project: Looking Back Over 15 Years of Supercomputing Experience". In: *Informatik-Spektrum* 31.3 (June 2008), pp. 203–222. ISSN: 1432-122X. DOI: 10.1007/s00287-008-0240-6. URL: https://doi.org/10.1007/s00287-008-0240-6.

[4] *The Top500 list*. URL: https://www.top500.org/lists/2019/06/.

[5] Y. Georgiou. "Contributions for resource and job management in high performance computing". In: ().

[6] A. B. Yoo, M. A. Jette, and M. Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[7] T. Plewa, T. Linde, and V. G. Weirs. "Adaptive mesh refinement-theory and applications". In: ().

[8] S. Müller. *Adaptive multiscale schemes for conservation laws*. Vol. 27. Springer Science & Business Media, 2012.

[9] *MPI: A Message-Passing Interface Standard*. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[10] *The Top500 list*. URL: https://www.top500.org/statistics/perfdevel/.

[11] *The Green500 list*. URL: https://www.top500.org/green500/.

[12] M. Chadha and M. Gerndt. "Modelling DVFS and UFS for Region-Based Energy Aware Tuning of HPC Applications". In: *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. 2019, pp. 805–814. DOI: 10.1109/IPDPS.2019.00089. URL: https://doi.org/10.1109/IPDPS.2019.00089.

[13] M. Chadha, T. Ilsche, M. Bielert, and W. E. Nagel. "A statistical approach to power estimation for x86 processors". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2017, pp. 1012–1019.

[14] M. Chadha, A. Srivastava, and S. Sarkar. "Unified power and energy measurement API for HPC co-processors". In: *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. Dec. 2016, pp. 1–8. DOI: 10.1109/PCCC.2016.7820633.

[15] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. "Invasive computing: An overview". In: *Multiprocessor System-on-Chip*. Springer, 2011, pp. 241–268.

[16] C. Ureña and I. Alberto. "Resource-Elasticity Support for Distributed Memory HPC Applications". PhD thesis. Technische Universität München, 2017.

[17] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee. "A survey of MPI usage in the US exascale computing project". In: *Concurrency and Computation: Practice and Experience* (2017), e4851.

[18] *MPICH: High-Performance portable MPI*. URL: https://www.mpich.org/.

[19] *OpenMPI: Open-Source High-Performance computing*. URL: https://www.open-mpi.org/.

[20] L. V. Kalé, S. Kumar, and J. DeSouza. "A malleable-job system for timeshared parallel machines". In: *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*. IEEE. 2002, pp. 230–230.

[21] L. V. Kale and S. Krishnan. "Charm++ A portable concurrent object oriented system based on C++". In: *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. 1993, pp. 91–108.

[22] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, et al. "Parallel programming with migratable objects: Charm++ in practice". In: *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 647–658.

[23] C. Huang, O. Lawlor, and L. V. Kale. "Adaptive mpi". In: *International workshop on languages and compilers for parallel computing*. Springer. 2003, pp. 306–322.

[24] C. Huang, G. Zheng, and L. V. Kalé. "Supporting adaptivity in MPI for dynamic parallel applications". In: *Rapport technique, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign* 14 (2007).

[25] C. Huang, G. Zheng, L. Kalé, and S. Kumar. "Performance evaluation of adaptive MPI". In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2006, pp. 12–21.

[26] S. Chakravorty, C. L. Mendes, and L. V. Kalé. "Proactive fault tolerance in MPI applications via task migration". In: *International Conference on High-Performance Computing*. Springer. 2006, pp. 485–496.

[27] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale. "Hierarchical load balancing for charm++ applications on large supercomputers". In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE. 2010, pp. 436–444.

[28] I. Comprés, A. Mo-Hellenbrand, M. Gerndt, and H.-J. Bungartz. "Infrastructure and api extensions for elastic execution of mpi applications". In: *Proceedings of the 23rd European MPI Users' Group Meeting*. 2016, pp. 82–97.

[29] R. Sudarsan and C. J. Ribbens. "ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment". In: *2007 International Conference on Parallel Processing (ICPP 2007)*. Sept. 2007, pp. 44–44. DOI: 10.1109/ICPP.2007.73.

[30] G. Martın, D. E. Singh, M.-C. Marinescu, and J. Carretero. "Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration". In: *Parallel Computing* 46 (2015), pp. 60–77.

[31] Y.-K. Kwok and I. Ahmad. "Static scheduling algorithms for allocating directed task graphs to multiprocessors". In: *ACM Computing Surveys (CSUR)* 31.4 (1999), pp. 406–471.

[32] K. Pruhs. "Competitive online scheduling for server systems". In: *ACM SIGMETRICS Performance Evaluation Review* 34.4 (2007), pp. 52–58.

[33] D. G. Feitelson and L. Rudolph. "Toward convergence in job schedulers for parallel supercomputers". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1996, pp. 1–26.

