DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Towards Dynamic Resource Management with MPI Sessions and PMIx

Dominik Huber

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Towards Dynamic Resource Management with MPI Sessions and PMIx

In Richtung Dynamisches Resourcen Management mit MPI Sessions und PMIx

Author:	Dominik Huber
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Prof. Dr. Martin Schreiber
	Dr. Isaías A. Comprés Ureña
Submission Date:	15.12.2021

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

Munich, 15.12.2021

Acknowledgments

Throughout my work on this master's thesis, I have received support and guidance from various sides.

First and foremost I like to thank my adviser Prof. Dr. Martin Schreiber, who consistently assisted me during the complete process of my work. Your feedback and tips on various professional, technical and organizational aspects were invaluable not only for this thesis but also for my academic path of the last years.

I like to thank Dr. Isaías A. Comprés Ureña for providing me with his expertise and experience on HPC systems and system components. Moreover, by proving me with a docker cluster development environment you greatly contributed to the development of the prototype.

I would like to acknowledge kindly Prof. Dr. Martin Schulz for giving me this research opportunity. I also appreciate that you welcomed me to join meetings of the MPI Sessions WG, which greatly improved my understanding of the MPI Sessions model.

Furthermore, I like to appreciate Howard Pritchard's contribution of introducing me to the MPI Sessions related efforts of the Open-MPI project.

Most importantly, this work would not have been possible without the help of my family and friends. To my parents and my sister I like to express my gratitude not only for supporting my work on this thesis but also for always standing by me throughout all phases of my life. In addition, I like to thank especially my best friend and flatmate Juri Littwin. Without your consistent emotional support and our enjoyable shared activities this accomplishment would not have been possible.

Abstract

In the last decade, the scale of HPC systems has increased substantially, often featuring thousands of compute nodes and first exascale systems are expected to soon be available. For such large-scale systems, efficiency of resource utilization is an important and complex objective, which motivates research on efficient solutions for both, hardware and software.

To this end, dynamic resource management of HPC jobs is a promising research direction. Dynamic resource management opens up the potential for better scheduling and load-balancing to improve e.g. system utilization and throughput. However, enabling dynamic resource management requires fundamental changes in several layers of the system software management stack, as well as the application.

This thesis focuses on resource adaptive solutions for MPI based applications and runtime systems to facilitate dynamic resource management on HPC systems. In this context, the potential of the new MPI Sessions model is explored, which was recently introduced in the MPI 4.0 Standard.

We propose a new concept based on extensions to the MPI Standard which allows the development of resource adaptive MPI applications by exploiting the flexible process management of the MPI Sessions model. Moreover, we describe extensions and usage of the Process Management Interface (PMIx) Standard for developing portable, dynamic MPI runtime environments supporting such applications.

We developed a prototype based on the widely used Open-MPI implementation. Our tests on a state-of-the-art HPC cluster demonstrate the practicability of our approach to dynamically vary the number of MPI processes. In preliminary performance tests we measured moderate overheads of 108 - 175 ms for resource addition and 88 - 126 ms for resource subtraction on up to four nodes and a total of 122 processes.

The formulation of our design in terms of the MPI and PMIx Standards provides an abstraction from the concrete implementation of host systems and MPI runtime environments, thus aiding integration into software management stacks supporting dynamic resource management. Moreover, due to the generality of the underlying mechanisms, application of our concept is not limited to malleable jobs. It also provides potential for evolving jobs, intra-job load-balancing and fault tolerance, which should be addressed in future work.

Contents

List of Abbreviations

DPM	Dynamic Process Management
DVM	Distributed Virtual Machine
HPC	High Performance Computing
MCA	Modular Component Architecture
MPI	Message Passing Interface
OPAL	Open Portability Access Layer
PMIx	Process Management Interface For Exascale
PRRTE	PMIx Reference Runtime Environment
RM	Resource Manager
RML	Remote Messaging Layer
RTE	Runtime Environment
SLURM	Simple Linux Utility For Resource Management
SMS	System Management Software Stack
SWE	Shallow Water Equations

1. Introduction

The scale of modern High Performance Computing (HPC) systems has increased over the last decades and the next generation of systems is expected to reach exascale performance. Such performance is achieved by massive parallelism, resulting in systems featuring millions of compute cores.

This advance in the scale of HPC systems has introduced several new challenges, such as achieving efficient system utilization and handling of low mean-time-to-failure. To this end, the research area of dynamic resource management has gained increasing interest, due to its potential to improve the energy efficiency of the system and to support fault tolerance. In particular, malleable HPC jobs - jobs which can dynamically vary the number of processes they are running on - play an important role in increasing energy efficiency of the system [22, 42, 34]. This is because malleable jobs allow for more flexible job scheduling, which reduces the time jobs are waiting in the job queue and enables load-balancing, thereby improving overall system throughput. To give a simple example, consider a cluster with ten compute nodes. When statically scheduling a job on six nodes of the cluster, a second job requiring a minimum of five nodes cannot be started until the first job has finished. However, if the first job would be malleable, dynamic scheduling would allow immediate start of the second job by shrinking the malleable job by just one node.

Moreover, today's HPC workloads tend to become more dynamic, exhibiting changing resource requirements over time. For instance, adaptive mesh-refinement methods often have unpredictable, dynamically changing degrees of parallelism. With traditional static batch scheduling such jobs are executed with a fixed number of processes. To ensure successful job completion, users typically use a pessimistic upper bound estimate of expected resources requirements. Clearly, this leads to inefficient resource usage, especially in phases where the workload provides less parallelism. Dynamically changing the number of processes of such a job would improve this by assigning resources based on the current resource requirements.

While the benefits of dynamic resource management are well-known, enabling it on HPC systems is still currently researched. One of the greatest challenges of dynamic resource management is the need for extensions to and coordination between several components of the system management software stack (SMS) on these systems. This includes the scheduler, resource manager (RM) and runtime environment (RTE) as well

as applications. With the Message Passing Interface (MPI) being the de facto standard for writing programs for distributed compute systems, dynamic MPI jobs, i.e. dynamic MPI applications and RTEs, are a major building block towards dynamic resource management.

Thus, this work is on investigating new ways to introduce resource dynamicity to MPI applications and their RTE. In particular, we focus on the potential of the new *MPI Sessions* model introduced in the fourth version of the MPI Standard. In contrast to the traditional *MPI World* model, the MPI Sessions model is not based on global initialization of MPI and improves runtime-awareness of MPI programs. A central concept of the new model are *process sets*, which provide an abstract description of ensembles of processes used by both, the MPI application and the MPI RTE.

Our approach is based on other work [12], which introduced an extended MPI Sessions interface allowing dynamic changes of the number of processes of MPI applications. In this approach, MPI processes can query the RTE for available resource changes which are represented by *process sets*. Through *set operations* of *process sets*, the application can reconfigure itself and subsequently establish adapted communication to apply these changes. So far, the functionality of this approach was only demonstrated in an emulated MPI Sessions environment.

We extend this approach and integrate it into a dynamic MPI RTE based on extensions of the MPI Sessions interface, the *process set* concept and the Process Management Interface for Exascale (PMIx) standard. PMIx is a widely used programming interface standard for common services on HPC systems, including the creation of MPI RTEs as well as coordination with RMs and other components of the SMS. In our concept, new and existing PMIx functionalities are used for coordination between the MPI application and the RTE, as well as for introducing dynamicity to the RTE itself. Thus, flexible integration into the SMS is facilitated, especially for systems already relying on PMIx.

Moreover, to demonstrate the basic functionalities of our design in a realistic setting, we developed a prototype extending the Open-MPI, Open-PMIx and PMIx Reference Runtime Environment (PRRTE) implementations. This prototype allows growing and shrinking the number of processes of MPI applications at runtime. We used a synthetic benchmark as well as a resource-adaptive PDE solver for a hyperbolic problem on a fixed grid to test the functionality and performance of our approach on a modern HPC cluster.

The remainder of this work is structured as follows:

In Sec. 2 relevant literature regarding resource adaptive MPI is reviewed.

Sec. 3 provides background about the typical components in the SMS on modern HPC clusters and their requirements in the context of dynamic resource management. Here, we also define the scope of our concept in terms of these components and

respective layers.

Subsequently, in Sec. 4 and Sec. 5 the concepts of MPI and PMIx are introduced and discussed in the context of this work.

Building on top of the background provided in previous sections, in Sec. 6 we describe in detail our concept for dynamic resource management with MPI Sessions and PMIx. We first introduce necessary definitions and extensions to the process set concept. Subsequently, we describe the new interface based on the usage and extension of MPI and PMIx on the application as well as the runtime level. Moreover, we illustrate the usage of our design based on two examples and discuss possible extensions for other use cases such as intra-job load-balancing and fault tolerance.

In Sec. 7 we describe our implementation of a prototype of this new concept on top of the Open-MPI project.

This prototype is evaluated in Sec. 8 based on the results of a synthetic benchmark as well as a resource-adaptive PDE solver run on up to 4 nodes of an state-of-the-art HPC system.

Finally, we discuss our work and possible directions for future research in Sec. 9 and conclude the thesis with a summary in Sec 10.

2. Related Work

In the literature numerous, diverse approaches towards enabling dynamic resource management with MPI exist.

Here, we differentiate four common strategies constituting the basis of most works: *Checkpoint and restart*, *process virtualization*, *MPI-2 Dynamic Process Management (DPM)* and *malleable* `MPI_COMM_WORLD` *communicators*. Often a combination of these strategies is used to fully exploit their potential.

Many approaches revolve around *checkpointing* techniques, borrowing concepts from research on fault tolerance. They use ckeckpointing in combination with application restart and/or reconfiguration to achieve adaptivity of MPI programs. However, the approaches vary in the particular techniques used for checkpointing and reconfiguration. One of the first approaches providing re-configurable MPI applications through checkpoint and restart mechanisms on heterogeneous machines was SRS [48]. However, a comparably large overhead resulted from storing checkpoints and re-initializing the application. In [10, 9] the authors introduce the Process Checkpointing and Migration library PCM. They use checkpointing in combination with MPI-2 Dynamic Process Management, thereby avoiding complete application restart. In [16] Linux shared memory is used to improve performance of checkpointing together with process virtualization and the authors also provide some integration into adaptive RMs.

Contrary to these works our concept does not rely on checkpointing.

Another common approach includes process virtualization. Here resource adaptation is achieved through migration of light-weight user level processes and over-decomposition. Utera et al. [47] demonstrated the benefits of Folding by JobType (FJT), which combines moldability and folding to adapt to changes in resources. In [4] the authors use a related technique called dynamic CPUSETs mapping for supporting malleability in MPI on the system level as well as MPI-2 Dynamic Process Management on the application level.

Charm++ [23] is a portable, object-oriented, parallel programming system providing migratability and adaptivity through over-decomposition and asynchronous message-driven execution. This allows for a dynamic parallel RTE capable of grow/shrink operations using task migration, which has been demonstrated to be integratable into dynamic RMs [16, 34].

Adaptive MPI (AMPI) [19] implements MPI on top of the Charm++ language and

RTE using its capabilities to map virtual MPI processes arbitrarily on physical processors. This allows for migration of (virtual) MPI processes, load-balancing and automatic checkpointing. As AMPI is based on Charm++, the aforementioned works on dynamic parallel Charm++ RTEs enable dynamic resource management with AMPI jobs. However, by virtualization of MPI processes one deviates from the MPI execution model of a distributed memory system. Moreover, it shifts certain control over the program execution from developers to the Charm++ RTE.

Our approach does not make use of process virtualization and strictly relies on the MPI paradigm.

The introduction of *DPM* capabilities in MPI-2 has inspired various approaches towards resource dynamic MPI applications. Most works make use of the `MPI_Comm_Spawn` routine to spawn new processes into a separate `MPI_COMM_WORLD` which is connected via an inter-communicator to the spawning process(es).

A simple approach was demonstrated in [31] using the OAR Resource Manager and `MPI_Comm_Spawn` to grow a job from an initial size to an extended size when resources become available. Shrinking was achieved via fault tolerance procedures, returning back to the initial size.

ReShape [41] and Flex-MPI [27] are examples of frameworks on top of MPI implementations providing load balancing through performance monitoring and resizing based on *MPI-2 DPM* functionalities. These approaches are limited by certain constraints of the MPI-2 Dynamic Process Management which we discuss in detail in Sec. 4.3. One example of such limitations is that processes sharing an `MPI_COMM_WORLD` can only be terminated collectively. Again, our approach is different as it does not make use of the MPI-2 DPM functionalities.

Finally, a fourth approach is to augment process managers/MPI RTEs with the capability of growing and shrinking the `MPI_COMM_WORLD` communicator in coordination with the application. This is different to MPI-2 Dynamic Process Management where processes are spawned into separate, static `MPI_COMM_WORLD` communicators. Most of these approaches are based on forms of Invasive MPI, combining MPI with the ideas of invasive computing [45]. To this end, several publications describe an extended MPI interface allowing for invasive MPI programs [46] and malleable MPI jobs integrated into the Simple Linux Utility for Resource Management (SLURM) workload manager [7, 5].

These works have some similarity with our approach as they use PMI(x) to realize the integration into process managers. However, these approaches are collective over the `MPI_COMM_WORLD` communicator, thus all processes need to participate in the adaption process. Although this might work well from a pure resource utilization point of view, this poses severe issues on applications which use more than one communicator. Such applications include, e.g., coupled simulations and the utilization of dedicated I/O

nodes to name just a few.

Thus, for our concept we use the new MPI Sessions model, which is not based on a global `MPI_COMM_WORLD` communicator but instead relies on *process sets*. Here, resource adaption is not collective over all MPI processes but only over concerned processes.

Our work is mainly based on the interface proposed in [12], which we extend and realize here. For this we formulate the integration into process managers and RTEs in terms of PMIx capabilities.

3. System Management Software Stack

Dynamic resource management on distributed systems requires support by and coordination between various components of the System Management Stack (SMS). This section summarizes the basic structure of modern HPC systems and discusses the requirements for basic SMS components in the context of dynamic resource management. Subsequently, the scope of this thesis is defined in terms of the concerned SMS components and its application on HPC systems.

3.1. Classification of HPC-Jobs

Typically, an HPC system allows users to submit jobs to be executed on the compute resources of the system. Such jobs can be classified regarding their dynamic behavior. A frequently used classification is shown in [14, Table 3.1]. This classification categorizes jobs in terms of the entity deciding about the resources and the time of the decision. Consequently the following types of jobs can be discriminated.

Rigid jobs can only be started with a certain number of resources, which does not change during execution. *Moldable jobs* are more flexible in the number of resources they are started with e.g. any number above a certain minimum. Similar to *rigid jobs*, once started, the number of resources stays constant during execution. Throughout this thesis we refer to these two types of jobs as *static*, as the resource allocation does not change during execution.

Evolving and *malleable* jobs change their resource allocation during execution. For *malleable jobs*, the resource allocation is controlled by the system and such jobs therefore need to dynamically adapt to changing resource availability. With *Evolving* jobs, the resource allocation is changed dynamically on behave of the job itself, requiring support

Who decides	**When is it decided**	
	at submittal	during execution
user	rigid	evolving
system	moldable	malleable

Table 3.1.: Adjusted table from [14]: **Classification of HPC-Jobs.**

of the system to react to corresponding allocation requests. As most systems strive to globally optimize the usage of the system's resources, real evolving jobs are uncommon, especially in the HPC domain. Instead, the system and jobs might cooperatively decide on the resource allocation which sometimes is referred to as *adaptive jobs*.

In this thesis, we use the term *dynamic jobs* for all jobs, which change their resource allocation during execution, i.e. *malleable*, *evolving* and *adaptive* jobs. The goal of this thesis is to develop a general concept applicable to *dynamic jobs*.

A concept that partly overlaps with dynamic resource management is *fault tolerance*. Fault tolerant jobs are jobs which are able to recover from faults such as hardware failure. Similarly to *malleable jobs*, fault tolerance requires dynamic adaptation of jobs to varying resource availability. Fault tolerance, however, poses numerous additional challenges beyond dynamic resource management, such as failure detection, checkpointing for data preservation etc. Thus, fault tolerance is not a primary target of the concept developed in this thesis, although approaches for fault tolerance might profit from the underlying mechanism of our approach.

3.2. Job Scheduler

The job scheduler of an HPC system is responsible for efficiently matching incoming jobs with available resources of the system. Especially in large scale, efficient system usage reduces the environmental impact (green computing), saves monetary resources for system operation and improves the system service for users.

HPC systems usually use batch scheduling, where users submit a job script describing the resource requirements of the job which is then used by the scheduler to decide where and when to run the job. The scheduler assigns resources to jobs according to particular performance metrics such as system throughput, response time (e.g. bounded slowdown), minimum makespan and fairness between users [13]. To this end, different scheduling policies are employed such as First-Come-First-Serve, Short-Job-First and Long-Job-First. These policies are often combined with backfilling and starvation prevention techniques to improve system utilization.

A central requirement for efficient scheduling is a precise modeling of jobs to e.g. assign corresponding priorities. Such models often consider two characteristics: job arrival and job workload/resource requirements. A large body of work has been dedicated to parallel job modeling. Often, historical traces of HPC system usage are used to characterize workloads. Recently, research especially focuses on probabilistic (machine learning) approaches such as regression models for workload classification [51] and Markov Chains for modeling inter-job correlations [38].

3.2.1. Traditional Batch Scheduling

Traditionally, HPC jobs are rigid (or moldable). Thus, jobs in the job queue have a fixed (range of) resource requirements for starting job execution, which cannot change during execution. Models of such jobs are therefore restricted, imposing natural limits on the potential for schedulers to find the most efficient job schedule.

For instance, "it is common for jobs to be stuck in the queue because they require just a few more processors than are currently available" [42]. Moreover, users usually overestimate the resource requirements of their job to ensure job completion, leading to inefficient usage of the compute cluster. This is especially problematic in the light of increasingly dynamic resource requirements of scientific workloads such as adaptive mesh refinement. Thus, although sophisticated scheduling strategies exist, the static nature of traditional HPC jobs limits the potential of efficient usage of HPC clusters.

3.2.2. Dynamic Batch Scheduling

The advant of malleable applications paves the way towards new scheduling strategies to "improve the utilization of clusters as well as an individual job's turn around time" [42]. However, especially workload modeling becomes more difficult when deviating from rigid jobs [13]. A key parameter for modeling dynamic jobs are parallel efficiency metrics such as gradients, i.e. the change of performance when expanding/shrinking jobs. Schedulers can use this parameter to incorporate the potential of resizing malleable jobs into their scheduling decision.

Scheduling strategies for malleable jobs are actively researched and several such strategies have been proposed in the literature. Sudarsan et al. [42] divides them into two categories: a "processor allocation strategy decides which applications to expand and by how much whereas a processor harvesting strategy decides which application to contract and by how much". A harvesting strategy can for instance be used to solve the problem of jobs stuck in the queue requiring a small amount of additional resources to start. The allocation strategy can be used to efficiently distribute unused resources between running jobs. A combination of both, potentially augmented with profiling data and locality considerations would allow efficient load balancing and overall improved scheduling performance.

Indeed, the potential of scheduling dynamic jobs has been demonstrated frequently. For instance, it was shown that assigning resources to malleable applications based on estimated parallel speedups is superior compared to scheduling of moldable jobs [22]. Similarly, in [33] the author presented a scheduling approach for malleable jobs that outperforms static scheduling for different application classes from a parallel benchmark suite. It was also shown that the performance improvements become more

prevalent when the fraction of malleable jobs in the system increases, emphasizing the importance of developing malleable jobs. Consequently, also performance in terms of power consumption can be improved by scheduling malleable jobs [28].

Thus, there clearly is a need for dynamic resource management on HPC systems to exploit the potential of such dynamic scheduling approaches, which is the main motivation of this work.

3.3. Resource Manager

While the scheduler decides which job to run when and on which resources, the RM is responsible for performing the according resource allocation and manage the execution of the job. Thus, it works closely with the scheduler, constituting the second part of the batch system.

3.3.1. General Tasks of the Resource Manager

General tasks of the RM include resource allocation, process startup, hardware and job monitoring, job control, job termination and resource cleanup. The RM is aware of the overall system's state on the hardware-level as well as the job-level. This allows it to provide system services to jobs, actualize resource allocation as determined by the scheduler and to manage hardware aspects such as power consumption and temperature.

It is also responsible for the creation and/or management of a parallel RTE which we describe in a separate section. While we make a conceptional distinction between the RM and the RTE in this thesis, it has to be noted that the RTE is often integrated into or strongly coupled with the RM. For instance, the widely used SLURM [50] workload manager provides integrated support for MPI using PMIx. However, it also supports MPI jobs running in dedicated runtimes such as the PRRTE inside of a SLURM allocation.

3.3.2. Resource Management of Dynamic Jobs

When dealing with malleable or evolving jobs, the RM needs to provide dynamic allocation of resources to jobs. On the one hand, this requires tighter interaction with running applications/runtimes to, for instance, declare the schedulers decision to remove certain resources from the job or to receive requests for resource adaption from the application. On the other hand, it requires more frequent interaction with the scheduling component of the system to pass on the jobs' dynamically changing resource requirements.

Moreover, execution environments themselves are becoming more adaptive, requiring the RM to perform additional tasks, such as dynamic power redistribution and management of networks to avoid contention.

This thesis does not deal with dynamic resource allocation but rather with the creation of dynamic MPI RTEs, thus resource management aspects are only discussed peripherally.

3.4. Runtime Environment

A RTE can be defined as "a software package that resides between the operating system (OS) and the application programming interface (API) and compiler. An instantiation of a runtime is dedicated to a given application execution" [39]. In the context of HPC systems, the RTE can be seen as a layer between the batch system and a concrete application running on the system. As MPI is a programming standard, which is widely used for HPC applications, here we focus on MPI RTEs.

3.4.1. MPI Runtime Environments

To run MPI applications on distributed compute systems an MPI RTE is needed. MPI RTEs provide support to "launch the application, to provide out-of-band communications, enabling I/O forwarding and bootstrapping of the connections of high-speed networks, and to control the correct termination of the parallel application" [1]. Therefore, they are often integrated to some degree into the RM or make use of the services of RMs.

As a layer between the batch system and applications, it hides application specifics from the batch system and system specifics from the application. For instance, the batch system might only assign a certain number of nodes to a job leaving the responsibility of creating and managing processes to the RTE. Such processes on the other hand are only aware of the provided environment, which is an abstraction of the actual system they are running on.

Both, MPI RTEs and RMs, can be built on the basis of PMIx which enables well defined interactions and varying degrees of integration of MPI RTEs and RMs.

3.4.2. Requirements of a Dynamic MPI Runtime Environment

Dynamic MPI applications require a dynamic MPI RTE. In particular, such an environment needs to be able to dynamically adapt the number of processes available to applications for communication purposes. To give an example, the scheduler might instruct the RM to grow or shrink the allocation for a particular MPI job. In this

context, the dynamic MPI RTE can act as a mediator between the system and the MPI job. It takes on the task of launching processes on newly allocated resources or terminates processes on resources to be excluded from the allocation, while sustaining a consistent environment for the running processes. A great challenge in this regard is the creation of a concise mechanism for interactions between the application and the RTE. In particular, this includes announcements of such changes to the application and the management of the applications' adaption to these changes without disrupting its execution.

Overcoming these challenges is the main objective of this work.

3.5. Dynamic MPI Applications

A prerequisite of dynamic resource management are *dynamic jobs* and in particular dynamic (MPI-)applications, i.e. applications that can vary the number of processes over time.

Dynamic parallelism is a property that depends on the underlying problem as well as the implementation to solve this problem. For simplicity, here we discuss two very general classes of parallel patterns: *data-parallelism* and *task-parallelism*.

In *data-parallelism*, a certain task needs to be executed on certain data. The parallelism arises from decomposing the data which allows the parallel execution of the same task on smaller pieces of data. This type of parallelism is extremely common in HPC applications such as physical simulations on grids. Typically, the execution is based on parallel loops, where each iteration performs a data parallel task on the input data followed by a global synchronization and data exchange. Here, a natural strategy to introduce dynamicity is to adjust the degree of parallelism at synchronization points in between iterations.

In *task-parallelism* on the other hand, the parallelism arises from the independence between tasks. Tasks that do not depend on each other can be executed in parallel while dependent tasks can only be executed in a particular order. From these dependencies a task-graph arises which defines the order and parallelism of the task execution. Such dependency graphs naturally arise in sparse linear algebra computations such as the factorization of spares matrices. Moreover, task dependencies graphs gain interest in HPC as they provide possible improvements in increasingly complex systems through dynamic runtime scheduling [6, 26]. Here, dynamicity of the parallelism arises from the varying number of tasks executed in parallel while traversing the task graph. That is, the inherent degree of parallelism of the application changes when transitioning between the nodes of the graph.

A concept enabling dynamic MPI applications needs to be general and flexible

enough to be applicable to the wide range of different parallel application types. The design developed in this thesis aims to support both, task-parallelism as well as data-parallelism.

3.6. Scope of this Thesis

In the preceding sections, we provided a general overview of the SMS on distributed compute systems. In this section we delimit the scope of this thesis and discuss an abstraction from the variety of system structures using a layered model. Fig. 3.1 illustrates the layered view of job execution on distributed compute systems on which we base our work. To represent the concept of a job running on the system, a cut out of two nodes hosting the job is shown, including the head node of the allocation. For this situation we differentiate five layers: *application*, *MPI*, *PMIx*, *runtime* and *batch system*.

