An Emulation Layer for Dynamic Resources with MPI Sessions

Jan Fecht, Martin Schreiber, Martin Schulz, Howard Pritchard, Daniel J Holmes

An Emulation Layer for Dynamic Resources with MPI Sessions

Jan Fecht[1 [0000−0001−9775−2222]], Martin Schreiber[1,2 [0000−0002−2390−6716]], Martin Schulz[1 [0000−0001−9013−435X]], Howard Pritchard[3], and Daniel J. Holmes[4 [0000−0002−2776−2609]]

[1] Technical University of Munich, Garching bei München, Germany
[2] Université Grenoble Alpes, Saint-Martin-d'Hères, France
[3] Los Alamos National Lab, Los Alamos, NM, USA
[4] Collis Holmes Innovations Ltd, Scotland, UK

Abstract. The current static job scheduling on supercomputers for MPI-based applications is well known to be a limiting factor for the exploitation of a system's top performance in terms of application throughput. Hence, allowing fully flexible and dynamically varying job sizes would provide multiple advantages compared to the current approach, e.g., by prioritizing jobs dynamically and optimizing resource usage by transferring resources economically.
A critical step in achieving dynamic resource management with MPI on supercomputers is the development of sound and robust interfaces between MPI applications and the runtime system. Our approach extends the concept of MPI Sessions, a new concept introduced with MPI 4.0, by adding new features to support varying computing resources via the MPI process set abstraction. We then show how these features can be used, as a proof of concept, to request (active) and cope with (passive) varying resources from an application's perspective. To validate of our approach, we develop *libmpidynres*, a C library providing an emulated MPI Sessions environment on top of existing MPI implementations without MPI Sessions support, which we then use to integrate our proposed extensions to the interface specification. Using this proof-of-concept environment, we show how an MPI Sessions enabled application can use process sets to handle dynamically varying resources.

Keywords: MPI · MPI Sessions · Dynamic resources · Resource management

1 Introduction

1.1 Motivation

Job scheduling systems for MPI-based applications allocate a fixed amount of resources (cores, nodes, GPUs, FPGAs, ...) for the job's runtime. This is a strong constraint on resource usage, leading to various inefficiencies, e.g., idling cores, lack of taking runtime-changing resource requirements into account, to name just a few.

To solve this issue, dynamic resources, meaning that the number of available resources can change during an application's execution, need to be introduced to and supported by applications as well as the runtime. For example, with dynamic resources, the job scheduler can withdraw resources from an application and transfer them to another application. This could potentially lead to a higher throughput of application on the entire system, hence an overall better parallel efficiency. Furthermore, this approach allows the job scheduler to prioritize certain jobs dynamically by having more flexibility and scheduling abilities than in the static resource allocation case.

To use dynamic resources in high-performance scenarios, the following components need to be carefully designed and realized:

1. **API**: a flexible, robust interface for dynamic resources that can be used by MPI applications.
2. **Runtime**: dynamic resource support in the runtime (MPI library, job scheduler, ...).
3. **Applications**: MPI-based software using this interface to handle dynamic resource changes including all required changes in the software.

Having all three components working together in the right way is a very long lasting process and this work is on the first component.

Our contributions are a proposal of an API for robust and flexible dynamic resource changes based on the new MPI Sessions concept introduced with MPI 4 [16]. In addition to this, we evaluate our proposal based on a library called *libmpidynres* which emulates a dynamic resource environment on top of an existing MPI communicator.

1.2 Related Work

MPI 2's Dynamic Process Model: The MPI Forum already addressed the need for a more dynamic process management approach with the introduction of the MPI dynamic process model in the MPI 2 standard [17]. However, the number of running MPI processes is still limited to `MPI_UNIVERSE_SIZE`, which is typically equal to the number of resources reserved by the job scheduler.

Task-based parallelization models: Another set of applications are task-based parallel programs. E.g. *DucTeip* is a framework for creating task-based MPI programs [19]. That could be exploited rather in a straight-forward manner by, e.g., executing different applications using the same MPI context in parallel. In such a scenario, having a separate MPI context with dynamic resources could be used not only to avoid idling MPI processes, but also to avoid applications influencing each other, e.g., due to a bug.

