Towards Dynamic Resource Management with MPI Sessions and PMIx

Dominik Huber, Maximilian Streubel, Isaías Comprés, Martin Schulz, Martin Schreiber, Howard Pritchard

▶ **To cite this version:**

Dominik Huber, Maximilian Streubel, Isaías Comprés, Martin Schulz, Martin Schreiber, et al.. Towards Dynamic Resource Management with MPI Sessions and PMIx. EuroMPI/USA'22 - 29th European MPI Users' Group Meeting, Sep 2022, Chattanooga, United States. pp.57-67, 10.1145/3555819.3555856 . hal-03856697

HAL Id: hal-03856697
https://hal.science/hal-03856697

Submitted on 16 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

This is a preprint version of our paper.
Please find the official one at
https://doi.org/10.1145/3555819.3555856

Towards Dynamic Resource Management with MPI Sessions and PMIx

Dominik Huber
Maximilian Streubel
Isaías Comprés
Martin Schulz
domi.huber@tum.de
maximilian.streubel@tum.de
isaias.compres@tum.de
schulzm@in.tum.de
Technical University of Munich
Garching, Germany

Martin Schreiber
martin.schreiber@univ-grenoble-alpes.fr
Universit Grenoble Alpes &
Laboratoire Jean Kuntzmann
Saint Martin d'Hres, France
Technical University of Munich
Garching, Germany

Howard Pritchard
howardp@lanl.gov
Los Alamos National Laboratory
USA

Abstract

Job management software on peta- and exascale supercomputers continues to provide static resource allocations, from a program's start until its end. Dynamic resource allocation and management is a research direction that has the potential to improve the efficiency of HPC systems and applications by dynamically adapting the resources of an application during its runtime. Resources can be adapted based on past, current or even future system conditions and matching optimization targets. However, the implementation of dynamic resource management is challenging as it requires support across many layers of the software stack, including the programming model.

In this paper, we focus on the latter and present our approach to extend MPI Sessions to support dynamic resource allocations within MPI applications. While some forms of dynamicity already exist in MPI, it is currently limited by requiring global synchronization, being application or application-domain specific, or by suffering from limited support in current HPC system software stacks.

We overcome these limitations with a simple, yet powerful abstraction: resources as process sets, and changes of resources as set operations leading to a graph-based perspective on resource changes. As the main contribution of this work, we provide an implementation of this approach based on MPI Sessions and PMIx. In addition, an illustration of its usage is provided, as well as a discussion about the required extensions of the PMIx standard. We report results based on a prototype implementation with Open MPI using a synthetic application, as well as a PDE solver benchmark on up to four nodes and a total of 112 cores. Overall, our results show the feasibility of our approach, which has only very moderate overheads. We see this first proof-of-concept as an important step towards resource adaptivity based on MPI Sessions.

Keywords

MPI Sessions, dynamic resources, malleability, PMIx

ACM Reference Format:
Dominik Huber, Maximilian Streubel, Isaías Comprés, Martin Schulz, Martin Schreiber, and Howard Pritchard. 2022. Towards Dynamic Resource Management with MPI Sessions and PMIx. In *EuroMPI/USA'22: 29th European MPI Users' Group Meeting (EuroMPI/USA'22), September 26–28, 2022, Chattanooga, TN, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3555819.3555856

1 Introduction

In the last decade the performance of HPC systems has increased significantly, with first systems (officially) reaching exascale in 2022. This is driven by heterogeneous and accelerated architectures, which require a careful resource management and resource allocation to application types and kernels. However, traditionally, resources are assigned statically at the start of a job and are then fixed for the duration of the job, which is often a limitation for the efficiency of resource usage on HPC systems due to variations in the scalability of applications, algorithmic kernels, intermediate idle times, or different suitability to accelerators, to name just a few issues.

As a consequence, resource management must become more dynamic and adaptive in order to be able to counteract these challenges. We will refer to this in the remainder of this paper as *dynamicity*, which has been shown to improve overall system efficiency with respect to various optimization goals, such as throughput, energy or job queue latency [13, 19, 20, 23]. Dynamicity can also be advantageous for individual applications: Avoiding a-priori resource fixation and using the global scheduler perspective can improve individual resource efficiency and support urgent computing [6, 15, 16, 20].

However, achieving dynamicity is challenging, since it not only increases the complexity of the system scheduler and

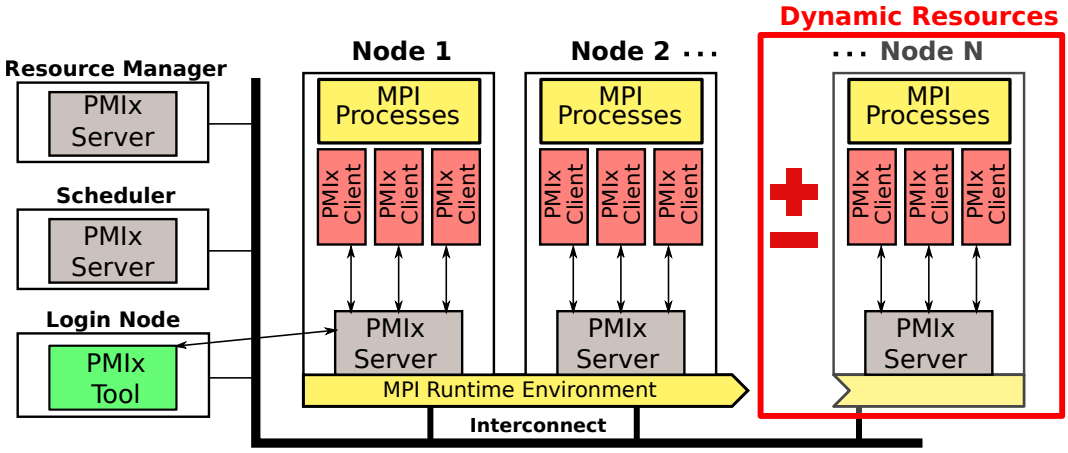

Figure 1: Illustration of resource dynamicity of MPI jobs on an PMIx-based HPC system. The HPC system consists of PMIx based components: a login node, scheduler, resource manager (left) and compute nodes executing the (dynamic) MPI job. All components are connected via a high-speed interconnect. To provide dynamicity, it is necessary to add/remove MPI processes (or, in most cases, full nodes with multiple MPI processes) to/from the MPI job (red box). As the figure shows, this requires not only an interface for writing resource dynamic MPI applications, but also a dynamic MPI runtime environment.

resource manager, but also requires support from the programming models used on the platforms, so that applications can request and react to resource changes accordingly. In this paper we focus on MPI [17], as the currently dominating programming model in HPC, and introduce support for resource-adapting MPI jobs, i.e., MPI jobs with a dynamically varying number of MPI processes.

