DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

A Simulation Layer for Dynamically Varying Computing Resources in MPI

Jan Fecht

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

A Simulation Layer for Dynamically Varying Computing Resources in MPI

Eine Simulationsschicht für Dynamisch Variierende Rechenressourcen in MPI

Author:	Jan Fecht
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Dr. rer. nat. Martin Schreiber
Submission Date:	April 15, 2021

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

Munich, April 15, 2021 Jan Fecht

Acknowledgments

I want to thank Martin Schreiber for his great guidance and invaluable support during the last months. This thesis would not be possible without him.

Also, my gratitude goes out to the MPI Sessions Workgroup which I had the honor to present this work to and which provided me with great feedback.

I want to thank the Leibniz-Rechenzentrum for allowing me to evaluate my work on their Linux-Cluster.

Finally, I am grateful for my friends and my family and their ongoing support.

Abstract

During the last two decades, the degree of parallelization in supercomputers has been increasing with a fast pace. In November 2000, not a single supercomputer contained more than ten thousand cores[14]. Twenty years later, the most powerful supercomputer in terms of FLOPS, the Fugaku Supercomputer in Japan, contains over seven million cores[15].

Modern job scheduling systems on supercomputers use fixed resource allocation which provides little flexibility and leads to inefficient usage of computing resources. These problems are amplified by the increasing parallelization. A dynamic resource approach which allows varying available resources at runtime could fix the shortcomings of space-sharing systems. However, this approach raises new questions on the design of parallel applications with regard to resource-adaptivity and scalability. New, robust interfaces and runtime components need to be specified and evaluated to make programming and running dynamically resource-adaptive applications possible.

To tackle this problem, we used the new MPI Sessions model to develop an interface for resource changes. The interface was inspired by recent work of the MPI Sessions Workgroup[1] and uses process sets to group and describe resource changes.

For testing the interface, we have developed a simulated runtime environment providing dynamic resources on top of MPI. This runtime environment is realized as a C library called **libmpidynres**. The library uses a client-server model to manage and schedule the application. We show that this library allows dynamic resource changes to be handled by a parallel, loop-based application. Furthermore, the library enables applications to use some concepts of MPI Sessions and therefore can be used to develop and experiment with the MPI Sessions API.

However, more work must be done to test other parallel programming patterns with this interface and to integrate this interface into the components of the runtime stack of supercomputer systems.

[1]`https://github.com/mpiwg-sessions/sessions-issues/wiki`

Contents

1. Introduction

1.1. Motivation

Supercomputers are expensive to maintain and are limited in their resources. Usually, job schedulers use a space sharing model in which they grant a job exclusive access to a set of resources on the supercomputer, e.g., CPU cores. This mechanism can lead to inefficiencies as jobs might use their resources uneconomically. This can happen when running an application on fewer resources than are reserved or by idling cores at specific sections of the application execution.

If the job scheduler would be able to dynamically remove resources from an application or add resources to an application, it would allow the scheduler to optimize the core usage by transferring idle resources to applications with high resource demand.

Additionally, this would allow the job scheduler to prioritize jobs dynamically and to grant high priority jobs more flexibility on their available resources.

In most cases, parallel applications can be split into two categories: Applications with static workload and applications with dynamic workload. A static workload means that the data the application is working on stays fixed. On the contrary, dynamic workload means that the data the application works on is changing dynamically in size and structure. In the following sections we take a closer look at these kinds of applications in the light of dynamic resource changes.

1.1.1. Jobs with Static Workload

In many parallel applications, the workload size of the application stays static, meaning that the data on which computation will be applied is of a fixed size. In these scenarios, the data is usually separated into chunks of similar size that are assigned to different processes. Each process then applies computations on the data. At the end, these chunks are aggregated at one rank which then produces usable output. Many applications use a loop based approach where a single data point computation depends on data points in its surrounding (convolution, partial differential equation solvers). In these cases, each loop iteration has to be synchronized to make the exchange of halo regions (data points that neighboring processes depend on) possible.

Once dynamic resources are introduced, the application can check for an available resource change during synchronizations steps. If there is a resource change, data has to be redistributed so that each process has a similar share of the workload after the resource change. Applications with static workload size can profit from dynamic resources by having more resources available during the computation loop and in exchange having less resources at initialization and finalization phases surrounding the computation loop.

Static Workload

```
s,e = get_my_section()
for i in 1...n:
  swap_halo()
  compute(data[s:e])
```

Static WL. + Resource Changes

```
for i in 1...n:
  res_change()
  s,e = get_my_section()
  load_balance(s,e)
  compute(data[s:e])
```

Figure 1.1.: Pseudocode for static loop-based application scenarios.

1.1.2. Jobs with Dynamic Workload

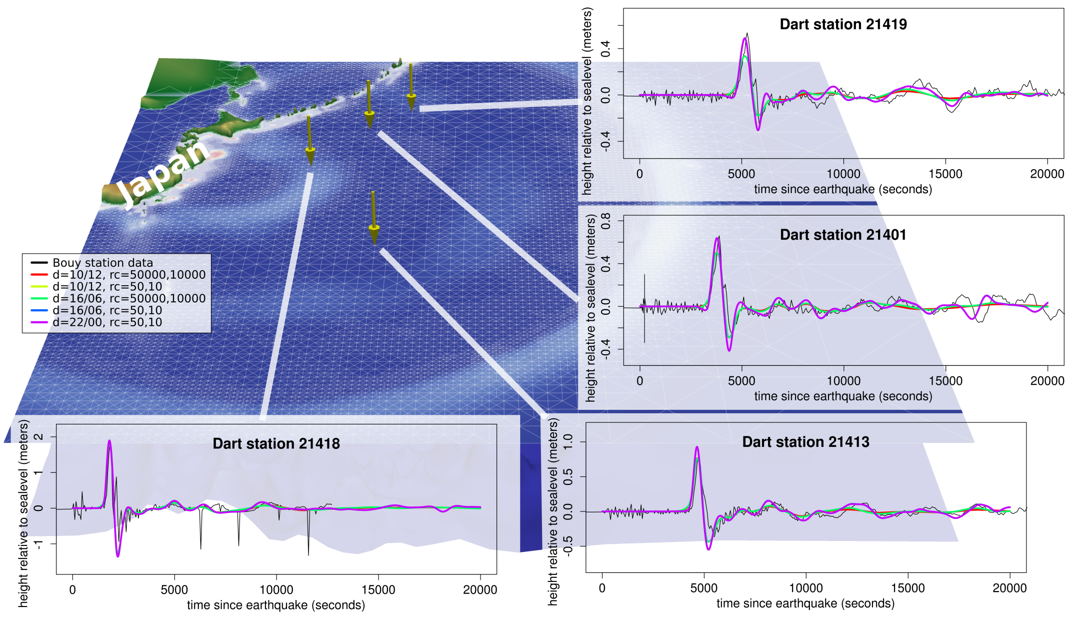


Figure 1.2.: Tōhoku Tsunami Simulation using adaptive grids on PDEs by Schreiber[34]. A finer grid can be seen at areas of high change.

The aforementioned loop-based programming patterns are often observed in the context of numeric computations, especially Partial Differential Equations (PDEs). PDEs are used for a multitude of purposes including weather simulations, automotive crash tests, and protein folding[17].

Using adaptive mesh refinement, PDEs can be solved more accurately by dynamically increasing the precision in sensitive areas. However, this approach changes the application workload size with each refinement step. These grid refinements lead to load imbalances and to varying efficiency of the application.

The task of dividing the workload into chunks of similar sizes during adaptive mesh refinement is difficult, especially in higher dimensions. Multiple software packages exist to solve the problem by assisting the creation of large scale HPC programs using adaptive mesh refinement.

p4est is an MPI library that manages data structures called *octree* in a *forest of octrees* to enable parallel adaptive mesh refinement on large scale computer systems[3]. The *forest of octrees* allows a programmer to describe the three-dimensional connectivity of the mesh used. This forest can then be adaptively refined using callback functions passed to *p4est*.

PETSc is a collection of libraries offering data structures and routines for parallel scientific applications, mainly targeting PDE problems[1]. It has support for MPI and can also use *p4est* as an adaptive mesh refinement backend.

Integrating these libraries with dynamic resources would ease the creation of efficient, scalable PDE solver applications significantly.

Dynamic resources can also simplify the load balancing step by spawning new processes at points where the grid becomes more dense and by removing processes at point where the grid becomes more sparse.

Dynamic Workload (AMR)

```
for i in 1...n:
  s,e = refine_grid()
  load_balance(s,e)
  compute(data[s:e])
```

Dyn. WL + Resource Changes

```
for i in 1...n:
  res_change()
  s,e = refine_grid()
  load_balance(s,e)
  compute(data[s:e])
```

Figure 1.3.: Pseudocode for adaptive mesh refinement application scanarios.

1.2. Thesis Goals

This work tries to accomplish the following goals with regard to dynamic resources:

Design of an Interface for Dynamic Resources

An MPI-like interface that uses new MPI Sessions concepts and allows a parallel application to adapt to dynamic resource changes should be designed. The interface should be well-defined and easy to use for programmers. At the same time, it should allow programmers to trade off between portability and additional functionality by providing optional `MPI_Info` type arguments.

Note that this work will focus on programs that follow the loop based parallel programming pattern and have high synchronization requirements. The synchronization allows all application processes to be aware of resource changes and to adapt simultaneously.

Development of a C Library Implementing the Interface

The new interface should be tested and adjusted in case of potential problems. For that, coping with the complexity of implementing a real dynamic scheduler on multiple layers of the MPI runtime stack (MPI library, job scheduler, ...) should be avoided. Avoiding these layers also allows for easy extensibility of the interface implementation.

To solve the problem, this work presents a C library on top of MPI that implements the proposed interface and a simulated dynamic resource environment. This assure portability and ease of extensibility.

Exploration of MPI Sessions

The simulated environment should implement an interface for the MPI-4.0 Sessions draft[10]. The draft already proposes an interface which should be implemented by our library. Furthermore, the MPI Sessions interface should be tested and adjusted in case of problems.

Testing the Interface

Finally, a proof-of-concept, loop-based application should be created. This application should use the new interface to correctly handle and adapt to resource changes. At the same time the application should be able to coordinate and communicate across all available resources.

2. Related Work

2.1. The Message Passing Interface

Modern supercomputers are highly parallelized. They contain cores in the number of millions[15]. Supercomputer applications need to be able to use these CPU cores efficiently. To accomplish this, work began in the early 90s with the goal of creating a standard library interface for efficient interprocess communication. This effort lead to the release of the first *Message Passing Interface* (MPI) specification in 1993[36].

MPI defines a C and Fortran interface for message passing between processes of an application. The interface allows programmers to efficiently use parallel programming patterns on supercomputers. Popular implementations of the Message Passing Interface include MPICH[26] and Open MPI[32]. Today, MPI is the dominant programming model used on supercomputers and much research relies on MPI to run scalable, distributed applications.

During the course of the years, multiple versions of the MPI Standard were released. While the MPI-1 standard (with the latest version being MPI-1.3) mostly focuses on Message Passing itself and defines Point-To-Point and collective communications and process topologies, MPI-2 introduces first concepts of parallel I/O, one-sided communications (using Remote Memory Access) and dynamic process management, on which a close look will be taken in Section 2.3.1[7, 8]. The latest MPI version as of April 2021 is MPI-3.1. MPI-3 introduces non-blocking collective concepts and further functions for one-sided communication[9].

The efforts behind the MPI standard are driven by the MPI Forum, a collection of experts on parallel computing from academia, governments and the computer industry[9]. Currently, the MPI Forum is working on the next standard version, MPI-4.0. MPI-4.0 "aim[s] at adding new techniques, approaches, or concepts to the MPI standard that will help MPI address the need of current and next generation applications and architectures."[25]. Some major goals include gaining better support for hybrid programming models, meaning programming models that both contain interprocess communication with MPI and shared-memory thread-based communication (e.g., using OpenMP), persistent collective operations which allow optimized reusage of function calls with the same arguments and better fault tolerance support[25].

2.2. MPI Sessions

MPI Sessions are a new concept introduced in 2016 by Daniel Holmes et al. at the 23rd European MPI Users' Group Meeting with *MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale*[20]. To further investigate the concepts of MPI Sessions and how to realize them with a sound API, a new MPI Forum working group concerning MPI Sessions was created. Since then, a first prototype of an MPI Sessions API was implemented in Open MPI combined with PMIx and PRRTE with support for initializing and finalizing sessions and creating groups from the process sets "mpi://SELF" and "mpi://WORLD"[19, 18]. In contrast, **libmpidynres** does not implement the MPI Sessions interface as part of an MPI library, but will act as a simulation layer on top of MPI, which allows for greater flexibility and experimentation with MPI Sessions concepts by not needing to modify other areas of the runtime stack.

The MPI-4.0 draft released in November 2020[10] includes an MPI Sessions API which is the main API reference used in this work. It was extended and slightly modified in **libmpidynres**(see Chapter 5).

A timeline of MPI and MPI Sessions releases is shown in Figure 2.1.

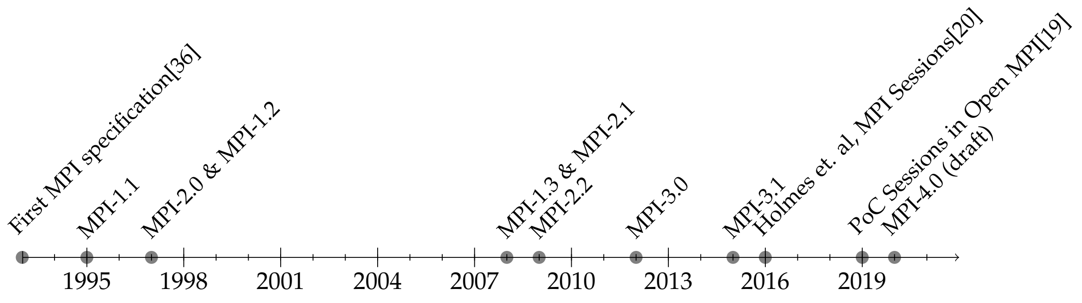


Figure 2.1.: Timeline of MPI & MPI Sessions

2.2.1. Global vs. Local Initialization

Traditionally, MPI must be initialized and finalized globally using `MPI_Init` and `MPI_Finalize`. There are multiple downsides to this. Firstly, MPI initialization needs to be coordinated. This makes it hard to create modular code and libraries. Also, MPI cannot be initialized more than once and after it has been finalized. Furthermore, global initialization is known to be bad at scaling efficiently with the size of ranks in `MPI_COMM_WORLD`[4]. Because the size of large-scale MPI jobs has increased drastically since the first version of MPI was released, a new approach is necessary. The proposed *MPI Sessions* aim to solve these problems and replace the global MPI initialization. MPI Sessions initialize MPI locally by creating a lightweight handle to the MPI runtime. The Session object can be initialized and finalized using two new functions called `MPI_Session_init` and `MPI_Session_finalize`. MPI Session related functions will then have to take a Session object as an argument to be able to access the runtime information stored in the Session object[20].