Invasive Computing (IC): The IC paradigm suggests varying resource utilization for embedded systems [15] with certain progress to adopt this also in HPC. As a first step, the OpenMP and Threading Building Blocks parallelization models have been modified to allow for varying resources, see e.g. [13]. Here, the underlying idea has been to allow a resource manager to improve the system-wide efficiency for concurrently running applications and to start applications at arbitrary points in time where the present work also borrows this

idea of a resource manager. Based on the aforementioned work, an extension for distributed-memory systems with MPI was developed [3]. However, this led to several drawbacks of this approach, such as that resource changes are solely based on `MPI_COMM_WORLD` (which is obviously a serious problem for, e.g., coupled simulations) and that only specialized cases have been taken into account.

MPI fault tolerance: Work on MPI fault tolerance approaches, including MPI *global restart* [9] and the Fenix project [5], share some of the features of *libmpidynres*. Both provide mechanisms for an application to recover from an initial loss of compute resources and utilize replacing resources when available, but their functionality is limited in scope to resilience. Our proposed approach, on the other hand, covers a much wider field, but can be used to implement the recovery models supported by these approaches to fault tolerance.

Malleability in MPI: Dynamic resource management in MPI has been studied intensively over the last years, usually under the umbrella term of "malleability". Since then, multiple frameworks have been created to support malleability in MPI applications [4, 14, 11, 6, 8]. These frameworks used different techniques and APIs to achieve malleability. For example, some authors propose process splitting and merging for expansion and shrinking of the application [4, 8]. Other authors start new processes while keeping the old, existing processes running [14]. Another approach is to use checkpointing systems and restart the actual MPI application for resizing it [10, 12].

Although there has been much research around malleability, there is still a lack of a highly flexible, efficient and future proof API. The work presented in this paper takes the attempt to propose such an API and further differs from previous approaches by its use of MPI Sessions for malleability.

2 MPI Sessions

We start with a brief introduction to MPI Sessions since this is at the core of our proposal.

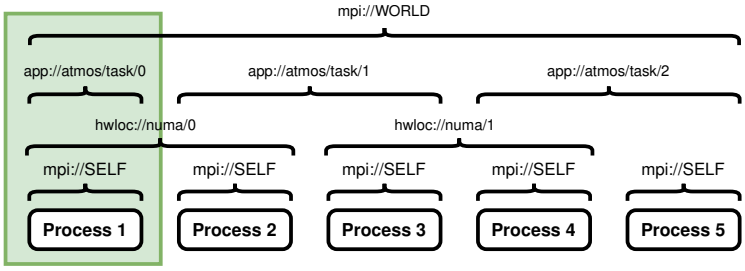

Fig. 1. The process sets of an application example. Process sets are represented as curly brackets. MPI Process 1's view on its process sets is highlighted in green.

The concept of MPI Sessions was first introduced in 2016 [7] and was later included into the MPI standard with the release of MPI 4.0 in June 2021 [7, 16]. It defines a new object, the *MPI Session*, which is a lightweight handle to the MPI runtime. Using a session, MPI can be initialized without the `MPI_COMM_WORLD` communicator. Further, an application and its libraries can allocate multiple independent sessions allowing for better isolation and a higher degree of composability compared to traditional global MPI intitialization. MPI Sessions also offer a tighter runtime integration by allowing the runtime to expose available resources via *process sets*. A process set groups together multiple potential MPI processes and is identified by a name in a URI-like format (e.g., "mpi://world", "mpi://self"). Using process sets, an application can create local MPI groups, which can be further used to create communicators that connect MPI processes contained in the respective process sets.

Process sets allow the runtime to expose available resources to the application. Fig. 1 shows an example view of an application's process sets. A process set can represent something of a static nature (e.g., a NUMA node, "mpi://numa/0" in Fig. 1), but it can also represent more dynamic groups of resources (e.g., a specific task in an application: "app://atmos/task/1" in Fig. 1). There is still ongoing discussion about the exact nature of process sets, their lifetime, scope and dynamic behavior. Our approach takes a look at process sets from a more dynamic perspective, which leads to the following assumptions for the remainder of the paper:

Immutability: A process set identified by a unique name will always represent the same resources, even if resources are not actually available to an MPI application, e.g., because the job scheduler removed the resource during a resource change. This property avoids race conditions in cases where an MPI group is created from the process sets by different MPI processes.