Fig. 1 illustrates a dynamic MPI job running on a typical HPC system. To give a concrete example, consider an MPI application (e.g., a parallel CFD solver on a structured regular grid) running on n nodes (*job1*). Now consider the situation where another job (*job2*) is waiting in the job queue because it requires $m \ll n$ more nodes than currently available. With dynamic resource management, a scheduler could initiate a resource change to remove m nodes from *job1* and to assign them to *job2*, which would allow *job2* to start its execution immediately. However, this requires the RTE of *job1* to interact with that job, to ensure that MPI processes running on the m nodes to be terminated and removed, and that the application can continue its execution correctly without these processes after the resource change.

Many different approaches to introduce dynamicity to MPI have been proposed in the literature using techniques such as checkpointing and process migration [7, 24, 25], process virtualization [10, 11], MPI-2 Dynamic Process Management [22] or dynamic `MPI_COMM_WORLD` communicators [5].

The approach we present in this paper is a continuation of previous work [8], which emulated dynamicity with MPI Sessions for a single application on-top of an MPI version lacking support for MPI Sessions. In this previous work, we introduced our approach to describing dynamicity with resource sets and set operations for the first time and we extend these concepts in this work.

So far, MPI Sessions is mainly a new model for MPI initialization that avoids the necessity of a global communicator (`MPI_COMM_WORLD`). Instead, communication can be derived from sets of processes, which are organized as a dictionary of URIs. Our first contribution is to use these MPI Sessions as a foundation to our dynamicity approach, which will be presented in Sec. 2.

Further, we describe the implementation of a resource dynamic MPI runtime environment based on PMIx. The Process Management Interface - Exascale (PMIx) [4] defines a standardized interface for portable and scalable interaction between components of the Software Management Stack (SMS). It is frequently used by MPI implementations for an MPI runtime environment (RTE) as well as in other system components, such as resource managers and schedulers (see Fig. 1). Thus, a dynamic MPI implementation based on PMIx can facilitate the integration and interactions of dynamic MPI on HPC systems. We use and extend PMIx to realize a dynamic MPI RTE allowing for generic integration with different SMSs, as shown in Sec. 3.

Finally, we provide a performance evaluation in Sec. 4 and an assessment of our approach in Sec. 5 followed by a summary in Sec. 6.

2 MPI application perspective on dynamic resources

In this section we propose a new approach for resource dynamicity for MPI applications. We start by describing our design goals in Sec. 2.1. Subsequently, we describe the set and graph-based abstraction we use to approach these design goals. Finally, we present possible MPI interfaces that realize this approach using the MPI Sessions model.

```
MPI_Session_init(&session)  // Create a session handle
MPI_Session_dyn_recv_res_change(  // Check if process was added dynamically (included in delta pset)
  session,          // IN: Session handle
  ''mpi://self'',   // IN: The name of the associated PSet, for which resource change info is queried
  &rc_type,         // OUT: The type of the resource change
  delta_pset,       // OUT: Name of delta PSet, describing the changing resources
  &incl             // OUT: Flag indicating if this process is included in the delta PSet
  )

if incl: // Dynamically added resource? -> Receive name of new PSet for communication with original procs
  MPI_Session_dyn_integrate_res_change(
    NULL,        // IN: mpi_info (optional)
    delta_pset, // IN: Name of the delta PSet of the resource change
    false,       // IN: Flag indicating if this process provides the name of the new PSet
    main_pset,   // IN/OUT: Name of the new PSet related to the resource change
    &terminate   // OUT: Flag indicating that this process needs to terminate
    )

MPI_Group_from_session_pset(session, main_pset, &main_group)            // PSet to group
MPI_Comm_create_from_group(main_group, "tag", NULL, NULL, &main_comm)   // group to communicator

while work_to_do: // MAIN LOOP
  ... // Application dependent work, potentially including data migration to new resources

  // Rest of while loop copes with resource change requests
  if my_rank == root_rank:
    MPI_Session_dyn_recv_res_change(session, main_pset, &rc_type, delta_pset, &incl)
    resources_changing = rc_type != MPI_RC_NONE

    if resources_changing:  // For each resource change, root process executes the set operation
      if rc_type == MPI_RC_ADD:
        MPI_Session_pset_create_op(
            MPI_PSETOP_UNION,  // IN: Requested set operation
            main_pset,         // IN: First argument to the set operation (PSet name)
            delta_pset,        // IN: Second argument to the set operation (PSet name)
            new_main_pset      // OUT: Result of the set operation (PSet name)
          )
      else if rc_type == MPI_RC_SUB:
        MPI_Session_pset_create_op(session, MPI_PSETOP_DIFF, main_pset, delta_pset, new_main_pset)

  // Provide information whether there is a resource change
  MPI_Bcast(&resources_changing, ..., root_rank, main_comm)

  if resources_changing:  // resource change operations executed by all existing processes
    MPI_Bcast(delta_pset, ..., root_rank, main_comm)  // Provide the name of the delta PSet
    MPI_Session_dyn_integrate_res_change(session, NULL, delta_pset, new_main_pset, my_rank==root_rank, &
   terminate)

    MPI_Comm_disconnect(&main_comm) // Destroy old communicator
    if terminate:
      break // This process is not part of the new pset => terminate

    main_pset = new_main_pset     // new main pset
    MPI_Group_from_session_pset(session, main_pset, &main_group)    // PSet to group
    MPI_Comm_create_from_group(main_group, NULL, NULL, &main_comm)  // group to communicator

MPI_Session_finalize(session);
```

Listing 1: Illustrative example of the proposed interface for a resource dynamic MPI application in pseudo code. In this example resource changes target the whole application. Arguments followed by IN/OUT are read/written, respectively. Although this code shows blocking API calls, *all of them can be realized as non-blocking API calls.*

2.1 Design Goals

Our approach follows three core design goals:

- **Generality:** MPI applications are diverse, from linear algebra solvers and graph workloads to adaptive meshes and large scale machine learning. It is not possible to foresee all different application requirements. Our primary goal, therefore, is the creation of a general approach for dynamic resources, which then provides a maximum of flexibility for MPI application developers.