This work focuses on the four layers *application*, *MPI*, *PMIx* and *runtime* and their interaction to enable dynamic MPI jobs. In particular, we describe how the RTE can handle changes of resources mandated by the batch system and extend the MPI and PMIx interfaces for the according interaction between the RTE and the application. The concrete realization of the implied interaction between the RTE and the batch system is beyond the scope of this work. We just assume that there exists some information flow between the batch system and the RTE describing the anticipated change in resources. We provide an interface for writing dynamic MPI applications and demonstrate its usability with some illustrative examples such as an grid-based iterative algorithm with load-balancing. However, an extensive study of its use for different kinds of dynamic applications is not a main area of investigation of this work.

The next two sections provide a detailed description of the MPI and PMIx layers, further outlining the interfaces shown in Fig 3.1.

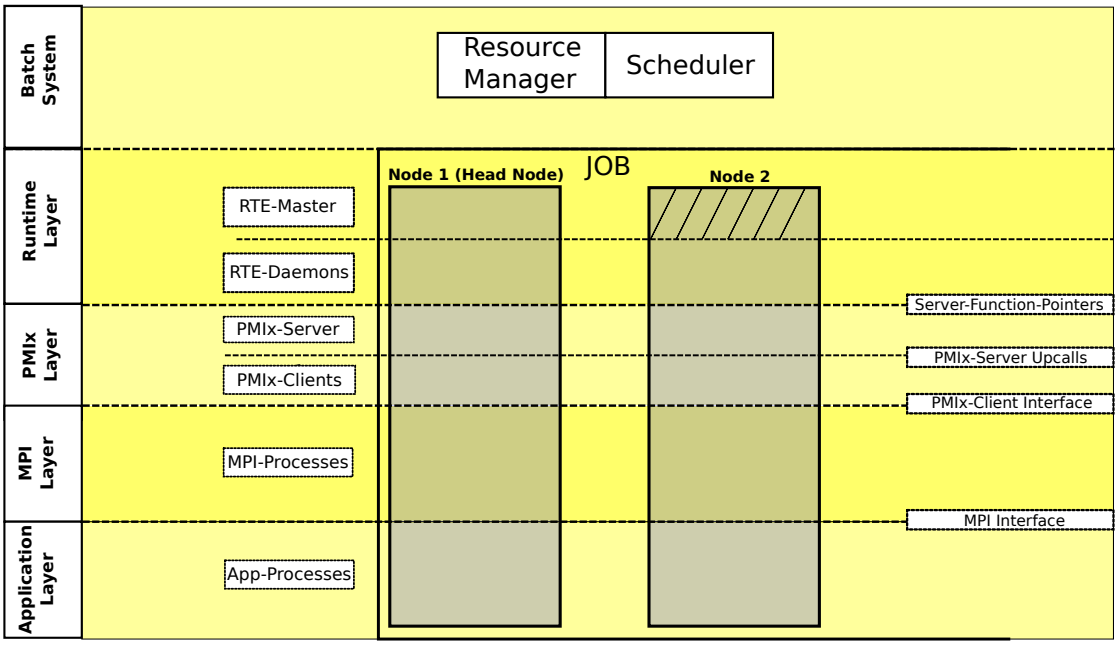


Figure 3.1.: **Simplified layer model of an MPI job running on an HPC system.** Here we assume the usage of an PMIx-enabled MPI implementation. The labels in the most left boxes are the names of the five layers: *application*, *MPI*, *PMIx*, *runtime* and *batch system*. The white boxes with text inside each layer denote the agents of these layers. The boxes on the right side at the borders between the layers denote the interface used by an agent of one layer to act as or communicate with an agent of the next higher layer. *Application processes* use the *MPI interface* to act as *MPI processes*. *MPI processes* use the *PMIx client interface* to act as *PMIx clients*. Most services for *PMIx clients* require interaction with the local *PMIx server* via PMIx-internal *PMIx server upcalls*. *PMIx servers* might delegate requests up to the *RTE daemon* which operates the server via *server function pointers* provided by the RTE implementation. The *RTE daemons* are usually spawned and coordinated by the *RTE master* process running on the head node of the job allocation. Beyond these four job-level layers, the batch system manages multiple such jobs, with the RM assigning resources to jobs based on the scheduler's decisions.

4. The Message Passing Interface

MPI is a standard interface for the message passing parallel programming model, which is widely used on distributed computing systems, and the de facto standard for HPC applications. Thus, to address this large portion of applications, efforts towards dynamic resource management on such systems need to focus on introducing new dynamic elements to MPI.

This work uses and extends features of the most recent version of the standard, the MPI 4.0 Standard [29], to enable the development of resource dynamic applications. As a starting point, this section gives an overview of the goals, progression, core concepts and limitations of MPI. Moreover, it provides a detailed introduction to the new MPI Sessions model, which constitutes the basis of our approach for dynamic resource management developed in this thesis.

4.1. MPI: A Standard Interface for the Message Passing Paradigm

A widely used programming paradigm for parallel multi-computers and distributed systems is the *Message Passing Paradigm*. This paradigm has its roots in the 1970s and assumes a distributed process memory model and explicit communication of data between processes through message exchange [49]. This makes it particularly well-suited for running programs on distributed hardware such as modern HPC on compute clusters. Its generality allows for the development of efficient and portable applications for a large number of different hardware architectures.

The MPI standard specifies a standard interface for the message passing paradigm. There exist several different implementations of the MPI standard, such as OpenMPI, MPICH or IntelMPI, which provide a parallel programming library for application programmers, consistent with the MPI Standard.

The goal of standardizing message passing is to provide a portable, scalable and easy-to-use interface to enable high performance message passing. In particular MPI's level of abstraction makes it language and vendor independent and applicable on heterogeneous environments.

The MPI Standard is developed by the *MPI Forum*, which unites the efforts of different

research groups and vendors. Its work on the MPI Standard in an ongoing process to meet the changing demands of the MPI community which we briefly outline here.

The first efforts towards MPI started 1992 and resulted in the MPI 1.0 Standard which was presented in November 1993. This first version specified basic point-to-point communication such as blocking and non-blocking send and receive as well as collective operations including synchronization, data movement and collective computation. Minor changes were introduced in the following years in versions 1.1, 1.2 and 1.3.

Version 2.0 includes several extensions such as one-sided communications, extended collective communications, external interfaces and parallel I/O. One of the core philosophies of MPI was and still is that it should only specify the message passing between processes by excluding runtime specifics such as process creation and resource management from the standard. However, from early on the user base of MPI demanded concepts beyond the sole message passing, to large parts driven by users migrating from Parallel Virtual Machine (PVM) [44]. PVM is a software package, which provides functionalities to create a large parallel computer from a heterogeneous collection of individual computers. In contrast to the MPI 1.0 standard, PVM supported a form of dynamic adaptability of the number of processes an application runs on. Consequently, the MPI 2.0 standard introduced process creation and management routines that go beyond message passing. This feature is also the basis for many proposals towards dynamic resource management in the literature, however there are also well know limitations which we will outline in detail in Sec. 4.3.

Version 3.0 extended the standard by non-blocking collectives, new one-sided communication operations and introduced Fortran 2008 bindings.

Currently the MPI forum works on version 4.0, which is expected to among other features add support for persistent collectives, partitioned communications, performance assertions, and improved error handling. Moreover, the MPI Sessions Working Group [30] is exploring an alternative model of MPI initialization: The *MPI Sessions* model. This new model is particularly relevant to our work as it provides the basis for our design of resource dynamic MPI. We will give a detailed overview of this new model in Sec. 4.4.

4.2. Key Features of MPI

Over the years the MPI Standard has grown significantly consisting of a vast number of different routines. Here, we will outline four basic concepts of MPI, which are particularly relevant to this work, and refer the reader to the official MPI Standard document [29] for a more complete overview. In this section we omit features and

changes related to the new MPI Sessions model as we discuss them separately in Sec. 4.4.

MPI Groups and Communicators

The traditional MPI model exhibits two concepts for ensembles of processes: *MPI groups* and *MPI communicators*.

MPI groups are ordered sets of processes and are local to each process. Therefore, every process in the group can be associated with a group-unique integer *rank*. Due to locality of groups, group creation and manipulation functions do not require synchronization between the processes of the group. To this end, MPI defines several routines to derive groups as subsets or supersets of other groups. The predefined, implicit `MPI_GROUP_WORLD` contains all processes that were started collectively. Thus, it is the basis for all other *MPI groups*.

An *MPI communicator* augments an *MPI group* with a globally unique context assigned for communication purposes. All communication between MPI processes happens via *MPI communicators*. In the *MPI World* model, the predefined communicator `MPI_COMM_WORLD` is constructed at startup and is associated with the implicit `MPI_GROUP_WORLD`. Thus, it allows communication between all processes that were created collectively. The unique communication context of communicators allows for unambiguous matching of messages send between processes. However, the creation of communicators requires global synchronization of the processes in the associated *MPI group* for deriving the unique communication context.

The MPI standard differentiates between two types of *MPI communicators*. An intra-communicator allows communication between processes of a single *MPI group*. Inter-communicators on the other hand allow communication between processes from two disjoint *MPI groups*. This is particularly relevant to MPI-2 DPM as outlined in Sec. 4.2. Inter-communicators can be merged into an intra-communicator using the `MPI_Intercomm_merge` function.

Communication

MPI defines several different types of communication.

Communication can be either blocking or non-blocking. Blocking communication functions do not return before the data buffer provided to the routine can be reused. In non-blocking communication on the other hand, the data buffer provided to the function may not be reused after the call returned until the operation has completed. Users can check for completion using `MPI_Test`.

Moreover, communication can be point-to-point or collective.

Point-to-point communication allows message passing from one process in the communicator to another by specifying the rank of the sender and receiver respectively. There are several different modes providing different assertions in regards to synchronization between sender and receiver. In *synchronous* mode the send operation does not complete before the matching receive has started. In *buffered* mode, the send operation returns without any assertion about the state of the receiving processes. In *standard* mode, either synchronous or buffered send is used, depending on implementation and context. In *ready mode* the send operation may only be started when the matching receive was already posted by the receiving process. In this case it behaves like the standard mode.

Collective communication involves all processes in a particular communicator. Thus, it must be called by all processes in the associated process group. Collective communication comprises several different operations such as *synchronization* operations, *1-to-many* communication, *many-to-many* communication and *global reduction* operations.

MPI Info Object

An *MPI info object* is an MPI object used in many MPI routines as a means of carrying additional information. They store key-value pairs, where both, keys and values are strings. However different data types can be stored using their string representation.

A typical use case of MPI info objects is the specification of additional function parameters to further control function behavior. Moreover, functions themselves can provide additional information about the execution state of the function as output in a provided MPI info object. This mechanism is especially useful for functions that interact with the runtime, as it respects the varying support of functionality provided by the host system.

Process managemnent

As early as in 1995, the MPI forum started to work on a more flexible approach for process management [15]. With MPI-2, the new MPI routines `MPI_Comm_spawn`, `MPI_Comm_accept` and `MPI_Comm_connect` were introduced. With this set of routines, MPI applications can create new processes at runtime and establish inter-communicators with processes that were not part of the initial `MPI_COMM_WORLD`.

Fig. 4.1 and Fig. 4.2 illustrate the concept of spawning new processes and connecting processes via the connect/accept API. `MPI_Comm_spawn` is a collective, blocking function over an intra-communicator and spawns a specified number of new MPI processes. When these new processes (right side in Fig. 4.1) initialize MPI via `MPI_Init` a new, separate `MPI_COMM_WORLD` communicator is created. In addition, an

inter-communicator is created connecting the new `MPI_COMM_WORLD` with the communicator used in the `MPI_Comm_spawn` call. This inter-communicator is returned to the original processes by the `MPI_Comm_spawn` function and can be accessed by the new process using `MPI_Comm_Get_Parent`. Thus, communication between original and spawned processes is established via an inter-communicator connecting the original communicator with the `MPI_COMM_WORLD` communicator of the spawned processes.

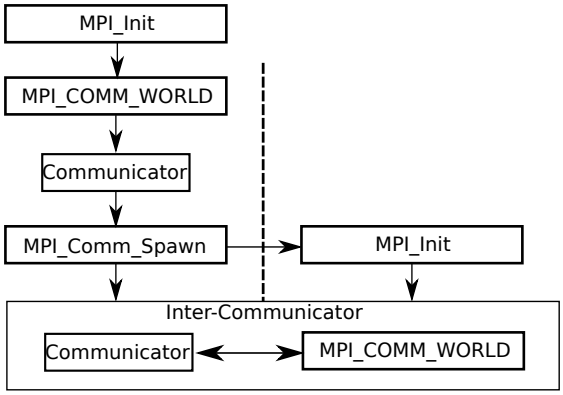


Figure 4.1.: **Concept of the `MPI_Comm_Spawn` API.**

Apart from spawning new processes, MPI further provides a general interface for establishing communication, with processes which do not already share a common communicator, i.e. an `MPI_COMM_WORLD`. Typical use cases are for instance communication between independently started application parts, tool connection and client-server applications. Here we use the client/server terminology solely for distinguishing the responsibilities of the two sides seeking a common communicator.

The server side (left side in Fig. 4.2) opens a *port* via the `MPI_Open_Port` function. Here the term *port* describes a very general concept of a communication endpoint. The particular realization of this concept is implementation-dependent. With `MPI_Comm_accept` the server side can "listen" for connection attempts by clients. The function is collective and blocking over the provided communicator.

The client-side (right side in Fig. 4.2) can connect to a port with a pending accept via the `MPI_Comm_connect` function. Again, this function is collective and blocking over the provided communicator.

Both functions return a common inter-communicator connecting the communicators provided to `MPI_Comm_accept` and `MPI_Comm_connect` respectively.

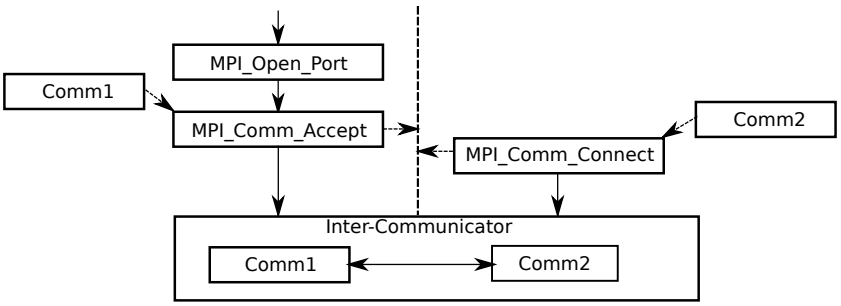


Figure 4.2.: **Concept of the** `MPI_Comm_accept/connect` **API.**

4.3. Limitations of the MPI-2 Dynamic Process Management

The MPI-2 DPM introduced some degree of adaptivity to MPI and inspired several approaches towards dynamic resource management. However, the feature is rarely used in HPC applications [43] with a recent large scale study finding that only 2% of applications make use of DPM [24]. Various authors have expressed limitations of the feature which might be a reason for this observation [43, 24, 46, 8, 27]. In the following, we summarize these ,imitations.

MPI-2 DPM does not provide any ad-hoc integration with dynamic resource allocation. The spawning of new processes can only be initiated by the application and typically new processes are spawned in the same resource allocation, possibly using oversubscription. External support is required to combine it with dynamic resource allocation and to enable malleability.

Further, MPI-2 DPM has a high latency of process creation. One reason for that is that the `MPI_Comm_Spawn` function is blocking, thus running processes are blocked until the new processes are launched and initialized.

Moreover, the design using inter-communicators is impractical especially when processes are spawned repeatedly. It either requires subsequent merging to create an intra-communicator associated with additional overhead or a complex management of various inter-communicators.

Finally, MPI-2 DPM only provides limited support for application shrinking. This is due to processes sharing an `MPI_COMM_WORLD` communicator are connected and the `MPI_Finalize` function being a collective call over connected processes. Consequently, it is not possible to remove subsets of processes of a particular `MPI_COMM_WORLD` communicator. As a result, all processes started at initial application launch can only be terminated collectively. Similarly, all processes spawned by a single `MPI_Comm_Spawn` call have to be terminated collectively. This can only be circumvented by incrementally growing applications by separately spawning single processes, at the price of large

overheads.

These limitations motivate a more flexible design of process managing providing low latency for running processes, direct integration with dynamic RMs, unconstrained shrinking and a convenient interface. The MPI Sessions model, which we introduce in the next section, provides a basis towards achieving these design goals.

4.4. MPI Sessions

The MPI Sessions model is an alternative approach to the original model of MPI initialization (the MPI World model), included in the MPI 4.0 Standard. The new model exhibits several features providing interesting starting points towards dynamic resource management. In this section, we will outline the motivation behind the MPI Sessions model, describe its key concepts and discuss its potential as a basis for this work.

4.4.1. Motivation for the MPI Sessions Model

The motivation for the MPI Sessions model was to overcome several limitations of the MPI World model, which we briefly summarize here:

- **Initialization by different entities:** A major limitation in the World Model is that initializing MPI by different entities in the same MPI process such as different application components and libraries requires a priori knowledge and coordination. This is especially problematic for complex and coupled applications.
- **Re-initialization of MPI:** Another problem is that it is not possible to initialize MPI more than once or to reinitialize it after MPI finalization. It further does not allow for differentiated control over differing requirements of application components such as the MPI threading-level or resource isolation. Clearly, this is a major limitation towards fault tolerance and dynamic MPI.
- **Global `MPI_COMM_WORLD` communicator**: Moreover, the requirement of the global `MPI_COMM_WORLD` communicator introduces severe scalability issues, especially when considering the trend towards loosely coupled applications and exascale systems.

The MPI Sessions model introduces a new approach towards MPI initialization which addresses these limitations.

4.4.2. MPI Session Handles

In the MPI Sessions model, MPI processes instantiate a local `MPI_Session` handle with the `MPI_Session_init` function, which is a local operation. This is a major difference to the MPI World Model, where the `MPI_Init` call is collective over all processes and point-to-point communication paths are established between all processes. Instead of providing an implicit communicator between all processes, `MPI_Session` handles can be used to query the runtime about characteristics of the job the process is running in. This allows different application components to instantiate MPI resources based on the specific communication needs of a certain component. Moreover, MPI processes are allowed to create and finalize multiple session handles.

`MPI_Session` handles also provides a domain of isolation, that is, MPI objects derived from different MPI Sessions handles are not allowed to be inter-mixed in a single MPI procedure. Different session handles can support different needs of application components such as thread-levels and provide the potential for isolated fault tolerance through partial re-initialization.

However, it is important to note that processes can become connected processes in the sense of the MPI definition through common communicators derived from different sessions. This is an important consideration as the `MPI_Session_finalize` function used to release a session handle is collective over all connected processes.

4.4.3. Process Sets, Groups & Communicators

As described in the last section, the Sessions model has no default process groups and communicator such as the `MPI_COMM/GROUP_WORLD` from which all communication is derived. Instead the *MPI Sessions* model uses a new concept for establishing communication.

The central component of this new concept are *process sets*. A *process set* is an unordered set of processes and is uniquely identified by a string in the *Uniform Resource Identifier (URI)* format. They provide a common concept for the runtime and MPI processes to associate information with a set of processes. In particular, it allows MPI processes to discover processes available for establishing communication.

Throughout this thesis we refer to the processes, which are included in a particular process set as the *members* of this process set.

The MPI standard defines two mandatory process sets: *mpi://WORLD* and *mpi://SELF*. Implementations/runtimes can define additional process sets and associate resources with them. Thus, process sets are classifications of processes based on arbitrary criteria. In the current version of the MPI 4.0 standard process sets are indelible, i.e., new process sets may be defined but process sets cannot be removed nor can the index of a

particular process set change.

The main purpose of process sets is to provide a starting point for MPI processes to establish communication. Fig. 4.3 demonstrates the typical procedure to create communicators in the *MPI Session* model.

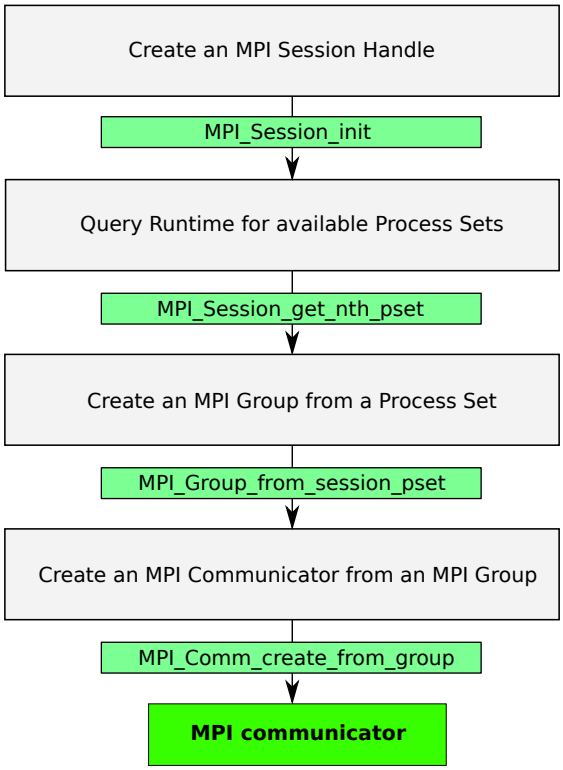


Figure 4.3.: **Steps for creating an MPI communicator in the MPI Sessions model** (adjusted figure from [29]).

An application first creates a local *MPI Session handle* using the `MPI_Session_init` function.

The *MPI Session handle* can subsequently be used to query the runtime for available process sets. The MPI standard defines two functions, which allow application processes to discover process sets: `MPI_Session_get_num_psets` returns the number of defined process sets the calling process is member of and `MPI_Session_get_nth_pset` returns the name of the process set based on its index. Information about a particular process set can be queried using the `MPI_Session_get_pset_info` function. Currently, the only mandatory, queryable attribute of process sets is the process set size using the `mpi_size` key.

To establish communication with the processes included in a particular process set, first an *MPI Group* is created by specifying the name of the process set in the `MPI_Group_from_session_pset` function.

Finally, an *MPI Communicator* is created for the processes in the *MPI Group* using the `MPI_Comm_create_from_group` function.

4.4.4. Potential for Resource-Adaptive MPI Sessions

Based on the last sections, two factors become apparent, which are the basis for the great potential for dynamic resources with MPI Sessions.

First, avoiding the collective `MPI_COMM_WORLD` opens up the potential for scalable adaption of the number of resources. As there is no need for manipulating a global communicator, addition and removal of processes can be achieved without global synchronization. This makes resource changes compatible with the idea of isolation, which is central to the MPI Sessions model.

Second, the concept of process sets provides a natural description of resources, which is shared by both, the application and the runtime. This enables the runtime to specify resource changes in terms of process sets, which simultaneously are the basis for applications to create communicators. While the current concept of process sets is relatively static, an extensions towards more dynamic process sets would therefore provide a basis for resource changes naturally embedded in the existing mechanisms of the MPI Sessions model.

Building on top of these considerations, in Section 6 we propose a design for dynamic resource management with MPI Sessions and PMIx.

5. The Process Management Interface - Exascale

The Process Management Interface - Exascale (PMIx) [3] defines a standardized interface for portable and scalable interaction between components of the SMS as well as for managing application processes.

To this end, PMIx defines abstractions for typical components of the SMS and operates as a messenger between those components. Therefore, PMIx enables independence from specific implementations of these components through a standardized interface, which increases reusability and portability.

A typical use case of PMIx is to provide RTE services to distributed applications such as the exchange of wire-up information of application processes, synchronization and distributed key-value storage services. Many MPI implementations use PMIx for exchange of MPI process wire-up information at startup and as a mediator between MPI and the RTE services. While these implementations usually also include their own RTE implementation, the PMIx abstractions also allow direct integration of MPI support into PMIx-supporting RMs such as SLURM. Thus, new concepts for dynamic resource management of MPI applications would clearly benefit from a realization based on PMIx.

The concept developed in this work extends and makes use of PMIx in various ways to support interaction between MPI applications and the RTE. To this end, this section provides an overview of PMIx concepts and functionalities which are particularly relevant to our developed concept and which we will therefore reference throughout this work.

5.1. Components

PMIx defines different components each with a standardized interface to take the needs of different SMS elements into account. In the following, we give a brief overview of these components. For the sake of illustration we provide a concrete example of the usage of these components in an HPC system in Fig. 5.1. However, it should be noted that PMIx only specifies an abstract interface for these components and their concrete utilization in the context of a SMS varies greatly.