Change of process sets: We expect the available process sets to change frequently during an application run. At the same time, the number of available process sets at any point in time is expected to remain small as process sets that become invalid (due to an MPI process exiting a process set) are removed in our model (compare with Sec. 4.2).

3 Dynamic Resources with MPI Sessions

3.1 MPI Sessions Advantages Compared to `MPI_COMM_WORLD`

Dynamic resources are non-trivial to implement in the traditional global MPI architecture with `MPI_COMM_WORLD`. This is because `MPI_COMM_WORLD` needs to be mutated or invalidated when resources are added or removed. As a consequence, communicators that originate from `MPI_COMM_WORLD` would need to adapt or get invalidated together with associated rank and size information, which is hard to do in a consistent fashion that is transparent to an application.

MPI Sessions provide one way to tackle the aforementioned problem by allowing `MPI_COMM_WORLD` to be avoided entirely. Besides various other benefits, we briefly discuss the main advantages in the context of varying resources.

In our work, process sets are used to globally express resource changes. Once an MPI object is invalidated, new MPI objects from process sets can be created without the need for complex application coordination. Also, there is no need to change the mechanism and semantics of MPI groups and communicators like we would may need to do in the mutable `MPI_COMM_WORLD` case. Another advantage of MPI Sessions is given by the current interfaces that already permit the dynamic modification of an applications point of view on available MPI processes by changing the process sets that are exposed to the application.

3.2 Resource Changes with Process Sets

Next, we discuss our strategy to realize dynamic resource changes. We would like to point out, though, that our approach focuses on loop-based applications similar to the application shown in Sec. 6.

Resource changes happen when the runtime removes or adds new resources from/to an application. For our dynamic resource model we assume that the runtime does not implicitly add new resources in the form of a new process set, but an explicit function call needs to be made by the application. From the application point of view, implicit adding of resources is problematic due to assumptions on a particular number of resources in a communicator, e.g. the number of ranks. This explicit approach is also useful as the application might need to do load balancing/process coordination work after each resource change. Once a resource change arrives, the application has a time window to do cleanup/load balancing and then accepts the resource change. This is especially important when MPI processes are being removed because the data from these MPI processes needs to be transmitted to avoid data loss.

A resource change consists of a *resource change type* and a *resource change process set*:

The *resource change type* indicates how the application's resources are modified. In the present work, we only investigate two resource change types: *addition* and *removal* of processes. To migrate resources, both operations have to be applied sequentially. Obviously, a *replace* resource change type could be also implemented that both removes and adds processes, as well as a *split/join* change that (de-)partitions existing process sets. However, these type of changes are not the focus of this work.

The *resource change process set*, on the other hand, describes the difference between the current set of processes and the set of processes after the resource change. In the case of MPI process addition, the resource change process set will contain all to-be-started MPI processes and in the case of MPI process removal all to-be-removed MPI processes.

4 Interface Design

4.1 MPI Sessions Interface

libmipdynres's MPI Sessions interface was developed along the lines of the Sessions interface in the MPI 4.0 draft from November 2020 [18]. The draft's inter-

face description matches the one that was finally published with the official MPI 4.0 standard in June 2021 [16].

The MPI 4.0 standard document defines multiple C signatures of MPI Sessions functions and explains the semantics of these functions. However, the document does not fully define all concepts of MPI Sessions. Many questions remain open in regard to process sets. Because of that, we have modified and extended the MPI Sessions interface to fit the way process sets are viewed in this work (see Sec. 2). The MPI Sessions interface that is included in our library, *libmpidynres*, contains the following functions:

- `MPI_Session_init` - initialize an MPI Session
- `MPI_Session_finalize` - finalize an MPI Session
- `MPI_Session_get_info` - query information about an MPI Session
- `MPI_Session_get_psets` - query for available process sets
- `MPI_Session_get_pset_info` - query information about a process set
- `MPI_Group_from_session_pset` - create an MPI group from a process set
- `MPI_Comm_create_from_group` - create an MPI communicator from an MPI group without a parent communicator

These functions match the functionality and semantics described in MPI 4.0 [16], except for `MPI_Session_get_psets`, which we discuss in the next section.