- **Separation-of-concerns:** The focus of this work is to target the MPI aspects of resource dynamicity, with the goal of providing a separation-of-concerns. By separating the MPI part from other parts, such as resource scheduling (or optimization) and application data reorganization, we enable clean differentiation and composability of solutions.
- **Performance:** Our goal is to provide a low-overhead mechanism to adapt to changing resources. To this end a) unnecessary, global synchronization between processes should be avoided and b) non-blocking interfaces should be provided, e.g., to avoid blocking during MPI process creation such as when using the `MPI_Spawn` API.

2.2 Underlying Abstractions

To achieve our goal of generality, we build our approach on top of a set of abstractions which are again based on concepts of MPI Sessions for extending them.

- **Sets:** We use **sets** to denote an unordered grouping of resources. Such sets can be potentially overlapping. For this definition, we refer to resources in an abstract sense describing a logical or physical component to be used as part of an HPC system. In the context of this paper we focus on the specific case of sets including (MPI) processes, with one of these sets referred to as process set (aka. PSet). This concept of PSets is already defined in the MPI standard [17] and is an important building block of the MPI Sessions model.
- **Set operations:** Set **operations** (union, difference, split, join, ...) can be **executed** on sets to create one or more new sets. A set operation thus can be used to express subsets or changes of sets, which can be used to indicate the scope of resources (re-)assignments.
- **Graphs:** A **graph-based** perspective on resources and resource changes can be taken by associating vertices with PSets and directed hyperedges with set operations. Having such a graph-theoretical description of resources and resource changes as an abstract building block, will enable us (and others) to layer dynamicity approaches on top of it, like, e.g., scheduling heuristics.

2.3 Dynamic MPI Interfaces and an Example

In this section we describe a first design of an interface for a dynamic MPI Sessions model, following the abstractions and design goals explained in the last sections. While the underlying model applies to all kinds of resources, for now, we only consider MPI processes as resources. This design introduces *three additional MPI procedures, all related to the MPI Sessions model*:
`MPI_Session_dyn_recv_res_change`,
`MPI_Session_pset_create_op` and
`MPI_Session_dyn_integrate_res_change`.
We also implemented asynchronous variants of these procedures which is trivial as these procedures can be executed in the background and MPI processes can use the `MPI_Request`, `MPI_Test` and `MPI_Wait` procedures accordingly. Thus, we omit the details of the asynchronous variants.

In the following we describe the interfaces in the course of which we also refer to the provided pseudo code example for a loop-based MPI Sessions application in Lst. 1. A brief description of the new function signatures is included in this code example in lines 2, 11 and 32 respectively. We use the terminology "*root process*" to denote the one particular process that takes the role of being in charge for processing resource changes. Note that this interface will very likely undergo further changes in the future to approach an even higher level of generality and performance.

`MPI_Session_dyn_recv_res_change`: This procedure allows an MPI process to query the RTE for *resource change information*.

Following the graph-like description from Sec. 2.2, resource change information is always associated with an existing set of resources (a vertex), thus allowing for fine-grained, independent resource management of different application parts. This information includes the type of resource change and the set including resources to be changed, i.e., the delta between the resources before and after the resource change (which we call "delta set").

In this work we only deal with MPI processes as resources. Accordingly, `MPI_Session_dyn_recv_res_change` queries the RTE for resource change information associated with the PSet name specified by the application (parameter 2). From hereon, we refer to this PSet as the *associated PSet*. Here, the PSet name *mpi://self* can be specified as a special means to query information about the most recent resource change where the calling MPI process is included in the delta set.

The procedure returns the type of the resource change (parameter 3) where we currently support

- `MPI_RC_TYPE_NONE`: No resource change information available,
- `MPI_RC_ADD`: Resources (MPI processes) are added,
- `MPI_RC_SUB`: Resources (MPI processes) are removed

with more operations such as replacement of MPI processes possibly defined in future versions.

Moreover, in case of available resource change information, it returns the name of a PSet representing the delta between the MPI processes before and after the resource change, i.e., the MPI processes to be added or removed. Hence, from hereon we refer to this PSet as the *delta PSet*. Additionally, the function returns a flag to indicate if the calling MPI process is included in the delta PSet.

The interface deals with PSets (resource sets including MPI processes), however, similar interfaces could be used to support other types of resource sets. Also, this interface does not enforce or interfere with how the application adapts to the resource change, as it only provides the name of the delta PSet. Thus, it provides the application with the flexibility to use this delta PSet in set operations according to the specific needs of the application.

In the example, this interface is used at two different places: First, by all processes at program start to allow MPI processes to determine their dynamic nature (line 2). Second, by the root process after each iteration of the main loop to check if a new resource change needs to be processed (line 27).

MPI_Session_dyn_pset_create_op: This procedure allows applications to request the creation of new PSets based on a specified set operation.

Using our abstraction, a resource change is represented as a delta set. However, the delta set provided by the query operation is not the only option; the set can be further manipulated based on application specific needs. This procedure enables such further manipulations of PSets, resulting in new PSets. Contrary to local set operations performed on MPI Groups, set operations on PSets result in new PSets visible beyond the local MPI process in all MPI processes that could be affected by the process change. Thus, other processes can query these Sets and then create groups from the discovered PSets without having to have the knowledge about the operations that created them.

In particular, this function enables applications to create a new PSet **pset_result** from a set operation of two existing PSets **pset1** and **pset2**, i.e., **pset_result** = **pset1** ∘ **pset2**. Currently possible PSet operations specified by the **pset_op** ∘ parameter are

- `MPI_PSETOP_UNION` (pset_result = pset1 ∪ pset2),
- `MPI_PSETOP_DIFF` (pset_result = pset1 \ pset2),
- `MPI_PSETOP_INTERSECT` (pset_result = pset1 ∩ pset2)

and further operations could be defined to broaden the generality of our approach.

In Lst. 1 the root MPI process uses set operations to create a `new_main_pset` based on the old `main_pset` and the queried `delta_pset` to reflect the addition or removal of resources to/from the application respectively (lines 32 and 39).

MPI_Session_dyn_integrate_res_change: This procedure is specific to MPI processes as it targets issues related to communication channels and synchronization. On one hand, when adding processes the concrete usage of dynamically added processes might not be known a-priori. Thus, these processes lack knowledge of the PSet name to be used for creating an initial communicator. On the other hand, when removing processes it might be necessary to have a synchronization barrier before processes terminate. Due to the flexibility provided to applications by the procedures defined above, it is not guaranteed that communication channels (MPI communicators) between processes in the delta PSet and/or the associated PSet exist or are desired.