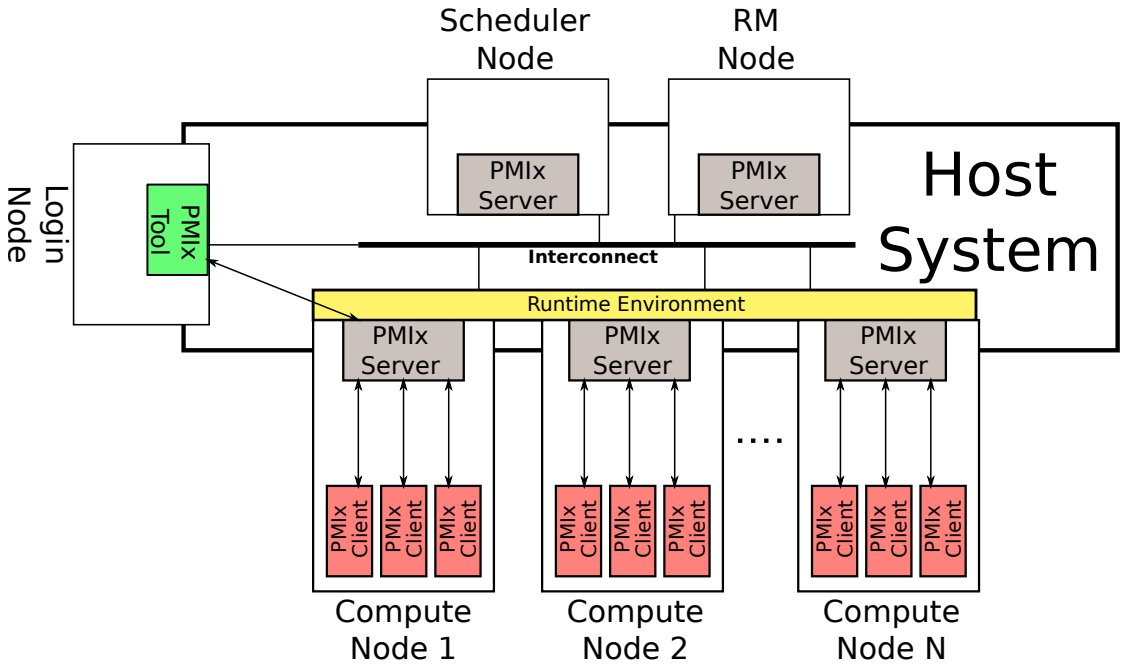


Figure 5.1.: **Example for the usage of PMIx components in an HPC system.**

5.1.1. Host System

PMIx recognizes the *host system* as an entity that uses PMIx to offer and realize system services, including resource management, job control, remote messaging etc. In Fig. 5.1, the host system comprises several hardware and software components of the system such as a RTE, the scheduler, the RM and the system interconnect. As this thesis is mainly on the realization of a dynamic RTE for dynamic MPI applications, for the most part of this work we only refer to the RTE part of the host system. The interaction with the RM, scheduler and fabric subsystems is only broadly indicated by our design and its concrete specification is beyond the scope of this work.

Host systems operate one or more *PMIx server(s)*, usually one on each PMIx enabled node in the system. PMIx servers define an interface for host system services and allows the host system to provide concrete implementations for these services. For this, host systems provide *server function pointers* (see also Fig. 3.1) for the implementation of these system services. These function pointers are passed to PMIx servers during their initialization in an instance of the `pmix_server_module_t` struct. The services can then be accessed by *PMIx clients*, *PMIx tools* and the host system via the PMIx server. Different levels of support of the host system are acknowledged by allowing `NULL` pointers for unsupported system services in the `pmix_server_module_t` struct.

The host system entity is not necessarily a single implementation. Instead, the abstraction of PMIx allows interoperability between different SMS components and vendors to provide a common host system. For instance, a particular RTE implementation might interact with the system-specific scheduler and RM implementations via the PMIx interface for enhanced RTE services.

5.1.2. PMIx Server

A PMIx server is operated by the host system and provides services to host system processes, PMIx tools and PMIx clients. The `PMIx_server_init` interface is used by the host system to setup a PMIx server. In distributed systems typically a local PMIx server is set up on each node. A PMIx server is shut down by calling the `PMIx_server_finalize` function.

Host system processes operating the PMIx server can use all services provided to clients as well as some additional server functionalities. This includes the collection of inventories, preconditioning of resources, creation of regular expressions and requesting information from servers e.g. to fulfill data access requests from remote processes. Moreover, the host system can set up a *PMIx namespace* using the `PMIx_server_(de)register_nspace` function. A PMIx namespace is a string identifier for a certain job e.g. an MPI job submitted by a user. Each PMIx server can manage multiple PMIx namespaces. A *PMIx job data object* is associated with each PMIx namespace and provides information about a particular job which is accessible by clients at startup. The host system can use the `PMIx_client_(de)register` functions to register/deregister PMIx clients with a particular PMIx namespace at the local PMIx server. Registered client processes can connect to the server and subsequently make use of its provided services via the PMIx client interface.

Fig. 5.1, for instance, represents a situation where an RTE implementation is hosting a PMIx server on each of the N compute nodes. Subsequently it registered a PMIx namespace and three PMIx clients (e.g. MPI processes) with each server. The *PMIx clients* can connect to the server and use the services provided by the RTE and PMIx server e.g. for exchange of wire-up information during the initialization of MPI.

5.1.3. PMIx Client

PMIx clients are processes that were registered with a PMIx server (usually by the host system operating the server) and which connected to the server by calling `PMIx_Init`. This function sets up the PMIx client library and returns a unique identifier for the process consisting of the PMIx namespace (the job identifier) and its unique rank in this namespace. This corresponds to the identifier assigned by the host system while

registering the client. Clients, which are connected to a server, can access information about their namespace included in the PMIx job data object provided by the host. Moreover, a wide range of services can be requested from the server and the hosting system of which we cover the most relevant ones for this work in Sec. 5.2. Clients can disconnect from a server using `PMIX_finalize`.

MPI implementations typically use the PMIx Client interface for bootstrapping MPI processes. Each MPI process is a PMIx client connected to the PMIx server on the local node. During initialization, MPI processes wire-up using information provided in the PMIx job data object. Moreover, many MPI routines rely on services accessed via the *PMIx client interface* (see Fig. 3.1). These services are either provided directly by the PMIx server (e.g. synchronization of local processes) or delivered to the RTE / host system for further processing (e.g. synchronization with remote processes).

5.1.4. PMIx Tool

Tools are programs interacting with the host system to have access to and control of various aspects of schedulers, resources, jobs and applications. Typically, tools are used for monitoring, debugging and launching of system parts.

PMIx provides a general interface for tools to interact with the host system and applications. PMIx tools are processes which initialize PMIx as a tool using the `PMIx_tool_init` function. Connections to PMIx servers can be established during initialization or using the `PMIx_tool_attach_to_server` function. In contrast to clients, tools are not registered by the host system, thus connection of tools to servers requires rendezvous files to discover connection information. Once connected to a server, a tool can use similar services as those provided by the PMIx client interface, possibly with tool specific attributes, to interact with the host system or applications. For instance it could send or receive event notifications, query the system or send allocation and job control requests. It further can launch new jobs directly or through intermediate launchers using the `PMIx_Spawn` interface. In addition, a tool-specific PMIx-API is provided for controlling IO forwarding from and to remote processes.

Fig. 5.1 provides an example of using a PMIx tool e.g. for debugging of a job. The PMIx tool is running on the login node and connected to a server running on one of the compute nodes which allows for access to job related information.

5.2. PMIx Functionalities Relevant to This Work

The PMIx standard defines a rich ensemble of functionalities covering most needs of SMS components. As this work mostly focuses on the interaction between MPI applications and MPI RTEs, here we only cover the functionalities particularly relevant

for our purpose. The reader is referred to the PMIx Standard 4.0 document [32] for a more complete and detailed description of the interface.

5.2.1. General Concepts

A general concept in the PMIx standard is to provide blocking and non-blocking versions of many functions. Non-blocking will return as soon as the request gets processed by the PMIx library. Upon completion of the request, a user provided callback function is called to deliver the status and results of the request. This allows processes to continue execution while the request is processed. Server module functions, for which the host systems provides pointers, always use a non-blocking approach. Thus, PMIx servers are able to respond to requests after an upcall to the host system for host system services. When the host system finishes a request it calls the callback function provided by the PMIx server.

Another general concept applying to most PMIx functions is the usage of an (array of) `pmix_info_t` object(s) to specify additional attributes (in the form of key-value pairs) to be considered by the function implementation. This is a concept similar to the `MPI_Info` object in MPI. The PMIx standard defines mandatory attribute keys, which have to be supported by all implementations and optional attribute keys allowing for different levels of support.

Moreover, for data access operations and event delivery, a certain *PMIx scope* and *PMIx data range* can be set explicitly by specifying a value of the `pmix_scope_t` and `pmix_data_range_t` type respectively. To give an example, delivery of an event can be restricted to the `PMIX_RANGE_RM` if an event is solely intended for the host system.

5.2.2. Synchronization and Data Access Operations

One of the most important PMIx functionalities are those related to (synchronized) data storage and retrieval which allows out-of-band exchange of information. Such information could for instance be the endpoint information of an MPI process which should be made accessible to other processes in the job.

A key-value approach is used for the data storage and retrieval. The PMIx standard defines various *reserved keys* starting with with the `pmix` prefix. These keys are exclusively provided by the host system and PMIx server and are made available to clients at startup. Non-reserved keys are keys which might also be used by clients and tools to store a key-value pair.

PMIx provides the following functionalities for data storage and retrieval.

- **Posting and retrieving data:** The `PMIx_Put` function can be used to post a non-reserved key-value pair to a specified PMIx scope, restricting its accessibility

e.g. to the processes on the local node or in the same namespace. The `PMIx_Get` function allows retrieval of values for reserved keys (e.g. from the job data object) as well as (non-reserved) keys posted by `PMIx_Put`, provided matching scopes. The behavior of the data access can be influenced by specifying according keys in the `info` parameter, for instance to restrict the scope of the access or to request update of cached values from remote processes. The process identifier of the process that posted the information needs to be specified when calling `PMIx_Get` to retrieve the information on demand from remote hosts. When the process, which posted the information, is not known to the process calling `PMIx_Get` the information can be retrieved based on the namespace. However, this requires prior synchronization with global data exchange.

- **Synchronized data exchange:** Synchronization and data exchange can be achieved using `PMIx_Fence`. This is a collective function across a range of processes, e.g. processes in a namespace. The operation does not complete before all processes in the specified range have called into the collective. If desired, a global data exchange can be executed, which makes all key value pairs posted via `PMIx_Put` available to the participating processes.

- **Asynchronous data exchange:** For the case that the posting and/or receiving processes are not known in advance or a synchronization operation is not desired, asynchronous distribution of information is enabled by the `PMIx_Publish/Lookup` functionality. `PMIx_Publish` allows processes to publish key-value pairs in a certain PMIx range and to restrict the access based on permission and persistence policies. `PMIx_Lookup` allows the retrieval of data from a certain PMIx scope if the corresponding permission requirements are met. Due to the asynchronous nature, the key to be looked up might have not yet been published. Through the specification of the `PMIx_WAIT` and `PMIx_TIMEOUT` attributes in the `info` parameter, the PMIx library can be instructed to wait a certain amount of time for the key to be published.

5.2.3. Host System Queries

The last section described ways of accessing job information or data made available via `PMIx_Put` and `PMIx_Publish`. However, often information about the host system (state) is required. To access general and possibly dynamic system information, such as the host system support levels, the job status and information related to scheduling and allocations, the `PMIx_Query_Info` interface is used.

The function takes one or more `pmix_query_t` structures consisting of an array of keys to be queried. Moreover, an array of `pmix_info_t` containing corresponding

qualifiers can be used for further specifying details of the query. Similar to the `PMIx_Get` function, retrieved values are cached. To skip the search in the local cache and directly query the host system the `PMIX_QUERY_REFRESH_CACHE` qualifier can be used.

5.2.4. Event Notification System

PMIx provides an event notification system as an "asynchronous out-of-band mechanism for communicating events between application processes and/or elements of the SMS" [32, p. 135]. This is especially useful for fault tolerance and asynchronous workflows.

The `PMIx_Notify_event` function can be used to send notifications of a particular event to the specified range. Additional attribute can be specified in the `info` to further control delivery or describe the event.

The event notification is received by all processes falling into the specified range which have registered a corresponding *event handler* using the `PMIx_register_event_handler` function. This function allows the registration of a callback function to be executed when receiving notification of the specified event. Events are delivered through an event chain, determining the order of delivery. The event chain can be influenced through the usage of corresponding attributes in the `info` parameter.

5.2.5. Process, Job and Allocation Control

PMIx provides an interface of several functionalities for controlling processes, jobs and resource allocations.

- **PMIx_Spawn:** The `PMIx_Spawn` function spawns a new job. Several attributes can be specified to describe the job and application as well as attributes specifying mapping and resource allocation. The name of the new namespace is returned to the caller. New processes are spawned into a new namespace and are connected to the calling process. Connected processes receive a copy of the job data of the parent/child processes respectively. Moreover, in case of a processes failure, connected processes are treated by the RTE as one ensemble of processes similar to a single application.
- **PMIx_Abort:** The `PMIx_Abort` function requests the system to abort the specified processes with a specified error code and error message.
- **PMIx_Job_control:** The `PMIx_Job_Control` function provides a mechanism for applications and host systems to coordinate job control actions. Various directives can be specified to control actions such as pausing, termination, checkpointing

and preemption of jobs. The job control actions are applied to the processes specified by the `target` parameter.

- **PMIx_Alloc_request:** The `PMIx_Alloc_request` function can be used to request a new allocation or a modification of an existing allocation from the RM. Possible allocation directives include requests of new allocations, time or resource extension of existing allocations, permanent or temporary release of allocated resources and reacquiring of temporarily released resources. Detailed specification of resources requirements can be achieved by specifying corresponding attributes in the `info` parameter.

The *PMIx Standard Tools Working Group* recently expressed their vision of using a combination of `PMIx_Job_Control`, `PMIx_Alloc_request` and `PMIx_Notify_event` to enable general purpose transactions, e.g. between the RM and applications.

Such a transaction API could for instance be used by applications and RMs for cooperative dynamic resource management. The PMIx Standard provides some examples for such use cases. We briefly discuss this possibility in the context of our developed concept in Sec. 6.6, however, a full elaboration of required transactions for cooperative resource management is beyond the scope of this work.

5.2.6. Process Sets

Similarly to the MPI standard, the PMIx standard uses the concept of process sets as a *label for a collection of processes*.

Though, there are certain differences to MPI process sets, which we will briefly discuss here. PMIx process sets are host system defined or defined by users at the start of an application, while MPI does not further specify this. Names of PMIx process sets are required to differ from system-assigned namespaces in the scope of the process set but are not required to use the URI format specified in MPI. Moreover, in PMIx once a process set is defined, its members cannot change. However, process sets can be defined and deleted by the host system. MPI process set on the other hand, once defined, cannot be deleted but their members could potentially change.

The host system needs to ensure the consistency of the knowledge about existing process sets and their members across all involved PMIx servers. For this, the `PMIx_server_define/delete_process_set` functions can be used to define/delete process sets on PMIx servers. The PMIx server notifies all local clients about definition and deletion of process sets via the `PMIX_PROCESS_SET_DEFINE/DELETE` event. The notification includes the name of the process set and in case of process set definition its members. The system can be queried about the number, names and members of

process sets in a particular *PMIx scope*. Moreover, a process can retrieve the names of process sets it is a member of using the `PMIX_PSET_NAMES` key in a call to `PMIx_Get`.

6. Proposal for Dynamic Resources with MPI and PMIx

In this section we describe our design for dynamically changing resources in MPI using MPI Sessions and PMIx.

In Sec. 6.1 we describe our basic design decisions and introduce necessary definitions. Sec. 6.2, 6.3 and 6.4 each handle one of the three phases of the resource change. In each we describe the general concept of the phase, the required functionality/extensions to MPI, PMIx and the RTE and discuss implications and possible extensions of the design.

Sec. 6.5 illustrates the usage of our design for dynamic resource management based on two representative examples.

The chapter is concluded by Sec. 6.6, in which possibilities to extend our proposal towards evolving jobs and fault tolerance are discussed.

6.1. Basic Design Decisions

In this section we briefly describe central concepts of our design which constitutes the basis for the following Sections.

6.1.1. Main Phases of Resource Changes

Our design is based on the work described in [12], which proposes an extension of the concept of MPI Sessions to support dynamically varying resources, i.e. a varying number of MPI processes. This model has been demonstrated to work in an emulated MPI Sessions environment. In this work, we adopt the three proposed, fundamental phases and adapt them to the requirements of a real MPI Sessions environment supported by a dynamic RTE. Here, we briefly summarize the basic sequence for resource changes as proposed in earlier work [12] :

- **Phase 1**: Applications query the RTE which provides explicit information about a change of available resources. Resource changes are unambiguously described by two attributes: a resource change type indicating addition or removal of resources and a *delta process set* containing the processes to be added or removed. A *delta*

process set therefore describes the delta between the processes before and after the resource change.

- **Phase 2**: The application prepares an adjusted communication structure to reflect the resource change. To this end, it creates new process sets, which are the basis for the creation of new communicators. This is achieved via set operations of processes sets, providing a generic mechanism for applications to manipulate communication patterns.
- **Phase 3**: The application establishes connection to new processes or terminates processes respectively and creates a new communicator, which finalizes the resource change.

The MPI interface we propose in the following sections is largely based on this basic structure. We use and extend the PMIx standard to realize the necessary coordination with the RTE. These extensions to the PMIx Standard also constitute a reference for the requirements for implementing a supporting RTE. Thus, we explicitly discuss these requirements for every phase of the resource change.

6.1.2. Targets of Resource Changes

HPC jobs can consist of multiple applications and MPI applications using the MPI Sessions model can consist of multiple loosely connected parts, which derive communication from different process sets. Thus, different potential targets for resource changes exist:

- **Job level**: *The resource change targets the whole job and would therefore be queryable by all processes in the job during phase 1.* This corresponds to the typical target of scheduler decisions, which assigns resources to a particular job. However, in multi-application jobs the optimal distribution of additional resources and fairness when subtracting resources needs to take application level measures into account. Moreover, making new resources available to the whole job would require inter-application coordination for deriving consensus for the assignment of new resources to applications.
- **Application level**: *The resource change targets a particular application in the job and is queryable by all processes of this application during phase 1.* Here, the RTE could use its knowledge about the job's applications for a refinement of the resource change announced by the scheduler. For instance, when the scheduler assigns some new resources to the job, the RTE could split up these resources and create two separate resource changes each queryable by the corresponding application.

As shown in a recent study [43], a large share of exascale applications exclusively make use of the `MPI_COMM_WORLD` communicator for communication. Thus, for such applications the assignment of new resources on application level would be sufficient.

However, an application might consist of multiple malleable parts based on different process sets, which run concurrently. Resource changes targeting the whole application would therefore require coordination between these parts for optimal assignments of resources. Moreover, subtracting resources could affect multiple process sets. Thus applications would be required to determine the necessary coordination for the safe termination of processes over different process sets themselves. Depending on the complexity of the applications communication patterns more fine-grained control of resource changes would be beneficial.

- **Process set level**: *The resource change targets a particular (set of) process set(s) and is only visible to the members of this (set of) process set(s).* Thus, processes query the RTE for resource changes that target a particular process set. The fine granularity of resource changes would acknowledge the distinct levels of parallelism of different application parts. Moreover, it would avoid the necessity for application programmers to coordinate resource changes across different process sets themselves.

  However, this approach adds complexity to the implementation of the RTE and requires an extended concept for resource changes and process sets.

In this work we aim to design an interface which acknowledges the need for both, application level and process set level resource changes. In particular, the interface should support application level resource changes without imposing the added complexity required for process set level resource changes. For this, we rely on the usage of the `MPI_info` object to make use of the full set of supported functionality.

6.1.3. Extension of the Process Set Concept

The process set concept in the MPI 4.0 standard is formulated in a very general manner. While our design would allow application-level resource changes based on the current definition of process sets, an extended definition is mandatory for support of process set level resource changes. To motivate and illustrate our extensions for process sets we give an example of a process set hierarchy in Fig. 6.1.

In this example a certain job consists of two applications, *app1* and *app2*. Our design allows the creation and manipulation of process sets on behave of the applications as well as the RTE. We therefore propose to define two separate domains based on the URI

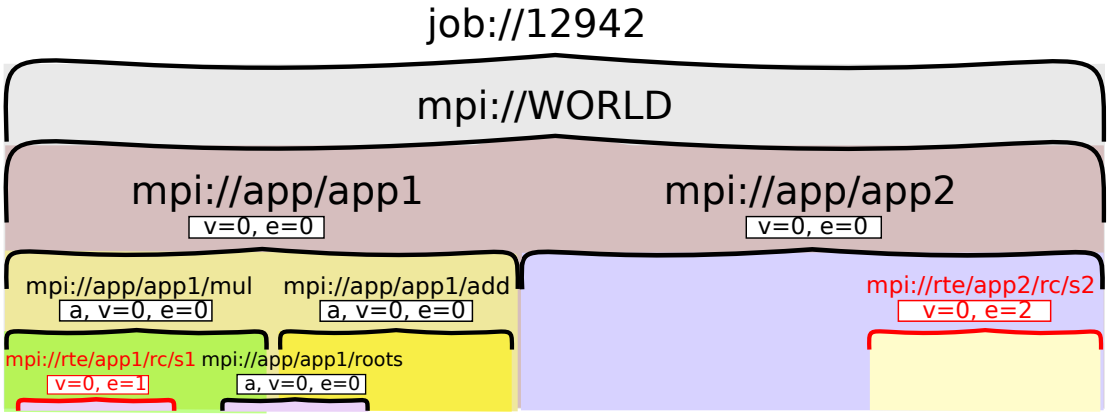


Figure 6.1.: **Example of a hierarchy of process sets.** The job consists of applications *app1* and *app2*. Red process sets represent *delta process sets*. Process sets cache the attributes `mpi_ps_version` (v), `mpi_ps_epoch` (e) and `mpi_ps_active` (a). In this example, the applications have not yet adapted to the resource changes.

name format of process sets. This allows the classification of process sets and prevents conflicts between runtime- and application-defined process sets.

The **mpi://app/{app_name}** domain should be reserved for process sets created on behave of the application *app_name*. At the start of an application, a predefined *mpi://app/{app_name}* process set should exist, which includes all processes of the application.

The **mpi://rte** domain should be reserved for process sets defined by the RTE. For convenience, the RTE implementations are advised to define an explicit subdomain for *delta process sets* representing a resource change of an application, e.g. **mpi://rte/{app_name}/rc**.

In our example in Fig. 6.1 *app1* created three new process sets in the *mpi://app/app1* domain: *mul*, *add* and *roots*. The RTE has defined a delta process set *s1* in the *mpi://rte/app1/rc* domain representing a subtraction of processes. *App2* has not defined any new process sets in the *mpi://app/app2* domain. The RTE has defined a delta process set *s2* in the *mpi://rte/app2/rc* domain representing a subtraction of processes.

Resource changes introduce dynamicity into this hierarchy of process sets, as processes are added or removed and applications adopt their communication patterns through set operations. To ensure consistency between processes as well as between the application and RTE, we extend the specification of mandatory key-value pairs cached on each process set. These are especially relevant to process set level resource changes, however, some of these extensions also benefit the usability of the design for

application level resource changes.

In addition to the `mpi_size` key, we define the following mandatory keys to be cache on process sets. If not stated otherwise, $S(x)$ denotes the value of attribute x cached on process set S.

- **Attributes assigned by the runtime environment**
  - **mpi_ps_version** (int): *The version number (v) of a process set S, ($S(v)$), should be incremented whenever the members of the process set change. For new process sets $S(v) = 0$.*
    The `mpi_ps_version` attributes provides the possibility to query if and how often the members of the process set have changed since its definition.
  - **mpi_ps_epoch** (int): *The epoch number (e) of a process set S, ($S(e)$), references the index of the most recent resource change that changed the members of S. The RTE should assign the epoch number according to Def. 1.*
    The `mpi_ps_epoch` attribute can be useful to synchronize job meta data between processes deriving communication from this process set. For instance, in Fig. 6.1 after applying the announced resource change to process set *mpi://app/app1/mul* its members might have a different version of job meta data than processes in process set *mpi://app/app1/add*. If the processes from both process sets need to communicate e.g. via process set *mpi://app/app1* the `mpi_ps_epoch` could inform the processes of process set *mpi://app/app1/add* about the necessity to update the job meta data for consistency. This idea is further outlined in Sec. 6.4
- **Attributes assigned by the application**
  - **mpi_ps_active** (bool): *The active flag (a) indicates that the process set S is actively managed by the application. The RTE is not allowed to change the members of active process sets. Applications are responsible for applying announced resource changes to the corresponding active process sets. By default $S(a) = true$.*
    The `mpi_ps_active` attribute, when set to `true` gives an application full control and responsibility over a process set. This is important to avoid conflicts and inconsistencies between the application and the RTE. In particular, when communication is derived from a process set it is important that the members of this process set do not change without coordination with the application. However, malleable applications are responsible for applying the resource changes announced by the RTE to the corresponding active process sets using our proposed design.

    When setting the `mpi_ps_active` attribute to `false`, the application gives up its control over the process set as well as the responsibility to check for and

apply resource changes corresponding to this process set. In this case, the RTE is responsible for maintaining consistency of inactive process sets when affected by resource changes.

Moreover, the `mpi_ps_active` attribute is the basis of Definitions 2, 3 and 4.

– **(mpi_ps_sched_{xxx})**: *(Not included in our proposal)*. For the sake of discussing possibilities of using our design for improved load-balancing and scheduling, in the following we refer to a not further specified class of attributes `mpi_ps_sched_xxx`. These attributes are expected to represent any information useful for efficient resource assignment.

Based on these attributes we can classify process sets as *malleable* (Def. 2), *reference counted* (Def. 3) and *recursively affected* (Def. 4). *Malleable* and *reference counted* process sets play a role in choosing target process sets for resource changes while the concept of *recursively affected* process sets is important in the context of process set operations. This is further outlined in Sec. 6.2.2 and 6.3.2.

When applying the classification to the example in Fig. 6.1, we would derive the following results. The set of malleable process sets of app1 is $X_m(app1) = \{mul, add, roots\}$. The set of reference counted process sets w.r.t. the delta process set *s1* is $X_c(s1) = \{mul\}$. The set of recursively affected process sets w.r.t the process set operation $op(mul, s1)$ is $X_a(op(mul, s1)) = \{mpi : //app/app1\}$. As application *app2* has not yet created any new process sets, only the mandatory *mpi://app/app2* set exists in the corresponding domain, which is `active` by default. Thus, $X_m(app2) = X_c(s2) = \{mpi : //app/app2\}$ and $X_a(op(app2, s2)) = \emptyset$. *App2* illustrates the simpler case of application level resource changes which is a special instance of process set level resource changes.