2.2.2. Runtime Information and Process Sets

In general, the MPI Forum is hesitant when it comes to tighter runtime integration in the MPI standard to keep the interface runtime-independent[20].

However, the current trends in High Performance Computing suggest that a tighter runtime integration leads to performance gains. For example, knowledge about the physical topology of the nodes can lead to a smart distribution of tasks to maximize transfer speed and remote memory accesses[20]. Tighter runtime integration is also a goal of the upcoming MPI-4.0 standard and part of it is to be realized using MPI Sessions.

One concept that aims to help with better runtime integration are *process sets*. Process sets allow the runtime to expose available computing resources in multiple named sets. These sets can be used to group together ranks with similar physical attributes (e.g., ranks sharing a rack) and thus give hints about the physical topology of the computer system. Also, process sets can be useful for managing available resources for an application as each application can have different process sets available. This aspect will play a significant role in this thesis and is the driver behind dynamic resource changes.

A closer look at process sets will be made in Section 3.4.

2.3. Existing Work on Dynamic Resources

Much research has been done to help the creation of dynamically scalable software. This research goes way beyond HPC use cases. With the rise of cloud computing and its service models, together with increased usage of containerization software, new tools like Kubernetes[22] play a big role in creating scalable, reliable software. Here, a closer look will be taken on two approaches targeting HPC: MPI's *dynamic process model* and a new paradigm to parallel computing and CPU design called *invasive computing*.

2.3.1. MPI's Dynamic Process Model

The MPI-2 standard introduced a new dynamic process model with multiple new functions including the ones shown in Figure 2.2. This dynamic process model was inspired by the work of PVM (Parallel Virtual Machine)[9, 8], a software package that enables parallel computing over computer networks[12].

```
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs,
       MPI_Info info, int root, MPI_Comm comm,
       MPI_Comm *intercomm, int array_of_errcodes[])

int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
       char **array_of_argv[], int array_of_maxprocs[],
       MPI_Info array_of_info[], int root, MPI_Comm comm,
       MPI_Comm *intercomm, int array_of_errcodes[])

int MPI_Comm_get_parent(MPI_Comm *parent)
```

Figure 2.2.: C signatures for dynamic programming model introduced in MPI-2

The dynamic process model allows an application to spawn new MPI applications, where the parent application is connected to the respective child application through an intercommunicator. Furthermore, it allows applications to discover other MPI applications that are not connected through the `MPI_Comm_spawn` mechanism, running on the same computer system, using a client-server model.

It is important to mention that MPI's dynamic process model does not provide a way of doing more complex tasks that may be provided by an operating system (querying running processes, killing running processes etc.). Environment-specific interfaces have to be used for this. However, MPI still assumes that an underlying runtime has a process-like concept and provides a small interface for interaction with it through the dynamic process model interface[9].

The new function `MPI_Comm_spawn` is used to spawn a child application. It allows to specify an executable (`command`) together with its arguments (`argv`) that the underlying runtime uses to start the application. The `maxprocs` argument specifies the number of processes that should be contained in the child application, i.e. the number of processes that should be started. By default, the number of child processes has to be equal to the `maxprocs` argument. If the MPI library fails to start all processes it will return an error code. Optionally, the `info` argument can be used to pass hints to the MPI library. The `info` argument can also tell the MPI library that spawning less than `maxprocs` children is valid. However, an MPI implementation does not have to support this.

`MPI_Comm_spawn_multiple` is a similar function. It tells the MPI library that a new child application should be started. However, the commands and/or arguments of the new child application's processes may differ. All started ranks are still part of the same `MPI_COMM_WORLD`.

Finally, `MPI_Comm_get_parent` allows an MPI application to query the intercommunicator that connects both the `MPI_COMM_WORLD` of the calling process and the

MPI_COMM_WORLD of the application that started the calling application using one of the MPI_Comm_spawn functions.

The downside of the spawn mechanism are mainly performance issues and lack of flexibility. Due to the high amount intercommunicators and separate process groups which may be created during a highly resource-dynamic application run, communication and synchronization become slow and inefficient[6]. Another downside is that the destruction of processes can only happen in groups. Also, the maximum amount of resources are limited by the MPI_UNIVERSE_SIZE parameter which represents the total number of available processes for the HPC job.

2.3.2. Invasive Computing

Invasive Computing is a novel approach to parallel programming, first proposed by Teich et al. in November 2010[35]. The core idea is that applications are resource-aware and self-adaptive. They can expand to new, neighboring computing resources at application runtime. The main goals behind Invasive Computing are dynamic resource adaption and fault tolerance. The work behind Invasive Computing is driven by the DFG (Deutsche Forschungsgemeinschaft) in the Transregional Collaborative Research Centre 89 as a joint effort by the Technical University of Munich, the Karlsruhe Institute of Technology and the FAU Erlangen[37].

The Invasive Computing approach defines multiple operations that a program can apply on computing resources, including the operations *invasion*, *infection* and *retreat*. During *invasion*, an application claims multiple resources in its surroundings. Afterwards, the application copies its program code onto the resource cells that were claimed (*infection*). Finally, the application can free resources by using the *retreat* operation.

Originally, the Invasive Computing approach was mainly aimed for embedded MPSoC devices[35], but work has been done to apply the approach to HPC.

In 2012, Ureña et. al modified the MPICH[26] library by extending the MPI-2 interface with Invasive Computing operations, namely MPI_Comm_invade, MPI_Comm_infect and MPI_Comm_retreat. After a successful *infect* operation, the new processes are made part of MPI_COMM_WORLD. This extension, called invasive MPI (iMPI), was then evaluated on an experimental Intel CPU containing 48 cores.

Gerndt et al. published an OpenMP extension called iOMP in 2013 which allows the usage of the *invade*, *infect*, *retreat* operations in programming languages supported by OpenMP[13, 31] and therefore in High-Performance Computing scenarios. One of the available logical CPU cores was reserved for a designated *resource manager*, a concept similar to the approach taken in this thesis.

In 2014, Martin Schreiber et al. successfully used the invasive computing approach on HPC systems, by using MPI and OpenMP to create a hybrid Invasive Computing environment. This environment was used to simulate tsunami wave propagation using adaptive grid refinement for solving partial differential equations[33]. The authors use a resource manager to distribute available resources among multiple applications.

Compres et al. implemented dynamic resources by modifying the MPICH library and the Slurm Workload Manager in 2016[6]. Similar to this work, resource changes are imposed by the runtime system and can be queried by the application. In contrast to **libmpidynres**, Compres et al. use two separate functions, `MPI_COMM_ADAPT_BEGIN` and `MPI_COMM_ADAPT_COMMIT` to apply the resource change. This approach starts new resources with the first function call and removes resources with the second. This allows the application to rebalance its workload. In **libmpidynres**, resource changes are either addition or removal only, thus the need for two function calls is avoided.

In this work, the library **libmpidynres** is presented. Unlike the mentioned work on dynamic resources, the library implements resource changes using process sets, which were introduced with MPI Sessions. Also, **libmpidynres** is implemented as a proof-of-concept on top of MPI which ensure potability and easy extensibility. The main concepts used by **libmpidynres** are presented in the following chapter.

3. Core Concepts

3.1. MPI Runtime Data

When an MPI library is initialized, it has to discover other processes to construct communication endpoints and MPI objects (e.g., Communicators, Groups, . . .). During this process, the library might need to communicate with the hardware, an operating system or the job scheduling system. As the library operates, the library creates global data stored in its address space. With MPI Sessions, the library runtime data will be stored partially in a local Session object.

Various MPI operations will act differently on the library data. There are multiple approaches to implement MPI message sending in the underlying library. For example, there are different ways to realize the `MPI_Send` function[9] in an MPI library. One example is to wait for the recipient rank to call `MPI_Recv` and then send the message. In this case, the `MPI_Send` function behaves like the synchronous `MPI_Ssend`. Another valid implementation of `MPI_Send` it to leverage direct memory access. In this case, the receiving process copies the data from a temporary buffer of the sender's process space into its own. Section 3.4 of the MPI-3 standard contains more information on different modes of message passing[9].

With MPI Sessions, runtime data like process set changes has to be distributed across all relevant ranks. It remains open how and when this is achieved.

In this work however, the simulation layer will make it possible for a rank to create new process sets. This is expected to occur quite frequently, namely with and after each resource change. This information together with the resource changes themselves must be available to all active processes of the application.

In our approach all process set and resource change data is stored centrally in the resource manager's process space. This approach will be further discussed in Section 4.2. This method takes away the need of synchronization and assures a consistent state across all ranks at the cost of slower access times.

3.2. MPI's Info Object

The Info object (`MPI_Info` type in C) was introduced with MPI-2 as an opaque object that represents a key-value dictionary. Both the key and value are string types (`char *` in C). Keys and values each have an implementation defined maximum length. MPI defines multiple functions to access and modify the dictionary's content[8]. Besides these functions, the Info object is mainly used as an argument to give the underlying MPI runtime some implementation-defined optimization hints. Some examples include `MPI_Comm_spawn` (see Section 2.3.1), `MPI_Comm_connect` and `MPI_File_delete`. This mechanism allows an application to trade off between portability and performance.

In the case of the MPI-4.0 draft, the Info object plays an important role and is an argument to all MPI Sessions-related functions except for `MPI_Session_finalize`[10]. The rationale behind this is that MPI Sessions and its process sets provide a tight interaction with the runtime. Many aspects of process sets are implementation defined, e.g., which specific process sets are available to the application. These questions among others can be answered by implementation defined keys in the Info objects passed to the Sessions-related functions.

The Info object is also an important aspect of the API provided by **libmpidynres**, the library that is presented in this thesis. Similar to the use case stated before, it is often used in the API to give some implementation defined hints. Furthermore, the Info object is used as a designated return value of a function, with respective key-value items (see `MPIDYRNES_get_psets` in Section 5.4).

As a consequence of the resource manager client-server model described in Section 4.2, the Info object has to be serialized internally by **libmpidynres** and communicated via MPI across ranks.

3.3. Computing Resource

In general, the term *Computing Resource* describes an arbitrary component of a computing system that can be measured and is limited by some aspect. Ideally, a dynamic resource scheduler would be able to not only dynamically change an application in terms of processing power, but also dynamically change other resources, e.g., available memory per process, GPU nodes, caches.

In this work, computing resources are limited to CPU cores and processes bound to these cores. **libmpidynres** uses existing processes to simulate computing resource addition and removal. This is achieved by taking the processes contained in `MPI_COMM_WORLD` and simulating the availability of a process by hiding or revealing the process from the application using process sets. Also, **libmpidynres** uses a client-server model with a designated resource manager hidden from the application. As a consequence, only the processes with rank `1...n-1` are available to be scheduled/managed.

3.4. Process Set

The MPI Sessions model introduces a new concept called *process set*[20].

A process set represents a set of computing resources provided by the runtime. It can be used to create an MPI group which then can be used to create an MPI communicator. It is uniquely identified by a name expressed in a URI-like format[16]. This format is useful, because it allows the separation of sets by common attributes into namespaces and subcategories. The MPI Standard draft reserves the "mpi://" prefix for process set names[10].

Additionally, the latest MPI-4.0 draft states that a process set caches key-value tuples which can be queried by the application as an MPI Info object[10].

To which extent process sets are shared and synchronized between processes is not specified. However, it is clear that different processes can have different views on the process sets provided by the runtime and have a different set associated with the same name. An example for that is "mpi://SELF", a process set that has to be defined by the runtime and should always contain solely the querying process[10]. Besides "mpi://SELF", there exists another predefined process set: "mpi://WORLD". "mpi://WORLD" contains all processes that were started initially by the runtime and would be part of `MPI_COMM_WORLD` in previous MPI versions.

Figure 3.1 shows an example how process sets of an application might look like. Five processes were started initially. All of them are contained in the "mpi://WORLD" process set. In the figure, the process sets prefixed with "hwloc://" represent process sets allocated by the mpi library by communicating with the runtime and discovering the hardware locations of processes. Process sets that start with "app://" stand for process sets created by the application by asking the MPI library to create the process set. According to the MPI-4.0 draft, a process should only be able to query for process sets that it is part of[10]. This is highlighted by the green box in the figure. When Process 1 queries for available process sets, the MPI library should respond with "mpi://SELF", "hwloc://numa/0", "app://atmos/task/0" and "mpi://WORLD".

In **libmpidynres**, an application can create new process sets by combining two process sets into a new one. The operations allowed are *union*, *intersection* and *difference*. This mechanism will be described further in Section 5.4.

Also, in **libmpidynres**, the simulated MPI environment can create new process sets besides "mpi://SELF" and "mpi://WORLD". This happens during resource changes. Simulating arbitrary process sets like hardware locations can be achieved easily by extending the source code of **libmpidynres**. The process sets created by **libmpidynres** are prefixed with "mpidynres://".