To this end, the `MPI_Session_dyn_integrate_res_change` procedure provides an out-of-band mechanism for processes to synchronize and exchange a PSet name, without creation of a common communicator. The procedure is *collective over the union of the MPI processes in the delta PSet and the associated PSet*.

All processes participating in the collective need to specify the same delta PSet name (parameter 2). A synchronization barrier is performed across the participating processes.

Moreover, one process from the associated PSet (*referred to as the provider from hereon*) can indicate its intention to act as provider via an input flag (parameter 3). Usually, this would be the root process. The PSet name provided by the provider (parameter 4) is published to a range accessible to all MPI processes involved in the collective.

The behavior of the procedure is undefined in the case that multiple MPI processes set the provider flag.

All other MPI processes can lookup the provided name, which will be stored in the buffer (parameter 4). Here, a `NULL` pointer as buffer is valid and results in no lookup by this process. If no MPI process acts as provider, all MPI processes involved in the collective are required to specify a `NULL` pointer as buffer.

An additional output flag indicates if the calling process is included in the delta PSet and that the delta PSet is related to a resource change of type `MPI_RC_SUB`. Thus this flag indicates if processes are required to finalize and terminate their execution.

An optional info object can be used to influence the behavior of the function, e.g., to indicate if the function should wait until the processes are synchronized before returning.

In our example, the original MPI processes call this procedure (line 46), with the root process providing the name of the `new_main_pset`. The dynamically added MPI processes call this procedure (line 11) to receive the name of the new main PSet. Subsequently both, original and dynamically created processes, create a common communicator from this PSet.

3 Implementation using Open MPI and PMIx

Next, we illustrate a possible implementation of the MPI procedures described in the previous section. We build on top of the MPI Sessions prototype in Open MPI [9] as well as its already existing usage of the PMIx interface [4]. The latter provides mechanisms for runtime environments to interact with resource managers and is used by a range of common software components, including Open MPI, as well as by other system software such as SLURM [26], for bootstraping MPI processes and to realize MPI runtime functionalities. As PMIx provides standardized APIs for portable access to typical system software services, it naturally simplifies the handling of resource dynamicity. Moreover, PMIx already supports the concept of PSets, which is a central component of our approach. Thus, in Sec. 3.1 we discuss the usage of PMIx to implement the new, dynamic MPI interface. Based on this, Sec. 3.2 describes our prototype implementation based on Open MPI, OpenPMIx [4] and the PMIx Reference RunTime Environment (PRRTE). Our particular contribution is to provide insight into the additional work and challenges one has in realizing these particular dynamicity interfaces also on the runtime levels.

3.1 PMIx

Even though PMIx already includes some support for dynamicity, we need a few more extensions to support the new MPI functionalities. PMIx acts as a messenger between SMS components and is based on the storage and transfer of key-value pairs, with some keys being predefined. Another important concept of PMIx is the PMIx server abstraction which gives well defined access to PMIx services via function

```
pmix_status PMIx_Pset_Op_request(
        psetop_directive,       // IN pmix_psetop_directive_t specifying the requested PSet operation
        info,                   // IN Array of pmix_info_t structures (array of handles)
        ninfo,                  // IN Number of elements in the info array (integer)
        results,                // OUT Array of pmix_info_t structures (array of handles)
        nresults                // OUT Number of elements in the results array (integer)
)
```

```
typedef pmix_status (*pmix_server_pset_operation_fn_t)(
        proc,                   // IN pmix_proc_t structure identifying the process requesting the PSet operation (handle)
        psetop_directive,       // IN pmix_psetop_directive_t specifying the requested PSet operation
        info,                   // IN Array of pmix_info_t structures (array of handles)
        ninfo,                  // IN Number of elements in the info array (integer)
        cbfunc,                 // IN Callback function pmix_op_cbfunc_t (function reference)
        cbdata                  // IN Data to be passed to the callback function (memory reference)
)
```

```
typedef void pmix_psetop_cbfunc_t(
        status,                 // IN pmix_status_t describing the status of the request
        psetop_directive,       // IN pmix_psetop_directive_t specifying the requested PSet operation
        info,                   // IN Array of pmix_info_t returned by the operation(pointer)
        ninfo,                  // IN Number of elements in the info array (size_t)
        cbdata,                 // IN Callback data passed to original API call (memory reference)
        release_fn,             // IN Function to be called when done with the info data (function pointer)
        release_cbdata          // IN Callback data to be passed to release_fn (memory reference)
)
Attributes required to be included in the passed info array:
        - PMIX_PSETOP_PRESULT       "pmix.psetop.p1" (char*)                // Name of the resulting PSet
        - PMIX_PSET_MEMBERS         "pmix.pset.mems" (pmix_data_array_t*)   // The members of the resulting PSet
```

Listing 2: Proposed PMIx functions related to PSet operations. `PMIx_Pset_Op_request` requests the execution of the specified process set operation. PMIx passes on this request to the host environment, for which the latter provides a pointer to its implementation of the `pmix_server_pset_operation_fn_t` function at initialization of the PMIx server. The host environment provides the results of the operation back to the PMIx server using the provided `pmix_psetop_cbfunc_t` callback.

pointers provided by the host environment. Here, we use the term *host environment* according to the PMIx definition "to refer to the process hosting the PMIx server library" [18, p. 7], which is the MPI RTE (daemon) in the context of this paper. For the sake of brevity we refer the interested reader to the PMIx 4.0 standard document [18] for further information.

MPI_Session_dyn_recv_res_change: This procedure interacts with the RTE to *receive information about resource changes*. PMIx supports queries for runtime/system information via the `PMIx_Query_info` function, allowing the user to provide query keys and qualifiers to specify a query request.

Therefore, for the *delta PSet*, *resource change type* and *associated PSet* parameters of the MPI procedure, we define the corresponding `PMIX_RC_PSET`, `PMIX_RC_TYPE` and `PMIX_RC_ASSOC_PSET` query/qualifier keys respectively.

The `PMIx_Query_info` function returns key-value pairs, with the values for each query key provided by the host environment. To facilitate the provision of a value for the *type of the resource change* we define the new enum data type `pmix_rc_type_t`. So far, this type is one of the three new PMIx constants: `PMIX_RC_NONE/ADD/SUB`.

MPI_Session_pset_create_op: In order to support the needed PSet operations introduced by this proposed MPI procedure, we require three new functions in PMIx. We show their signatures in Lst. 2.