Definition 1.

At start of an application, $S(e) = 0$ for all its process sets. Let D be the set of delta process sets in a particular
mpi://rte/{app_name}/delta domain before adding d_{new} to D . The epoch number of d_{new} shall be

$$d_{new}(e) = \begin{cases} max(T(e)) + 1 | T \in D & \text{if } D \neq \emptyset \\ 1, & \text{otherwise} \end{cases}$$

For a process set S_{new} which is created via a set operation of two process sets A, B the epoch number shall be $S_{new}(e) = max(A(e), B(e))$.

Definition 2.

***Malleable process sets**: Let S, T be process sets in a particular mpi://app/app_name domain. We define the **set of malleable process sets** for this application as*

$$X_m(app\_name) : \{S\} | S(a) \wedge \nexists T | T(a) \wedge S \subset T$$

Definition 3.

***Reference counted process sets**: Let S, T be process sets in a particular mpi://app/app_name domain. Let U be a process set in the mpi://rte/app_name/delta domain. We define the **set of reference counted process sets** for the delta process set U as*

$$X_c(U) : \{S\} | S \cap U \neq \emptyset \wedge S(a) \wedge \nexists T | T(a) \wedge S \subset T$$

Definition 4.

***Recursivley affected process sets**: Let S, T be process sets in a particular mpi://app/app_name domain. Let U be a process set in the mpi://rte/app_name/delta domain. We define the **set of recursively affected process sets** for a set operation of S and U op(S,U) as*

$$X_a(op(S,U)) : \{T\} | \neg T(a) \wedge (T \cap U \neq \emptyset \vee S \subset T)$$

6.2. Phase 1: Initiation of Resource Changes

In the first phase, resource changes need to be initiated by the RTE and the according information needs to be made available to MPI processes. Here we assume that the scheduler has decided on a particular change of the resources of a job. The certain policy on which this decision is based on is not further specified here and our approach is independent from scheduling policies. Thus, support for evolving applications could be realized on top of our approach as we briefly describe in Sec. 6.6. However, we assume that once the scheduler decided on a resource change, there is no application-sided negotiation possible.

6.2.1. Concept

Similar to [12] in our proposal resource changes are explicitly represented to the application instead of an implicit representation and automatic adaption of the application.

A resource change is represented by a set of processes and the type of the resource change. In this proposal, we differentiate between two basic types of resource changes: *Addition* and *Subtraction* of resources. The corresponding set of processes consists of the processes which are to be added or subtracted. We adopt the naming of this process set as *delta process set* as it represents the delta between the jobs processes before and after the resource change.

We use a query approach that allows applications to request relevant information about resource changes on demand. This is in tune with typical HPC workloads where a change of the number of processes is restricted to certain program points e.g. during synchronization in between iterations of loop based workloads. Processes can either query for resource changes targeting the application or targeting a particular process set.

6.2.2. Realization

To realize the aforementioned concept we propose changes to the MPI, PMIx and Runtime Layer.

MPI Layer

We propose a new MPI Sessions routine, which allows applications to query the RTE about current resource changes for the job. The signature of this routine is shown in Listing 6.1.

The function returns a globally unique description of a resource change consisting of the type of the resource change `rc_type` and the name of the *delta process set* `delta_pset`. A variable of type `MPI_rc_type` can be assigned one of the three predefined values `MPI_RC_ADD`, `MPI_RC_SUB` or `MPI_RC_NULL` indicating addition, subtraction or no change of resources respectively.

Moreover, the `status` of the resource change is returned. The `status` of a resource change is one of the following predefined values:

- `MPI_RC_STATUS_NULL`: No valid resource change is currently available.
- `MPI_RC_STATUS_ANNOUNCED`: The resource change was announced by the RTE and adaption of the job is required.
- `MPI_RC_STATUS_PENDING`: Running processes have published connection information but the newly spawned processes have not yet confirmed their readiness to establish communication with existing processes.
- `MPI_RC_STATUS_FINALIZED`: The resource change was applied successfully.
- `MPI_RC_STATUS_ABORTED`: The resource change was aborted.

The `incl` parameter is a flag indicating if the calling process is a member of the *delta process set*. Processes can use this information to determine if they are newly spawned in case of a resource addition or if they are to be aborted in case of a resource

subtraction. Alternatively processes could access this information by searching for the specified *delta process set* via calls to `MPI_Session_get_nth_pset`.

Additional information to influence the behavior of the function can be specified in the `info` object. By default, this function queries for application level resource changes.

To support process set level targets for resource changes, the proposal defines the `MPI_RC_BOUND_PSET` key to specify the name of the process set for which resource change information is to be queried. When specifying *mpi://self*, the function queries for a resource change containing the calling process. If the `MPI_RC_BOUND_PSET` is not present in the MPI info object, the function queries for application level process sets, i.e. for resource changes bound to process set *mpi://app/{app_name}*. If no appropriate resource change was found, the value returned in `rc_type` will be `MPI_RC_NULL`.

```
int MPI_Session_get_res_change(MPI_Session session,
          MPI_rc_type *rc_type, char delta_pset[],
          bool *incl, MPI_rc_status_t *status,
          MPI_Info *info)

IN     session          session used
OUT    rc_type          type of the resource change
OUT    delta_pset       name of the new delta process set describing the resource change
OUT    incl             true, if the calling process is a member of delta_pset
OUT    status           current status of the resource change
INOUT  info              optional additional information
```

Listing 6.1: **MPI Interface for phase 1 of resource changes.** The function queries the RTE for available resource changes.

PMIx Layer

The details of the interaction of the RTE and the RM are beyond the scope of this work. Here, we only specify a simple, uni-directional mechanism for initiating resource changes via event notification. We define the `PMIX_RES_CHANGE_DEFINE` event to be used in the `PMIx_Notify_event` function. The `PMIX_RC_TYPE` and `PMIX_PSET_SIZE` keys should be used in the `info` parameter to specify the details of the resource change. Dynamic RTE implementations should register a corresponding event handler to react to `PMIX_RES_CHANGE_DEFINE` events.

To allow an application to request information about resource changes we extend the specification of the existing `PMIx_Query_info` function (Sec. 5.2.3). These extensions are used to realize the interaction between the `MPI_Session_get_res_change` function and the RTE.

Three new query keys are proposed:

- `PMIX_RC_PSET`: (`char*`) Name of the *delta process set*

- PMIX_RC_TYPE: (pmix_rc_type_t) A uint8_t type that defines the type of the resource change. The following constants can be used to set a variable of the type pmix_rc_type_t:
  - PMIX_RC_ADD: New processes should be added to the job.
  - PMIX_RC_SUB: Existing processes should be removed from the job.
- PMIX_RC_STATUS: (pmix_rc_status_t) A uint8_t type that defines the status of the resource change. The following constants can be used to set a variable of the type pmix_rc_status_t:
  - PMIX_RC_STATUS_NULL: No valid resource change is currently available.
  - PMIX_RC_STATUS_ANNOUNCED: The resource change was announced by the RTE and adaption of the job is required.
  - PMIX_RC_STATUS_PENDING: Running processes have published connection information but the newly spawned processes have not yet confirmed their readiness to establish communication with existing processes.
  - PMIX_RC_STATUS_FINALIZED: The resource change was applied successfully.
  - PMIX_RC_STATUS_ABORTED: The resource change was aborted.

Moreover, a new query qualifier key is proposed:

- PMIX_RC_BOUND_PSET: (char*) Name of the process set for which an available resource change should be queried.

To ensure that the query is handed to the RTE the PMIX_QUERY_REFRESH_CACHE qualifier should be specified via the info parameter.

Runtime Layer

The RTE is responsible for initiating resource changes as specified by the host system (scheduler/RM) and to provide the required information for a resource change in a consistent manner to the application processes.

When using the initiation mechanism via a PMIx event notification, the procedure described in this section could be triggered by the callback function registered for the PMIX_RES_CHANGE_DEFINE event.

The RTE is required to perform the following steps for initiating a resource change:

- **Definition of the *delta process set***: The RTE is responsible for defining the *delta process set* which describes the resource change. In case of a resource addition, the RTE has to ensure unique PMIx process identifiers for the new processes.

For this, a unique *rank* in the *namespace* of the given job has to be assigned. For resource subtraction, if not further specified by the requesting entity (e.g. the RM), processes to be removed should be chosen based on the `mpi_ps_sched_xxx` attributes cached on the process sets.

The existing `PMIx_define_process_set` function can be used to define the new process sets (Sec. 5.2.6). The host is free to choose a name for the new process set, however, it would be most convenient to use a particular subdomain in the URI format, e.g. *mpi://rte/{app_name}/delta*. It is important to define the *delta process set* before announcing the resource change to allow processes to query information about the *delta process set* such as its size or members.

- **Update of the PMIx Job Data**: The RTE needs to update the PMIx job data object on the PMIx server(s) to reflect the new job state. This is necessary, as application processes need to be able to access the updated job meta data during the adaption procedure and during startup respectively. Thus, it is important to perform this step before the resource change is announced to the application. The PMIx job data object can be set via the `PMIx_register_namespace` interface. The RTE should expect a certain time interval in which processes "adopt" this new job meta data. Application processes might still rely on the old job data until the adaption has been finalized.

- **Announcement of resource change**: After the previous two steps have been carried out, the RTE can announce the resource change by providing corresponding responses to `PMIx_Query_info` calls for the `PMIX_RC_PSET`, `PMIX_RC_TYPE` and `PMIX_RC_STATUS` keys. It is important that the resource change is announced before any new processes are spawned, as these processes need to determine their dynamic nature via the `MPI_Session_get_res_change` function during initialization.

  Moreover, the RTE needs to determine the process sets target by the resource change, to correctly respond to queries which specified the `PMIX_RC_BOUND_PSET` qualifier. When adding resources, the resource change should be bound to one of the *malleable process sets* (Def. 2) based on the `mpi_ps_sched_xxx` attributes cached on these process sets. When resources are subtracted, the resource change is bound to the *reference counted process sets* (Def. 3) w.r.t. the *delta process set*.

- **Process Management** Finally the RTE needs to manage the creation and termination of processes. New processes can be launched using the usual launch sequence, as MPI processes are able to establish communication by themselves in phase 2 and phase 3. As processes are added to an already running job, the RTE

needs to update its corresponding internal data for this job. Here, the details are dependent on the specific RTE implementation.

When processes are subtracted, the RTE needs to grant a certain time for the application to terminate these processes. For this, the state of the specified processes needs to be tracked to detect when all processes for the resource change have terminated. To this end, we provide further information in Sec. 6.4.

6.2.3. Discussion of the Design

The polling approach we use has the advantage that applications are able to receive information about available resource changes at any time. However, clearly over-usage of this functionality could impose stress on the RTE as it needs to frequently answer these queries. The intended usage of this function is therefore to restrict its usage to a particular process e.g. the root process of a communicator. The `MPI_Bcast` function can be used to distribute the queried information among the according processes.

A consequence of the polling approach is that the RM's performance of deallocating resources from a job depends on the application's interval of querying for resource changes. To include this into the scheduling decision, a valuable attribute to be cached on process sets would be the expected interval the application queries for resource changes for this process set.

Another valuable extension with regards to topology-aware resource allocation and fault tolerance would be resource replacement. With the current design, resource replacement could be realized as a sequence of resource subtraction and addition. In future work a dedicated third type of resource changes could be developed to enable resource replacement through a single resource change.

6.3. Phase 2: Application-driven Creation of New Process Sets

In the second phase, an application needs to prepare an adapted communication pattern, which takes the change of resources into account. That is, processes to be removed need to be excluded from current communication patterns and added processes need to be included.

6.3.1. Concept

In the MPI Session model, communication is derived from process sets. Thus, updating communication patterns requires dynamic updating of available process sets.

Similar to [12] we propose the usage of set operations to manipulate the set of available process sets.

Definition 5. *Let A, B be arbitrary sets of processes. We allow the creation of a new process set C based on the following set operations on A and B:*

- *UNION(A, B):* $C = \{p\} | p \in A \vee p \in B$
- *DIFFERENCE(A, B):* $C = \{p\} | p \in A \wedge p \notin B$
- *INTERSECTION(A, B):* $C = \{p\} | p \in A \wedge p \in B$

We further impose the condition that $C \neq \emptyset$.

6.3.2. Realization

Application-driven process set manipulation is a major difference to the current MPI specification, which assumes process sets to be defined by the RTE/MPI implementation only. Thus, allowing the creation of new process sets on behave of an MPI process requires a new level of coordination between the MPI processes and the RTE. Especially in the case of process set level resource changes a clear definition of responsibilities and permissions for process set operations is needed to sustain consistency. To this end we make use of the additional process set attributes defined in Sec. 6.1.3

MPI Layer

We propose a new MPI Sessions function based on [12] to request a process set operation of two process sets. The signature of this function is provided in Listing 6.2.

Possible values for the `op` parameter are the predefined constants `MPI_PSETOP_UNION`, `MPI_PSETOP_DIFFERENCE` and `MPI_PSETOP_INTERSECTION`. These constants represent the corresponding set operation as defined in Def. 5. Moreover, a value of `MPI_PSETOP_NULL` can be specified when updating process set attributes.

The parameters `pset1` and `pset2` represent the sets A and B from Def. 5 respectively. The user may specify a preferred name for the resulting process set (set C in Definition 5) using the `MPI_PSET_TARGET` key in the `info` object. The RTE is only obliged to use this name for the new process set if it is from the *mpi://app/{app_name}* domain. If successful, the RTE executes the operation on behave of the calling process and the name of the new process set is returned in `pset_result`.

To support the extended process set definition and process set level resource changes, the behavior of the function can be influenced through the usage of the `info` object. The user may specify attributes in the MPI info object to be applied to the process set specified by the `MPI_PSET_TARGET` key using the following keys:

- `mpi_ps_active` (bool)

- `mpi_ps_sched_xxx` (class of attributes providing scheduling hints)

If these keys are not provided in the MPI info object, the new process set will inherit the corresponding attributes from *pset1*.

To create a new version of a process set (i.e. to increment the version number) the name specified by the `MPI_PSET_TARGET` key should equal the name provided in the `pset1` argument. A value of `MPI_PSETOP_NULL` may be specified to solely update the attributes of an existing process set specified by the `MPI_PSET_TARGET` key. In this case the parameters `pset1` and `pset2` are ignored.

If `pset2` is a *delta process set* the same operation will be applied by the RTE to the *set of recursively affected process sets*. Thus, by according usage of the `mpi_ps_active` attributes, applications can delegate the task of maintaining the semantics of the process set hierarchy to the RTE.

```
int MPI_Session_pset_create_op(MPI_Session session, MPI_Info info,
              MPI_pset_op op, const char *pset1,
              const char *pset2, char *pset_result)

IN    session          session used
IN    info             optional additional information for the operation
IN    op               requested set operation
IN    pset1            name of the first process set of the operation
IN    pset2            name of the second process set of the operation
OUT   pset_result      name of the newly created process set
```

Listing 6.2: **MPI Interface for phase 2 of resource changes.** The function requests a process set operation from the RTE.

PMIx Layer

To realize the above MPI function, coordination between the calling MPI process and the RTE is required. To this end, we propose the following extensions to the PMIx Standard:

- **Pset Operation Directives:** The `pmix_psetop_directive_t` is a `uint8_t` type defining the set operation requested in a process set operation request. The following constants can be used to set a variable of the type `pmix_psetop_directive_t`:
  - `PMIX_PSETOP_UNION`: The union of two process sets is requested.
  - `PMIX_PSETOP_DIFFERENCE`: The difference of two process sets is requested.
  - `PMIX_PSETOP_INTERSECTION`: The intersection of two process sets is requested.

- **Pset Operation Attributes:** Attributes used with the new process set operation functions to specify the request and results of a process set operation.
  - `PMIX_PSETOP_P1`: The first process set in a process set operation.
  - `PMIX_PSETOP_P2`: The second process set in a process set operation.
  - `PMIX_PSETOP_PREF_NAME`: The preferred name of the resulting process set (optional).
  - `PMIX_PSETOP_PRESULT`: The name of the resulting process set assigned by the host system.
  - `PMIX_PSET_ACTIVE`: The value of the *active* attribute of the resulting process set.
  - `PMIX_PSET_SCHED_XXX`: Scheduling hints to be cache on the resulting process set.
- **Pset Operation Functions:**
  - `PMIx_Pset_Op_request`: Request a process set operation.
  - `pmix_server_pset_operation_fn_t`: Server function pointer definition for process set operations. The host system provides a pointer to the corresponding implementation during initialization of the PMIx server.
  - `pmix_psetop_cbfunc_t`: Callback function provided by the PMIx server to be called by the host system after executing the process set operation.

We propose a new PMIx client function to request a process set operation and to receive a corresponding answer with the results of the operation. The signature of this function is shown in Lst. 6.3. If successful, the `results` array will contain at least the name of the resulting process set, specified with the `PMIX_PSETOP_PRESULT` key. Further details on specifying attributes for the process set operation is provided in Lst. A.1. In our approach, we use this function to realize the functionality of the `MPI_Sessions_pset_create_op` interface.

The request is delegated to the local PMIx server, which will request the operation from the host system via the provided `pmix_server_pset_operation_fn_t` function pointer which is defined in Listing 6.4. A more detailed specification of this function is provided in Lst. A.2 in the Appendix.

After execution of the process set operation, the host system is required to call the `pmix_psetop_cbfunc_t` callback function specified by the PMIx server in the `cbfunc` parameter. For the sake of brevity, we do not include the signature of the callback function in this section. The interested reader is referred to Lst. A.3 in the Appendix.

```
pmix_status
PMIx_Pset_Op_request( pmix_psetop_directive_t directive,
                      const pmix_info_t info[],  size_t ninfo,
                      const pmix_info_t *results[],  size_t *nresults)

IN    directive
     pmix_psetop_directive_t specifying the requested pset operation
IN    data
     Array of info structures (array of handles)
IN    ninfo
     Number of elements in the info array (integer)
OUT    results
     Array of info structures (array of handles)
OUT    nresults
     Number of elements in the results array (integer)

Description:
Request a process set operation from the host environment. Possible operations are
PMIX_PSETOP_UNION, PMIX_PSETOP_DIFFERENCE and PMIX_PSETOP_INTERSECTION. If
successful, the returned results for the request must include the name of the
created process set (PMIX_PSETOP_PRESULT) and its size (PMIX_PSET_SIZE).
```

Listing 6.3: **PMIx function for requesting a process set operation.**

```
typedef pmix_status (*pmix_server_pset_operation_fn_t)(
                                 const pmix_proc_t *proc
                                 pmix_psetop_directive_t directive,
                                 const pmix_info_t data[],  size_t ndata,
                                 pmix_psetop_cbfunc_t cfunc,  void *cbdata);

IN    proc
     pmix_proc_t structure identifying the process requesting the process set operation (handle)
IN    directive
     pmix_psetop_directive_t specifying the requested process set operation
IN    data
     Array of info structures (array of handles)
IN    ninfo
     Number of elements in the data array (integer)
IN    cbfunc
     Callback function pmix_op_cbfunc_t (function reference)
IN    cbdata
     Data to be passed to the callback function (memory reference)

Description:
Request a process set operation from the host system. The request will include the
namespace/rank of the process that is requesting the operation, an array of
pmix_info_t describing the operation, and a callback function/data for the return.
```

Listing 6.4: **PMIx server function pointer to request a process set operation from the host system.** The host system provides a corresponding implementation.

Runtime Layer

RTEs should provide an implementation of the `pmix_server_pset_operation_fn_t` function following the specifications in Listing 6.4. When initializing the `PMIx server`, a corresponding pointer has to be passed in the `pmix_server_module_t` struct.

The specific implementation is allowed to use any criteria to decide wether it accepts the request. Moreover, if the `PMIX_PSETOP_PREF_NAME` key is not provided in the `info` object, the RTE is free to choose any name for the new process set as long as it does not conflict with any known namespace or process set.

However, RTEs that support malleable jobs should as a rule accept the request for operations in the *mpi://app/{app_name}* domain to enable applications to adapt themselves to resource changes. If the `PMIX_PSETOP_PREF_NAME` key is present in the info object, the RTE should use the specified value for the new process set if it does not conflict with the PMIx specification of process set names or any other system policies.

The RTE is responsible for preserving consistency when receiving multiple requests for process set operations, e.g. by serializing their execution.

The RTE should cache the specified attributes for the resulting process set and assign the version number and epoch attribute as specified in Sec. 6.1. Moreover, in case the process set operation involves a *delta process set*, the RTE is required to apply the same operation to all *recursively affected process sets*.

6.3.3. Discussion of the Design

The definition of process sets of the MPI Standard 4.0 (Sec. 4.4.3) does not permit deletion of process sets. Clearly this could be problematic if applications regularly create new process sets during execution.

The versioning approach we use in our proposal attenuates this issue. Especially when complex hierarchies of process sets are used, versioning is important to circumvent the creation of a large number of new process sets due to cascading effects for recursively affected process sets.

Such recursive application of process set operations is required to ensures that process sets remain valid e.g. when removing subtracted resources. Moreover, this preserves the logic expressed by the hierarchy of process sets.

However, even with versioning there could exist process sets which are not needed anymore throughout the execution of an application. The issue of removing process sets was already discussed in the MPI Sessions WG. A possible solution was proposed allowing applications to prune the list of process sets, which would "delegate the responsibility to the user to determine when it is safe to reduce the length of [...] the list of process sets" [18]. For instance, the RTE could delete a process set from its own

list of process sets, while this process set is only marked as invalid in the local list of process sets of an MPI process. The user could later remove invalid process sets from the local list when it is save reduce its length.

Notwithstanding the above discussion, RTE implementations have to provide efficient mechanisms for storage and manipulation of process sets. This is especially important considering the trend towards exascale applications, where process sets could possibly comprise millions of processes.

Furthermore, for efficiency, the `MPI_Session_pset_create_op` function could be extended to allow the execution of process set operations on multiple process sets in a single request.

For applications with more complex process set hierarchies, application developers have to take care of beneficial use of process set attributes. Most importantly, the `mpi_ps_active` attribute should be used carefully as it distributes responsibilities between the application and the RTE. In particular, this influences the *set of malleable process sets* and consequently (the granularity of) the malleability of the whole MPI job. However, for applications relying on application level resource changes our design does not impose this additional complexity due to the usage of the `MPI info` object and appropriate default values.

To achieve balanced assignment of resources, useful `mpi_ps_sched_xxx` attributes need to be assigned to process sets. To this end, future work is required to extend the proposed design with such attributes by taking into account the requirements of both, dynamic applications and batch system schedulers.

Beyond the usage of process set operations to reflect resource changes, a wider range of operations for manipulating process sets would be useful. In particular, operations to split process sets based on certain criteria are required to enable application developers to setup a hierarchy of malleable application parts in more complex applications such as task graphs.

6.4. Phase 3: Application-side Execution of Resource Changes

In the last phase, application processes need to execute the resource change. For this, processes need to coordinate the switch from old communication patterns to new ones to approach a state that is consistent with the change in resources announced by the RTE.

6.4.1. Concept

Establishing adopted communication is achieved via a coordinated switch from communicators derived from the *bound process set* of a resource change to communicators based

on updated or newly created process sets. Applications are responsible for creating such process sets via the `MPI_Session_pset_create_op` function during phase 2.

In the case of a resource addition, as no common `MPI_COMM_WORLD` exists, the new processes need to receive information "out-of-band" from running processes specifying how communication can be established. For this we use a *resource change specific rendezvous point* that is used to exchange the required information.

In the case of resource subtraction the remaining processes need to switch to a new communicator, while the processes included in the *delta process set* need to finalize. This requires disconnecting processes and cleanup of invalid communicators.

Throughout this process a consistent state of resources for all cooperating processes needs to be guaranteed.

6.4.2. Realization

We realize this concept by requiring processes in the *bound process set(s)* to call a new MPI routine for completing a resource change. Newly spawned processes need to call another new MPI routine for setting up connection to running processes. The interaction between these routines is non-blocking. The only interaction of the application with the RTE in this phase is the update of the resource change status.

MPI Layer

We propose two new functions: `MPI_Session_accept_res_change` (Lst. 6.5) and `MPI_Session_confirm_res_change` (Lst. 6.6). An overview of their functionality and interaction is given in Fig. 6.2.

The `MPI_Session_accept_res_change` function is called collectively by the processes in the *bound process set* to apply a particular resource change. Similar to the `MPI_Bcast` function, one process is considered the root of the communicator. The root process is responsible for specifying the name of the *delta process set* identifying the resource change in the `delta_pset` parameter as well as the name of the new process set from which a new communicator will be derived in the `new_pset` parameter. All other processes will receive the specified names in the buffers provided for the `delta_pset` and `new_pset` parameters.

In case of resource subtraction all processes will disconnect from the communicator `comm` and the `terminate` flag will be set for all processes in the *delta process set*. The `MPI_Session_accept_res_change` has to be called once for every process set in the *set of reference counted process sets*, in which case the status of the resource change will be changed to `MPI_RC_FINALIZED`. Subsequently processes for which the `terminate flag was set` call `MPI_Session_finalize` and terminate.