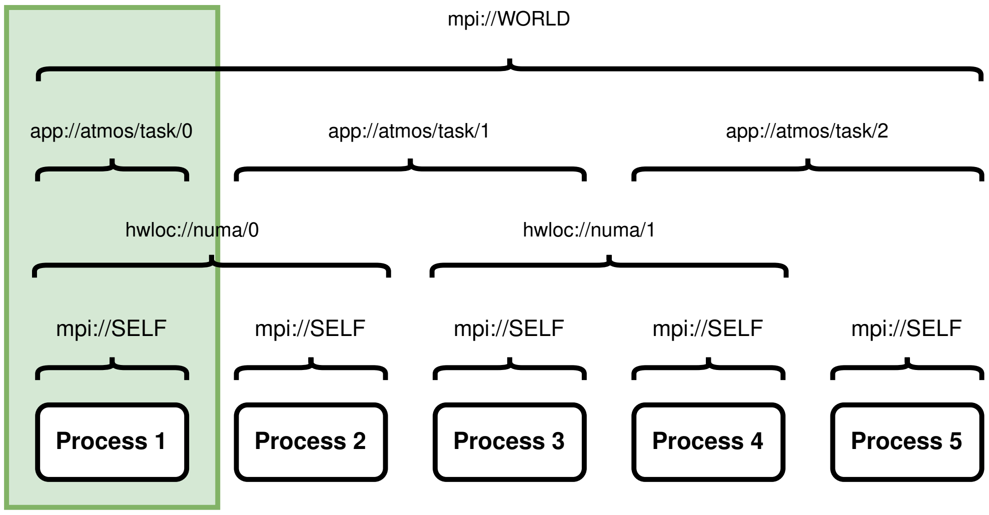


Figure 3.1.: The process sets of an example application. Process sets are represented as curly brackets. Process 1's view on the process sets is highlighted in green.

In this work, process sets are *immutable*. This means that the name of a process set specifies only the same set of computing resources while its valid. This avoids race conditions when creating groups out of process sets. Once one of the processes exits, the process set becomes invalid and trying to create a group out of a process set fails.

In the MPI Sessions workgroup, a naming scheme using version numbers was discussed. For example, "mpi://WORLD/0" contains the initially started computing resources. As the number of processes for an application changes, new process sets appear named "mpi://WORLD/1", "mpi://WORLD/2" and so on. Each of these process sets contain all active computing resources at the moment of the process set's creation. This naming scheme solves the problem of race conditions in mutable process sets. However, the processes still have to agree on a version number to use. Currently, **libmpidynres** does not support these versioned process sets. Process set names besides "mpi://WORLD" are created by **libmpidynres** and consist of a "mpidynres://" prefix, followed by a random string.

3.5. Resource Change

A *resource change* describes a change to the available computing resources of an application during its runtime. Resource changes allow the job scheduling system to optimize the usage of available cores.

In **libmpidynres**, resource changes are driven by process sets. For each resource change, a new process set is created. If the resource change adds new resources, the new process set will contain all processes that will be added to the application. If the resource change removes existing resources, the new process set will contain the processes that are currently part of the application that should be removed.

In this work, the application has to query for resource changes on a regular basis. Resource changes are imposed by the runtime and in the case of a new resource change, the resource change type and process set name will be returned by the query. To apply the resource change, the application *accepts* the resource change. Processes that are added will then be started by the runtime. If resources are removed, the application has to quit the processes by itself.

libmpidynres only supports either addition or removal of resources with a process set and not both addition and removal at once. As a consequence, load balancing is simplified. When resources are added, using process set operations, a connection between the main application and the newly created process set can be accomplished. Now, the application can load balance together with the newly created processes. In the case of resource removal, the application will have to load balance before accepting the resource change. This means that data is transferred from the processes that will shut down to the processes that will keep on running.

Due to the proof-of-concept nature of **libmpidynres** and its implementation as a simulation layer, the runtime cannot preemptively "kill" running processes. Malicious applications that do not correctly shutdown their resources cannot be handled by **libmpidynres**.

4. Realization of the Simulation Layer

In this chapter, an overview of the implementation of **libmpidynres** is given. First, the general architecture and the library-application interaction is described. Then, the internal communication of **libmpidynres** is explained. Afterwards, a closer look is taken on how process sets and resource changes are implemented. Finally, a short overview of the software development approach taken is provided.

4.1. Simulating the Runtime on top of MPI

The process of creating an MPI Sessions simulation layer for MPI applications can follow different goals and provides multiple challenges.

In this case, one of the goals of the simulation layer is the experimentation with the MPI Sessions API and environment. Therefore, a C program using the simulation layer ideally should be able to use the MPI Sessions API as if the API was part of the underlying MPI implementation itself. Furthermore, the simulation layer runtime behavior ideally should resemble the MPI Sessions runtime as close as possible. This means that the logical changes that result from API calls should be close to the specification in the MPI-4.0 draft[10]. On the other hand, the usage of the simulation layer should remain simple and as platform independent as possible.

To reach these goals, the simulation layer is implemented as a C library called **libmpidynres**, consisting of C header files and the library shared object which depends on MPI. The source code is written in C and only relies on MPI and a valid C runtime. As a consequence, **libmpidynres** can be compiled on all systems with MPI support. Furthermore, this ensures ABI compatibility and ease of use, because compiling an application with the library only consists of linking the application's source file against the library. Alternatively, the application can be compiled against **libmpidynres**'s source code if there are compatibility issues between the MPI library that **libmpidynres** was dynamically linked against and the MPI library the application wants to use.

To keep close to the ideal API of MPI Sessions, one solution would be to expose the MPI Sessions API in a header called `mpi.h`. This would allow the application program to resemble an MPI application if MPI Sessions was included in the MPI standard and would allow the simulation layer to disable/wrap real MPI functions. However, there are multiple downsides to this approach.

First, installing the header under this name into a system will either overwrite the existing MPI library's header file or at least create conflicts when compiling MPI programs (a compiler might use the wrong header file).

Another problem is the increased complexity of the library's code: If the simulation layer library does not wrap a function from the MPI library, both libraries may be linked together statically, leading to less flexibility as users cannot change (i.e. update) the used MPI library independently. If the simulation layer library wants to wrap certain functions from the underlying runtime library it will either have to use complex symbol renaming or adjust the MPI library's source code to rename functions as otherwise there will be duplicate symbols when linking the libraries.

A third problem of using the `mpi.h` header name is that due to the simulating nature of the library, some setup/cleanup will have to be performed before the application environment will resemble the MPI Sessions environment. For example, the underlying MPI runtime still has to be globally initialized and finalized.

4.1.1. Application Wrapper

Because global MPI initialization and cleanup is still necessary when simulating MPI Sessions on top of MPI, the design of **libmpidynres** splits the application program into two distinct parts: The application wrapper and the simulated application.

The application wrapper uses the `mpidynres_sim.h` header and is aware of the simulation layer. It has to initialize MPI, configure **libmpidynres** and then start the simulation layer, passing a communicator object to **libmpidynres**. The latter is achieved with the function `MPIDYNRES_SIM_start` which is collective, i.e. has to be called by all ranks of the communicator used. After the simulated application is done, the wrapper has to clean up and finalize MPI. The application wrapper implements the C `main` function and passes the simulated application entry point to **libmpidynres**.

The simulated application uses the `mpidynres.h` header which exposes the MPI Sessions API and a process set and resource change management API. It has a separate entry point which is passed to `libmpidynres` by the application wrapper. The simulated application is running in a simulated Sessions environment that provides process sets and resource changes. How this is implemented will be discussed in Section 4.2.

Figure 4.1 illustrates how these two header files are used and how entry point information is passed to **libmpidynres**.

To ease the usage of **libmpidynres**, when defining `MPIDYNRES_MAIN` before including `mpidynres_sim.h` in the C code, a default wrapper is included which implements a main function which will pass `MPI_COMM_MAIN` and a user defined `int MPIDYNRES_main(int argc, char *argv[])` to **libmpidynres**.

application_wrapper.c

```c
#include "mpi.h"
#include "mpidynres_sim.h"
/*
 * Simulation aware code uses the "mpidynres_sim.h" header
 */

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  ...
  myconfig.base_communicator = MPI_COMM_WORLD;
  MPIDYNRES_SIM_start(&myconfig, argc, argv, application_entry);
  ...
  MPI_Finalize();
}
```

application.c

```c
#include "mpi.h"
#include "mpidynres.h"
/*
 * The MPI Sessions API is available in the "mpidynres.h" header
 */

int application_entry(int argc, char *argv[]) {
  // Use your favorite MPI Sessions functions here
  return 0;
}
```

Figure 4.1.: Application wrapper example using libmpidynres.

4.1.2. Library Interfaces

Due to the design of **libmpidynres**, both the application wrapper and the simulated application should include the original MPI header themselves, as general MPI functions are not implemented by **libmpidynres**.

Both the application and **libmpidynres** access the MPI library independently. That is why the wrapper has to initialize the MPI environment and also choose a communicator to be used.

As a consequence, the application itself should not call any MPI functions that deal with the initialization or finalization of MPI. Furthermore, **libmpidynres** cannot deal with an application where a process suddenly exits (e.g., by using the `exit` syscall) or fails in another non intended way, for example with a segmentation fault. A non-simulated MPI Sessions implementation may be able to detect this and simply remove the resource from the application. As a replacement for `exit`, a special function called `MPIDYNRES_exit` is available in **libmpidynres** which a simulated application can use to return from any point in its execution from the entry point function. In the code example in Figure 4.1, this has the same effect as the `return` statement in line 9 in `application.c`. This mechanism is implemented using the C standard library functions `setjmp` and `longjmp`.

Also, the application should not use any MPI objects which were not generated by **libmpidynres**, e.g., `MPI_COMM_WORLD`. However, it can create new MPI objects based on objects returned by **libmpidynres** calls. These limitations cannot be enforced by **libmpidynres**. It expects a non-malicious application that respects these rules.

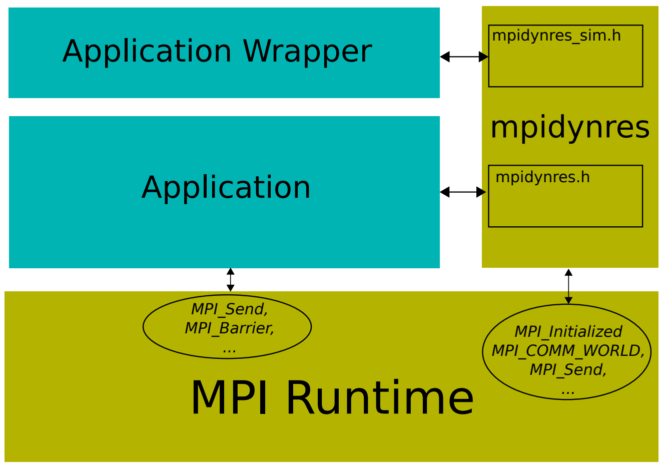


Figure 4.2.: The different components of a simulated MPI Sessions environment using **libmpidynres**.

4.2. Resource Manager and Computing Resources

To achieve process set and resource change consistency, **libmpidynres** uses a client-server architecture with a dedicated *resource manager* acting as a server and *computing resources* acting as clients. The resource manager is responsible for the process set management, the resource scheduling and the computing resource startup. While a more complicated peer-to-peer approach is possible, the resource manager avoids unnecessary race conditions, is more feasible to implement and assures a consistent state across all ranks.

4.2.1. Communication

The communication between the resource manager and the computing resources is realized with MPI and is opaque to the application as it happens within **libmpidynres**. It takes place on top of the communicator passed to `MPIDYNRES_SIM_start`. The communicator used is included as a global symbol inside **libmpidynres**. This is necessary because the communicator has to be hidden from the application and storing it inside the `MPI_Sessions` object is not possible as it will be constructed while the simulated application is already running.

Communication exchanges usually start by a computing resource sending an MPI message with a request-specific tag and request-specific data. The only time a communication exchange is not started by a computing resource is when the resource manager sends a *change-state* command to a computing resource. A communication exchange usually ends with the resource manager sending an answer message with a request-specific answer tag. This answer can contain either request-specific data or an error code. During a communication exchange, more MPI messages may be sent between the resource manager and the computing resource depending on the request. This is due to variable-length data that has to be sent in some cases and special serialization routines for the exchange of `MPI_Info` objects.

Depending on the data, either MPI's primitive datatypes are used or new datatypes are constructed using `MPI_Type_create_struct`. The MPI tags used for these messages are defined with an `enum` statement at the beginning of `comm.h`, an internal C header. The corresponding C source file `comm.c` contains the implementation of the `MPI_Info` serialization functions and the datatype construction mechanism. The resource manager communication endpoint and the computing resource communication endpoint are implemented in the `mpidynres.c` and `scheduler.c` files respectively.

4.2.2. Control Flow & State Signaling

After the application wrapper has called `MPIDYNRES_SIM_start`, the processes enter an MPI barrier on the communicator passed to the call. This is done to make sure all processes are ready to start the simulated environment and to make sure other non-obvious MPI functions used later will not get stuck due to a process not being active.

After the barrier is passed, the processes assume different roles depending on their rank in the communicator used. Rank 0 will act as the resource manager. Ranks 1...n-1 act as available computing resources. A computing resource starts in the idle state. This means that **libmpidynres** calls `MPI_Recv` with a special tag reserved for the *change-state* message type. Depending on the MPI implementation and the underlying runtime used, the `MPI_Recv` could even lead to an actual idling CPU core in the computer system. In the case of Open MPI, this is effect does not occur as it schedules events using spinlocks in the user space leading to a busy loop[30].

A *change-state* message can either tell the computing resource to become active, meaning to run the application, or to shutdown, meaning that the simulated environment is ending, and the control should be returned to the application wrapper. In the case of a computing resource becoming active, `setjmp` is called and the application entry point is called with the `argc` and `argv` commands passed to `MPIDYNRES_SIM_start` function by the application wrapper.

When an application process is done, either by calling `MPIDYNRES_exit` or by returning from the entry point function, a message is sent to the resource manager, so it can keep track of active and idle resources correctly and the computing resource goes back to waiting for *change-state* messages using `MPI_Recv`.

A state diagram describing these steps can be seen in figure 4.3.

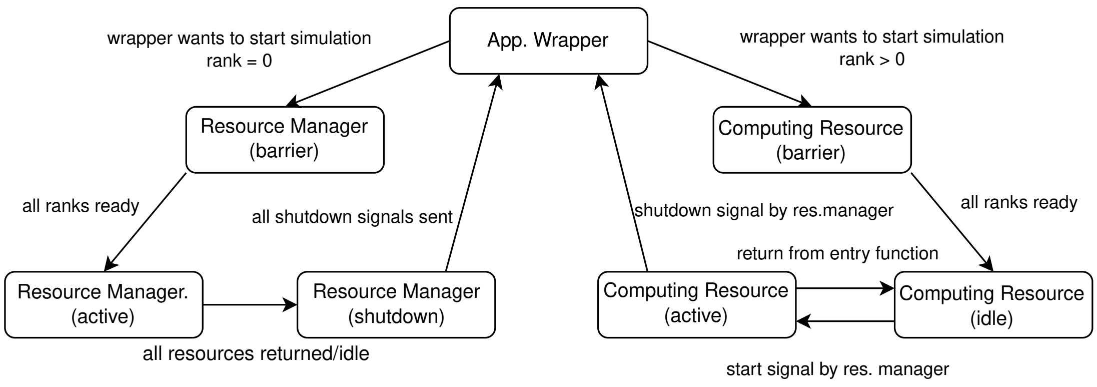


Figure 4.3.: State diagram showing different process stages during a simulation run.