- `PMIx_Pset_Op_request`: This function allows to request a PSet operation from the host environment. The operation can be specified as one of the three new PMIx constants `PMIX_PSETOP_UNION/DIFF/INTERSECT`. The `pset1`/`pset2` arguments of the set operation are provided in the `info` parameter using the new `PMIX_PSETOP_P1/P2` keys. The name of the resulting PSet is returned in the `result` parameter with the new `PMIX_PSETOP_PRESULT` key.
- `pmix_server_pset_operation_fn_t`: This function definition allows to pass on the request from the PMIx server to the host environment. The host environment provides a pointer to a valid implementation of this function definition during the initialization of the PMIx server. The implementation is expected a) to perform the specified PSet operation; b) to make the information about the resulting PSet available in the given scope; and c) to call back into the local PMIx server via a callback function.
- `pmix_psetop_cbfunc_t`: This callback function specification allows the host environment to pass the operation results to the PMIx server.

MPI_Session_dyn_integrate_res_change:

The implementation of this MPI procedure does not require any extensions to the PMIx standard.

For the out-of-band transmission of the PSet name, the existing `PMIx_Publish/Lookup` interface can be used. This interface provides access to a global key-value datastore, where data can be retrieved solely by referral to its key. Thus, in the MPI procedure the provider publishes the PSet name as key-value pair with `PMIx_Publish`. The used key is required to be well-defined and unique for each resource

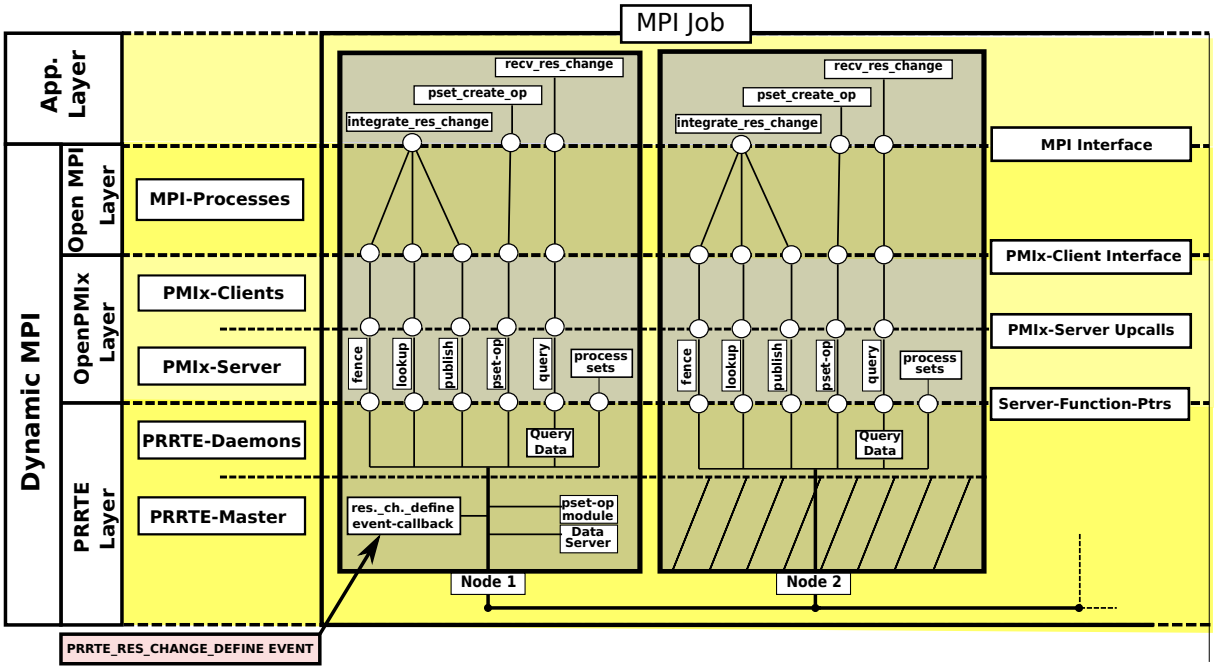

Figure 2: Overview of the prototype implementation of our approach with Open MPI. This example shows two nodes running a (dynamic) MPI job. We differentiate between four layers (see labels on the left), which are each accessible from the layer above via the interfaces indicated on the right side. The interior of the nodes delineate the required interactions of our dynamicity extensions to MPI with PRRTE using PMIx.

change, e.g. by deriving it from the corresponding delta PSet name. This way, the published PSet name can be retrieved by a corresponding `PMIx_Lookup` by other processes in the `MPI_Session_dyn_integrate_res_change` procedure. The `PMIx_WAIT` attribute has to be provided in `PMIx_Lookup` to request that the PMIx server should wait until the value is found, e.g., in case it was not yet published.

Subsequently, the `PMIx_Fence` function is entered, which performs a PMIx barrier across all the processes specified in the `pmix_proc_t` array, i.e. the processes in the union of the delta PSet and the associated PSet).

After the barrier, the provider uses `PMIx_Unpublish` to unpublish the corresponding PSet name, which avoids an unnecessary accumulation of key-value pairs in long-running PMIx environments and hence prevents memory leaks.

3.2 Implementation with Open MPI, OpenPMIx and PRRTE

In this section, we illustrate how the extended interfaces of MPI and PMIx achieve resource dynamicity with Open MPI. Fig. 2 provides an high-level overview of our prototype. We differentiate between the application layer, the MPI (Open MPI) layer, the PMIx layer (OpenPMIx) and the runtime layer (PRRTE). In the following we describe the necessary actions and interactions of these layers to implement the proposed MPI extensions.

PRRTE master layer: To launch the MPI runtime, an initial process starts PRRTE daemon on each node of the allocation, excepting the one on which it is executing. Here, we call this process *PRRTE master*, as it is responsible for the orchestration of the runtime services and the control of the resource change procedure. To this end, the PRRTE master provides an entry point for triggering resource changes via *PMIx event notifications*. The *PMIx event handler* performs the following actions required for the initiation of a resource change: a) it defines the corresponding delta PSet b) it sends all necessary meta and query data to the daemons and c) it sends the launch command for the corresponding processes in case of a resource addition. In our design, also the (centralized) PSet operation module for executing PSet operations, as well as the PMIx data-server for the global dictionary (publish/lookup) are also hosted by the PRRTE master.