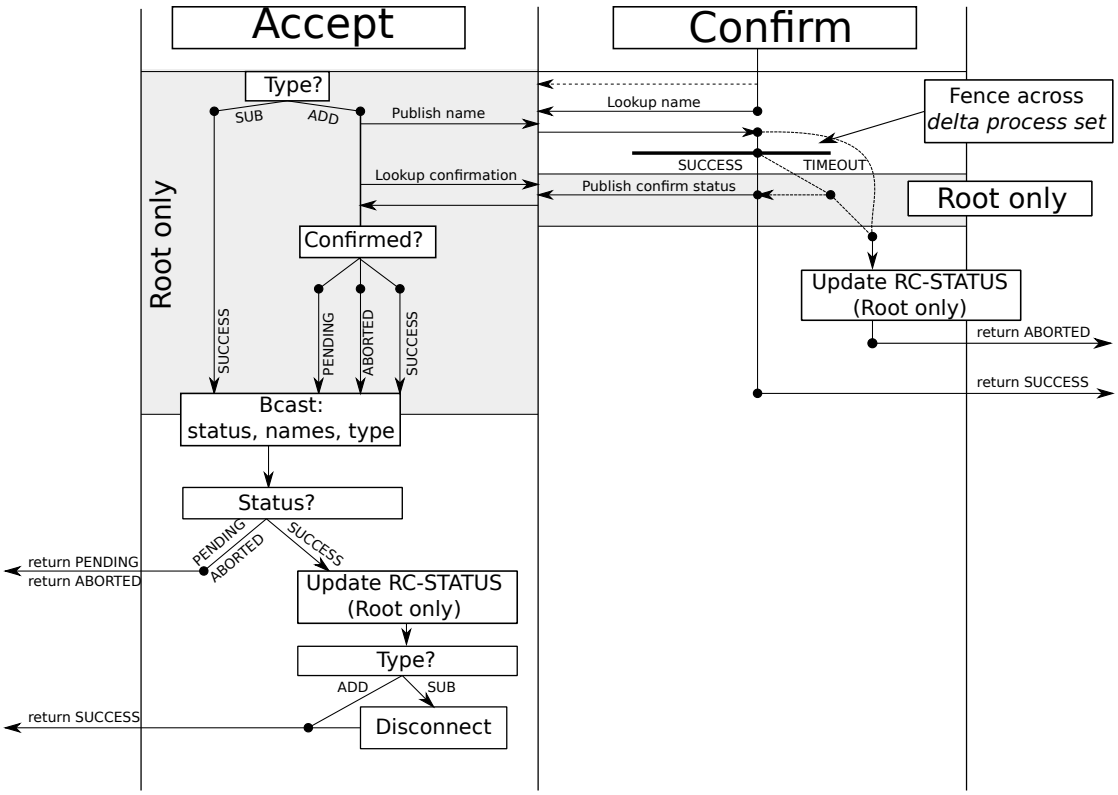


Figure 6.2.: **Concept of the** `MPI_Session_accept/confirm_res_change` **API.**

Advice to users. As the `MPI_Session_finalize` function is collective over all connected processes, processes first need to disconnect from all communicators other than the one specified by `comm` in the `MPI_Session_accept_res_change` calls. Information about disconnecting processes can be found in the corresponding section of the MPI Standard (*End of advice to users.*)

For resource additions the `MPI_Session_accept_res_change` function interacts with its counterpart `MPI_Session_confim_res_change` which is called by the dynamically added processes (Lst. 6.6). This interaction has some similarities to the existing `MPI_Comm_accept/connect` API as "two parts of an application that are started independently need to communicate" [29]. However, in our approach communication is not established via an inter-communicator. Instead, the name of a process set is exchanged, from which a common intra-communicator can be derived. Fig. 6.2 illustrates the realization of the necessary interaction of the `MPI_Session_accept_res_change` and the `MPI_Session_confirm_res_change` functions to exchange the information about the particular process set to be used for communication.

The root process of the communicator in the accept call publishes the process set name specified by the `new_pset` parameter with a well defined, resource change specific key (e.g. *{name of delta process set}:accept*). This changes the state of the resource change from `MPI_RC_ANNOUNCED` to `MPI_RC_PENDING`.

New processes perform a blocking lookup for the published process set name using the well defined key. Subsequently they participate in a fence operation across the *delta process set*, to ensure all new processes have received this information. The lookup and fence operation should eventually timeout after an implementation-defined time. In this case the status of the resource change is changed to `MPI_RC_ABORTED`. One of the new processes publishes a value indicating success or failure with a well defined key (e.g. *{name of delta process set}:confirm*). The function returns `MPI_SUCCESS` if all processes successfully looked up the name of the new process set and `MPI_ERR_PORT` otherwise.

The root process of the `MPI_Session_accept_res_change` call checks for the confirmation value. By default this lookup is non-blocking and in case the confirmation value was not yet published the function will return `MPI_ERR_PENDING`. Subsequent calls to `MPI_Session_accept_res_change` should be used to check again for the confirmation at a later point in time. The looked up confirmation status as well as the process set names and resource change type are broadcasted to all processes in the communicator. If the looked up confirmation value indicates success, the status of the resource change is changed to `MPI_RC_FINALIZED` and the function returns `MPI_SUCCESS`. Otherwise `MPI_ERR_PORT` is returned.

In case of success, running and new processes can establish communication via the process set specified in `new_pset`.

Advice to implementers. A high quality implementation may provide a mechanism, through the info argument to `MPI_Session_accept_res_change`, for the user to specify a time for the lookup to wait for the confirmation to be published. (*End of advice to implementers.*)

Advice to implementers. Implementations might need to update implementation-dependent job meta data and/or endpoint information before returning from the `MPI_Session_accept_res_change` with `MPI_SUCCESS`. When using PMIx, the RTE will have updated the *PMIx job data object* on the PMIx server during Phase 1. (*End of advice to implementers.*)

```
int MPI_Session_accept_res_change(MPI_Session session, MPI_Info info,
          const char *delta_pset, const char *new_pset
          int root, MPI_Comm comm, int terminate)

IN    session          session used
IN    info             optional additional information for the operation
IN    delta_pset       name of the delta process set identifying the resource change
IN    new_pset         name of the process set to be use for establishing communication
IN    root             the rank of the process publishing the accept
IN    comm             communicator for the collective operation
OUT   terminate        flag indicating that the process needs to terminate
```

Listing 6.5: **MPI interface for phase 3 of resource changes.** `MPI_Session_accept_res_change` is a non-blocking function, collective over the communicator `comm`. It is called by the processes in the `bound process set(s)` to accept a resource change.

```
int MPI_Session_confim_res_change(MPI_Session session, MPI_Info info,
          const char *delta_pset, char *new_pset)

IN    session           session used
IN    info              optional additional info for the operation
IN    delta_pset        name of the delta process set name identifying the resource change
OUT   new_pset          name of the process set to be used for establishing communication
```

Listing 6.6: **MPI interface for phase 3 of resource changes.** `MPI_Session_confirm_res_change` is a blocking function, collective over processes in the `delta_pset`. It is called by dynamically added processes to connect to running processes.

PMIx Layer

The realization of the MPI interface described in the previous section can mostly be based on the following, already existing PMIx functionalities:

- `PMIx_Publish/PMIx_Lookup`: As described in Sec. 5.2.2 PMIx provides a mechanisms to publish and lookup key-value pairs for a specific `PMIx_Range`. The `PMIX_RANGE_NAMESPACE` is sufficient for this particular use case. An array of `pmix_info_t` structs can be passed to the functions to provide additional information. Thus, a timeout mechanism for the `PMIx_Lookup` can be achieved through the usage of the `PMIX_WAIT` and `PMIX_TIMEOUT` keys.
- `PMIx_Fence`: The `PMIx_Fence` operation can be used to realize the barrier required for synchronization in the `MPI_Session_confirm_res_change` function. To specify the processes participating in this operation, the members of the corresponding

delta process set first need to be queried using the `PMIx_Query_info` function. Again, a timeout mechanism can be achieved by specifying a corresponding value for the `PMIX_TIMEOUT` key in the `info` object.

- `PMIx_Notify_event`: To realize the updates of the resource change status, the `PMIx_Notify_event` function can be used to inform the RTE about such updates. To this end we define the event code `PMIX_RC_UPDATE_EVENT`. Required information to be passed via the `info` parameter are the name of the *delta process set* (`PMIX_PSET_NAME`) and the new status of the resource change (`PMIX_RC_STATUS`). To restrict the delivery of the event to the host system, the `PMIX_RANGE_RM` should be specified. The RTE is required to register a corresponding event handler for the `PMIX_RC_UPDATE_EVENT`.

Runtime Layer

During the third phase, the RTE is responsible for managing and reacting to changes of the resource change status. The following illustrates the requirements for an event handler of the `PMIX_RC_UPDATE_EVENT`.

For **resource additions** the RTE needs to handle three status updates.

- In case of a notification for the `PMIX_RC_STATUS_PENDING` status, the RTE simply updates the status in the resource change query data.
- In case of a notification for the `PMIX_RC_STATUS_FINALIZED` status, it updates the status of the resource change and makes it unavailable for queries by the `MPI_Session_get_res_change` function.
- In case of a notification for the `PMIX_RC_STATUS_ABORTED` status, it first needs to update the status in the resource change query data accordingly. Resources of an aborted resource change are expected to be removed from the allocation to be rescheduled. Thus, the termination of processes included in the *delta process set* needs to be tracked and eventually enforced. When all processes of the *delta process set* have terminated, the resource change needs to be made unavailable for queries by the `MPI_Session_get_res_change` function. Moreover, the internal job data as well as the `PMIX_job_data` on the PMIx server(s) needs to be updated to reflect the removal of the previously added processes.

In case of a **resource subtraction**, notification with the `PMIX_RC_STATUS_FINALIZED` status indicates that the resource change was accepted by the processes in one process sets in the *set of reference counted process sets* $X_c(U)$. The RTE needs to count such notifications for a particular *delta process set* U. If $|X_c(U)|$ notifications arrived, the RTE

updates the status of the resource change to `PMIX_RC_STATUS_FINALIZED` and makes it unavailable for queries by the `MPI_Session_get_res_change` function. The termination of processes included in the *delta process set* needs to be tracked and eventually enforced. When all processes of the *delta process set* have terminated the runtime-internal job data needs to be updated accordingly and resources can be de-allocated from the job.

6.4.3. Discussion of the Design

We use a (by default) non-blocking design of the `MPI_Session_accept_res_change` function to enable applications to continue their work until the new processes are actually ready to establish communication. Especially in large scale systems the startup time for MPI processes can be significant in which case a blocking approach could lead to significant overhead for the application.

The design of the `MPI_Session_accept_res_change` function as a collective call based on an MPI communicator could potentially be limiting. Here, our design is somewhat based on the assumption of the existence of an MPI communicator in a 1:1 relation to the *bound* process set. However, in general this might not always be the case and creation of such a communicator might impose an unnecessary overhead. In the future work, our design might be extended to take this special case into account.

With regards to implementations of the design, our work on a prototype has revealed the need for updating implementation specific job meta data end/or endpoint information (Sec. 7.4.1). Especially when extending the design towards resource replacement, changing endpoint and topology information needs to be distributed consistently. It is relatively straight-forward to explicitly update this data during the `MPI_Session_accept_res_change` function for processes in the *bound process set(s)*. However, subsequent merging of process sets could introduce inconsistencies for processes that did not "accept" the resource change. This could be circumvented by updating the job data during each call to `MPI_group_from_session_pset`. However, a more efficient solution would use the `mpi_ps_epoch` attribute cached on process sets to determine the necessity of updating job data and endpoint information. As the *epoch* number represents a particular state of resources, each process can determine differences between its own resource state and the one represented by a process set. For this, processes store a local *epoch* number, which is the maximum *epoch* number encountered for process sets during calls to `MPI_group_from_session_pset`. Whenever the epoch number of the process set in the `MPI_group_from_session_pset` call is larger then the local epoch number a job data/endpoint update is required.

6.5. Illustration of Usage

In this section we provide two examples for the usage of our design.

The first example (Lst. 6.7) illustrates a simple use case of application level resource changes for loop based workloads, such as iterative solvers on grids with load-balancing.

At the entry point of the application, processes check if they were started dynamically using the `MPI_Session_get_res_change` function. Note, that `NULL` is passed for the `info` parameter as all resource changes target the application. Dynamically added processes need to call `MPI_Session_confirm_res_change` to receive the name of the process set to be used to establish communication.

After creation of a communicator the processes enter the main loop. Each iteration consists of a re-balance, work and resource change step. The re-balance and work steps are application dependent.

During the resource change step the root process of the communicator checks for resource changes via the `MPI_Session_get_res_change` function and eventually creates a corresponding new process set via the `MPI_Session_pset_creat op` function. Then it broadcasts the received resource change status to all other processes.

In case of an announced or pending request all processes collectively call the `MPI_Session_accept_res_change` function. If successful, processes derive a new communicator from the new process set or terminate respectively.

The general structure would be the same for process set level resource changes except that additional information would be passed to the MPI routines via MPI info objects.

To illustrate the potential of using our design for more complex applications we consider dynamic resource management for a task graph as shown in Fig. 6.3. Many parallel scientific workloads can be express by a task graph. In this example we consider graphs that exhibit not only inter-task parallelism but also intra-task parallelism as they arise for instance from factorization of sparse matrices using the multi-frontal method [26]. The task graph in our example consists of 10 nodes, with each node representing a parallel, malleable task. The edges in the graph represent task dependencies, that is, a task can only be started if the predecessor task has been finished. For the sake of illustration we assume that tasks at a certain depth d, start and finish simultaneously (e.g. nodes `N1.x` all finish simultaneously, subsequently all nodes `N2.x` execute simultaneously etc.). The numbers in the boxes above a node represent the number of processes at the start of a task. The numbers in the boxes next to a node represent the number of processes executing the task after a resource change has occurred.

```
/* create a session handle */
MPI_Session_create(&session);

/* check if process was added dynamically. If so, confirm the resource change */
MPI_Session_get_res_change(session, &rc_type, &delta_pset, &incl, &status, NULL);
if(incl){
  MPI_Session_confirm_res_change(session, NULL, delta_pset, main_pset);
}

/* Derive communicator from process set */
MPI_group_from_session_pset(session, main_pset, &main_group);
MPI_Comm_create_from_group(main_group, "tag", NULL, NULL, main_comm);

/* MAIN LOOP */
while(work_to_do){

  /* application dependent load-balancing and work step */
  rebalance_step(num_procs);
  work_step();

  /* -------------------- RESOURCE CHANGE STEP -------------------- */
  if(my_rank == root){
    /* -------------------------- Phase 1 --------------------------*/
    MPI_Session_get_res_change(session, &rc_type, &delta_pset, &incl, &rc_status,
        NULL);
    /* -------------------------- Phase 2 --------------------------*/
    if(rc_type != MPI_RC_NULL && rc_status == MPI_RC_ANNOUNCED){
      op = rc_type == MPI_RC_ADD ? MPI_PSETOP_UNION : MPI_PSETOP_DIFFERENCE;
      MPI_Session_pset_create_op(session, op, NULL, main_pset, delta_pset,
          result_pset);
    }
  }
  MPI_Bcast(&rc_status, 1, MPI_INT, root, main_comm);
  if(rc_status == MPI_RC_STATUS_ANNOUNCED || rc_status == MPI_RC_STATUS_PENDING){
    /* -------------------------- Phase 3 --------------------------*/
    rc = MPI_Session_accept_res_change(session, NULL, delta_pset, new_pset, root,
        main_comm);
    if(rc == MPI_SUCCESS){
      if(terminate)
        break;
      /* derive a communicator from the new process set */
      strcpy(main_pset, new_pset);
      MPI_group_from_session_pset(session, main_pset, &main_group);
      MPI_Comm_create_from_group(main_group, "tag", NULL, NULL, main_comm);
    }
  }
  /* ---------------- END OF RESOURCE CHANGE STEP ---------------- */
}

MPI_Session_finalize(session);
```

Listing 6.7: **Illustrative example of using our approach for writing malleable, iterative MPI applications.** In this example resource changes target the whole application.

The middle column of the figure illustrates the process set hierarchy during each execution phase. At the start of the execution only one process set exists representing the 128 processes executing node `N.0.0`. After finishing this task, two new process sets are created to represent nodes `N1.0` and `N1.1` by splitting the N.0.0 process set. For this, our current design would need to be extended to allow splitting of process sets in addition to the set operations. We assume that only process sets representing currently executed tasks are *active* (red process sets in Fig. 6.3), while process sets of finished tasks are set to *inactive*. The numbers inside the process sets, e.g. 128 for process set `N.0.0`, represent the optimal number of processes for the corresponding task. (This could for instance be cached as `mpi_ps_sched_opt` attribute on the process sets).

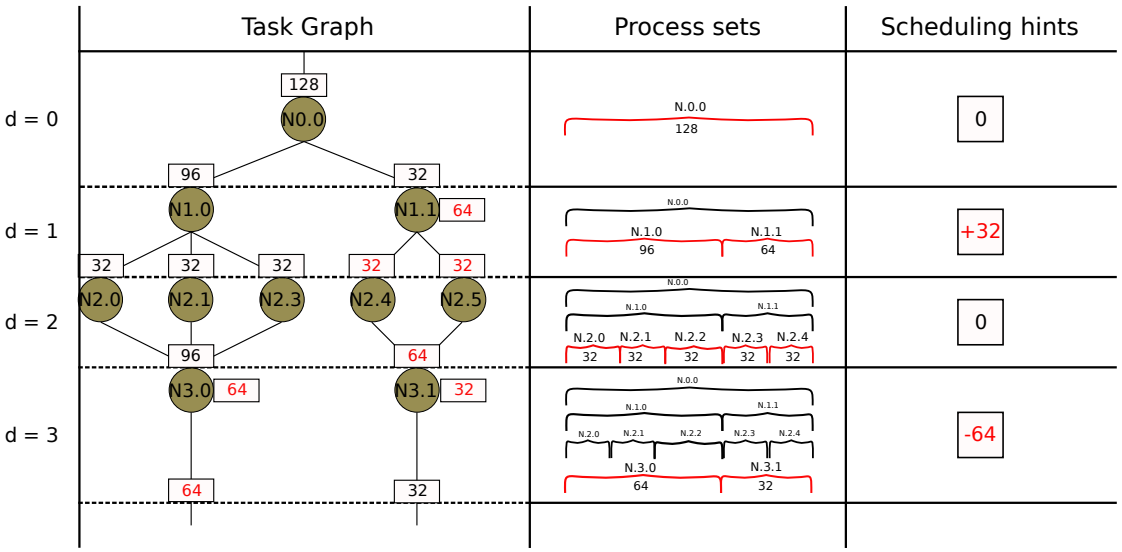


Figure 6.3.: **Illustration of using our approach for dynamic resource management of task graphs.** The task graph (left) is executed from top to bottom. Numbers above a node represent the number of processes at the start of a task. Numbers next to a node represent the number of processes executing the task after a resource change has occurred. In the middle column the distribution of processes to tasks is represented by a process set hierarchy, which grows over time. Only red process sets are active, corresponding to the tasks which are currently executed. The number inside a process set represents the optimal number of processes for the corresponding task. The right column shows the aggregated resource requirements of the application at each level of the task graph.

By comparing the current number of processes in a process set with the optimal

number of processes specified for this process set, the RTE could pass on aggregated scheduling hints to the scheduler. For instance at level $d = 1$ (second column), `N1.0` has the optimal number of 96 processes, but `N1.1` is only executed by 32 instead of the optimal 64 processes specified for this process set. Therefore, the overall requirements to optimize performance at level $d = 1$ are 32 additional processes. In this example the scheduler satisfies the requirement by assigning additional resources to the job, and the RTE announces a corresponding resource change for `N.1.1`.

Similarly, on level $d = 3$, the RTE aggregates the requirements of process sets `N3.0` and `N3.1`. The RTE could inform the scheduler that a total of 64 processes can be removed from the job. Upon approval of the scheduler the RTE could define resource changes for process sets `N3.0` and `N3.1` removing 32 processes each. After completion of the resource changes, the resources are handed over to the RM for reallocation.

While this example simplifies certain aspects of a real application it demonstrates the flexibility of our design and motivates future work on advanced use cases such as graph based applications.

6.6. Possible Extensions and Integration on HPC Systems

The design as proposed in the last sections does not impose any restrictions on the entity responsible for deciding over resource changes. Exploiting the generality of the PMIx interface, different entities could trigger resource changes by connecting to corresponding PMIx servers as a PMIx tool. Thus, our approach can be used beyond the classical notion of malleable jobs, where the scheduler of an HPC system exclusively triggers resource changes to optimize for system-wide objectives. In this section we discuss the potential of using and extending our design towards evolving jobs, intra-job load-balancing and fault tolerance.

Evolving jobs are jobs which self-reliantly and dynamically adjust their assigned resources. In typical HPC systems, real evolving jobs are uncommon as allowing applications to initiate resource changes could conflict with overall system efficiency. However, our design offers possibilities for cooperative resource management of HPC jobs on HPC system.

One such possibility arises from the caching of scheduling hints on process sets. By specifying useful `mpi_ps_sched_xxx` attributes on each process set, the RTE could create aggregated scheduling hints applying to the whole job. The example with a task-graph in the last section illustrated a simple realization of such an attribute, which is aggregated by the RTE to describe the resource requirements of the whole job.

To enable the inclusion of such requirements into system-wide scheduling decisions, interaction between the RTE and the batch system is required. In the case where an MPI

job is directly launched and managed by the batch system (e.g. when using SLURM's `srun` command to launch an MPI job) this is straightforward as the RTE is provided by the batch system itself. However, a more generic mechanism for such interaction could be realized using PMIx. This would for instance enable MPI jobs which were launched inside a SLURM allocation via `mpirun` to interact with the SLURM workload manager for dynamic resource management.

The need for such interactions has been recognized by the PMIx community and they proposed the usage of a combination of the notification system, allocation requests and job control actions building a general transaction API. To give an example, the batch system could connect as a *PMIx tool* to a *PMIx server* operated by an MPI RTE. This way, the scheduler could first send a notification of its intended resource change action to the RTE. The RTE, having knowledge of the jobs requirements based on an aggregation of the process set attributes, could send back a notification describing conditions for its willingness to execute the proposed resource change. Similarly, the RTE could use job control actions to mark resources as preemptive or it could send resource allocation requests for additional resources. In future work such interactions based on this transaction-like API could be defined in more detail to enable cooperative resource management on top of our approach.

The scheduling hints cached on process sets further open up the potential for intra-job resource management and load-balancing. Especially, when using performance-based dynamic models of resource requirements for each process set, dynamically redistributing resources between different application parts could improve overall application performance. To this end, for instance, a dedicated "resource change" type could be developed which. Instead of representing a real change in resources, this new type could indicate the migration of processes between process sets for re-balancing purposes. The task of performance profiling and definition of according "resource changes" for load balancing could be undertaken by the application itself or delegated to external tools. Such tools could be written as PMIx tools connecting to a PMIx server operated by the RTE.

Finally, our design could also be extended towards usage for fault tolerance mechanisms. For instance, the dynamic resource adaptivity of our approach fits well with pro-active fault tolerance schemes, where hardware parts which are expected to fail soon are replaced by the RM. Moreover, an integration of our approach with the ideas of fault-tolerant MPI (FT-MPI)[11] could be explored to further improve fault tolerance for MPI.

The discussion in this section shows, that possible use cases of our concept are numerous and diverse which motivates future work to explore its full potential for HPC systems and applications.

7. Implementation of a Prototype with Open-MPI, Open-PMIx and PRRTE

We implemented a prototype for evaluating the proposed design, which we describe in this section.

Our prototype extends the implementation of Open-MPI (OMPI), thus in Sec. 7.1, we briefly introduce the major modules of the Open-MPI project.

Next, we discuss the objective and scope of our prototype implementation.

Sec. 7.3 gives a high level overview of our prototype, illustrating the interaction and data flow between the different layers and components.

A more detailed explanation the implementation and necessary changes to the original code is given in Sec. 7.4. Here we particularly focus on the implementation of the RTE, which provides a reference for integrating our approach into the SMS.

7.1. The Open-MPI Project

Our implementation is based on Open-MPI, an open source MPI implementation. Recently, an experimental implementation of the MPI Sessions model has been realized with Open-MPI [17]. Our prototype was built on top of the Open-MPI *sessions pr* developing branch involved in the *Sessions pr #59* pull request [35]. Thus, in this section we briefly introduce the Open-MPI project.

A general concept used in the Open-MPI project is the Modular Component Architecture (MCA). The idea is to define an abstract interface for a set of functionalities (framework), that can be loaded/unloaded at runtime (components) and supports different component instances (modules). To give an example the Remote Messaging Layer (RML) framework defines functionalities for the routing of out-of-band messages. At runtime a corresponding component e.g. the *out-of-band* component is loaded which could be realized with a TCP module. This architecture allows for independent development of Open-MPI components and enables fine-grained control to tune Open-MPI towards specific requirements of the system and use case at runtime.

Beyond the MCA, Open-MPI consists of different layers and submodules.

The OMPI project layer provides a full implementation of the interface specified by the MPI standard, which can be used to write parallel applications for distributed

compute systems. To avoid confusion, we use the abbreviation OMPI solely to refer to this layer of the *Open-MPI* project, while the term Open-MPI always refers to the overall project.

For interaction with the hardware and OS, the OMPI layer makes use of utilities from the Open Portability Access Layer (Open Portability Access Layer (OPAL)), such as memory management and container classes.