4.3. Resource Manager Realization

The resource manager is used for process coordination, process set management and resource change management.

The internal header `scheduler.h` declares the `struct MPIDYNRES_scheduler` type which represents the resource manager. The header also declares a constructor and a `MPIDYNRES_scheduler_free` function for the resource manager type. The resource manager uses multiple datatypes and data structures for storing information about the simulation state. These are declared in the internal header `scheduler_datatypes.h`. Most data structures like sets and dictionaries are constructed using *ctl*, a 3rd-party, header-only C data structure library[23]. Dictionaries are implemented with *ctl*'s set datatype by storing the key inside the set element struct and exclusively using the key field for the set element comparison function.

4.3.1. Request Handling

When the resource manager becomes active, it spends most of its time in the `MPIDYNRES_scheduler_schedule` function which contains the main request loop. In this function, multiple non-blocking, receiving MPI calls are made using `MPI_Irecv`, one for each initial communication exchange message potentially coming from a computing resource. Then, `MPI_Waitany` is used to wait for a request to complete. Once a request completes, the resource manager starts to call a handler function declared in `scheduler_handlers.h`. The handler usually interacts with the simulation state and the requesting computing resource as described in section 4.2.1. After the request has been handled, a new `MPI_Irecv` call is made so new requests of the same type can be received again. Once there are no more active computing resources, the loop is broken and the non-blocking requests freed.

Using this combination of `MPI_Irecv` and `MPI_Waitany` allows the scheduler to only receive valid messages with a correct tag and correct MPI datatype. An alternative could be a `MPI_Recv` using a single `MPI_INT` type. The received integer would then represent the type of request to be performed and a request specific `MPI_Recv` call could be used. However, this approach requires one more message to be sent during the communication exchange, namely this first integer.

A consequence of the request handling design in **libmpidynres** is that the resource manager is blocked during the handling of a request. While this approach guarantees state consistency and avoids locks and race conditions on the resource manager's side, it has multiple downsides. There is a large performance loss by this blocking design. Multiple requests arriving in a short timespan will be handled sequentially. There are multiple scenarios where this situation is likely. One example is the querying of process sets at the application start. Another downside is, if one of the communication exchanges fails and the resource manager is left in a blocking MPI call, the resource manager is stalled and further requests will never complete.

4.3.2. Process Set Management

Computing resources are internally identified by their rank inside the communicator used by **libmpidynres**. A process set is represented as a `struct pset_node`, containing the name, a set of integers representing the computing resources in the process set and finally an `MPI_Info` object. The resource manager holds a dictionary with the process set names as keys and with `struct pset_nodes` as values. "mpi://SELF" is not stored in this dictionary as its contents vary depending on the process querying the process sets. As a consequence, the resource manager's request handlers have to consider the edge case of "mpi://SELF". To support multiple simulated applications, a new dictionary would have to be added to separate the process sets from each application.

The resource manager also tracks the state of process sets using a set of `struct process state` objects. To conserve memory, only active and reserved processes are managed. A struct process state also holds the names and sizes of process sets that the process is part of. This is mainly used for performance reasons. As a consequence, the addition and removal of process sets must modify these process states.

In this work, process sets are assumed *immutable*, meaning the computing resources in a process set and the name of a process set never change. This avoids race conditions that might occur when different computing resources query a process set by its name , e.g., for creating a group. However, this approach means that process set names change frequently and there cannot be a dynamic process set with the same name. The latter could be useful for many scenarios, for example when the environment wants to offer the application a set of resources with specific dynamic characteristics.

The immutability of process sets also raises the question how process sets are treated that are invalid because they contain inactive processes. This can easily happen when a process calls `MPIDYNRES_exit` or returns from the simulation entry function. As a consequence, all process sets that contained the process become invalid.

In **libmpidynres**, invalid process sets are removed globally after a process exits. Groups created from the process sets are still valid. Communicators based on this group become invalid. However, some MPI calls with the communicator will still succeed. This is due to the fact that the process is still technically running in the simulated idle state from MPI's point of view. This means the underlying MPI library will not complain when the communicator is used. However, an application should not use these communicators anymore and in a non-simulated implementation of this process set model, the communicator should become invalid so MPI functions that are use this communicator as an argument fail. As a consequence, a programmer has to be careful and use `MPI_Barrier` or similar synchronization functions between querying process sets and process exits. For example, the resource manager might tell the application that the processes contained in the process set "mpidynres://change12345" have to exit. Now a process would query the process sets its contained in and then call `MPIDYNRES_exit` if "mpidynres://change12345" is in the result. If one process exits before the other processes have called the process set query function, the process set will be freed and the other processes will not see "mpidynres://change12345" in their query result although they might have been part of the process set. This leads to an unintended state.

On the other hand, process sets containing *reserved* computing resources are valid in **libmpidynres**. This is necessary so that the addition of new computing resources under a new process name is possible. Creating a group from such a process set will succeed. Creating a communicator from the group will fail, as there are no active (in the **libmpidynres** terminology) processes that are part of the group. Once the processes are started by the resource manager, they are active and no longer reserved and communicators can be created.

Process sets are created in three circumstances: First, the "mpi://WORLD" is created with all processes that are starting initially. Then, the resource manager will create process sets for resource changes with the *delta* between the old processes available and the new processes available after the resource change is applied. Lastly, the application can use process set operations to ask the resource manager to create new process sets out of existing process sets.

There are three process set operations available: *union*, *intersection* and *difference*. The logic of these operations corresponds to the mathematical set operations of the same name. The resulting set's info object will hold information about the operation. The info object can be queried by a process by calling the `MPI_get_pset_info` function.

4.3.3. Resource Change Management

In **libmpidynres**, processes are managed by the resource manager. With the startup and shutdown mechanism described in 4.2.2, the manager can keep track of the process state and turn idle processes active.

Resource changes are imposed by the resource manager on the application which has to accept them. However, the resource manager cannot enforce resource changes that remove processes from the application. This is due to the fact that the application is in control of active processes. Malicious applications are not considered by **libmpidynres**.

The resource change mechanism described in Section 3.5 means that an application has to query for new resource changes using the `MPIDYNRES_rc_get` function on a regular basis. If there are no resource changes, `MPIDYNRES_RC_NONE` is returned. If processes are to be added or to be removed, `MPIDYNRES_RC_ADD` or `MPIDYNRES_RC_SUB` are returned respectively and additionally the resource change process set name and a resource change tag. Currently, only one pending resource change is allowed. If a resource change is pending, any call to `MPIDYNRES_rc_get` will return no resource change. If there is a resource change, the application has time to inspect the new process set and to do load balancing before it calls the `MPIDYNRES_rc_accept` function using the resource change tag obtained previously. Once the accept function is called, the resource manager starts the new processes or expects processes to have exited, i.e. becoming idle.

In the case of resources being removed, the resource manager will keep track whether all necessary resources have been shutdown or not. As long as there are still resources running that should be removed, all resource change queries will result in no resource change.

This mechanism leads to the multiple possible states a computing resource can have from the resource manager's point of view. These states are shown in Figure 4.5.

Internally, the resource manager saves resource changes in a dictionary holding `struct rc_info` objects. These objects contain a resource change tag, a process set name and a resource change type. The dictionary uses the resource change tag as the key. Due to the current design, the dictionary will only contain one entry at most. However, the dictionary data structure was chosen because it will make future extension of **libmpidynres** easier, especially when it comes to multiple application support.

4.3.4. Scheduling Approaches

The header `scheduler_mgmt.h` exposes a generic API for a scheduling interface. Because of that, new scheduling mechanisms can be implemented easily. This interface is used by the resource manager for resource change decisions. It can track the application state by accessing the resource manager's `MPIDYNRES_scheduler` object that holds the dictionaries discussed in the previous sections.

After the application wrapper starts the simulation and the barrier is passed, the `scheduler_mgmt.h` function `MPIDYNRES_manager_get_initial_pset` is used by the resource manager to get the initially active processes. Then, the "mpi://WORLD" process set is created containing these processes. Finally the *start* message is sent to these initial computing resources so that they become active. In the next step, the scheduler enters the request handling loop (see 4.3.1).

Once the application queries for resource changes, the resource manager calls the `MPIDYNRES_manager_handle_rc_msg` function from the `scheduler_mgmt.h` interface. The scheduling interface implementeation then decides on the next resource change, which then will be returned by the function call. Afterwards, the resource manager registers the resource change in its data structures and returns the resource change to the calling computing resource which then returns it to the application.

To allow more direct communication between the scheduling interface and the application, **libmpidynres** offers a function called `MPIDYNRES_add_scheduling_hints`. Using this function, the application can send an `MPI_Info` object to the scheduling

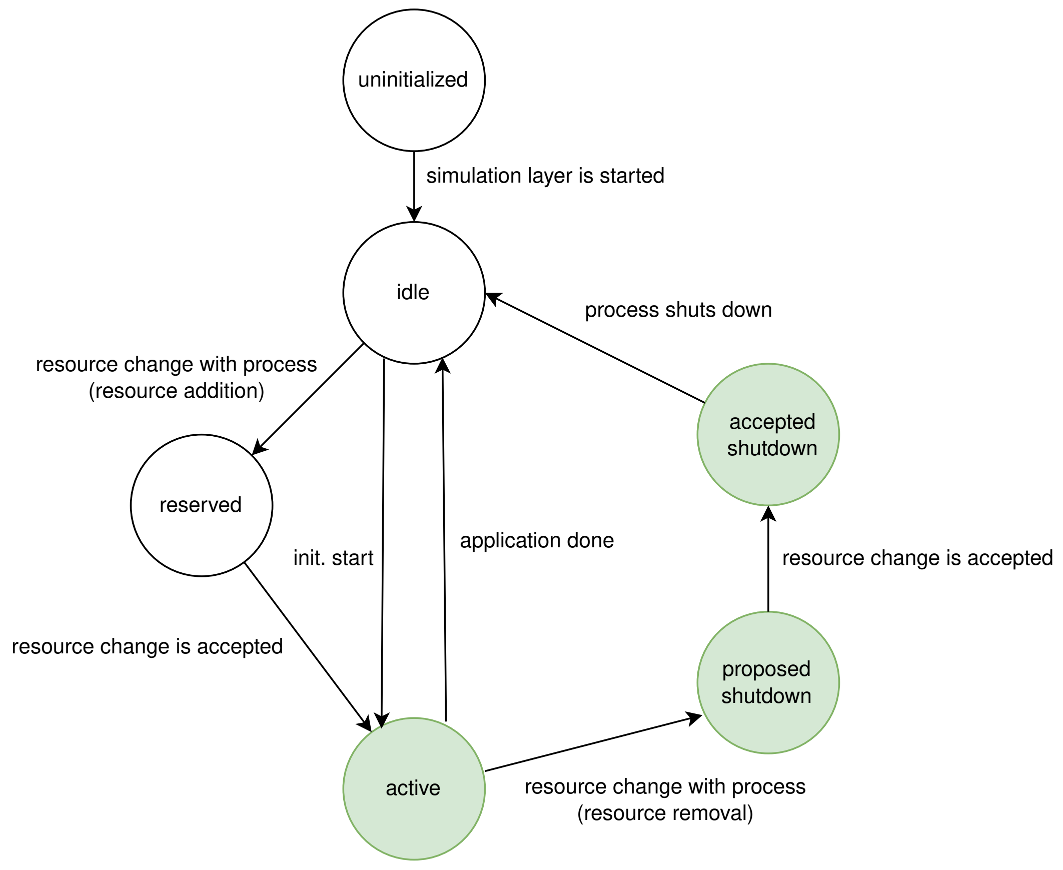


Figure 4.4.: Diagram showing the different states a computing resources can be in. States where the application has control over the resource are highlighted in green.

interface and obtains an `MPI_Info` object as an answer. This mechanism can be used in the future to generate resource changes that fit the application needs.

Another way the application can configure the scheduling interface consists of the application wrapper passing an `MPI_Info` object to the resource manager in the config argument of `MPIDYNRES_SIM_start`. This info object is also accessible by the `scheduler_mgmt.h` interface. This can be useful to specify properties of the initial process set or change scheduling parameters without having to change the simulated application's code by adding `MPIDYNRES_add_scheduling_hints` calls.

Two example implementations of the `scheduler_mgmt.h` interface are included in the **libmpidynres** source code. These can be found in the `src/managers` directory. To use one of them, a copy or symbolic link should be placed in `src` directory with the name `scheduler_mgmt.c`.

Increasing-Decreasing Scheduling

This scheduling types starts with one initial active computing resource and incrementally adds one computing resource each until all available computing resources are active. After that, computing resources are removed one by one until one computing resource is left. Then, resources are added again one by one and so on.

In the **libmpidynres** implementation of this scheduling approach, the initial computing resource that is started is the one with rank "1" in the underlying communicator used. Afterwards, the computing resources are added in the order of their ranks in the communicator. Removal happens in the opposite direction. This means that the highest rank will be removed first, then the rank one lower and so on.

Random-Difference Scheduling

The random-difference scheduling approach adds and removes resources randomly.

For that, a normal distribution is sampled using the Box-Muller algorithm[2]. A negative value means the removal of resources, a positive value the addition of resources.

The resulting value is rounded and sanitized to avoid invalid values, e.g., by removing equal or more resources than are active or adding more resources than available.

Finally, the process set is generated by either applying a random permutation on the active resources' internal id (in the case of removing resources) or on the available resources' internal id (in the case of adding resources) using the Fisher-Yates shuffle[21]. Again, the internal id corresponds to the rank inside the underlying communicator used. Then, the number of processes necessary are taken from the beginning of the permutation result.