PRRTE daemon layer: On each node of the allocation a *PRRTE daemon* is running, providing runtime services for the MPI job. The PRRTE master also acts as PRRTE daemon on the node on which it is executing. Each daemon operates a local *PMIx server*, thus providing function pointers for the PMIx services during the initialization of the server. Particularly relevant for resource changes are the function pointers for five PMIx services: queries, PSet operations, publish/lookup requests and fence operations. We indicate these services in Fig. 2 at the interface between the OpenPMIx and PRRTE layers.

MPI and OpenPMIx layers: In Fig. 2, we denote the three new MPI procedures for resource changes (Sec. 2)

at the MPI API level. The implementation of these procedures makes use of the five PMIx services that we list at the PMIx server / PRRTE daemon interface. The lines between the MPI interface and the PRRTE layer illustrate how these services are used by the MPI procedures via PMIx as discussed in Sec. 3.1. To give one example, the implementation of the `MPI_Session_pset_create_op` procedure uses the PMIx client interface (`PMIx_Pset_Op_request`) to send the request to the local PMIx server. The PMIx server passes on the request to the local PRRTE daemon (`pmix_server_psetop_request_fn_t`), which further transmits the request to the PRRTE master. Here, the PSet operation is executed in the PSet operation module and the result is broadcasted to all daemons. The daemon from which the request originated uses the `pmix_pset_op_cbfunc_t` callback function to call back into its local PMIx server to provide the results to the MPI process.

4 Performance Evaluation

Next, we provide performance results for a synthetic benchmark as well as for a use case of a Partial Differential Equation (PDE) solver, which has been enhanced to exploit resource dynamicity. For sake of reproducibility we provide the code used for the benchmarks [12].

4.1 System Setup

We use the CoolMuc2 Linux cluster at the *Leibniz Supercomputing Centre* [14] for all experiments , which features 28 cores (56 hardware threads) per node (2 sockets per node each containing Intel Xeon E5-2697V3 processors with 14 cores and 28 hardware threads) and a 64 GB DDR4 memory per node. The interconnect between the nodes is based on FDR14 Infiniband (bandwidth to interconnect per node up to 13.64 GB/s). The cluster runs an instance of the SLURM workload manager for resource scheduling and management. We create an interactive SLURM shell with the `salloc` command, allocating four nodes and a total of 112 cores. Following that, we launch a *daemon job* (runtime layer) spanning these four nodes using the `prterun` command. Our benchmarks (*user jobs*) run on top of this runtime layer with their size being grown and shrunk dynamically. Thus, the presented performance results do not include any resource allocation and management of the resource management system, but rather provide insight into the overheads of the adaption process of dynamic MPI jobs using our approach.

4.2 Benchmarks

We implement a **synthetic application** similar to the example in Lst. 1 to evaluate the performance characteristics of the suggested interface. As workload we use a problem size of 10^{10} elements and ≈ 10 FLOPs per element. The elements are equally distributed over the available processes, however, no communication is required for redistributing elements after a resource change. Resource changes are triggered by the application every 10-th iteration. We report performance results for adding/removing $n_{delta} \in \{28, 56, 84\}$ MPI processes.

We further use the proposed interface to provide dynamic resources to a **PDE solver** with fixed workload and load-balancing to evaluate the impact on application performance in a realistic setting. For this we build on top of prior work [21], which combined an emulation layer for dynamic MPI Sessions [8] with a Shallow-Water-Equation solver [1] and the P4est library [3]. We replace the emulation layer with our prototype implementation for dynamic MPI Sessions. We use a uniform grid of size $S = 4^{10}$, a simulation time of $t = 10$ seconds which is split in $c = 10$ phases, and the application triggering resource changes each phase. We report the results for a simulation run starting with $n_{start} = 28$ processes and adding $n_{delta} = 28$ processes per resource change up to a maximum of $n_{end} = 112$ processes.

4.3 Performance Results

Synthetic Benchmark: We first discuss the performance results of the synthetic benchmark, which we list in Tab. 1.

For *resource addition*, we measure a total time of up to 1.69 seconds on the RTE level, which is dominated by the MPI initialization (`MPI_Session_init`) of the added processes (up to 1.42 seconds). The sole overhead introduced on the RTE level for initiating the resource change was $11.56 - 18.49$ ms. We use an asynchronous implementation of `MPI_Session_dyn_integrate_res_change`, which allows the application to resume its work (i.e., perform further iterations) until the dynamically added processes have been initialized. Based on this, the overhead of resource changes on the application level, including the creation of the new MPI communicator, is $109.39 - 181.1$ ms. Both overheads slightly increase with increasing n_{delta}.

For *resource subtraction* no new processes need to be initialized, which naturally leads to a lower total time required for the resource change (88.27 - 125.86 ms). However, also the overheads are slightly smaller with $7.29 - 14.85$ ms on the RTE level and $71.4 - 112.78$ ms on the application level. Again, increasing n_{delta} leads to slightly larger overheads.

In Fig. 3 we show the implications of resource subtraction on the performance of the synthetic loop-based benchmark. This synthetic benchmark represents a near optimal application, which has almost no load-balancing overhead and shows a straight-forward relationship between the number of processes and the application performance. This simplifies the assessment of the performance impact of our approach. As expected, the time required for the workload (black bars) increases after resources are removed in iterations 11, 22 and 33 respectively. In these iterations, overheads of around 100 ms related to processing the resource change are incurred (blue bars). These overheads are moderate relative to the overall time required for one iteration and the interval in which resource changes occur.

PDE solver: Fig. 4 shows the performance results of the resource-adaptive PDE solver run. Similar to the synthetic benchmark the application overhead for resource changes including the creation of the new communicator (blue bars) is between 100 and 125 ms. Additional overheads arise from the adaption of the grid with p4est (red and grey bars).

Type of resource change	Addition			Subtraction		
n_{delta}	28	56	84	28	56	84
Total time of res. change (RTE)	1670 ms	1692 ms	1690 ms	88.27 ms	108.80 ms	125.86 ms
Initiation of resource change (RTE)	11.56 ms	16.89 ms	18.49 ms	7.29 ms	11.13 ms	14.85 ms
Application overhead	109.39 ms	141.75 ms	181.1 ms	71.4 ms	108.7 ms	112.78 ms
max. MPI_Session_init (dyn. procs)	1209 ms	1417 ms	1375 ms		-	

Table 1: Performance results of the synthetic benchmark for resource addition and subtraction. We denote as *Total time of resource change* the time measured on the RTE level from receiving the notification of a resource change, until the application has adapted to the resource change. The *initiation of the resource change* is the time required by the RTE to process the resource change and prepare the data for the interaction with the application. The *application overhead* is the time the application spends with processing the resource change using the proposed MPI interface and creating a corresponding new communicator. For resource addition we also provide the maximum time the dynamic processes required for the MPI initialization (*MPI_Session_init*).