Moreover, a PMIx-enabled MPI RTE is included, the so-called PMIx Reference RunTime Environment (PRRTE). PRRTE, formally known as ORTE, was forked from Open-MPI several years ago and now is a standalone project within the PMIx community. It provides full-range PMIx support and is the default RTE for launching Open-MPI programs. PRRTE launches daemons on all allocated nodes, creating a persistent Distributed Virtual Machine (DVM), which we refer to as the *daemon job* in the following. Subsequently, *user jobs* i.e. applications using Open-MPI are run against this DVM. Thus, PRRTE provides a portable RTE based on PMIx which is independent from the batch system.

Both, the OMPI and PRRTE layer make use the Open-PMIx implementation, an open-source reference implementation of the PMIx standard.

Open-MPI can be configured to either use internal or external versions of PRRTE and Open-PMIx.

7.2. Prototype Objective, Scope and Limitations

As the realization of our proposed design requires major changes in several layers of the SMS, the developed prototype does not cover the full range of functionalities supported by our design. Thus, in this section we outline the objectives, scope and limitations of our prototype, to provide a concise delimitation from the proposal in the last section.

- **Objectives:**
  - The primary goal of the prototype is to demonstrate the basic capability of dynamically changing the number of processes of MPI applications during runtime using our proposed concept.
  - As a secondary goal we aimed to evaluate the basic performance characteristics. However, although we perform preliminary performance tests of our prototype, a high performance implementation was not a primary objective of the development. Several changes in the original as well as our code extensions will most certainly improve the performance of our prototype.

Instead, our performance tests are meant to identify the main performance bottlenecks of our approach and motivate future work.

- **Scope:**
  - We focused on realizing the proposed concept for application-level resource changes. Application level resource changes are sufficient to demonstrate and evaluate the basic sequence and characteristics of the operations of our proposed design. This includes runtime queries, process set operations, process launch, process state monitoring and meta data updates. For supporting process set level resource changes additional implementation effort would be required mainly at the runtime layer for managing and evaluating process set attributes to target particular application parts. In future work, such support for process set level resource changes could be implemented on top of the basic mechanisms developed for this prototype.
  - Our prototype is not yet integrated into a dynamic RM. Thus, the role of the RM has to be simulated to some degree in the runtime layer as we outline in the following. For our approach, the persistent DVM created by PRRTE represents the full set of available resources in the system. When running beneath a RM, these are the resource assigned to the user by the RM. The *daemon job* undertakes the task of dynamic resource allocation for the *user job*. For this, we use a stack approach as illustrated in Fig. 7.1 to decide where processes are added or removed. The approach is based on enumerating the nodes and the node slots of the DVM such that each slot in the system has a unique index, leading to a total order of slots in the system. When processes are added, they are started on the next free slots in the order, allocating new resources from the DVM if necessary. Similary, when processes are removed, the processes running on the slots with the highest index in the *user job* are terminated and "empty" nodes are deallocated from the *user job*.

- **Limitations:**
  - Due to implementational issues, a small subset of the parameterization described in Sec. 8.2 was not yet reliably supported by our prototype at the time of running the benchmarks. In particular, this concerned resource subtraction in the *SWE benchmark* (Sec. 8.1.2) on more than 3 nodes and benchmarks in *step mode*, when starting with resource subtraction. For reference we provide the code used for the prototype evaluation [21]. In addition, we like to refer the reader to the Github repository dedicated to the ongoing work on our prototype [20], in which we already resolved some of the aforementioned issues.

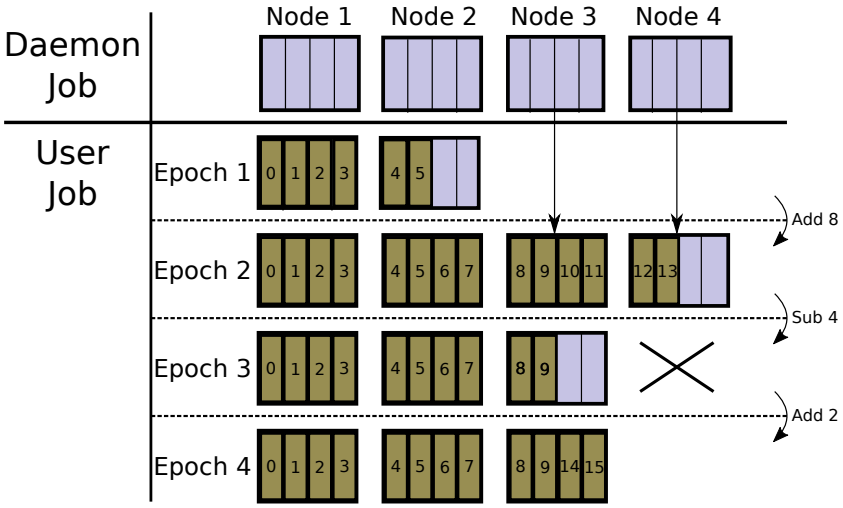


Figure 7.1.: **Stack approach used by the prototype for dynamic resource allocations**. In this example, the simulated system consists of four nodes, each providing four slots, thus the slots in the can be enumerated with indices $0-15$. In epoch 1, the user job (specified with the `prterun` command) is started against this DVM with four processes running on node 1 and two process running on node 2, i.e. in slots $0-5$. Note, that the numbers inside the node slots denote the ranks of processes running in these slots. In epoch 2, eight processes are started on the next free slots of the total ordering slots ($6-13$). First the remaining 2 slots in node 2 are filled up (slot 6 + 7). To start the other six processes, the allocation for the *user job* is extended by two nodes from the DVM. Similarly, when removing four processes in epoch 3, the processes running on the slots with the highest index in the total ordering (slots 13 - 9) are removed. As no processes are running on node 4 anymore, it can be deallocated from the user job and returned to the *deamon job* (representing the RM).

7.3. Overview of the Prototype

In this section, we provide a high-level overview of our prototype.

Fig. 7.2 shows a cutout of two nodes from an MPI job using our prototype, analogous to Fig. 3.1. Similarly, we differentiate between the application layer, the MPI layer (OMPI), the PMIx layer (Open-PMIx) and the runtime layer (PRRTE).

We start by describing the **PRRTE master layer**. The PRRTE master process is running on the head node of the allocation. It executes the`prterun` command which first sets up

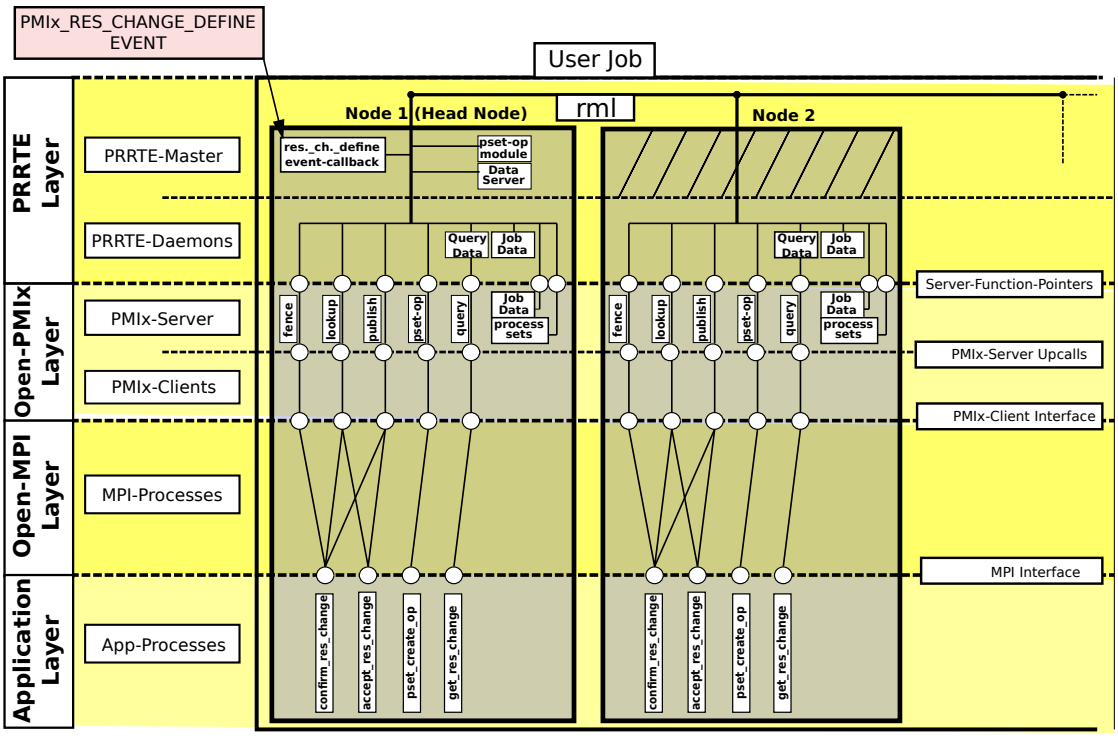


Figure 7.2.: **Overview of the prototype implementation.**

the *daemon job* (starting one daemon on each node of the allocation) and subsequently sends the launch command for the *user job* to the daemons. Note, however, that the PRRTE master simultaneously executes the role of a PRRTE daemon. The *PRRTE master* is responsible for initiating the resource changes. For this, it listens for resource change commands and sends corresponding commands to the daemons. Moreover, in our design the PRRTE master hosts a process set operation module responsible for executing process set operations as well as the data-server. In principle, however, these components could be hosted by any daemon or even on designated hardware providing these services.

Next, we describe the **PRRTE daemon layer**. On each node an instance of a PRRTE daemon is running which operates a local *PMIx server*. At startup of the PMIx server, function pointers are provided in the `pmix_server_module_t` struct. In particular relevant for resource changes are the function pointers for five PMIx services: queries, process set operations, publish/lookup requests and fence operations. These are indicated in Fig. 7.2 at the interface between the Open-PMIx and PRRTE layer.

Moreover, each daemon stores some local meta data. This includes a local *PRRTE*

job data object (not to be confused with the *PMIx job data*) which stores job meta data and tracks the job and process states (not included in the figure). Particularly relevant for resource changes, each daemon stores queryable data about process sets and resource changes, which allows them to answer such queries locally. The PRRTE master sends commands for updating query data (e.g. when a new resource change command arrived) to the daemons to ensure consistency. Process set operations and publish/lookup operations require interaction with the process hosting the process set operation module and the data server (here the master process) using communication via the RML. Fence operations between remote processes require interaction between the corresponding daemons via RML communication.

Finally, we consider the **OMPI and Open-PMIx layers**. In Fig. 7.2, at the interface between the application layer and the OMPI layer, we denoted the four new MPI functions for resource changes which we described in Sec. 6. These functions make use of the five PMIx services present at the PRRTE daemon layer. The lines between the MPI interface and the PRRTE daemon layer illustrate how the MPI functions make use of these services using the PMIx client interface. Some of these services can be provided at the Open-PMIx layer, e.g. when a local fence operation is requested or certain data has been cached. If services cannot be provided on the Open-PMIx layer, the request is forwarded to the PRRTE layer via the registered function pointers.

7.4. Details of the Implementation

In this section, we provide more detailed information about the realization of the implementation. For our prototype we needed to extend the implementations of all three layers.

A general issue, which appeared in all layers was the assumption of continuous ranking in a job. Especially loops over all processes, i.e. process ranks, in a job were based on this assumption. We changed these occurrences accordingly to account for arbitrary ranks inside a job.

In the OMPI/OPAL and Open-PMIx layer our implementation follows closely the proposed design for application level resource changes. Thus, in Sec. 7.4.1 and 7.4.2 we mainly describe issues and required changes of the existing code, which are not included in the proposal. Consequently, these sections mainly target readers familiar with and/or interested in the very details of these implementations.

On the runtime layer, i.e. PRRTE, our proposed design only defines the functionality via the interface to the PMIx server to account for the heterogeneity of RTE implementations. We therefore provide a more detailed explanation of our concrete realization with PRRTE. Consequently, this section is a valuable reference for readers particularly

interested in the realization and integration of MPI RTEs to support our proposed design.

7.4.1. OMPI and OPAL

Most MPI functions of the proposed design could mainly be implemented as wrappers for the corresponding PMIx functions as described in Sec. 7.3 and illustrated by Fig. 7.2.

However, the newly introduced dynamic nature of processes made some changes to the existing code necessary.

The original implementation of `MPI_Session_init` uses a collective fence operation over all processes in the job. This is implemented as a `PMIx_Fence` operation with the `PMIX_RANGE_NSPACE` scope. This was based on the assumption that processes which are started together, e.g. by mpiexec or MPI spawn, share a common, unique PMIx namespace. In our approach, however, new processes are added to the namespace of the running processes. In the original implementation, adding new processes would therefore require already running processes to participate in this fence operation. We changed the scope of the fence operation to only span the newly created processes. For this, during `MPI_Session_init` the RTE is queried for resource changes which include the calling process. In case such a resource change exists, the fence operation is performed across all processes which are members of the according *delta process set*. If no such resource change exists, the fence operation is performed over the complete namespace as in the original code.

Further changes were required for updating job meta data on the OPAL and OMPI layers such as the local peers information during `MPI_Session_accept_res_change`. First, the PMIx job-data, which is received by *PMIx clients* during initialization, needs to be updated. To this end, we used a new PMIx function which is described in more detail in the next section. Subsequently, the OPAL/OMPI `process_info` struct is updated with the new information for the job size, local peers and local rank.

Moreover, some effort was required to handle issues arising from introducing dynamicity into selected frameworks and components. It has to be noted that due to the large number of available components we focused only on the particular components relevant to the configuration of the development and evaluation systems. We have not yet tested compatibility with all possible components of the Open-MPI project.

On the byte transfer layer our prototype uses the shared memory component for communication on the local node. Here some issues needed to be addressed when dynamically changing the number of processes on the local node. This is due to the design of the component using the local rank as index into a fixed-sized array of endpoints. The size of the array of endpoints is initialized based on the number of local peers during `MPI_Session_init`. To allow for adding of new local endpoints, we

needed to adjust the initial size of this array. For simplicity we still rely on a fixed size, however this size is chosen accounting for the addition of processes at runtime. The fact, that the local rank of a process plays a role in locating shared memory segments further required updating endpoints. Here we rely on deleting and re-initializing shared memory endpoints for every resource change involving local processes.

We further made changes to the internal representation of process sets in the OMPI layer. This change was not strictly mandatory for implementing the proposed design. However, it simplified the implementation and adds flexibility to support future work towards high-performance implementations, e.g. through caching and shared memory techniques.

7.4.2. Open-PMIx

Besides the implementation of the new process set operation API as specified in Sec. 6.3 and new keys and attributes for the `PMIx_Query_info` and `PMIx_Notify_event` functions, only minor changes were required in the Open-PMIx layer.

One of these changes is a dedicated function for updating the local job data of a `PMIx client`, required for the corresponding updates in the OMPI and OPAL layers. Here we used the same mechanism present in the `PMIx_Init` function for retrieving the job data from the local *PMIx server*. Our prototype uses the `hash` module for the `gds` component however due to the modular design, this new function is independent of the used module.

Moreover, an issue requiring some caution arose from the implementation of the `PMIx_register_nspace` function. The original implementation provides explicit support of updating job data only for `PMIX_PROC_DATA` and `PMIX_GROUP_CONTEXT_IDs`. Although probably not the intended use case, by just calling the function with the updated job data object for an existing namespace the stored job data is replaced. This, however, also triggers other actions not necessary for solely updating the job data and which might corresponds to some overhead. Minor changes were required to restrict the actions to only replacing the job data. Moreover, the semantic of the `nlocal` parameter of the `PMIx_register_nspace` function slightly changes when using it for updating job data for resource subtraction. Originally, this parameter represents the number of local processes in the namespace to be registered. In case of a resource subtraction, however, this parameter needs to correspond to the current number of local processes to maintain the internal accounting. We circumvent this by usage of a special value, `INT_MIN`, indicating preservation of the internal accounting for the local processes.

7.4.3. PRRTE

To realize the proposed design on the runtime layer, coordination between the PRRTE master and the PRRTE daemons is required. This is mostly achieved by sending commands via the RML. Accordingly we implemented the following new commands and corresponding callback functions:

- `PRTE_DYNRES_DEFINE_PSET`: *Commands the definition of a new process set*. The command is sent by the master to all daemons. The message contains the name of the process set and its members. The callback function defines the process set on the local PMIx server via `PMIx_server_register_nspace` and updates the local query data accordingly.
- `PRTE_DYNRES_DEFINE_RES_CHANGE`: *Commands the definition of a new resource change*. The command is sent by the master to all daemons. The message contains the type of the resource change and the name of the *delta process set*. The callback function updates the local query data accordingly.
- `PRTE_DYNRES_SERVER_PSETOP`: *Commands a process set operation on behave of a local client*. The command is sent by a daemon to the master. The message contains the type of the process set operation, the names of the process sets invloved in the operation and the client's preferred name for the resulting process set. Moreover, the room number for the `prte_hotel_t` (a data storage structure) is included, so when a corresponding answer arrives it can be associated with the server's provided `pmix_psetop_cbfunct_t` callback function and data. The receiver's callback function (here the master) executes the specified process set operation and sends a corresponding answer.
- `PRTE_DYNRES_CLIENT_PSETOP`: *Commands the release of a client that requested a process set operation*. The command is sent by the master to the particular daemon who requested a process set operation. The message contains all the data received in the `PRTE_DYNRES_SERVER_PSETOP` command message, but with the clients preferred name replaced by the actually used name for the result process set. Moreover, the members of the new process set are included. The receiver's callback function executes the server's provided `pmix_psetop_cbfunct_t` function and updates the query data accordingly.
- `PRTE_DYNRES_UNPUBLISH_RES_CHANGE`: *Commands that a particular resource change becomes unqueryable*. The command is sent by the master to all daemons. The message contains the name of the *delta process set*. The receiver's callback function updates the query data accordingly.

- `PRTE_DYNRES_FINALIZE_RES_CHANGE`: *Commands that a resource change is finalized*. The command is sent by the master to all daemons. The message contains the name of the *delta process set*. The receiver's callback removes all data related to the resource change. In case of a resource subtraction, the local `prte_job_t` object is updated correspondingly.
- `PRTE_DAEMON_DVM_SUB_PROCS`: *Commands the update of the PMIx server job data in case of a resource subtraction*: This command is sent by the master to all daemons. The message contains a launch message including a serialized, updated job data object and further information describing the job after the resource change. The receiver's callback function deserializes the job data, includes additional local information and updates the PMIx job data object accordingly via `PMIx_register_nspace`. (This command is an adjusted version of the existing `PRTE_DAEMON_DVM_ADD_LOCAL_PROCS` command and procedure, which we use for resource addition.)

In the following subsections we illustrate the usage of these commands for the three phases of resource changes.

Phase 1: Initialization of Resource Changes

The initialization of resource changes is managed by the PRRTE master, which is responsible for performing the steps described in Sec. 6.2.2.

For this, it first creates an updated PRRTE job data object that represents the job after the resource change. We use the stack approach (Fig. 7.1) to determine which nodes/processes are removed or added respectively. For new processes we use ranks based on a monotonically increasing counter, thus, ranks are not recycled. During this step the members of the *delta process set* are determined and stored.

The created job data object is given as input to a function provided by the *Open-RTE daemon local launch subsystem* (odls) framework module, which creates a buffer containing the specific launch data required by the active odls component.

In case of resource subtraction, this buffer is sent to the daemons with the `PRTE_DAEMON_DVM_SUB_PROCS` command. The daemons use the provided info in the buffer to create a corresponding *PMIx job data* object and to replace the PMIx job data on the PMIx server using the `PMIx_register_nspace` function.

Next, the `PRTE_DYNRES_DEFINE_PSET` command is used to define the *delta process set* containing processes determined during construction of the PRRTE job data object.

Subsequently, the `PRTE_DYNRES_DEFINE_RES_CHANGE` command is sent to make the resource change queryable for clients.

Finally, in case of a resource addition, the buffer containing the launch data is sent to the daemons with the `PRTE_DAEMON_DVM_ADD_LOCAL_PROCS` command, which already existed in the original code. Only minor changes were required. The receiving daemons update their local PRRTE job data object based on the received information. Similarly, they create a *PMIx job data* object and register it via `PMIx_register_nspace`. Finally, in the case that new processes need to be started on the local node, the active launch agent is commanded to launch the specified processes.

Phase 2: Process Set Operations

In the second phase of resource changes the RTE needs to service process set operation requests of clients. Fig. 7.3 illustrates the necessary sequence of actions taken by the RTE.

An arbitrary daemon receives a process set operation request from a local client via the `pmix_server_pset_operation_fn_t` function pointer registered at the local PMIx server. As process set operations are executed centrally by the process hosting the *pset operation module* (here the PRRTE master) a corresponding request needs to be sent using the `PRTE_DYNRES_SERVER_PSETOP` command. Note, that the daemon does not yet call the PMIx server's callback as it needs to wait for a response from the master. Instead, the necessary data such as the callback function pointer is stored in a special data structure. The index where the data is stored in the data structure is included in the message to the master.

Upon receiving the request, the PRRTE master executes the process set operation. For our prototype we used a naive implementation for the process set operation module based on the comparison of the process identifiers, which are explicitly stored for each process set. Subsequently, the master distributes the result to all daemons. The requesting daemon receives an answer with the `PRTE_DYNRES_CLIENT_PSETOP` command, including the result as well as the index into the data structure, where the required data for the callback to the PMIx server is stored. The daemon executes the callback, which finally releases the PMIx client / MPI process which requested the operation. The remaining daemons just receive the default `PRTE_DYNRES_DEFINE_PSET` command, to define the process set locally.

Phase 3: Resource Change Finalization

In the third phase, the PRRTE master reacts to PMIx event notifications of the `PMIX_RES_CHANGE_FINALIZED` event.

In case of a resource subtraction it sends the `PRTE_DYNRES_UNPUBLISH_RES_CHANGE` command to all daemons which makes the resource change unavailable for queries.

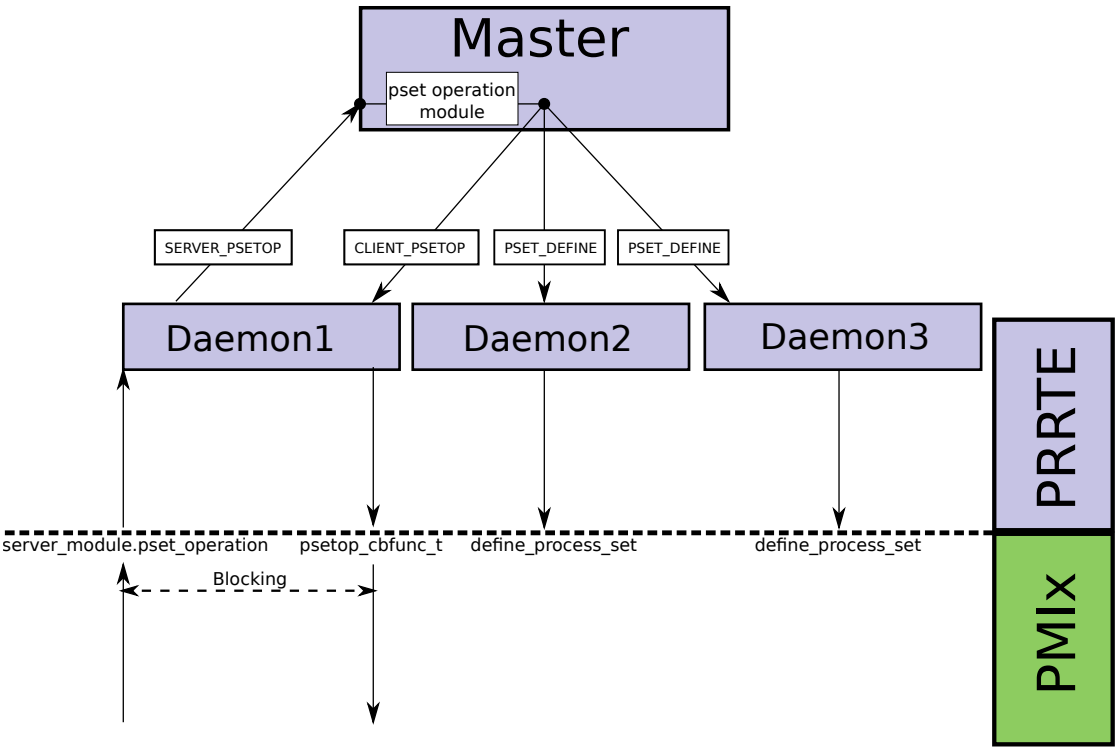


Figure 7.3.: **Illustration of the actions taken by the PRRTE layer to serve a process set operation request.** The sequence starts with the PMIx server calling the `pset_operation` function provided by the RTE (most left arrows in the figure) to serve a PMIx client's request for a process operation. The daemon operating the corresponding PMIx server delegates the request to the process hosting the process set operation module (master). After executing the operation, the master sends corresponding commands to the daemons. The daemon who delegated the request calls the PMIx server's callback function, which releases the requesting PMIx client.

However, this does not remove the local internal data about the resource change stored by each daemon. This is only relevant for resource subtractions, because this data is needed later when all processes of the resource change have terminated and the PRRTE job data object is updated. Thus, in case of a resource addition, also the `PRTE_DYNRES_FINALIZE_RES_CHANGE` command is sent to all demons, which removes the internal local data about the resource change.

To track if processes which are meant to be subtracted by a resource change have actually terminated, we adjusted the existing PRRTE functionality for process state

monitoring. In the original code, a daemon would only inform the master when all local process have terminated. We changed this such that every local process termination is reported to the master. If the terminated process is a member of the *delta process set* for a resource subtraction, a counter for this resource change is incremented. When the counter reaches the number of processes in the *delta process set* the `PRTE_DYNRES_FINALIZE_RES_CHANGE` command is sent to all daemons. The daemons then update the PRRTE job data object accordingly and remove the local data for this resource change.