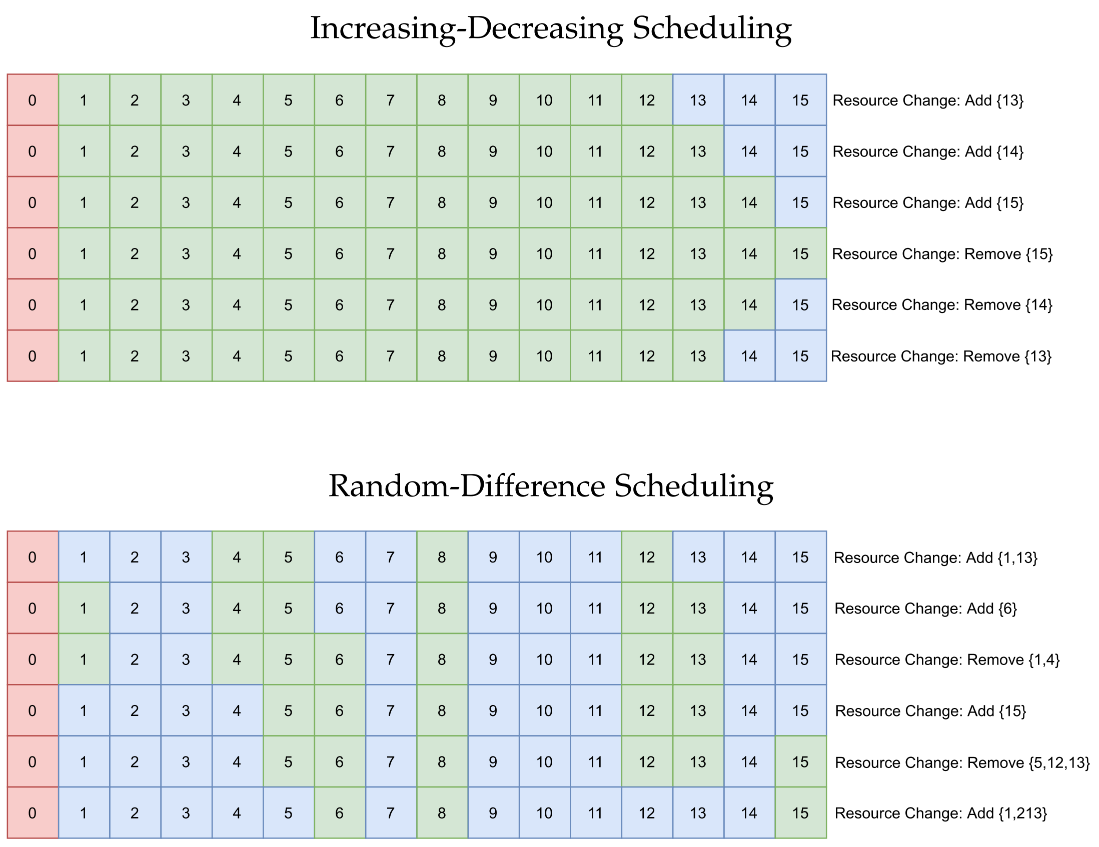

Increasing-Decreasing Scheduling
Resource Change: Add {13}
Resource Change: Add {14}
Resource Change: Add {15}
Resource Change: Remove {15}
Resource Change: Remove {14}
Resource Change: Remove {13}
Random-Difference Scheduling
Resource Change: Add {1,13}
Resource Change: Add {6}
Resource Change: Remove {1,4}
Resource Change: Add {15}
Resource Change: Remove {5,12,13}
Resource Change: Add {1,213}

Figure 4.5.: The two scheduling algorithms illustrated on a communicator of size 16. The red process is the resource manager. Idle processes are shown in blue and active processes are shown in green.

4.4. Computing Resource Realization

libmpidynres uses a designated resource manager rank to manage the simulated runtime data. Other ranks are available as computing resources for the resource manager to schedule. This section will focus on some aspects of the implementation of the computing resource interface that is available to the application.

4.4.1. Session Object

One of the core ideas behind MPI Sessions is the introduction of a `MPI_Session` object which acts as a local handle to the MPI library and stores MPI runtime data.

In **libmpidynres**, the MPI runtime is already initialized globally and the session object does not need to be used for local MPI initialization. As a consequence, the `MPI_Session` object only holds a unique session id which is passed together with most requests to the Resource Manager. Currently, the session id is ignored by the Resource Manager but future work that targets multiple sessions and applications support can use the session id to determine the request's source.

The special value `MPI_SESSION_NULL` is used as a designated error value. In **libmpidynres** it equals the C value `NULL`. When a session object is finalized, it is replaced with the `MPI_SESSION_NULL` object. The application interface checks the passed session object against this value and will return an error if it is used. Using this mechanism, the usage of a session after the `MPI_Session_finalize` call can be detected and avoided.

The `MPI_Session` object is initialized and finalized using `MPI_Session_init` and `MPI_Session_finalize`.

4.4.2. From Process Set to Communicator

Traditionally, MPI communicators are created by the application when calling an MPI function using an existing communicator as an argument. As a consequence, communicators are indirectly linked to `MPI_COMM_WORLD` which is not a part of the MPI Sessions model. To solve this issue, a new function `MPI_Comm_create_from_group` was proposed by the MPI forum in the MPI-4.0 draft[10]. Unlike previous MPI functions, this function allows to create a communicator from a group without the need of a parent communicator.

There are two steps a MPI Sessions application must complete to successfully create a communicator based on a process set. First, the function `MPI_Group_from_session_pset` is called with the process set name as an argument to create an `MPI_Group` object containing the processes of the process set. Afterwards, the `MPI_Comm_create_from_group` function is called to create a communicator from this group.

In **libmpidynres**, the same mechanism is available to the simulated application. The `MPI_Comm_create_from_group` function simply wraps the `MPI_Comm_create_group` function and passes **libmpidynres**' communicator as the parent communicator. The `MPI_Comm_create_group` function is only collective over the members of the passed group. The same is therefore true for the `MPI_Comm_create_from_group` function. This has the disadvantage that when a group is passed to the function while one of the group members is in **libmpidynres**' idle state, the application runs into a deadlock situation. Ideally, an error value should be returned by this function instead which would be possible by additional communication with the resource manager. However, this approach would suffer from race conditions and **libmpidynres** assumes a simulated application with correct behavior anyway. Also, this deadlock can be easily detected with a debugger.

The `MPI_Group_from_session_pset` function is the most complex routine on the computing resource's side of **libmpidynres**. As all process set information is stored in the resource manager, it first has to be sent to the computing resource. The set is sent as an `int` array with respective size. Due to the immutability of process sets, a caching mechanism can be implemented either by the application (reusing the group object from the first call instead of making another call) or in future versions of **libmpidynres** by **libmpidynres** itself. Afterwards, the `MPI_Comm_group` function is called on **libmpidynres**' communicator to get the group used by **libmpidynres**. Finally, `MPI_Group_incl` is used with the group previously created and the received `int` array to get a group containing the right subset of the **libmpidynres** group that includes the processes contained in the process set.

4.5. Software Development Approach

In this section, the structure and features of **libmpidynres** are explained from a software development standpoint.

4.5.1. Debugging Features of libmpidynres

Debugging parallel applications is difficult. Due to non-deterministic behavior, bugs like race conditions can occur in one application run and disappear in the next one. To help the development of **libmpidynres** applications and **libmpidynres** itself, **libmpidynres** introduces a handful of features useful for tracing and finding bugs.

Error Handling

Most of **libmpidynres**' functions return either a 0 value on success or an integer $\neq 0$ on failure. If an irrecoverable error occurs, the operating system process is shutdown and a meaningful error together with the source line number and source file of the error will be printed to `stderr`.

Because **libmpidynres** is a separate library on top of MPI, it cannot throw custom MPI errors on its own. As a consequence, not all functions can satisfy their complete specification in the MPI-4.0 draft. For example, the `errhandler` argument to `MPI_Sessions_init` is ignored.

Debug Output

libmpidynres supports debug output to `stderr`. This feature can be enabled and disabled by setting or unsetting the `MPIDYNRES_DEBUG` environment variable.

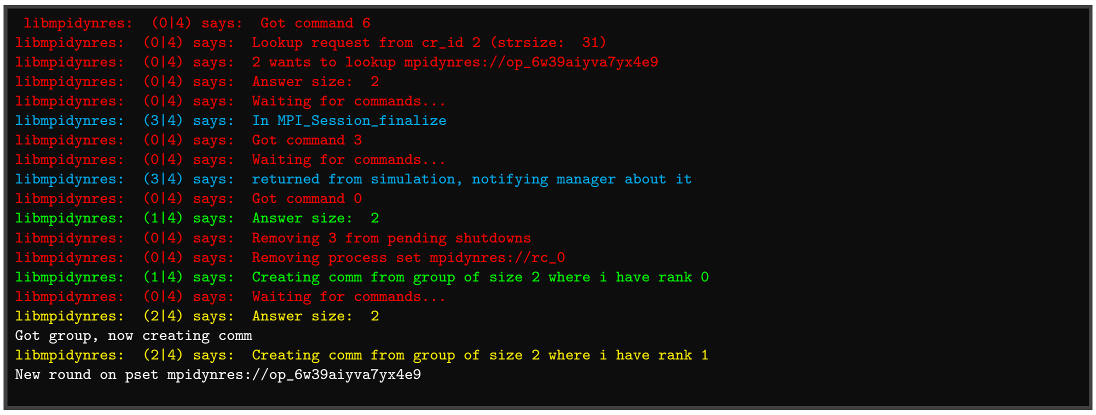
The debug output consists of a small prefix indicating the process rank in the **libmpidynres** communicator. Also, consecutive ranks will have their debug output printed in different colors to ease the debugging when all MPI processes print to a single terminal. Example debug output is shown in Figure 4.6.

In a GUI environment, it is generally recommended to start each process in a different terminal. This can be achieved by issuing:

```
$ mpirun <MPIRUN_ARGS> <TERMINAL_APPLICATION> -e ./myapp
```

instead of:

```
$ mpirun <MPIRUN_ARGS> ./myapp
```



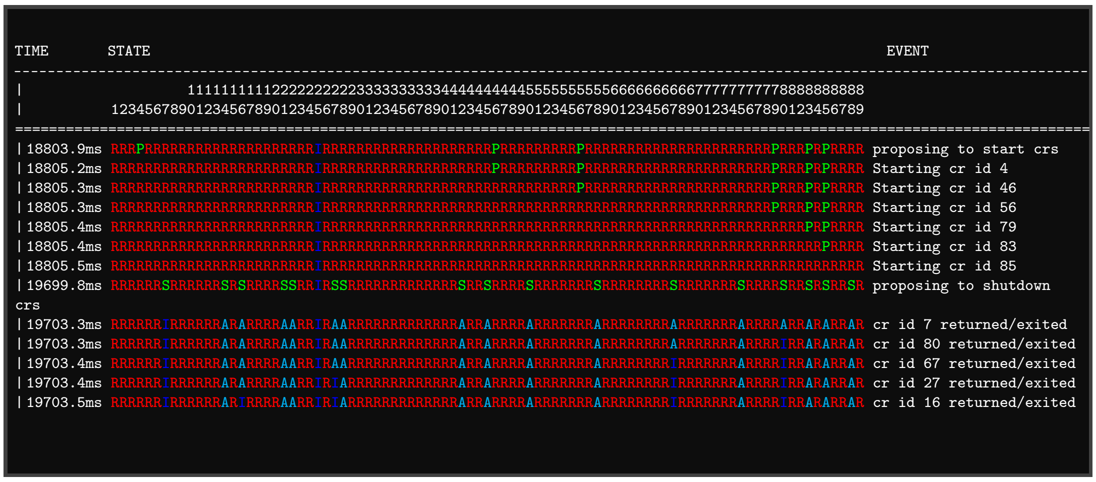
```
 libmpidynres:  (0|4) says:  Got command 6
libmpidynres:  (0|4) says:  Lookup request from cr_id 2 (strsize:  31)
libmpidynres:  (0|4) says:  2 wants to lookup mpidynres://op_6w39aiyva7yx4e9
libmpidynres:  (0|4) says:  Answer size:  2
libmpidynres:  (0|4) says:  Waiting for commands...
libmpidynres:  (3|4) says:  In MPI_Session_finalize
libmpidynres:  (0|4) says:  Got command 3
libmpidynres:  (0|4) says:  Waiting for commands...
libmpidynres:  (3|4) says:  returned from simulation, notifying manager about it
libmpidynres:  (0|4) says:  Got command 0
libmpidynres:  (1|4) says:  Answer size:  2
libmpidynres:  (0|4) says:  Removing 3 from pending shutdowns
libmpidynres:  (0|4) says:  Removing process set mpidynres://rc_0
libmpidynres:  (1|4) says:  Creating comm from group of size 2 where i have rank 0
libmpidynres:  (0|4) says:  Waiting for commands...
libmpidynres:  (2|4) says:  Answer size:  2
Got group, now creating comm
libmpidynres:  (2|4) says:  Creating comm from group of size 2 where i have rank 1
New round on pset mpidynres://op_6w39aiyva7yx4e9
```

Figure 4.6.: Colored debug output of **libmpidynres**. White text is application output.

Process State Logging

Another feature of **libmpidynres** is the logging and visualization of process state changes to a file. This feature can be enabled by setting the `MPIDYNRES_STATELOG` environment variable to a filename.