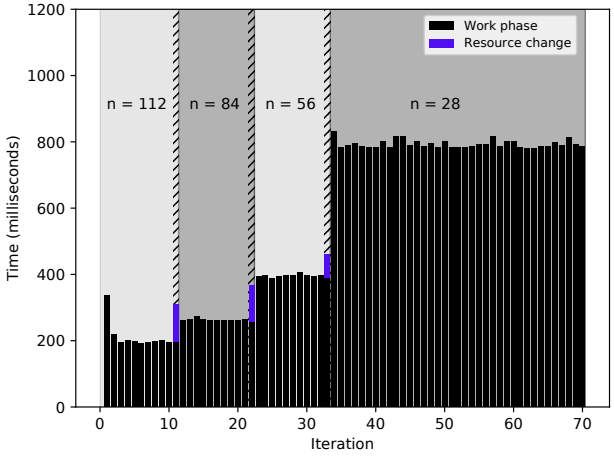

Figure 3: Performance implications of dynamic resource subtraction in the synthetic benchmark. The bars indicate the timings for each of the 70 iterations. The colored areas in the background are labeled with the current number of processes n. The black bars denote the time required for the synthetic workload in each iteration. The hatched areas mark the iterations where the resource change is processed by the application. The blue bars indicate the application overhead related to processing resource changes.

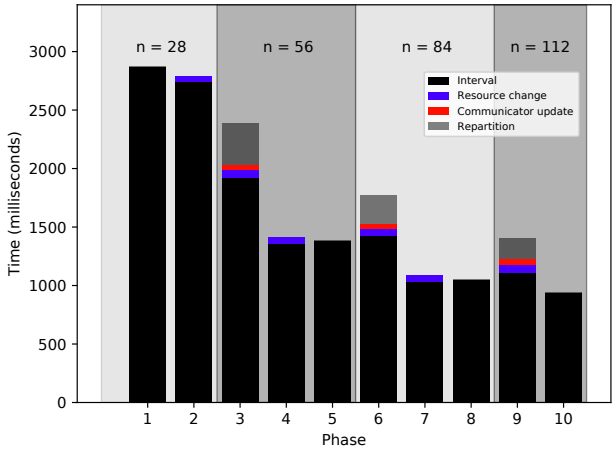

Figure 4: Performance implications of dynamic resource addition in a PDE solver for a hyperbolic problem. The total simulation time frame is split into ten phases, in which a certain time interval is simulated with fixed resources (black bars). The colored areas in the background are labeled with the current number of processes n. The blue bars indicate the application overhead related to processing resource changes. As we use a non-blocking implementation of the dynamic MPI interface, overheads related to resource changes are distributed over multiple iterations. The red and gray bars denote the overhead related to the grid adaptions with p4est.

However, due to additional resources, the time required for one phase after the resource changes decreases significantly by $G_{rc1} = 50.37\%$, $G_{rc2} = 25.33\%$ and $G_{rc3} = 10.68\%$ respectively. Thus, the total overhead of the resource change, can be amortized due to the improved performance with additional resources. In our benchmark, the total overhead of the resource change including repartitioning with p4est, would be amortized after $n_{rc1} > 0.37$, $n_{rc2} > 1.13$ and $n_{rc3} > 3.11$ subsequent phases with additional resources, pointing towards the practicability of our approach.

5 Discussion of Proposed Interfaces

The interface we provide in this work is the result of one of the first attempts to use the set and graph-based approach and is intended to help drive and support the discussions in the MPI Forum. As part of this discussion, in the following we focus on how our proposal is in alignment with our stated design goals, which we describe in Sec. 2.1 and infer possible changes and extensions.

Data redistribution: Changing resources of an application might require corresponding redistribution/reorganization of application data. Our approach does not provide implicit data redistribution, which follows the goal of separation-of-concerns. Instead it gives application programmers full control over the data reorganization in response to

changing resources to maintain generality. However, our approach provides applications with the necessary information about resource changes and mechanisms to create appropriate communicators, which can be used for standard MPI communication to redistribute any application data.

Resource Optimization: Again following the design goal of separation-of-concerns, our proposed interface is independent from the resource optimization process. In particular, our approach does not make any assumptions about the entity initiating the resource changes. Instead, it defines a general mechanism for MPI applications and RTEs to adapt to resource changes. Such a resource change could be triggered by any SMS component, e.g., by the scheduler to improve overall system utilization or by an evolving application to adapt to dynamic workloads.

Moreover, to achieve the goal of high performance, a sophisticated, potentially cooperative scheduling mechanism needs to be added on top of our graph-based approach. As our approach uses PMIx, resource changes and PSets could be made transparent to entities in the SMS (such as schedulers and tools) if they attach to the corresponding PMIx server. Exploring this potential could be valuable future work.

Application-driven set operations: In our current design, the *application drives the set operations* related to resource changes based on the information received via the `MPI_Session_dyn_recv_res_change`, such as the name of the *delta process set*. This gives applications the flexibility to control the resource usage in response to a resource change. For performance reasons, it might be preferable to perform set operations representing resource changes on a higher level of the SMS, e.g., the system scheduler, which would require minor changes of `MPI_Session_dyn_recv_res_change`.

Granularity of interfaces & fault tolerance: The current state of the suggested interface requires only three additional MPI procedures (and three asynchronous variants), which eases the complexity for application programmers to adopt the approach. However, we plan to achieve further generality by dividing the interface into smaller, but more general functions. To give one example, in our current design, `MPI_Session_dyn_integrate_res_change` performs the transmission of a PSet name as well as synchronization. More flexibility could be achieved by implementing these functionalities in separate procedures. Such a more fine-granular interface could also facilitate support for fault tolerance.

Root process: So far, our approach uses a root process, which performs the query for resource change information and the potentially required set operations. A consensus-based approach involving all processes of one PSets could be an alternative in particular due to fault tolerant aspects. We do not see both approaches excluding each other, but rather complementary to further improve generality.

6 Summary and Future Work

Dynamic resource management is an active research area which can improve the efficiency of HPC systems, as well as HPC applications. In this context, we proposed and evaluated a new approach towards dynamic resource management of MPI jobs based on MPI Sessions and PMIx.