[34] A. Gupta, B. Acun, O. Sarood, and L. V. Kalé. "Towards realizing the potential of malleable jobs". In: *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE. 2014, pp. 1–10.

[35] J. Hungershofer. "On the combined scheduling of malleable and rigid jobs". In: *16th Symposium on Computer Architecture and High Performance Computing*. IEEE. 2004, pp. 206–213.

[36] T. E. Carroll and D. Grosu. "Incentive Compatible Online Scheduling of Malleable Parallel Jobs with Individual Deadlines". In: *2010 39th International Conference on Parallel Processing*. Sept. 2010, pp. 516–524. DOI: 10.1109/ICPP.2010.60.

[37] H. Sun, Y. Cao, and W.-J. Hsu. "Fair and efficient online adaptive scheduling for multiple sets of parallel applications". In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE. 2011, pp. 64–71.

[38] G. Utrera, S. Tabik, J. Corbalan, and J. Labarta. "A job scheduling approach for multi-core clusters based on virtual malleability". In: *European Conference on Parallel Processing*. Springer. 2012, pp. 191–203.

[39] S. Prabhakaran, M. Iqbal, S. Rinke, C. Windisch, and F. Wolf. "A batch system with fair scheduling for evolving applications". In: *2014 43rd International Conference on Parallel Processing*. IEEE. 2014, pp. 351–360.

[40] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale. "A batch system with efficient adaptive scheduling for malleable and evolving applications". In: *2015 IEEE international parallel and distributed processing symposium*. IEEE. 2015, pp. 429–438.

[41] A. T. Wong, L. Oliker, W. T. Kramer, T. L. Kaltz, and D. H. Bailey. "ESP: A system utilization benchmark". In: *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE. 2000, pp. 15–15.

[42] O. Sonmez, H. Mohamed, W. Lammers, D. Epema, et al. "Scheduling malleable applications in multicluster systems". In: *2007 IEEE International Conference on Cluster Computing*. IEEE. 2007, pp. 372–381.

[43] T. Hoefler and M. Snir. "Writing Parallel Libraries with MPI - Common Practice, Issues, and Extensions". In: *Recent Advances in the Message Passing Interface*. Ed. by Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 345–355. ISBN: 978-3-642-24449-0.

[44] *MVAPICH:MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE*. URL: http://mvapich.cse.ohio-state.edu/.

[45] J. John. "The Elastic Phase Oriented Programming Model for Elastic HPC Applications". In: 2018.

[46] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[47] I. Foster and C. Kesselman. "The Globus project: A status report". In: *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98)*. IEEE. 1998, pp. 4–18.

[48] J. Kim, W. J. Dally, S. Scott, and D. Abts. "Technology-driven, highly-scalable dragonfly topology". In: *2008 International Symposium on Computer Architecture*. IEEE. 2008, pp. 77–88.

[49] T. Shanley. *InfiniBand network architecture*. Addison-Wesley Professional, 2003.

[50] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. *LULESH Programming Model and Performance Ports Overview*. Tech. rep. LLNL-TR-608824. Livermore, CA, Dec. 2012, pp. 1–17.

[51] *The Top500 list*. URL: https://www.top500.org/statistics/efficiency-power-cores/.

[52] K. Agrawal, I. A. Lee, J. Li, K. Lu, and B. Moseley. "Practically Efficient Scheduler for Minimizing Average Flow Time of Parallel Jobs". In: *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. 2019, pp. 134–144. DOI: 10.1109/IPDPS.2019.00024. URL: https://doi.org/10.1109/IPDPS.2019.00024.

[53] K. Jansen, M. Maack, and A. Mäcker. "Scheduling on (Un-)Related Machines with Setup Times". In: *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. 2019, pp. 145–154. DOI: 10.1109/IPDPS.2019.00025. URL: https://doi.org/10.1109/IPDPS.2019.00025.

[54] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson. "An application of bin-packing to multiprocessor scheduling". In: *SIAM Journal on Computing* 7.1 (1978), pp. 1–17.

[55] R. I. Davis and A. Burns. "A survey of hard real-time scheduling for multiprocessor systems". In: *ACM computing surveys (CSUR)* 43.4 (2011), pp. 1–44.

[56] *Slurm Workload Manager*. URL: https://slurm.schedmd.com/sched_config.html.

[57] *Slurm Workload Manager*. URL: https://slurm.schedmd.com/select_design.html.

[58] *SuperMUC-NG*. URL: https://doku.lrz.de/display/PUBLIC/SuperMUC-NG.

[59] B. Schroeder and G. A. Gibson. "Understanding failures in petascale computers". In: *Journal of Physics: Conference Series*. Vol. 78. 1. IOP Publishing. 2007, p. 012022.

[60] 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*. IEEE. 2014, pp. 9–17.