4.2 `MPI_Session_get_psets`

The signature of `MPI_Session_get_psets` is shown in Fig. 2.
The function replaces two functions in MPI 4.0: `MPI_Session_get_num_psets` and `MPI_Session_get_nth_pset`. These two functions assume a more static behavior of process sets, as they use a virtual array model for querying process sets. With the `MPI_Session_get_num_psets` function one can query the length of the virtual array and with the `MPI_Session_get_nth_pset` one can query a process set at a specific index. The runtime can only append new process sets to the array, an index can become invalid if the process set does not exist anymore. This approach has multiple disadvantages with our assumed process set properties (see Sec. 2):

1. The frequent change of process sets leads to an ever-growing array that will lead to increasing memory usage and increasing access times.
2. The frequent change will also lead to most indices being invalidated at some point in time. This in return increases the chance of invalid requests and creates an additional management overhead on the application side.

To adapt the API to our model, we replace the two function calls with one. Instead of querying each process set on its own, the application queries the names and sizes of available process sets at once. The result is returned in the `psets` argument of `MPI_Session_get_psets`. It consists of an MPI Info object with process set names as the keys and the respective process set size as the value, basically representing a snapshot of the current process set state. While this leads to more data being transferred, we expect the number of active process sets at any point in time to remain low. However, to make the API more future-proof and allow for more complex process set situations, an MPI Info object can be passed to the function. This object could be used to filter the results and only return a subset of available process sets.

```
int MPI_Session_get_psets(MPI_Session session, MPI_Info info,
                          MPI_Info *psets);
IN    info  Info object containing runtime hints
OUT psets Info object containing process set names as keys
          and process set sizes as decimal values
```

Fig. 2. *libmpidynres* API for querying available process sets.

```
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    hints       Hints passed to runtime
IN    pset1       Name of first process set argument
IN    pset2       Name of second process set argument
IN    op          Operation type to apply
OUT pset_result Name or resulting process set
```

Fig. 3. Proposed API for creating process sets by applying a set operation on existing process sets.

4.3 Process Set Management Interface

When dealing with resource changes, an application must be able to establish communication with new resources. In our work, new resources are expressed via process sets. To establish communication, an application can create an MPI Group from the new process set and use MPI group operations to create a group that both contains the new processes and old application processes. However, this approach has to be made by each process in the new group. This can become quite complex with increasingly more resource changes and is hard to coordinate. This is especially a problem for the newly created processes as they need to know which process sets they need to use to create MPI groups containing old application processes. To avoid these problems, we allow the application to create new process sets.

In our design, only one MPI process, which we will refer to as "main process", is responsible for operations on process sets. Note that these operations could be executed by *all* involved MPI processes, but leave this to future work and here strictly follow the "main process" approach.

To create new process sets, the application must call the `MPIDYNRES_pset_create_op` function, whose signature is shown in Fig. 3. The names of two existing process sets need to be given in the `pset1` and `pset2` arguments. Additionally, a set operation to be applied needs to be passed in the `op` argument. Calling the function has the effect that, if the arguments are valid, the runtime will create a new process set containing the result of the set operation on the process sets. Currently, three set operations are supported, see also Fig. 4:

- **Union**: The result contains all processes from both psets. This can be used to add new processes from a resource change set to the application's main process set.
- **Difference**: The result contains all processes from `pset1` that are not in `pset2`. This is useful to remove a resource change set (if the resource change takes away resources) from the application's main process set.

– **Intersection**: The result contains all processes that are both in `pset1` and `pset2`.

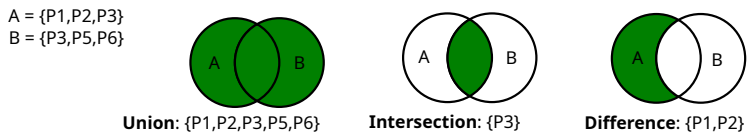

Fig. 4. Venn diagrams of the process set operations. Process set A contains the processes P1,P2,P3 and P4; Process set B contains the processes P3,P4,P5 and P6. The operation's result is written underneath each Venn diagram.