7.4.4. External Initiation of Resource Changes

The proposed design includes a flexible docking point for the initiation of resource changes via the PMIx event notification system using the `PMIX_RES_CHANGE_DEFINE` event. This could for instance be used by the batch system to request a resource change action from the RTE.

Accordingly, in our prototype we implemented the steps of phase 1 described in the last section as callback function which is registered for the `PMIX_RES_CHANGE_DEFINE` event by the PRRTE master. As our prototype is not yet integrated into a dynamic RM, resource adaption of the user job is limited to the allocation initially granted by the RM.

We use two alternative approaches to control such resource resources changes which are particularly useful for debugging and testing.

First, we wrote a (PMIx-)tool which allows sending corresponding notifications based on command line input. The tool connects to the PMIx server on the head node of the allocation where the PRRTE master resides. Thus, resource addition and subtraction can be triggered in real time using the command line. We used this approach mainly during development and debugging.

The second approach is more suited for the tests we describe in the next section. Here the resource changes are triggered by the application. We wrote a simple MPI wrapper function (`MPI_Session_request_res_change`) which sends an according PMIx notification to the PRRTE master. Thus, resource addition and subtraction can be triggered by the application code.

8. Prototype Evaluation

8.1. Test System and Setup

Our tests were performed on the Linux Cluster CoolMuc2 segment of the *Leipniz Supercomputing Centre* [25]. CoolMuc2 features 28-way Haswell-based nodes with 64 GB DDR4 memory per node. The nodes are connected via FDR14 Infiniband interconnect providing a bandwidth of 13.64 GB/s. The cluster's resources are scheduled and managed by an instance of the SLURM workload manager.

For our tests we allocated a SLURM job allocation consisting of four nodes using the `salloc` command. Similar to the example in Fig. 7.1 we use PRRTE to create a *daemon job* (DVM) spanning these four nodes and a total of 112 cores. In our benchmarks the size of the *user job* is grown and shrunk inside of this DVM.

8.1.1. Synthetic Benchmark

For the synthetic benchmark we use an application structure similar to Lst. 6.7. The only difference is that in our benchmark the resource changes are first requested by the application at start of the resource change step.

For the synthetic benchmark we aimed to exclude any influences of data redistribution such that the performance of the work step is purely determined by the computational work. However, we account for typical synchronization requirements using `MPI_Barriers` before and after the work phase.

To perform a sufficient amount of computational work in each phase, without relying on any data-read, -write or -distribution operations, we used the `rebalance` and `work step` functions shown in Lst. 8.1.

We use a total `PROBLEM_SIZE` of 10^{10} with a workload of ≈ 10 FLOPs per element. Thus, the calculation requires a total of ≈ 100 GFLOPs per iteration. The `PROBLEM_SIZE` is equally distributed across the available processes. Due to this synthetic workload we expect the performance to scale linearly with the number of available processes, such that the effects and overheads of resource changes should become especially apparent.

Although this benchmark is mainly based on a synthetic workload, we like to briefly note that it could be interpreted physically as a naive collision detection of an object moving on a parabolic trajectory and a particular bounding volume. To this end, the

PROBLEM_SIZE divides a total time interval into discrete sampling points and each process works on the sampling points of its particular sub interval. The position at each sampling point is given by the formula

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} v_0 t \cos\beta + x_0 \\ v_0 t \sin\beta - \frac{g}{2}t^2 + y_0 \end{pmatrix}.$$

The calculated position is compared against the extends of a bounding box specified by x_{min}, x_{max}, y_{min} and y_{max}. The final result is the time the moving object is located inside the bounding volume during the particular time interval.

```
void rebalance(){
  PROBLEM_SIZE = 10000000000;   // 10 Mrd
  size  = PROBLEM_SIZE / num_procs;
  partition_start = size * myrank;
  partition_end = (my_rank + 1 == num_procs) ? PROBLEM_SIZE : (my_rank +1) * size;
}

int work_step(){
  double a, b, c, d, e; // initialized to some arbitrary values
  double x, y, t;
  int res=0, i;

  for(i = partition_start, i = partition_end, i++){

    t = (double) i;
    x =  a*t + b;
    y = c*t*t + d*t + e;

    if(x > x_min && x < x_max && y > y_min && y < y_max){
      res ++;
    }
  }
  return res;
}
```

Listing 8.1: **Work and rebalance phases used in the synthetic benchmark.**

8.1.2. Resource-Adaptive PDE Solver for a Hyperbolic Problem

To demonstrate the usability of our approach in a more realistic use case, we test it for solving the shallow water equations (SWE).

For this, we build on other work [40], which provides a resource adaptive extension of the *p4est library* [2]. *P4est* facilitates parallel dynamic mesh refinement through dynamic management of a collection of quad-/octrees (a *p4est*). In [40], the *p4est library* was extended to support dynamic changes of resources and it is used for a simulation of the SWE based on the SWE Teaching Code [36] and the SWE solvers [37] from the

Chair of Scientific Computing at the Technical University of Munich. This support for dynamic resource changes was demonstrated using the emulation layer for dynamic MPI Sessions [12].

For our tests we replaced the emulation layer with our prototype implementation, which required some changes to the extensions of the *p4est library*, such as disconnection from MPI communicators. We use one of the example benchmarks from [40], which solves the SWE to simulate the scenario of a radial dam break. The simulation is performed on a fixed, uniform grid consisting of 4^{10} cells with load balancing and without adaptive mesh refinement. Throughout the following we refer to this benchmark as the *SWE benchmark*

The central functions executed in each iteration of the main loop of the *SWE benchmark* are shown in Lst. 8.2.

```
while(current_phase < num_phases){

  /* apply resource change if available */
  changed = resource_change();

  /* update p4est communicator according to resource change */
  if(changed)
    update_p4est_communicator();

  /* Equally partition the p4est between all available processes */
  p4est_partition();

  /* Simulate the interval of the current phase */
  simulate_interval();
}
```

Listing 8.2: **Main loop of the SWE benchmark.**

8.2. Benchmark Parametrization

We use mostly similar parameterization for both, the *synthetic benchmark* as well as the *SWE benchmark*, which is described in this sections.

Our benchmark is set up to measure the overhead of the resource change procedure as well as the impact of dynamically varying resources on application performance. Both benchmarks are loop-base applications and make use of application-level resource changes. The resource changes are initiated by the application using the approach described in Sec. 7.4.4 which we include into the measurements for the overhead of resource changes.

To test several different scenarios of resource changes our benchmarks are parameterized by the parameters listed in Tab. 8.1.

Parameter	Possible values	Description
mode	$\{inc, step\}$	Mode of resources changes
n_{start}	$\mathbb{N}_{>2}$	The initial number of processes
n_{limit}	$\mathbb{N}_{>0}$	An upper or lower limit of processes
n_{delta}	$\mathbb{N}_{>0}$	Influences the number of processes added/removed
f	$\mathbb{N}_{>0}$	Frequency of res. changes w.r.t. the main loop iterations
b	{0,1}	Flag indicating the usage of non-blocking/blocking accept
i	$\mathbb{N}_{>0}$	The number of iterations the test should run
t	$\mathbb{R}_{>0}$	Total time to be simulated
c	$\mathbb{N}_{>0}$	Number of phases (main loop iterations)

Table 8.1.: **Parameterization of the benchmarks.** Parameter i is exclusive to the synthetic benchmark. Parameters t and c are exclusive to the SWE benchmark.

The parameter b determines if phase 3 of the resource changes is performed blocking or non-blocking, thus, whether the `MPI_Session_accept_res_change` will block until the confirmation is published by dynamically added processes.

The benchmark is started with n_{start} processes and an upper/lower limit of processes can be set via n_{limit}. If $n_{limit} > n_{start}$ resources are added and vice versa.

We use two different modes for the benchmark. In *incremental (inc)* mode, in each resource change n_{delta} processes are added/removed towards the n_{limit}. In *step* mode, each step consists of two resource changes, where the first resource change is reversed by the second resource change. Thus, after each step (after every second resource change) the number of processes is again n_{start}. The number of processes added/removed in the i-th step is $i * n_{delta}$, thus growing in each step. This allows us to systematically evaluate the influence of different sizes of resource changes.

The total number of iterations of the outer loop is determined by parameter i in the synthetic benchmark and by parameter c in the SWE benchmark. After each iteration of the outer loop, a resource change could potentially be executed. By specifying $f > 1$, this frequency can be changed to perform several iterations with constant resources between resource changes.

Finally, for the SWE benchmark the total time to be simulated can be set using the t parameter, which influences the workload in each phase.

We measure the time needed for the individual steps of the resource changes as well as the time needed for computational work, re-balance, and communicator re-initialization.

8.3. Results of the Synthetic Benchmark

In this section we present the performance results of the synthetic benchmark. First, we discuss the overhead of the resource changes in the RTE layer. In Sec. 8.3.2 the influence of resource changes on the performance of the synthetic benchmark is evaluated and further details regarding the overhead of individual phases of resource changes are provided in Sec. 8.3.3.

8.3.1. Performance of Resource Changes in the RTE Layer

n_{delta}	28	56	84
total time of res. change (b/nb)	1445/1670 ms	1642/1692 ms	1606/1690 ms
Time until all processes started	11.56 ms	16.89 ms	18.49 ms
MPI_Session_init	1083 - 1209 ms	1172 - 1417 ms	1170 - 1375 ms

Table 8.2.: **Performance for resource addition in the runtime layer.**

n_{delta}	28	56	84
total time of res. change	88.27 ms	108.80 ms	125.86 ms
Time until published	7.29 ms	11.13 ms	14.85 ms

Table 8.3.: **Performance for resource subtraction in the runtime layer.**

This section provides an evaluation of the performance of resource changes and process management from the perspective of the RTE.

Table 8.2 presents the timings for resource addition for $n_{delta} \in \{28, 56, 84\}$. The total time required for the execution of the resource change is between 1445 ms ($n_{delta} = 28$, blocking) and 1692 ms ($n_{delta} = 56$, non-blocking). The non-blocking approach consistently requires slightly larger amounts of time. This is an expected result, as the application only checks for confirmation of the newly spawned processes after each iteration of the main loop. The time from reception of the resource change request until all processes are started is comparably small, increasing from 11.56 ms for 28 processes to 18.49 ms for 84 processes. The subsequent MPI initialization of the new processes (`MPI_Sessions_init`) requires more than one second, constituting $\approx 85 - 90\%$ of the total time required for resource addition. However, by using the non blocking approach, applications can continue their execution until the new processes are ready to establish communication.

Resource subtraction is considerably faster than resource addition as no new MPI processes need to be initialized. The required time from reception of the resource change

request until termination of all processes increases from 88.27 ms for 28 processes to 125.86 ms for 84 processes. Considering the time until the resource change is queryable by the application, $\approx 90\%$ of the total required time for resource subtraction can be attributed to the adaption process of the application.

8.3.2. Impact of Resource Changes on Application Performance

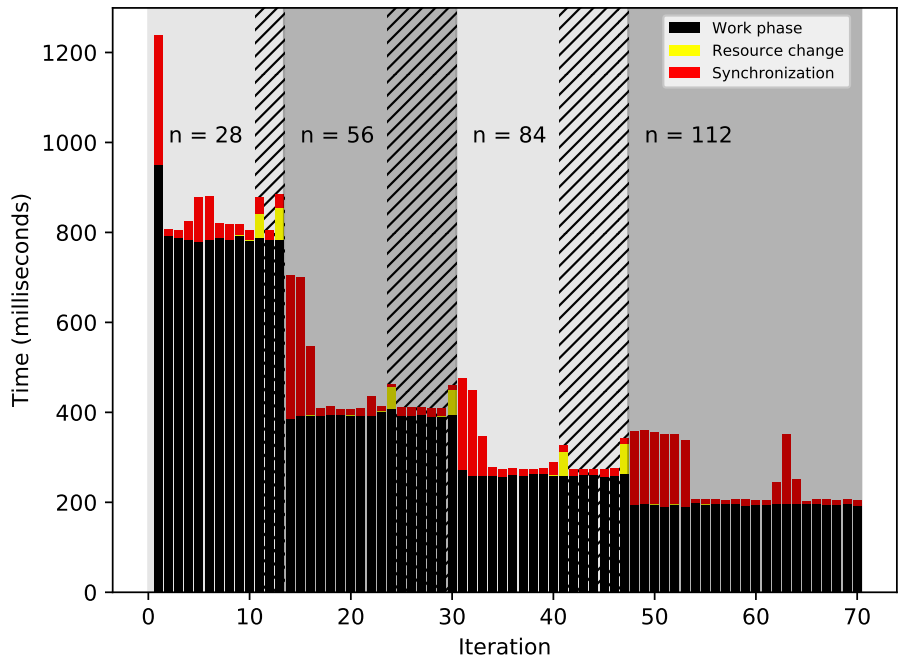
To study the impact of resource changes on the application performance we evaluate the results for the synthetic benchmark in *incremental* mode for resource addition/subtraction. We used the parameters $n_{start} = 28/112$, $n_{limit} = 112/28$, $n_{delta} = 28$, $f = 10$, $i = 70$ and $b = 0$.

Fig. 8.1a shows the results for resource addition. We used the non-blocking approach to avoid blocking during MPI initialization of the new processes, leading to the resource changes pending during several iterations indicated by the hatched areas in the figure.

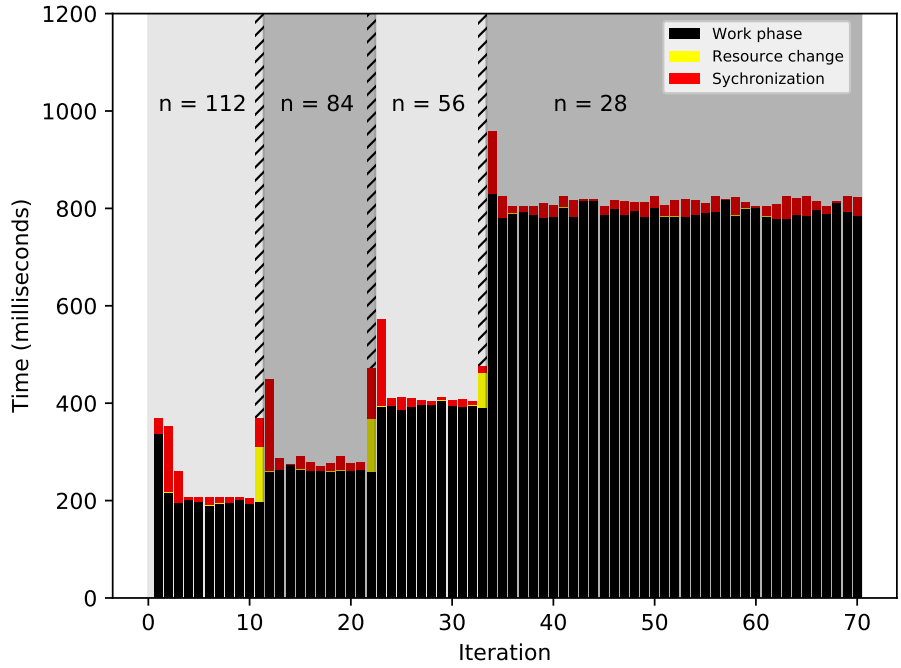
For the work phase it can be seen that the performance increases with the number of processes, with an almost optimal parallel speedup.

An overhead of resource changes is mainly measured in the first and last iteration of the hatched areas, i.e. at the beginning and end of a resource change. During the first iteration in the hatch areas, phase 1 and phase 2 of the resource change are performed, the new process set name is published and the confirmation of the new processes is looked up. However, in this iteration the new processes have not yet confirmed their readiness to establish communication, leading to a pending status of the resource change. The overhead in this iteration is between 48.5 ms and 53.8 ms. While the resource change is pending, the overhead for the lookup of the status of the new processes is comparably small with 0.3 - 0.4 ms per iteration. During the last iteration in the hatched areas, phase 3 of the resource change is performed and a new communicator is created. Here, the overhead is between 55.0 ms and 70.5 ms.

We also measured the total time required for the synchronization at the `MPI_Barriers` before and after the work phase. During the first iterations with new resource assignment the synchronization overhead is significantly larger than in the following iterations. We account for the overhead in the first iteration after a resource change by the connection "warm-up" used by PMIx at first use of a communicator. This setup of point-to-point connections depends on the size of the communicator and thus changes after each resource change. The same phenomen is therefore also experienced with resource subtraction. In case of resource addition the synchronization overhead is still enlarged for several subsequent iterations. As this is not present with resource subtraction we attribute this to frequency scaling on the newly added nodes.



(a) Resource Addition



(b) Resource subtraction

Figure 8.1.: **Application performance and overhead of resource changes in the synthetic benchmark.** The hatched areas are the time period with a pending resource change.

Analogously, the results for resource subtraction are shown in Fig. 8.1b. Here, the resource change is performed in a single iteration. As expected, the application performance decreases linearly with the numbers of processes. The overall overhead for the resource changes is 112.78 ms, 108.7 ms and 71.4 ms respectively, thus decreasing with the number of remaining processes.

8.3.3. Performance of Individual Phases of Resource Changes

In this section we study in more detail the sources of the measured overheads for resource addition and subtraction. For this we used the *step* mode of the benchmarks with the parameters $n_{start} \in \{28, 56, 84\}$, $n_{limit} = 112$, $n_{delta} = 28$, $f = 10$, $i = 200$ and $b = 0$.

Fig. 8.2 shows the performance measurements for the different phases of resource addition. The overhead of phase 1 (Fig. 8.2a) is between 5 ms and 11 ms, accounting only for a comparably small portion of the overall overhead of resource addition. The overhead of phase 1 increases with increasing n_{start} and n_{delta} and includes the resource change request as well as the time until the resource change is queried. For malleable jobs, only the query for resource changes is required, probably leading to an even smaller overhead.

During phase 2, the process set operation is performed, for which we measured the smallest overhead (Fig. 8.2a), ranging from 0.8 ms to 1.5 ms. The overhead slightly increases with increasing sizes of the involved process sets. However, it has to be noted that we expect this overhead to increase more significantly on larger scales as our implementation is based on a naive approach. This approach performs set operations in $O(m \times n)$, where m, n are the sizes of the process sets. Moreover, the 260 byte PMIx process names of the processes included in a process set are stored explicitly for each process set, leading to significant space overhead for larger scales. A better, alternative solution would be to store all process names of the job only once and use referencing bit arrays to denote the membership of process sets. This would be more space efficient and would allow for more efficient implementations of set operations.

Phase 3 produces the largest portion of the overhead of resource addition. The performance is quite similar for different values of n_{start}, but increases from 73 ms to 135 ms when increasing n_{delta} from 28 to 84 processes. We mainly attribute this overhead to the process of discovering the endpoint information of the new processes.

The last action during a resource change is the creation of a new communicator. This process requires between 29 ms and 35 ms and seems to be mostly independent from the number of involved processes.

Fig. 8.3 shows analogous performance measurements for resource subtraction, with

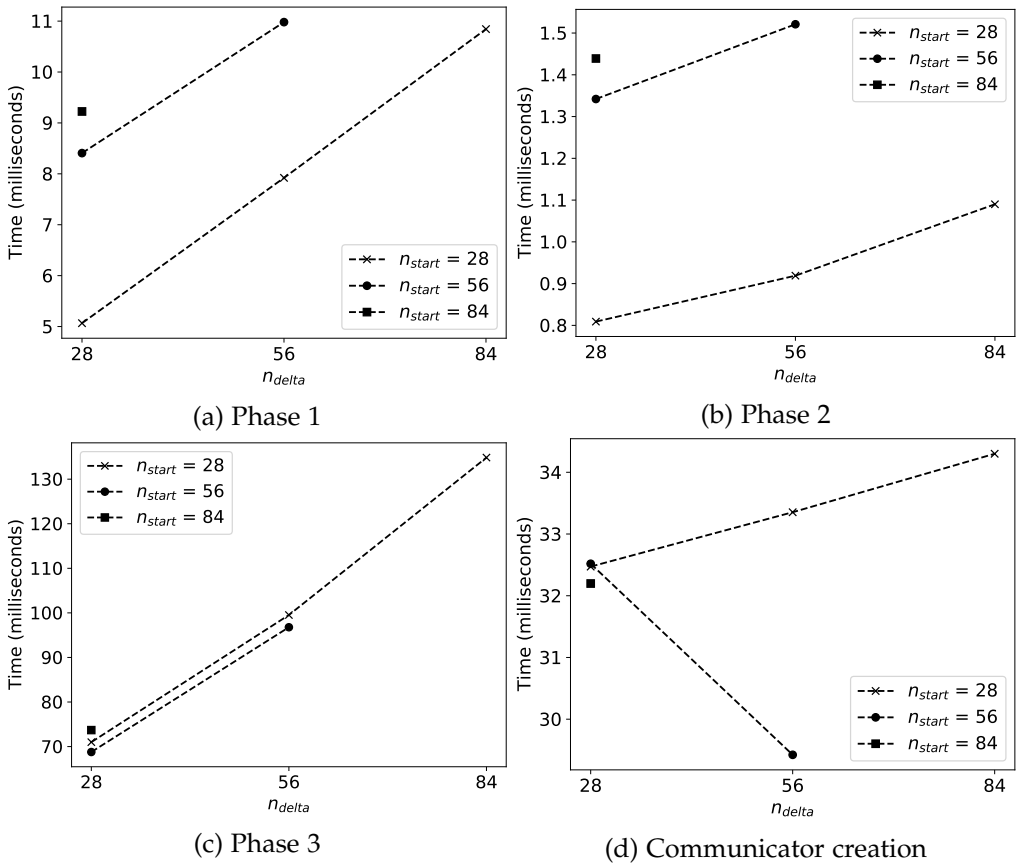


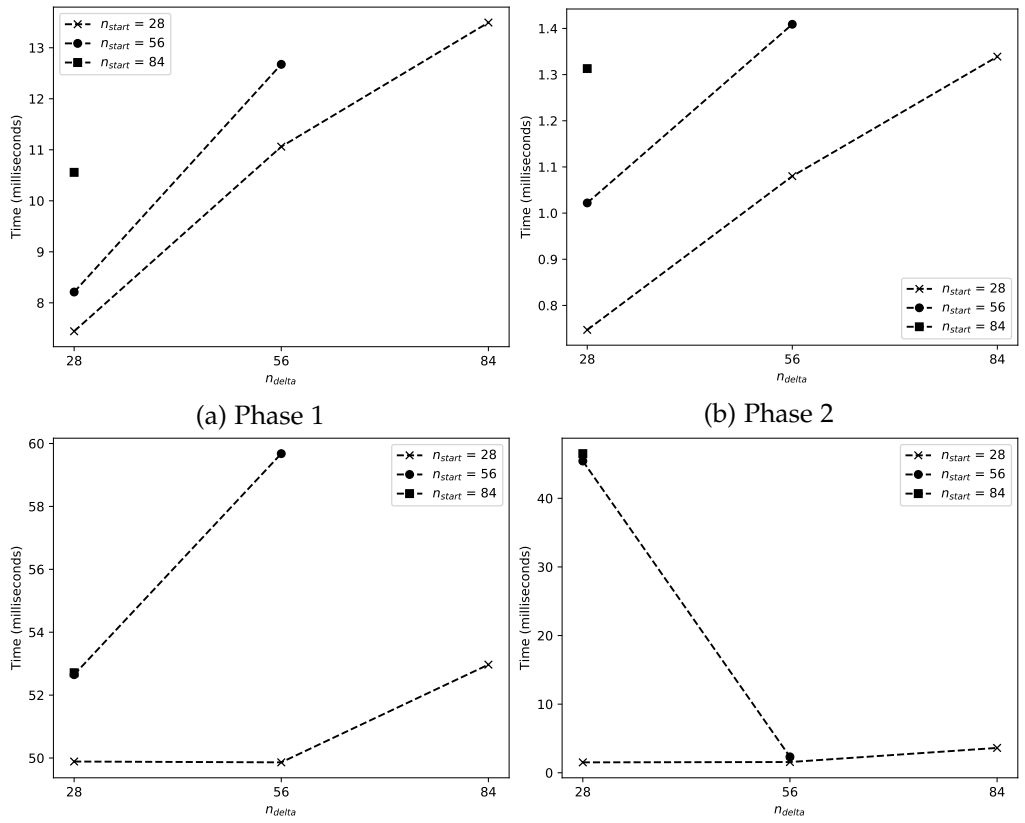
Figure 8.2.: **Performance of the individual resource change phases for resource addition.** n_{start} is the number of processes before the resource change and n_{delta} is the number of added processes.

n_{dest} denoting the number of processes after the resource change.

Phase 1 and phase 2 of the resource subtraction overall exhibit similar performance characteristics as for resource addition.

The overhead of phase 3 of resource subtraction (Fig. 8.3c) ranges from 50 - 60 ms. This overhead is significantly smaller compared to resource addition especially for larger n_{delta} values. For resource subtraction, phase 3 seems to be insensitive w.r.t. to n_{delta}, presumably because no new endpoint information needs to be discovered.

The measurements for the overhead related to communicator creation show unexpected results. Reducing the number of processes by 28 to 84 and 56 processes respectively results in overheads for communicator creation of more than 40 ms. For other combinations of n_{delta} and n_{dest} the overhead for the same operation is less than



(a) Phase 1

(b) Phase 2

(c) Phase 3

(d) Communicator creation

Figure 8.3.: **Performance of the individual resource change phases for resource subtraction.** n_{dest} is the number of processes after the resource change and n_{delta} is the number of removed processes.