A line of the statelog output will contain the time passed from the start, a list of states for each process, each with a color and an uppercase letter, similar to the linux `top`/`htop` utilities, and a small text describing the event that lead to the state change. An example state log output is shown in Figure 4.7



```
TIME       STATE                                                                                      EVENT
----------------------------------------------------------------------------------------------------------------------------
|                   11111111112222222222333333333344444444445555555555666666666677777777778888888888
|          12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
============================================================================================================================
|18803.9ms RRRPRRRRRRRRRRRRRRRRRRRRIRRRRRRRRRRRRRRRRRRRRPRRRRRRRRRPRRRRRRRRRRRRRRRRRRRRRRPRRRPRPRRRR proposing to start crs
|18805.2ms RRRRRRRRRRRRRRRRRRRRRRRRIRRRRRRRRRRRRRRRRRRRRPRRRRRRRRRPRRRRRRRRRRRRRRRRRRRRRRPRRRPRPRRRR Starting cr id 4
|18805.3ms RRRRRRRRRRRRRRRRRRRRRRRRIRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRPRRRRRRRRRRRRRRRRRRRRRRPRRRPRPRRRR Starting cr id 46
|18805.3ms RRRRRRRRRRRRRRRRRRRRRRRRIRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRPRRRPRPRRRR Starting cr id 56
|18805.4ms RRRRRRRRRRRRRRRRRRRRRRRRIRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRPRPRRRR Starting cr id 79
|18805.4ms RRRRRRRRRRRRRRRRRRRRRRRRIRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRPRRRR Starting cr id 83
|18805.5ms RRRRRRRRRRRRRRRRRRRRRRRRIRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR Starting cr id 85
|19699.8ms RRRRRRSRRRRRRSRSRRRRSSRRIRSSRRRRRRRRRRRRRSRRSRRRRSRRRRRRRSRRRRRRRRSRRRRRRRSRRRRSRRSRSRRSR proposing to shutdown
crs
|19703.3ms RRRRRRIRRRRRRARARRRRAARRIRAARRRRRRRRRRRRRARRARRRRARRRRRRRARRRRRRRRARRRRRRRARRRRARRARARRAR cr id 7 returned/exited
|19703.3ms RRRRRRIRRRRRRARARRRRAARRIRAARRRRRRRRRRRRRARRARRRRARRRRRRRARRRRRRRRARRRRRRRARRRRIRRARARRAR cr id 80 returned/exited
|19703.4ms RRRRRRIRRRRRRARARRRRAARRIRAARRRRRRRRRRRRRARRARRRRARRRRRRRARRRRRRRRIRRRRRRRARRRRIRRARARRAR cr id 67 returned/exited
|19703.4ms RRRRRRIRRRRRRARARRRRAARRIRIARRRRRRRRRRRRRARRARRRRARRRRRRRARRRRRRRRIRRRRRRRARRRRIRRARARRAR cr id 27 returned/exited
|19703.5ms RRRRRRIRRRRRRARIRRRRAARRIRIARRRRRRRRRRRRRARRARRRRARRRRRRRARRRRRRRRIRRRRRRRARRRRIRRARARRAR cr id 16 returned/exited
```

Figure 4.7.: Example state log of **libmpidynres**. Legend: R=running, I=idle, P=reserved, S=proposed shutdown, A=accepted shutdown.

4.5.2. Source Code and Project Structure

During the course of this thesis, **libmpidynres** has grown from being a single C file with a small amount of functions to a larger software project, containing about 2.500 single lines of C code in 18 different files in its `src` directory and about 600 single lines of C code in its `example` directory.

To handle the increasing complexity and the management requirements, a solid project structure and source code structure was designed. The repository structure can be seen in Table 4.1.

Directory Item	Explanation
3rdparty	Third-party code. The ctl project headers used for data structures are in this directory (MIT license) [23].
build	Compiled code. Will be created during compilation.
doc	Documentation. Doxygen output will be generated in this directory.
examples	Examples. Multiple C files that show example applications with increasingly more features of **libmpidynres**.
public	Application Headers. Symbolic links to the headers that will be installed to the system (`mpidynres_sim.h`, `mpidynres.h`).
src	Source Code. The source code of **libmpidynres**.
tests	Unit Tests. Some unit tests are contained in this direcotry.
ARCHITECTURE.md	A Markdown file describing how to navigate the source code.
Makefile	The Makefile used for compiling, installing, testing, and documentation (see Appendix A).
README.md	A Markdown file giving a small introduction to the project.
run_tests.sh	A Bash script for running and evaluating unit tests from the `tests` directory.
.clang-format	`clangfmt` configuration file.

Table 4.1.: Directory items of the repository root.

Build artifacts either go into the `build` or the `doc` directory, depending on whether the source code is compiled or the documentation is built. The `Makefile` contains targets for building the source, installing the source into a POSIX-like system, building and running tests, building examples and generating and viewing the documentation. How these targets are used is explained in the Appendix A.

The source code is structured into logical components. Resource Manager code is prefixed with `scheduler_`. Application interface implementations begin with `mpidynres`. `comm.h` contains serialization code and MPI datatype construction code.

`logging.h` and `util.h` contain debugging and miscellaneous code.

4.5.3. Software Tools

The library repository is tracked via the version control tool `git`[29].

For linting and autoformatting C code, the tool `clangfmt` was used[5]. This assures a consistent code style among the source files. The code follows the Google C++ Style Guide[1].

For documenation, Doxygen-Style comments are contained in the source code. The tool Doxygen can then be used to extract these comments and output an HTML document[28].

The GNU Debugger (gdb) was used for debugging[11]. For this, Open MPI was compiled with debug symbols and **libmpidynres** with the compiler flags `-ggdb -O0`. Also, the Arm DDT debugger was used on the Linux Cluster of the Leibniz Rechenzentrum to debug **libmpidynres**[27]. The debugger is aimed for parallel programs and contains a GUI interface with access to the source code position and variables of the MPI processes. Valgrind's *memcheck* tool was used to detect potential memory leaks which can often occur when constructing MPI Objects and not freeing them afterwards[24].

To avoid time-wasting implementation and debugging of data structures, a third-party library called `ctl` was used[23]. `ctl` is a header-only C library which implements multiple container data structures.

4.5.4. Software Evaluation

The library was developed on a linux desktop system running on an *Intel Core i6200U* CPU, a dual core processor with four hardware threads. To support jobs containing more ranks, Open MPI's *oversubscribe* feature was used.

Furthermore, the library was run and tested on the Linux Cluster of the Leibniz Rechenzentrum with job sizes up to 256 cores in total using both `Open MPI` and `Intel MPI`.

[1]`https://google.github.io/styleguide/cppguide.html`

4.5.5. Included Examples

The `examples` directory contains multiple examples shown in Table 4.2. The examples can be compiled via `make examples`. The compiled binaries will be placed into the `build/examples` directory.

The examples contain commented source code showing different features of the **libmpidynres** API and should be read carefully by all users of **libmpidynres**.

Example	Explanation
01_hello_world	Example showing the Application Wrapper and the Simulation start.
02_sessions	Example showing how to create and finalize the `MPI_Session` object.
03_pset_handling	Example showing how to query and display process set information.
04_static_communicator	Example showing the creation of a functioning communicator from "mpi://WORLD".
05_pset_operations	Example showing how to create new process sets with process set operations.
06_resource_changes	Example showing how resource changes can be handled in a loop based application.
07_simple_changes	Simplified version of the `06_resource_changes` example. This application is presented in Chapter 6.

Table 4.2.: Examples contained in the **libmpidynres** repository.

5. Library Interface

In this chapter, the interface provided by **libmpidynres** is presented.

The functions are presented in a similar style to the one used in the MPI standard document[9]. For each function, the C signature is shown. Then, the function arguments are listed and explained. Arguments that are passed to the function and are not modified are marked as IN(for input). When the function modifies the object that an argument points to, the argument is marked as OUT(for output). If the argument is both used as an input and output argument, it is marked as INOUT. Also, a short overview of the differences between the MPI-4.0 draft version of the function und the **libmpidynres** version is given. These differences are either due to implementation restrictions (for example, the simulated layer cannot invoke errhandlers) or due to shortcomings (in the context of dynamic resources) of the MPI-4.0 functions proposed by the MPI Forum.

5.1. Environment Functions

```
int MPI_Session_init(MPI_Info info, MPI_Errhandler errhandler,
       MPI_Session *session);
```

IN	info	info object to be associated with the session
IN	errhandler	dummy argument for compatability
OUT	session	new session object

Differences to MPI-4.0 draft: The `info` argument does not have to contain the *mpi_thread_support_level* key and does not affect the behavior of **libmpidynres**. The `errhandler` argument is ignored.

The `MPI_Session_init` function is used to create a new `MPI_Session` object. It should be called at the beginning of the application before any MPI communication takes place. Multiple session objects can be constructed and nested in **libmpidynres**. However, the behavior of **libmpidynres** functions will not differ between the session objects. Future work may distinguish between different sessions using the session id that is sent to the resource manager with each call. The `info` argument currently does not affect the behavior of **libmpidynres**. Future work could define keys with a specific names to change the behavior of **libmpidynres**. However, the `info` argument will still be copied and stay associated with the session. The passed info object must still be freed by the application). The `errhandler` is currently unsupported as the `MPI_Errhandler` type is opaque to **libmpidynres** as it is built on top of MPI. As a consequence, **libmpidynres** cannot call the error handler itself.

```
int MPI_Session_finalize(MPI_Session *session);
```

INOUT	session	session to finalize

Differences to MPI-4.0 draft: The session argument passed was changed from `IN` to `INOUT` as the argument will be set to `MPI_SESSION_NULL`. `MPI_Session_finalize` is not collective in **libmpidynres**.

`MPI_Session_finalize` is used to destruct an MPI session and has to be called on each constructed session object before a process exits. It frees the `session` argument and replaces it with `MPI_SESSION_NULL`. Although the draft specifies the replacement of `session` with `MPI_SESSION_NULL`, the argument is marked as "IN" in the draft. This is fixed in this specification. The function call is not collective in **libmpidynres** because dynamic resources lead to the removal of only a subset of available processes at a time. If the call was collective, these processes would not be able to shut down correctly.

```
int MPI_Session_get_info(MPI_Session session, MPI_Info *info_used);
```

IN	session	session info to get info about
OUT	info_used	info associated with session

Differences to MPI-4.0 draft: No differences.

`MPI_Session_get_info` is used to obtain hints associated with the `session` argument. A newly created `MPI_Info` object is created that contains key-value pairs and returned in the `info_used` argument. The info object has to be freed by the application.

According to the MPI-4.0 draft, the key-value pairs returned are implementation defined but have to contain all implementation defined hints actively used or hints containing default values. While not explicitly stated in the draft, it is assumed the mentioned hints are the ones passed to `MPI_Session_init`.

In **libmpidynres**, there are no supported keys for `MPI_Session_init`'s `info` argument. Instead, all key-value pairs passed to the initialization function stay associated with the session object and will be contained in the `info` object returned by `MPI_Session_get_info`. This can prove useful for coordination among different software components sharing the same MPI process.

This function also plays a key role in resource change coordination and process discovery. The key-value pairs passed with the `info` object of `MPIDYNRES_RC_accept` (see section 5.5) will be available for processes started through a resource change. This can be useful to tell the newly created processes the name of the process set that the processes can use for communication with other application processes. This mechanism is used in the application demonstrated in Chapter 6.

```
void MPIDYNRES_exit();
```

Differences to MPI-4.0 draft: This function does not exist in the MPI-4.0 draft.

This function is **libmpidynres** specific and has the same effect as returning from the application entry function, i.e. terminating the simulated process. It should be used by the application instead of the `exit` function provided by the C standard library. The latter would lead the OS process to terminate and **libmpidynres** to break.

5.2. Process Set Discovery Functions

Process sets are used to group available processes and to manage dynamic resource changes. As previously shown in Figure 3.1, a process can be part of multiple process sets. The MPI-4.0 draft specifies how to query these process sets. The functions proposed in the draft seem to expect the available process sets to be of static nature and do no consider dynamic resources using process sets. As a consequence, the process set query mechanism was changed in **libmpidynres**.

```
int MPI_Session_get_psets(MPI_Session session, MPI_Info info,
       MPI_Info *psets);
```

IN	session	session used
IN	info	info object containing runtime hints
OUT	psets	info object containing process set names as keys and process set sizes as decimal values

Differences to MPI-4.0 draft: This function replaces `MPI_Session_get_num_pset` and `MPI_Session_get_nth_pset` from the MPI-4.0 draft.

The `MPI_Session_get_psets` function is used to query process sets that the calling process is part of. The `info` argument can be used to add hints to the query. In **libmpidynres**, there are no keys supported yet. In the future, a possible use case of the `info` argument could be used to filter the process sets by some criteria. The resulting process sets will be available in a new `MPI_Info` object in the `psets` argument which must be freed by the application. The keys of the `MPI_Info` object are the names of the process sets and the corresponding value is a string containing the decimal representation of the process set size , i.e. the number of processes contained in the process set. The string can be converted to an integer using the `atoi` function from the C standard library. Note that this mechanism implies that `MPI_MAX_PSET_NAME_LEN` $\leq$ `MPI_MAX_KEY_LEN`.

The MPI-4.0 draft uses `MPI_Session_get_num_keys` and `MPI_Session_get_nth_pset` to query process sets. The idea is to have a virtual array of process sets, to which the runtime can only append new entries. The `MPI_Session_get_num_keys` function returns the current length of the array and the `MPI_Session_get_nth_pset` returns the process set at a specific index. To avoid race conditions, the draft is very restrictive when it comes to changes to the array:

> "An MPI implementation is allowed to increase the number of available process sets during the execution of an MPI application when new process sets become available. However, MPI implementations are not allowed to change the index of a particular process set name, or to change the name of the process set at a particular index, or to delete a process set name once it has been added." (MPI-4.0 draft, p.496[10])

This strictness combined with the immutability and frequent change of process sets in **libmpidynres** would lead to a continuously growing array which would need an increasing amount of memory to maintain and, depending on the implementation, would lead to slower access times. Furthermore, during an application run with **libmpidynres**, the number of active process sets is expected to stay low. Consequently, the majority of the array's indices would consist of invalid process sets which already would have been freed by the resource manager.

The usage of one function to query the process sets removes the danger of a race condition during two separate calls (which is likely one of the reasons for the immutability of the virtual array in the MPI-4.0 draft). One issue that could be seen with this single call is that all process set names and sizes have to be provided by the runtime and transferred to the application which could be inefficient. However, as previously stated, the number of active process sets is expected to stay low in **libmpidynres**. This fact justifies that the answer contains all process sets. For increased efficiency, some filtering mechanism could be implemented using the `info` argument.

```
int MPI_Session_get_pset_info(MPI_Session session,
       const char *pset, MPI_Info *info);
```

IN	session	session used
IN	pset	name of process set to query
OUT	info	info associated with the process set

Differences to MPI-4.0 draft: No differences.

This function allows the application to query for properties of a process sets. The `pset` argument contains the name of the process set to query. The property will be returned in the `info` argument which has to be freed by the application.

libmpidynres supports the keys shown in Table 5.1.