In our design, **resources** are represented by (potentially overlapping) **sets**, which can be modified using **set operations**. This allows for a **graph-based** perspective on resource changes where sets of resources are represented by vertices and set operations by directed hyperedges. Our expectations are that this will cover various requirements of different HPC applications.

We showed how our concept can be implemented on top of the standardized PMIx interface, and what extensions are necessary to support dynamicity. The latter are minimal, as PMIx already provides many of the needed features to support dynamicity and, hence, forms an excellent basis for our work. It is used by most MPI implementations as well as many software components for resource management on HPC systems, and it provides portable access to common system services. Thus, our approach can easily be integrated into such PMIx-based software management stacks.

We developed a prototype of our approach extending Open MPI, OpenPMIx and PRRTE and demonstrated its functionality to dynamically add/remove resources to/from MPI jobs on a modern HPC system on up to 4 nodes and a total of 112 cores. The application overheads were modest, with $109-181$ ms for resource addition and $71-113$ ms for resource subtraction.

Overall, this work is a proof of concept, providing a first demonstration of a novel set- and graph-based approach towards resource management based on MPI Sessions and PMIx.

In future work, we plan to continue working on the proposed interfaces to further improve on its generality and performance. To this end, we will also need to extend our performance evaluations in terms of scale, resource types and application use cases. Moreover, integration of our approach for resource dynamic MPI Sessions with HPC resource management systems is a natural next step to assess its applicability on modern HPC systems. Here, we will particularly focus on usage and extensions of PMIx to facilitate cooperation between HPC applications and other system components for improved resource optimization.

Acknowledgments

We like to gratefully acknowledge in particular Dan Holmes for his valuable feedback and discussions as part of the meetings of the MPI Session workgroup. We like to thank the anonymous reviewers and in particular Jeffrey M. Squyres for the valuable feedback. This project has received funding from the Federal Ministry of Education and Research and the European HPC Joint Undertaking (JU) under grant agreement No 955701, Time-X and No 955606, DEEP-SEA. The JU receives support from the European Unions Horizon 2020 research and innovation programme and Belgium, France, Germany, Switzerland.

References

[1] 2020. *Shallow Water Equations Teaching Code*. https://github.com/TUM-I5/SWE

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

[3] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. 2011. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing* 33, 3 (2011), 1103–1133. https://doi.org/10.1137/100791634

[4] Ralph Castain, Joshua Hursey, Aurelien Bouteiller, and David Solt. 2018. PMIx: Process Management for Exascale Environments. *Parallel Comput.* 79 (08 2018). https://doi.org/10.1016/j.parco.2018.08.002

[5] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. 2016. Infrastructure and API Extensions for Elastic Execution of MPI Applications. In *Proceedings of the 23rd European MPI Users' Group Meeting* (Edinburgh, United Kingdom) *(EuroMPI 2016)*. Association for Computing Machinery, New York, NY, USA, 8297. https://doi.org/10.1145/2966884.2966917

[6] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. 2007. Dynamic Malleability in Iterative MPI Applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*. 591–598. https://doi.org/10.1109/CCGRID.2007.45

[7] Kaoutar El Maghraoui, Boleslaw K. Szymanski, and Carlos Varela. 2006. An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–271.

[8] Jan. Fecht, Martin. Schreiber, Martin Schulz, Howard Pritchard, and Daniel J. Holmes. 2022. An Emulation Layer for Dynamic Resources with MPI Sessions. In *HPCMALL 2022 Workshop*.

[9] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–104.

[10] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kal. 2014. Towards realizing the potential of malleable jobs. In *2014 21st International Conference on High Performance Computing (HiPC)*. 1–10. https://doi.org/10.1109/HiPC.2014.7116905

[11] Chao Huang, Orion Lawlor, and L. V. Kalé. 2004. Adaptive MPI. In *Languages and Compilers for Parallel Computing*, Lawrence Rauchwerger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 306–322.

[12] Dominik Huber. 2021. *DynMPI Prototype (version used for benchmarks)*. https://github.com/DynMPISessions/dynmpi_prototype

[13] Jan Hungershfer, Achim Streit, and Jens-Michael Wierum. 2002. *Efficient resource management for malleable applications*. Paderborn Center for Parallel Computing.

[14] Leibniz-Rechenzentrum. 2022. *LRZ Linux Cluster*. https://doku.lrz.de/display/PUBLIC/Linux+Cluster

[15] Loris Marchal, Bertrand Simon, Oliver Sinnen, and Frédéric Vivien. 2018. Malleable task-graph scheduling with a practical speed-up model. *IEEE Transactions on Parallel and Distributed Systems* 29, 6 (June 2018), 1357–1370. https://doi.org/10.1109/TPDS.2018.2793886

[16] Gonzalo Martn, David E. Singh, Maria-Cristina Marinescu, and Jess Carretero. 2015. Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. *Parallel Comput.* 46 (2015), 60–77. https://doi.org/10.1016/j.parco.2015.04.003

[17] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard Version 4.0*. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[18] PMIx Administrative Steering Committee (ASC). 2020. *Process Management Interface for Exascale (PMIx) Standard Version 4.0*. https://pmix.github.io/uploads/2020/12/pmix-standard-v4.0.pdf

[19] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant Kal. 2015. A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 429–438. https://doi.org/10.1109/IPDPS.2015.34

[20] Martin Schreiber. 2014. *Cluster-Based Parallelization of Simulations on Dynamically Adaptive Grids and Dynamic Resource Management*. mediaTUM. 198 pages. https://mediatum.ub.tum.de/node?id=1189784

[21] Maximilian Streubel. 2021. Dynamic Resource Management Using MPI Sessions on p4est.

[22] Rajesh Sudarsan and Calvin J. Ribbens. 2007. ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. In *2007 International Conference on Parallel Processing (ICPP 2007)*. https://doi.org/10.1109/ICPP.2007.73

[23] Rajesh Sudarsan and Calvin J. Ribbens. 2009. Scheduling resizable parallel applications. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–10. https://doi.org/10.1109/IPDPS.2009.5161077

[24] Gladys Utrera, Julita Corbalan, and Jess Labarta. 2004. Implementing malleability on MPI jobs. In *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*. 215–224. https://doi.org/10.1109/PACT.2004.1342555

[25] Sathish Vadhiyar and Jack Dongarra. 2003. SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. *Parallel Processing Letters* 13 (04 2003). https://doi.org/10.1142/S0129626403001288

[26] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60.