5 ms. We attribute this to the step mode we used for the measurements. As the number of processes is n_{start} after each step, the connections are already "warmed up", which speeds up the creation of the communicator.

8.4. Results of the SWE benchmark

Here we present the results of the SWE benchmark.

As mentioned in Sec. 7.2, due to implementational issues, we only provide results for resource addition in the SWE benchmark.

Fig. 8.4 shows the results for parameters $t = 10$, $c = 10$, $n_{start} = 28$, $n_{delta} = 28$, $n_{limit} = 122$, $f = 1$ and $b = 0$. Due to the asynchronous approach, the resource

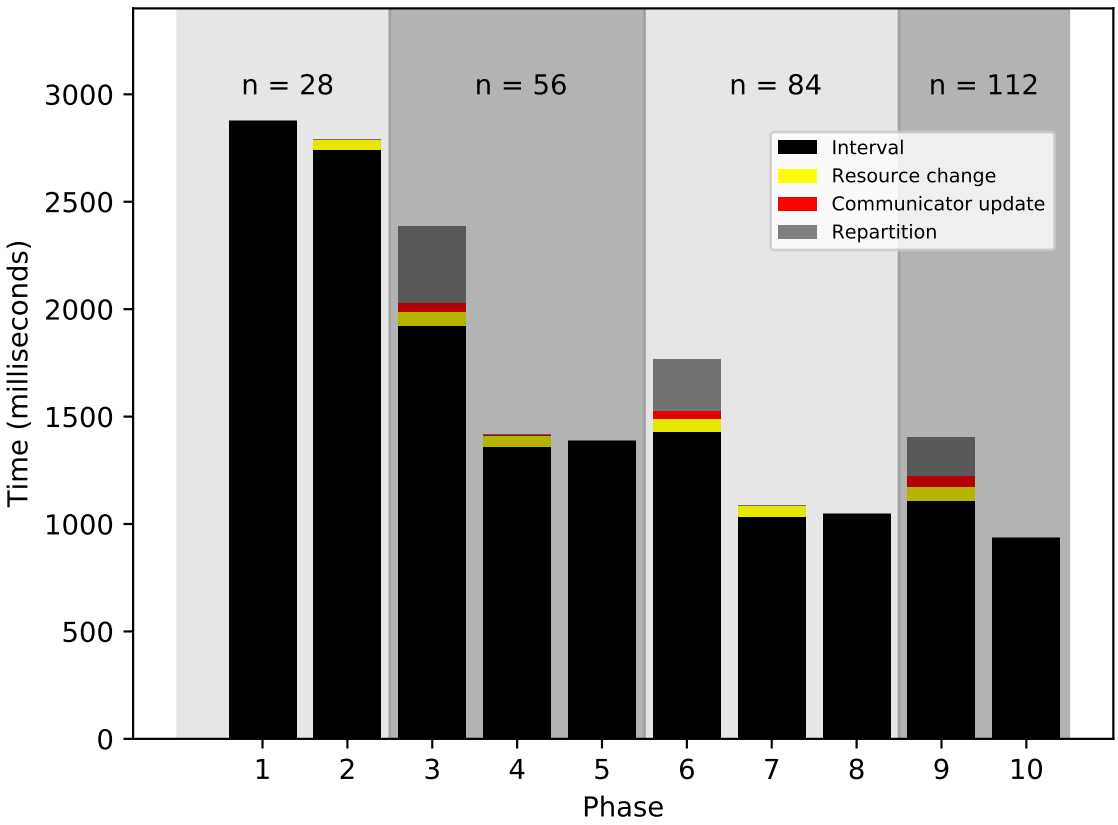


Figure 8.4.: Performance of the SWE benchmark simulating $t = 10$ seconds using 10 phases. The application starts on one node (28 processes) and additional nodes are added in phase 3, 6 and 9.

changes span multiple phases. The resource changes are finalized in phase 3, 6 and 9 respectively.

Similar to the synthetic benchmark, the overhead of the resource changes is in the range of 100 - 125 ms. An additional significant overhead of 385, 240 and 181 ms respectively is related to the re-partitioning of the *p4est*. Although the resource change and re-partitioning happens before the interval simulation in each phase, the performance improvements related to the additional processes becomes apparent only in subsequent phases. We account for this by the connection "warm-up" of the new communicator and the frequency scaling on the newly added nodes as we already dicussed in the context of the synthetic benchmark.

The time required for one phase decreases through the resource additions by $G_{rc1} = 50.37$ %, $G_{rc2} = 25.33$ % and $G_{rc3} = 10.68$ % respectively. The overhead of the resource

change and re-partitioning w.r.t. the time required for one phase before the resource change is $O_{rc1} = 18.68$ %, $O_{rc1} = 28.5$ % and $O_{rc1} = 33.19$ % respectively.

An interesting metric for application in practice is the number of iterations required to compensate the overhead, i.e. when the application starts to profit from the additional resources. The overhead of the resource change rc can be amortized over the n_{rc} subsequent, accelerated phases. We can therefore calculate the minimum number of phases after which the resource changes provide net performance improvement as

$$G_{rc} > \frac{O_{rc}}{n_{rc}} \rightarrow n_{rc} > \frac{O_{rc}}{G_{rc}}$$

For our measured results, we derive $n_{rc1} > 0.37$, $n_{rc2} > 1.13$ and $n_{rc3} > 3.11$. Thus, for the three resource changes in this benchmark, net performance gains are achieved after 1, 2 and 4 subsequent phases respectively.

9. Discussion and Future Work

Motivated by the challenge of improving resource-efficiency on large-scale HPC systems, this thesis deals with dynamic resource management in such systems. The main goal of this work was to explore the potential of the MPI Sessions model and PMIx for designing mechanisms for resource adaptive MPI applications and RTEs, which are a prerequisite for dynamic resource management. For this, a possible design of such a mechanism was proposed and its usability demonstrated with a prototype implementation.

The development of this approach has revealed a major advantage of the MPI Sessions model compared to the MPI World model being the integration of runtime information through the usage of *process sets*. Based on process sets, changes of resources can be realized in a more fine-grained and scalable manner, as there does not necessarily exist a global communicator such as `MPI_COMM_WORLD`. Moreover, they provide a common concept for MPI applications and RTEs to exchange information about available resources, resource requirements and communication patterns. These advantages introduce the necessary flexibility to develop resource dynamic MPI applications.

In this context, PMIx provides a concise and flexible interface to realize the interaction between MPI applications and RTEs necessary for adapting to changing resource availability. As PMIx can further be used for realizing interaction with other components in the SMS, using it as basis for our approach facilitates the management of resource dynamic MPI jobs on HPC systems.

Our implementation of a prototype based on an extension of Open-MPI has confirmed the practicability of the basic mechanisms of the proposed design to change the resources of MPI applications dynamically. However, so far only application level resource changes are supported by the prototype. For more comprehensive conclusions on the usability of this approach, the prototype implementation needs to be extended to provide the full set of functionality proposed by our design. In particular, this concerns the proposed extensions of the process set concept, whose practicability is not yet validated experimentally.

In our synthetic performance tests on up to four nodes and a total of 122 processes we measured overheads of 108 - 175 ms for resource addition and 88 - 126 ms for resource subtraction. While this is acceptable for a prototype, our performance analysis reveals the positives and negatives of our design/implementation. The performance analysis proved the benefits of using a non-blocking approach in phase 3 during resource

addition. This approach avoids blocking during MPI initialization phase of dynamically added processes, for which we measured overheads of more than 1 second. However, the results indicate that phase 3 is still responsible for the largest part of the overhead associated with resource changes for both, resource addition and subtraction. We suppose that there is still great potential for reducing this overhead by requiring less synchronization and optimizing the collective data exchange during this phase.

Thus, one direction for future work could be the enhancement and extension of the prototype implementation. This should also include performance and scalability tests for more complex, large-scale scenarios to further evaluate the potential of the approach for the intended use case.

Moreover, we think our research on a design for dynamic resource management with MPI Sessions and PMIx could also motivate future work on its usage for related topics. In particular, further use cases of the provided resource dynamicity could be explored in the context of evolving jobs, fault-tolerance and intra-job load-balancing.

10. Summary

In this work, we developed a new concept based on the MPI Sessions model and evaluated its potential in the context of dynamic resource management on distributed compute systems.

We first discussed the importance of flexible, resource adaptive MPI applications and RTEs as a major building block of the software management stack on such systems. This discussion revealed the need for approaches providing an MPI interface general enough to support various types of parallel workloads as well as the requirement of flexible integration of the RTE into existing software stacks.

In this regard, we have shown several limitations of the MPI World model and introduced the MPI Sessions model as a possibly better fitting approach. Moreover, the PMIx standard and its potential to facilitate the implementation and integration of dynamic RTEs was introduced and discussed.

Based on this background, we proposed a concept, which extends the MPI and PMIx standards, and allows applications to adapt to varying resource availability through the usage of an explicit interface.

For this, we adopted the three main phases of resource changes proposed in other work, which enable applications to *query the RTE for resource changes*, prepare adapted communication patterns through *process set operations* and *consistently apply resource changes*. We extended the current process set concept and defined an MPI interface for the three phases. On top of this interface we used and extended the PMIx interface to facilitate the necessary interaction with dynamic MPI RTEs.

A prototype of the concept was developed based on extended implementations of Open-MPI, Open-PMIx and PRRTE. Preliminary tests were run on a modern HPC cluster using a synthetic benchmark and a PDE solver for a hyperbolic problem, demonstrating the practicability of our approach to add processes to or remove processes from an application at runtime. On up to four nodes and a total of 112 cores we measured overheads of 108 - 175 ms for resource addition and 88 - 126 ms for resource subtraction. These overheads are modest when considering that such resource changes typically occur in a frequency of several seconds. Thus, these results hint towards the profitability of our approach for dynamic resource management of MPI jobs on distributed compute systems.

In future work, this prototype could be developed further to support all features of

our proposed concept and to further increase its performance. Additional large-scale performance tests are required to evaluate the effectiveness of our approach for more realistic use cases.

Moreover, further integration of the dynamic RTE into the SMS on HPC systems is required, for which the PMIx standard provides interesting possibilities. As our approach is generic in terms of the entity deciding on the change of resources, its underlying mechanisms could possibly be used not only for malleable and evolving jobs but also for fault tolerance and intra-job load-balancing, constituting promising directions for future research.

Overall, this work has shown that the usage of both, the MPI Sessions model and the PMIx standard, provides great potential as a basis towards a holistic approach for dynamic resource management on large-scale distributed compute systems.

Bibliography

[1] G. Bosilca, T. Herault, P. Lemarinier, A. Rezmerita, and J. J. Dongarra. "Scalable Runtime for MPI: Efficiently Building the Communication Infrastructure." 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. 342–344.

[2] C. Burstedde, L. C. Wilcox, and O. Ghattas. "p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees." In: *SIAM Journal on Scientific Computing* 33.3 (2011), pp. 1103–1133. DOI: 10.1137/100791634.

[3] R. Castain, J. Hursey, A. Bouteiller, and D. Solt. "PMIx: Process Management for Exascale Environments." In: *Parallel Computing* 79 (Aug. 2018). DOI: 10.1016/j.parco.2018.08.002.

[4] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux. "Supporting Malleability in Parallel Architectures with Dynamic CPUSETsMapping and Dynamic MPI." In: *Distributed Computing and Networking*. Ed. by K. Kant, S. V. Pemmaraju, K. M. Sivalingam, and J. Wu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 242–257. ISBN: 978-3-642-11322-2.

[5] M. Chadha, J. John, and M. Gerndt. "Extending SLURM for Dynamic Resource-Aware Adaptive Batch Scheduling." In: *CoRR* abs/2009.08289 (2020). arXiv: 2009.08289.

[6] C. P. Chan, J. D. Bachan, J. P. Kenny, J. J. Wilke, V. E. Beckner, A. S. Almgren, and J. B. Bell. "Topology-Aware Performance Optimization and Modeling of Adaptive Mesh Refinement Codes for Exascale." In: *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. 2016, pp. 17–28. DOI: 10.1109/COMHPC.2016.008.

[7] 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*. EuroMPI 2016. Edinburgh, United Kingdom: Association for Computing Machinery, 2016, 82–97. ISBN: 9781450342346. DOI: 10.1145/2966884.2966917.

[8] I. A. Comprés Ureña. "Resource-Elasticity Support for Distributed Memory HPC Applications." PhD thesis. Technical University Munich, 2017.

[9] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. "Dynamic Malleability in Iterative MPI Applications." In: *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*. 2007, pp. 591–598. DOI: 10.1109/CCGRID.2007.45.

[10] K. El Maghraoui, B. K. Szymanski, and C. Varela. "An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments." In: *Parallel Processing and Applied Mathematics*. Ed. by R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 258–271.

[11] G. E. Fagg and J. J. Dongarra. "FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world." In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by J. Dongarra, P. Kacsuk, and N. Podhorszki. Berlin, Heidelberg: Springer Berlin Heidelberg, Nov. 2000, pp. 346–353. ISBN: 978-3-540-41010-2. DOI: 10.1007/3-540-45255-9_47.

[12] J. Fecht. "A Simulation Layer for Dynamically Varying Computing Resources in MPI." Technical University Munich, 2020.

[13] D. G. Feitelson and L. Rudolph. "Metrics and benchmarking for parallel job scheduling." In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. G. Feitelson and L. Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–24. ISBN: 978-3-540-68536-4.

[14] D. G. Feitelson and L. Rudolph. "Toward convergence in job schedulers for parallel supercomputers." In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. G. Feitelson and L. Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, p. 7.

[15] W. Gropp and E. Lusk. "Dynamic process management in an MPI setting." In: *Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing*. 1995, pp. 530–533. DOI: 10.1109/SPDP.1995.530729.

[16] 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)*. 2014, pp. 1–10. DOI: 10.1109/HiPC.2014.7116905.

[17] N. Hjelm, H. Pritchard, S. K. Gutiérrez, D. J. Holmes, R. Castain, and A. Skjellum. "MPI Sessions: Evaluation of an Implementation in Open MPI." In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019, pp. 1–11. DOI: 10.1109/CLUSTER.2019.8891002.

[18] D. Holmes. *[Mpi-forum] MPI Sessions - final (hopefully) changes (for MPI-4.0 anyway)*. visited on 2021-10-11. 2019. URL: https://lists.mpi-forum.org/pipermail/mpi-forum/2019-September/006997.html.

[19] C. Huang, O. Lawlor, and L. V. Kalé. "Adaptive MPI." In: *Languages and Compilers for Parallel Computing*. Ed. by L. Rauchwerger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 306–322. ISBN: 978-3-540-24644-2.

[20] D. Huber. *DynMPI Prototype (Active Development Repository)*. 2021. URL: https://github.com/HawkmoonEternal/dynmpi_prototype_dev.git.

[21] D. Huber. *DynMPI Prototype (version used for benchmarks)*. 2021. URL: https://github.com/HawkmoonEternal/dynmpi_prototype.git.

[22] J. Hungershöfer, A. Streit, and J.-M. Wierum. *Efficient resource management for malleable applications*. Paderborn Center for Parallel Computing, Jan. 2002.

[23] L. V. Kale and S. Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based on C++." In: *SIGPLAN Not.* 28.10 (Oct. 1993), 91–108. ISSN: 0362-1340. DOI: 10.1145/167962.165874.

[24] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana. "A Large-Scale Study of MPI Usage in Open-Source HPC Applications." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356176.

[25] Leibniz-Rechenzentrum. *LRZ Linux Cluster*. URL: https://doku.lrz.de/display/PUBLIC/Linux+Cluster.

[26] L. Marchal, B. Simon, O. Sinnen, and F. Vivien. "Malleable task-graph scheduling with a practical speed-up model." In: *IEEE Transactions on Parallel and Distributed Systems* 29.6 (June 2018), pp. 1357–1370. DOI: 10.1109/TPDS.2018.2793886.

[27] G. Martín, 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. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2015.04.003.

[28] N. Mejri, B. Dupont, and G. Da Costa. "Energy-aware scheduling of malleable HPC applications using a Particle Swarm optimised greedy algorithm." In: *Sustainable Computing : Informatics and Systems* 28 (Dec. 2020), p. 100447. DOI: 10.1016/j.suscom.2020.100447.

[29] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021.

[30] *MPI Sessions Working Group*. URL: https://github.com/mpiwg-sessions/sessions-issues/wiki.

[31] C. Márcia, M. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. Navaux. "Supporting MPI Malleable Applications upon the OAR Resource Manager." In: *Colibri: Colloque d'Informatique: Brésil / INRIA, Coopérations, Avancées et Défis*. Rio Grande do Sul, Brazil, June 2009.

[32] PMIx Administrative Steering Committee (ASC). *Process Management Interface for Exascale (PMIx) Standard Version 4.0*. Dec. 2020.

[33] S. Prabhakaran. "Dynamic Resource Management and Job Scheduling for High Performance Computing." PhD thesis. Technical University Darmstadt, 2016.

[34] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. Kalé. "A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications." In: *2015 IEEE International Parallel and Distributed Processing Symposium*. May 2015, pp. 429–438. DOI: 10.1109/IPDPS.2015.34.

[35] H. Pritchard. *Sessions pr #59*. 2021. URL: https://github.com/hpc/ompi/pull/59.

[36] *Shallow Water Equations Teaching Code*. 2020. URL: https://github.com/TUM-I5/SWE.

[37] *Solvers for the Shallow Water Equations*. 2020. URL: https://github.com/TUM-I5/swe_solvers.

[38] B. Song, C. Ernemann, and R. Yahyapour. "Parallel Computer Workload Modeling with Markov Chains." In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 47–62. ISBN: 978-3-540-31795-1.

[39] T. Sterling, M. Anderson, and M. Brodowicz. "A Survey: Runtime Software Systems for High Performance Computing." In: *Supercomputing Frontiers and Innovations* 4 (Mar. 2017), pp. 48–68. DOI: 10.14529/jsfi170103.

[40] M. Streubel. "Dynamic Resource Management Using MPI Sessions on p4est." Technical University Munich, 2020.

[41] 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)*. Mar. 2007. DOI: 10.1109/ICPP.2007.73.

[42] R. Sudarsan and C. J. Ribbens. "Scheduling resizable parallel applications." In: *2009 IEEE International Symposium on Parallel Distributed Processing*. 2009, pp. 1–10. DOI: 10.1109/IPDPS.2009.5161077.

[43] N. Sultana, M. Rüfenacht, A. Skjellum, P. Bangalore, I. Laguna, and K. Mohror. "Understanding the use of message passing interface in exascale proxy applications." In: *Concurrency and Computation: Practice and Experience* 33.14 (2021), e5901. DOI: https://doi.org/10.1002/cpe.5901.

[44] V. S. Sunderam. "PVM: A framework for parallel distributed computing." In: *Concurrency: Practice and Experience* 2.4 (1990), pp. 315–339. DOI: https://doi.org/10.1002/cpe.4330020404.

[45] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. "Invasive Computing: An Overview." In: *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Ed. by M. Hübner and J. Becker. New York, NY: Springer New York, 2011, pp. 241–268. ISBN: 978-1-4419-6460-1. DOI: 10.1007/978-1-4419-6460-1_11.

[46] I. A. C. Ureña, M. Riepen, M. Konow, and M. Gerndt. "Invasive MPI on Intel's Single-Chip Cloud Computer." In: ARCS'12. Munich, Germany: Springer-Verlag, 2012, 74–85. ISBN: 9783642282928. DOI: 10.1007/978-3-642-28293-5_7.

[47] G. Utrera, J. Corbalan, and J. Labarta. "Implementing malleability on MPI jobs." In: *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*. Jan. 2004, pp. 215–224. ISBN: 0-7695-2229-7. DOI: 10.1109/PACT.2004.1342555.

[48] S. Vadhiyar and J. Dongarra. "SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems." In: *Parallel Processing Letters* 13 (Apr. 2003). DOI: 10.1142/S0129626403001288.

[49] D. Walker. "An Introduction To Message Passing Paradigms." In: *Proceedings of the 1995 CERN School of Computing*. Ed. by C. E. Vandoni. CERN. Arles, France, 1996, pp. 165–184.

[50] A. B. Yoo, M. A. Jette, and M. Grondona. "SLURM: Simple Linux Utility for Resource Management." In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[51] S. Zrigui, R. Y. De Camargo, A. Legrand, and D. Trystram. "Improving the Performance of Batch Schedulers Using Online Job Runtime Classification." working paper or preprint. Oct. 2020.

A. Appendix

A.1. Specification of the PMIx_Pset_Op_request Function

```
pmix_status_t
PMIx_Pset_Op_request( pmix_psetop_directive_t directive,
                      const pmix_info_t info[],  size_t ninfo,
                      const pmix_info_t *results[],  size_t *nresults)

IN    directive
     pmix_psetop_directive_t specifying the requested pset operation
IN    data
     Array of info structures (array of handles)
IN    ninfo
     Number of elements in the info array (integer)
OUT    results
     Array of info structures (array of handles)
OUT    nresults
     Number of elements in the results array (integer)

Returns one of the following:
  PMIX_SUCCESS The request was successfully executed by the host system.
  a PMIx error constant indicating either an error in the input or that the request
       was refused

PMIx libraries are required to pass any provided attributes to the host
environment for processing. In addition, the following attributes are required to
be included in the passed info array:
  PMIX_PSETOP_P1  "pmix.psetop.p1" (char*)
     Name of the first pset of the operation
  PMIX_PSETOP_P2  "pmix.psetop.p2" (char*)
     Name of the second pset of the operation
Optional:
  PMIX_PSETOP_PREF_NAME  "pmix.psetop.prefname" (char*)
     Clients prefered name of the resulting process set.

Description:
Request a process set operation from the host system. Possible operations are
PMIX_PSETOP_UNION, PMIX_PSETOP_DIFFERENCE and PMIX_PSETOP_INTERSECTION. If
successful, the returned results for the request must include the name of the
created pset (PMIX_PSETOP_PRESULT) and its size (PMIX_PSET_SIZE).
```

Listing A.1: **PMIx function for requesting a process set operation.**

A.2. Specification of the pmix_server_pset_operation_fn_t Function

```
typedef pmix_status (*pmix_server_pset_operation_fn_t)(
                                const pmix_proc_t *proc
                                pmix_psetop_directive_t directive,
                                const pmix_info_t data[], size_t ndata,
                                pmix_psetop_cbfunc_t cfunc, void *cbdata);

IN   proc
    pmix_proc_t structure identifying the process requesting the process set operation (handle)
IN   directive
    pmix_psetop_directive_t specifying the requested pset operation
IN   data
    Array of info structures (array of handles)
IN   ninfo
    Number of elements in the data array (integer)
IN   cbfunc
    Callback function pmix_op_cbfunc_t (function reference)
IN  cbdata
    Data to be passed to the callback function (memory reference)

Returns one of the following:
  PMIX_SUCCESS, indicating that the request is being processed by the host
      environment - result will be returned in the provided cbfunc. Note, that the
       host must not invoke the callback function prior to returning from the API.
  PMIX_OPERATION_SUCCEEDED, indicating that the request was immediately processed and
      returned success - the cbfunc will not be called
  PMIX_ERR_NOT_SUPPORTED, indicating that the host environment does not support the
      request, even though the function entry was provided in the server module -
      the cbfunc will not be called
  a PMIx error constant, indicating either an error in the input or that the request
      was immediately processed and failed - the cbfunc will not be called

PMIx libraries are required to pass any provided attributes to the host
environment for processing. In addition, the following attributes are required to
be included in the passed info array:
  PMIX_PSETOP_P1  "pmix.psetop.p1" (char*)
    Name of the first pset of the operation
  PMIX_PSETOP_P2  "pmix.psetop.p2" (char*)
    Name of the second pset of the operation
Optional:
  PMIX_PSETOP_PREF_NAME  "pmix.psetop.prefname" (char*)
    Clients prefered name of the resulting process set.

Description:
Request a process set operation from the host system. The request will include
the namespace/rank of the process that is requesting the operarion, an array of
pmix_info_t describing the operation, and a callback function/data for the return.
```

Listing A.2: **PMIx server function pointer to request a process set operation from the host system.** The host system provides a corresponding implementation.

A.3. Specification of the pmix_psetop_cbfunct_t Function

```
typedef void psetop_cbfunc(pmix_status_t status,
                           pmix_psetop_directive_t directive,
                           pmix_info_t *info, size_t ninfo,
                           void *cbdata,
                           pmix_release_cbfunc_t release_fn, void *release_cbdata)

IN    proc
      pmix_proc_t structure identifying the process requesting the process set operation (handle)
IN    directive
      pmix_psetop_directive_t specifying the requested pset operation
IN    data
      Array of pmix_info_t returned by the operation(pointer)
IN    ninfo
      Number of elements in the info array (size_t)
IN    cbdata
      Callback data passed to original API call (memory reference)
IN    release_fn
      Function to be called when done with the info data (function pointer)
IN    release_cbdata
      Callback data to be passed to release_fn (memory reference)

PMIx libraries are required to cache the specified resulting process set.
The following attributes are required to be included in the passed info array:
  PMIX_PSETOP_PRESULT   "pmix.psetop.p1" (char*)
    Name of the resulting process set.
  PMIX_PSET_MEMBERS "pmix.pset.mems" (pmix_data_array_t*)
    The members of the resulting process set.

Description:
The status indicates if the requested process set operation was executed. An array
of pmix_info_t will contain information describing the resulting process set.
```

Listing A.3: **PMIx callback function to be called by the host system after applying the process set operation.**