Key	Process Set Type	Value
`mpi_size`	all	Decimal representation of number of processes contained in the process set.
`mpidynres_name`	all	The name of the process set.
`mpidynres_op`	created by pset operation	The type of operation applied (One of `union`, `intersect`, `difference`).
`mpidynres_op_parent1`	created by pset operation	First argument to the process set operation.
`mpidynres_op_parent2`	created by pset operation	Second argument to the process set operation.
`mpidynres_rc`	created by resource change	Always the value `true`.
`mpidynres_rc_type`	created by resource change	The type of resource change (either `add` or `sub`).
`mpidynres_rc_tag`	created by resource change	Decimal representation of tag used for resource change.

Table 5.1.: Info keys supported for process sets.

5.3. Group and Communicator Functions

```
int MPI_Group_from_session_pset(MPI_Session session,
       const char *pset, MPI_Group *group);
```

IN	session	session used
IN	pset	name of process set to create group from
OUT	group	newly created group

Differences to MPI-4.0 draft: No differences.

`MPI_Group_from_session_pset` is used to create a new group containing the processes in the process set with the name passed in the `pset` argument. If an error occurs, `group` will be set to `MPI_GROUP_EMPTY` and a non-zero return value are returned.

```
int MPI_Comm_create_from_group(MPI_Group group, const char *stringtag,
       MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *comm);
```

IN	group	group to create communicator from
IN	stringtag	dummy argument for compatibility
IN	info	optional hints
IN	errhandler	error handler that will be attached to the communicator
OUT	comm	newly created communicator

Differences to MPI-4.0 draft: Here, the errhandler will not be invoked on an error in this function. The `stringtag` argument is ignored. Currently, no keys are supported for the *info* argument.

The `MPI_Comm_create_from_group` function is used to construct a new communicator containing the processes in the `group` argument. This function allows to create communicators solely based on the group and no parent communicator is needed. It is collective over the processes contained in the group.

In **libmpidynres**, this function will lock when the `group` argument contains a process that is currently in the **libmpidynres** idle state.

According to the MPI-4.0 draft, the error handler should be invoked if an error occurs during the function call and should be attached to the new communicator. Due to the opaqueness of the `MPI_Errhandler` type, **libmpidynres** cannot invoke the error handler itself. However, it will be attached to the communicator. The `stringtag` argument is ignored by **libmpidynres** as it does not support multithreading.

5.4. Process Set Management Functions

Process Sets are created in three scenarios: When the simulated environment starts, the process sets "mpi://WORLD" and "mpi://SELF" are created. More process sets can be created by the application by applying set operations to existing process sets. Finally, process sets are created by **libmpidynres** for each resource change.

```
int MPIDYNRES_pset_create_op(MPI_Session session, MPI_Info hints,
        const char pset1[], const char pset2[],
        MPIDYNRES_pset_op op, char pset_result[]);
```

IN	session	session used
IN	hints	hints passed to runtime
IN	pset1	name of first argument to process set operation
IN	pset2	name of second argument to process set operation
IN	op	operation type to apply
OUT	pset_result	name or resulting process set

Differences to MPI-4.0 draft: This function is not part of the MPI-4.0 draft. Process set management is implementation defined in the MPI-4.0 draft.

In **libmpidynres**, the application can ask the runtime to create new process sets by combining existing process sets. For that, the function `MPIDYNRES_pset_create_op` is used. The function is not collective and should only be called by a single process for a specific process set operation. Consequently, the call has to be coordinated in the application. The arguments `pset1` and `pset2` specify the two process set operands of the set operations. The `op` argument specifies the set operation to apply. It has to take one of the following values: `MPIDYNRES_PSET_UNION`, `MPIDYNRES_PSET_INTERSECT`, `MPIDYNRES_PSET_DIFFERENCE`. The resulting process set name will be returned in the `pset_result` argument and has the form of "mpidynres://op_" followed by a random string. Therefore, `pset_result` has to be able to hold at least `MPI_MAX_PSET_NAME_LEN` characters. The `hints` argument currently supports the key `mpidynres_proposed_name` which can be used to ask the resource manager to name the new process set by the given value. If the process set operation fails, a string of zero length will be placed in `pset_result`.

The process set operation mechanism is useful for querying and creating communicators that combine process sets directly through a new process set. This process name can be shared by the application by message passing. For example, the application shown in Chapter 6 uses process set operations to combine resource change process sets with the process set that contains all other active ranks to create a new process set containing all active processes including the new, dynamically created ones.

An alternative interface design could consist of creating communicators by combining MPI groups with existing MPI functions (`MPI_Group_union`, `MPI_Group_intersection`, `MPI_Group_difference`) together with `MPI_Comm_create_from_group`. Additionally, a function for creating process sets from MPI groups would be necessary to ease the creation and usage of the same communicator on future processes. Future work could try to implement this interface.

```
int MPIDYNRES_pset_free(MPI_Session session, char pset[]);
```

IN	session	session used
INOUT	pset	the name of the process set to be freed

Differences to MPI-4.0 draft: This function is not part of the MPI-4.0 draft. Process set management is implementation defined in the MPI-4.0 draft.

This function can be used to explicitly free a process set. Note that process sets are freed implicitly by **libmpidynres** once one of the processes that is part of the process set exits. Communicators based on the process set given in the `pset` argument stay valid as long as all processes are in the **libmpidynres** active state.

The `MPIDYNRES_pset_free` function is not collective and should only be called by one process to free a specific process set.

5.5. Resource Change Management Functions

Resource Changes are proposed by the runtime and have to be accepted by the application. The application can query for new resource changes. A new resource change will then be exposed as a new process set. Process set operations can be applied on the new process set to create common communicators. Before accepting the resource change, the application has time to clean up and rebalance its workload.

In **libmpidynres**, resource change function call will be sent to the resource manager which then calls the `scheduler_mgmt.h` interface which then does scheduling decisions (see 4.3.4). This design allows for easy extension of **libmpidynres** by creating new scheduling mechanisms.

It is important to note that the resource change related functions are not collective and should be only invoked by a single process for a specific resource change.

```
int MPIDYNRES_add_scheduling_hints(MPI_Session session,
       MPI_Info hints, MPI_Info *answer);
```

IN	session	session used
IN	hints	hints passed to the scheduling component of **libmpidynres**
OUT	answer	optional answer by scheduling component

Differences to MPI-4.0 draft: This function is not part of the MPI-4.0 draft. Resource change management is not specified in the MPI-4.0 draft.

This function can be used to give the scheduling part of the runtime clues with the `hints` argument. This can be used to dynamically change scheduling parameters or notify the scheduler about resource requirements. The latter aspect allows **libmpidynres** to adapt dynamically to the resource requirements of the application. To keep the interface general purpose, an arbitrary info object can be sent back as an answer by the scheduler. Note that the answer can also be `MPI_INFO_NULL`.

The included scheduling implementations (see Section 4.3.4) currently do not support this currently. Application-provided resource requirements can be part of future work of **libmpidynres**.

Both info objects have to be freed by the application if they are not equal to `MPI_INFO_NULL`.

```
int MPIDYNRES_RC_get(MPI_Session session,
       MPIDYNRES_RC_type *rc_type, char delta_pset[],
       MPIDYNRES_RC_tag *tag, MPI_Info *info);
```

IN	session	session used
OUT	rc_type	type of resource change
OUT	delta_pset	name of the new resource change process set
OUT	tag	identifier for the resource change
OUT	info	optional additional information about the resource change

Differences to MPI-4.0 draft: This function is not part of the MPI-4.0 draft. Resource change management is not specified in the MPI-4.0 draft.

The `MPIDYNRES_RC_get` function is used to query for a resource change. It should be called by the application on a regular basis, e.g., during each loop iteration. It is not collective and should only be called by one process at a time.

There are multiple values returned by the runtime. The `rc_type` return value will tell the application what kind of resource change is about to happen (one of `MPIDYNRES_RC_NONE`, `MPIDYNRES_RC_ADD`, `MPIDYNRES_RC_SUB`).

If the type is not equal to `MPIDYNRES_RC_NONE`, there are further return values. The `delta_pset` return value will contain the name of the resource change process set. In the case of resource removal, it will contain the processes that are currently active and part of the application that will need to shut down. In the case of resource addition, it will contain the processes that will be started once the resource change is accepted. The `tag` return value is used as an identifier for the resource change and has to be passed to the `MPIDYNRES_RC_accept` function. The `info` argument can hold additional information from the scheduler about the resource change. In the two example schedulers provided by **libmpidynres** this value will always be `MPI_INFO_NULL`. Future use cases of this object could be to pass information about the reason or trigger for the resource change. If a previous resource change accept is still pending or there are processes that should be removed and have not exited yet the function will return `MPIDYNRES_RC_NONE`. The `info` return value has to be freed by the application if it is not equal to `MPI_INFO_NULL`.

```
int MPIDYNRES_RC_accept(MPI_Session session, MPIDYNRES_RC_tag tag,
       MPI_Info info);
```

IN	session	session used
IN	tag	identifier of the resource change to accept
IN	info	runtime hints and hints for newly created processes

Differences to MPI-4.0 draft: This function is not part of the MPI-4.0 draft. Resource change management is not specified in the MPI-4.0 draft.

The `MPIDYNRES_RC_accept` function is used to accept a resource change. Before calling this function, the application should make sure that important load balancing steps have taken place and in the case of resource removal, the processes that need to shut down are about to shut down. In the case of resource addition, the new processes will be started once this function is called.

The `tag` argument should contain the `rc_tag` returned by the `MPIDYNRES_RC_get` function. The `info` argument can contain hints for the scheduler. Furthermore, in the case of resource addition, the key-value pairs passed in the info object will be available for newly started processes when calling `MPI_Session_get_info`. This is crucial for process coordination as important information concerning the application state and process set names for communication can be passed that way.

6. Application Demonstration

In this chapter, an example application is shown that uses the **libmpidynres** library to handle resource changes. The example application is based on a simple loop where a check for resource changes takes place during each iteration. The code is divided into multiple snippets of C code with a short explanation underneath each. Note that for readability, error checking has been omitted. This example application can be found in the `src/examples` directory and is named `07_simple_changes`. A more complex version with error checking can be found in the same directory under the name `06_resource_changes`.

The general idea of the application is to have all available processes together in a common communicator. The work is then split between the processes in the communicator. If resources are added, the application uses a `union` operation to combine the current main process set with the new delta process set. If resources are removed, the application uses a `difference` operation to remove the processes from the delta set from the main set. Then, a new communicator can be created from the resulting process set.

When accepting a resource change, the name of the main process set and application-specific information will be passed to **libmpidynres**. Newly spawned processes can then access the information and construct the main communicator.

Process set operations and resource change functions are called by rank 0 in the main communicator. The results of these function calls are broadcasted to the other processes of the communicator.

Globals

```
MPI_Session session;  // session object
MPI_Comm main_comm;   // the communicator where all the work happens
int main_rank;        // rank in main_comm
int main_iter;        // current application iteration
char main_pset[MPI_MAX_PSET_NAME_LEN];

// return whether info object contains key
bool info_contains(MPI_Info info, const char *key);
```

The application will keep track of its current state in global variables. These variables contain information about the current loop iteraton (main_iter) and the current communicator used (main_comm, main_rank, main_pset)

The function info_contains returns a boolean value indicating whether the string key is contained in the info argument. Its implementaton is omitted here.

Main Communicator Creation

```
void update_main_comm() {
  MPI_Group mygroup;
  MPI_Group_from_session_pset(session, main_pset, &mygroup);
  MPI_Comm_create_from_group(mygroup, NULL, MPI_INFO_NULL,
   ↪  MPI_ERRORS_ARE_FATAL, &main_comm);
  MPI_Comm_rank(main_comm, &main_rank);
  MPI_Group_free(&mygroup);
}
```

The update_main_comm function is used to create a new communicator once main_pset has changed. The function uses the MPI Sessions mechanism to create a group from a process set name and then a communicator from that group. As the functions for this mechanism are collective, the update_main_comm function is collective aswell.

The function is called once when a process is initializing and once for every resource change.

Main Loop

```
int application_entry(int argc, char *argv[]) {
  bool need_to_break = false;

  initialization_phase();

  for (; main_iter < 1000 && !need_to_break; main_iter++) {
    work_step();
    resource_changes_step(&need_to_break);
    rebalance_step();
  }

  finalization_phase();

  return EXIT_SUCCESS;
}
```

This function is the entry point passed to **libmpidynres** as described in Section 4.1.1.

It is divided into multiple phases. At the beginning, the MPI Sessions environment, the global state and application specific variables are initialized in the `initialization_phase`. At the end, these variables are freed in the `finalization_phase`.

In between, the main loop is run. In this example, the main loop uses 1000 iterations. During each iteration, the following steps are being executed:

- *work step*: Here, application specific work and MPI communication takes place. This function is application specific and its implementation is omitted here.
- *resource change step*: In this step, resource changes are queried and handled. If the current process needs to be removed for a resource change, the variable `need_to_break` is set. A closer look on this step is taken below.
- *rebalance step*: After the number of processes has changed, the application may need to rebalance its workload. This happens here. The implementation of this step is also omitted here.

Initialization & Finalization Phases

```c
void initialization_phase() {
  MPI_Info session_info, psets;
  int unused;
  char main_iter_buf[0x20];

  MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &session);

  MPI_Session_get_psets(session, MPI_INFO_NULL, &psets);

  if (info_contains(psets, "mpi://WORLD")) {
    strcpy(main_pset, "mpi://WORLD");
    main_iter = 0;
  } else {
    MPI_Session_get_info(session, &session_info);

    MPI_Info_get(session_info, "app_main_pset", MPI_MAX_PSET_NAME_LEN - 1,
     ↪  main_pset, &unused);
    MPI_Info_get(session_info, "app_main_iter", 0x20 - 1, main_iter_buf,
     ↪  &unused);
    main_iter = atoi(main_iter_buf);
    MPI_Info_free(&session_info);
  }

  // create main_comm from main_pset (collective!)
  update_main_comm();

  // application specific initialization/load balancing here

  MPI_Info_free(&psets);
}
```

In the initialization phase, the process needs to find out if the process invocation has been dynamic (based on a resource change) or initial (invoked with the application start). For this, the process queries its process sets and checks whether its part of "mpi://WORLD". In the case of the initial invocation, the "mpi://WORLD" process set is used as the main process set and the `main_iter` variable is set to zero. If it is a dynamic process start, the session information is queried with the `MPI_Session_get_info` function. In **libmpidynres**, the returned info object will hold keys passed by the application to the `MPIDYNRES_RC_accept`. In this way, the process can find out the current loop iteration and the current main process set.

Once `main_pset` has been set, `update_main_comm` is used to create a communicator from `main_pset`.

After the initializtion phase, the process has a communicator that connects all active processes of the application. The process also knows at which loop iteration the application is.

```c
void finalization_phase() {
  // application specific cleanup here

  // cleanup
  MPI_Comm_free(&main_comm);
  MPI_Session_finalize(&session);
}
```

The finalization phase is used for application specific cleanup and to free MPI objects.

Resource Change Step

```
void resource_changes_step(bool *need_to_break) {
  int rc_tag;
  MPIDYNRES_RC_type rc_type;
  char delta_pset[MPI_MAX_PSET_NAME_LEN];

  MPI_Barrier(main_comm);
  fetch_resource_changes(&rc_tag, &rc_type, delta_pset);
  handle_resource_changes(rc_type, delta_pset, need_to_break);
  accept_resource_change(rc_tag);
  if (rc_type != MPIDYNRES_RC_NONE && !*need_to_break) {
    update_main_comm();
  }
}
```

The resource change step can be subdivided into three steps:

- *fetch_resource_changes:* In this step, the application queries for a new resource change. A new resource change process set will be created (the `delta_pset` variable) and a resource change type will be returned.
- *handle_resource_changes:* In this step, the new resource change will be handled. The next `main_pset` is created by combining the current `main_pset` with the `delta_pset`.
- *accept_resource_changes:* Finally, the resource change is accepted and processes that need to shutdown break the main loop and new processes are started.

After these steps, the `main_pset` may have changed. Because of that, a new communicator is created with a call to `update_main_comm`. This call will be collective over the active processes and the newly started resources in the case of resource addition.

```
void fetch_resource_changes(int *rc_tag, MPIDYNRES_RC_type *rc_type,
                            char delta_pset[]) {
  MPI_Info rc_info;

  if (main_rank == 0) {
    MPIDYNRES_RC_get(session, rc_type, delta_pset, rc_tag, &rc_info);
    if (rc_info != MPI_INFO_NULL) {
      MPI_Info_free(&rc_info);
    }
  }
  MPI_Bcast(rc_type, 1, MPI_INT, 0, main_comm);
  MPI_Bcast(rc_tag, 1, MPI_INT, 0, main_comm);
  MPI_Bcast(delta_pset, MPI_MAX_PSET_NAME_LEN, MPI_CHAR, 0, main_comm);
}
```

In the `fetch_resource_changes` step, rank 0 will query **libmpidynres** for resource changes. These resource changes will then be broadcasted to the other ranks.

```
void handle_resource_changes(MPIDYNRES_RC_type rc_type, char delta_pset[],
                             bool *need_to_break) {
  MPI_Info mypsets;

  switch (rc_type) {
    case MPIDYNRES_RC_ADD: {
      if (main_rank == 0) {
        MPIDYNRES_pset_create_op(session, MPI_INFO_NULL, main_pset,
         ↪  delta_pset, MPIDYNRES_PSET_UNION, main_pset);
      }
      MPI_Bcast(main_pset, MPI_MAX_PSET_NAME_LEN, MPI_CHAR, 0, main_comm);
      break;
    }
    case MPIDYNRES_RC_SUB: {
      if (main_rank == 0) {
        MPIDYNRES_pset_create_op(session, MPI_INFO_NULL, main_pset,
         ↪  delta_pset, MPIDYNRES_PSET_DIFFERENCE, main_pset);
      }
      MPI_Bcast(main_pset, MPI_MAX_PSET_NAME_LEN, MPI_CHAR, 0, main_comm);

      MPI_Session_get_psets(session, MPI_INFO_NULL, &mypsets);
      if (!info_contains(mypsets, main_pset)) {
        *need_to_break = true;
      }
      MPI_Info_free(&mypsets);
      break;
    }
    case MPIDYNRES_RC_NONE {
      break;
    }
  }
}
```

The `handle_resource_changes` step depends on the type of resource change. In this code, a `switch-case` statement is used to change the behavior based on the type. In the case of an addition of resources, a new process set is created by constructing the union set of both the current main process set and the resource change set. In the case of a resource removal, a new process set is created by constructing the difference set of the current main process set and the resource change set. Furthermore, for resource removal, each process checks whether it is part of this new difference set. If not, the process will have to be removed. In that case, the variable `need_to_break` is set.

```
void accept_resource_change(MPIDYNRES_RC_tag rc_tag) {

  if (main_rank == 0) {
    MPI_Info new_processes_info;
    char buf[0x20];

    MPI_Info_create(&new_processes_info);
    MPI_Info_set(new_processes_info, "app_main_pset", main_pset);
    snprintf(buf, 0x1f, "%d\n", main_iter);
    MPI_Info_set(new_processes_info, "app_main_iter", buf);

    // more application specific keys can be set here

    MPIDYNRES_RC_accept(session, rc_tag, new_processes_info);

    MPI_Info_free(&new_processes_info);
  }
}
```

Finally, in the `accept_resource_change` step, the `MPIDYNRES_RC_accept` function is called by rank 0 of the main communicator. An info object containing the current application iteration and the next main process set is passesd to this function meaning it will be available for newly created processes during the `initialization` phase.

7. Conclusion

7.1. Summary

A new interface for dynamic resource changes using the process set concept of MPI Sessions was presented. Using MPI, a C library was implemented to provide a simulated dynamic resource environment in which applications can use this interface. Due to the library acting as an additional layer between MPI and the application, it is portable and can be used on any valid MPI implementation. The library also provides a simple interface for creating new scheduling mechanisms which helps in testing the resource adaptivity of applications in different scenarios. Two scheduler implementations are included in this work.

To help programmers use this new C library, multiple examples are included providing an incremental introduction into features of **libmpidynres**. Furthermore, the library provides additional debugging features to help programmers locating their implementation errors.

A parallel loop based application was presented in this work that successfully uses the new interface to query and apply new resource changes. At the same time, the application makes sure that all processes stay connected and coordinated. Also, the library exposes the MPI Sessions API to the example application. The application can create and finalize sessions, query session info and manage and use process sets.

7.2. Future Work

This work only contains a proof-of-concept library. To integrate dynamic resources into supercomputers, support has to be added to all layers of the MPI runtime stack.

Also, more work must be done on the MPI Sessions interface, especially in regard to multiple sessions in the same application and to process set logic. Should there be a distrinction between static or dynamic process sets? Does the current interface for querying process sets make sense?

Finally, **libmpidynres** can be further extended and used to experiment with the MPI Sessions interface. Multiple application support and a multithreaded resource manager are possible extensions for the future.

A. Build Instructions and Usage

A.1. Building and Installation

Please make sure that you have the source code of **libmpidynres** available. It should be contained in a directory called `libmpidynres` with a Makefile and a `src` subdirectory. Also, make sure that your system meets the following requirements:

- It is running a unix-like operating system (with posix shell utilities and a FHS-compatible filesystem).
- An MPI library and an MPI compiler (ideally Open MPI) are installed.
- The software package `rsync` is installed.
- The software package `make` is installed (tested with `gnumake`).

Before building **libmpidynres**, take a look at the variables set at the beginning of the Makefile and adjust them to your liking (for example adjust the `INSTALL_PREFIX` variable to change the installation root).

To build **libmpidynres**, run:

```
$ make
```

This will install the file `libmpidynres.so` into the `build/lib` directory and the headers into the `build/include` directory.

To install **libmpidynres**, run:

```
$ make install
```

This will install the following files:

- `$INSTALL_PREFIX/lib/libmpidynres.so`
- `$INSTALL_PREFIX/include/mpidynres.h`
- `$INSTALL_PREFIX/include/mpidynres_sim.h`

To update **libmpidynres**, run the same steps as above with newer source code. To remove **libmpidynres** from your system, delete the listed files manually.

A.2. Documentation, Tests and Examples

Documentation

To build the documentation, make sure to have `doxygen` installed. You can build the documentation by running:

```
$ make docs
```

This will install an HTML and a Latex version of the documentation into the `doc` folder. You can run

```
$ make viewdocs
```

to open the HTML version in firefox.

Tests

There are two unit tests for the serialization of the `MPI_Info` object. These can be found in the `tests` subdirectory.

To build the tests, run:

```
$ make tests
```

To run the tests, run:

```
$ make test
```

Examples

Multiple examples are included in the `examples` subdirectory. These illustrate how to use the library in an application.

To build the examples, run:

```
$ make examples
```

The example binaries are placed in the `build/examples` directory.

A.3. Usage and Debugging

The examples illustrate how an application can use the interface provided by **libmpidynres**. Chapter 5 describes each function available inside the simulated runtime. Before building your application, make sure you have installed the library into your system at a location where your MPI compiler can find it.

Two scheduling implementations are shipped with the **libmpidynres** source code. These can be found in the `src/managers` directory and are futher explained in Section 4.3.4.

To use the increasing-decreasing scheduler, run

```
$ ln -sv managers/inc_dec_manager.c scheduler_mgmt.c
```

in the `src` directory.

To use the random-difference scheduler, run

```
$ ln -sv managers/inc_dec_manager.c scheduler_mgmt.c
```

in the `src` directory. The `-f` flag can be added to overwrite an existing symlink.

If your application is contained in a C file called `application.c` you can compile it with the following command:

```
$ mpicc <COMPILER OPTIONS> application.c -o application -lmpidynres
```

Now you can run your application by issuing:

```
$ mpirun <MPIRUN_OPTIONS> ./application
```

Refer to the manual of `mpirun` for possible options.

By setting the environment variable `MPIDYNRES_DEBUG` before the application start, **libmpidynres** will output colorful debug output to `stderr`. By setting the environment variable `MPIDYNRES_STATELOG` to a filename, **libmpidynres** will output a log of the process states to the filename provided. These features are also explained further in Section 4.5.1.

Bibliography

[1] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. *PETSc Web page*. `https://www.mcs.anl.gov/petsc`. 2019.

[2] G. E. P. Box and M. E. Muller. "A Note on the Generation of Random Normal Deviates." In: *The Annals of Mathematical Statistics* 29.2 (1958), pp. 610–611. DOI: `10.1214/aoms/1177706645`.

[3] 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`.

[4] S. Chakraborty. "Job Startup at ExaScale: Challenges and Solutions." In: Sept. 2016.

[5] *ClangFormat — Clang 12 documentation - LLVM*. `https://clang.llvm.org/docs/ClangFormat.html`.

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

[7] T. M. Forum. "MPI: A Message-Passing Interface Standard Version 1.3." In: May 2008.

[8] T. M. Forum. "MPI: A Message-Passing Interface Standard Version 2.2." In: Sept. 2009.

[9] T. M. Forum. "MPI: A Message-Passing Interface Standard Version 3.1." In: June 2015.

[10] T. M. Forum. "MPI: A Message-Passing Interface Standard Version 4.0 (Draft)." In: Nov. 2020.

[11] *GDB: The GNU Project Debugger*. `https://www.gnu.org/software/gdb/`.

[12] G. Geist, J. A. Kohl, and P. M. Papadopoulos. "PVM and MPI: A comparison of features." In: *Calculateurs Paralleles* 8.2 (1996), pp. 137–150.

[13] M. Gerndt, A. Hollmann, M. Meyer, M. Schreiber, and J. Weidendorfer. "Invasive computing with iOMP." In: *Proceeding of the 2012 Forum on Specification and Design Languages*. 2012, pp. 225–231.

[14] P. GmbH. *Top 500 November 2000*. `https://www.top500.org/lists/top500/2000/11/`. 2020.

[15] P. GmbH. *Top 500 November 2020*. `https://www.top500.org/lists/top500/2020/11/`. 2020.

[16] N. W. Group. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. RFC Editor, Jan. 2005.

[17] D. Hans, B. Joachim, M. Zimmer, and D. Buchhol. *Modeling and simulation: an application-oriented introduction*. 2013.

[18] N. Hjelm, H. Pritchard, S. K. Gutiérrez, D. J. Holmes, R. Castain, and A. Skjellum. *MPI Sessions Open MPI Prototype*. `https://github.com/hpc/ompi/tree/sessions_new`. 2021.

[19] 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`.

[20] D. Holmes, K. Mohror, R. Grant, A. Skjellum, M. Schulz, W. Bland, and J. Squyres. "MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale." In: Sept. 2016, pp. 121–129. DOI: `10.1145/2966884.2966915`.

[21] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Third. Boston: Addison-Wesley, 1997. ISBN: 0201896842 9780201896848.

[22] *Kubernetes Project Homepage*. `https://kubernetes.io`.

[23] G. Louw. *ctl - The C Template Library*. `https://github.com/glouw/ctl`. 2021.

[24] *Memcheck: a memory error detector*. `https://valgrind.org/docs/manual/mc-manual.html`.

[25] *MPI-4.0*. `https://www.mpi-forum.org/mpi-40/`.

[26] *MPICH | High-Performance Portable MPI*. `https://www.mpich.org/`.

[27] *Official Website of the ARM DDT Tool*. `https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt`.

[28] *Official Website of the Doxygen Documentation Generator*. `https://www.doxygen.nl/index.html`.

[29] *Official Website of the GIT Version Control System*. `https://git-scm.com/`.

[30] *Open MPI source code on Github opal_progress.c.* `https://github.com/open-mpi/ompi/blob/26c136ae59e2235b1187fd3d9897498860320d68/opal/runtime/opal_progress.c`.

[31] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 5.1.* `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf`. Nov. 2020.

[32] T. O. M. Project. *Open MPI: Open Source High Performance Computing.* `https://www.open-mpi.org/`.

[33] M. Schreiber, C. Riesinger, T. Neckel, and H. Bungartz. "Invasive Compute Balancing for Applications with Hybrid Parallelization." In: *2013 25th International Symposium on Computer Architecture and High Performance Computing*. 2013, pp. 136–143. DOI: `10.1109/SBAC-PAD.2013.20`.

[34] M. Schreiber. "Cluster-Based Parallelization of Simulations on Dynamically Adaptive Grids and Dynamic Resource Management." PhD thesis. Technische Universität München, 2014.

[35] 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`.

[36] C. The MPI Forum. "MPI: A Message Passing Interface." In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 878–883. ISBN: 0818643404. DOI: `10.1145/169627.169855`.

[37] *Transregional Collaborative Research Centre 89 — Invasive Computing.* `http://invasic.informatik.uni-erlangen.de/`.