Allowing these fundamental set operations has multiple advantages which we like to summarize as follows:

– Process set changes do not rely on collective operations involving, e.g., processes which have not yet been started. Therefore, process sets including processes not yet available to the application (see Fig. 8) can be created.
– Since resource changes can be abstractly described as a directed acyclic graph as transitions on resource sets, designing an interface supporting such resource changes should also cover the typical requirements of such resource changes without taking application-specifics into account.
– A "main process" driven change of resources makes the process coordination easier since all management can happen in a single process. The only information that needs to be shared with other processes are process set names which are available globally. An extension to a consensus-based management should still be possible.
– Process set operations fit nicely into applications that rely on a single communicator during their runtime. There, the operations can be used to derive a new "main process set" from the previous "main process set" and the resource change process set (see Fig. 5).

Our main goal is the creation of rather generic interfaces to cover various requirements on resource change patterns. As usual, there are always optimizations possible by providing specialized interfaces or more feature-rich interfaces, however this is left for future work.

4.4 Resource Change Management Interface

To implement the mechanisms described in Sec. 3.2, the resource change API has to provide a way to a) query for pending resource changes and b) accept and apply these resource changes.

For a), *libmpidynres* offers the `MPIDYNRES_RC_get` function. Its signature is shown in Fig. 6. If there is a resource change, the type of resource change and the resource change process set are returned in the `rc_type` and `delta_pset` respectively. Furthermore, a handle to the resource change is returned in the `tag` argument.

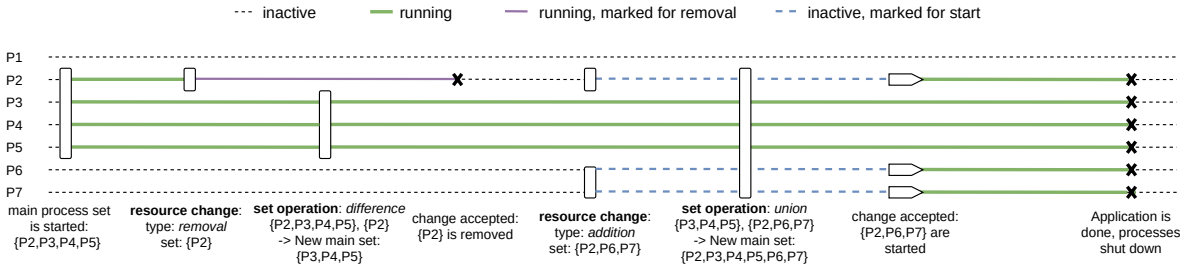

Fig. 5. Diagram showing how to use process set operations for resource changes.

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

int MPIDYNRES_RC_accept(MPI_Session session,
        MPIDYNRES_RC_tag tag, MPI_Info info);
IN tag  Identifier of the resource change to accept
IN info Runtime hints and hints for newly created processes
```

Fig. 6. Proposed API for managing resource changes.

For b), *libmpidynres* offers the `MPIDYNRES_RC_accept` function. Using this function, the application can tell the runtime to *apply* the resource change referenced by the `tag` argument. The `info` argument can be used to pass information new processes. Once this function is called with valid arguments, the runtime will start new processes in the case of resource addition. If the resource change removes resources, the application has to shutdown the relevant processes itself. If possible, the runtime can try to enforce the shutdown by forcefully shutting down running processes after a specific amount of time. In the case of *libmpidynres*, due to its architecture, the shutdown cannot be enforced.

An example application execution with both resource changes and process set operations in shown in Fig. 5. The application shown constructs a new "main pset" after each resource change. Note that the "main process" main thread is not highlighted, as it is application dependent to choose a main rank. One possible way to choose a "main process" is to use rank 0 of the communicator based on the "main pset".

5 libmpidynres

In order to evaluate our proposal, we implement the runtime component in the form of a C library, called *libmpidynres*, that emulates a dynamic resource environment on top of an existing MPI communicator.

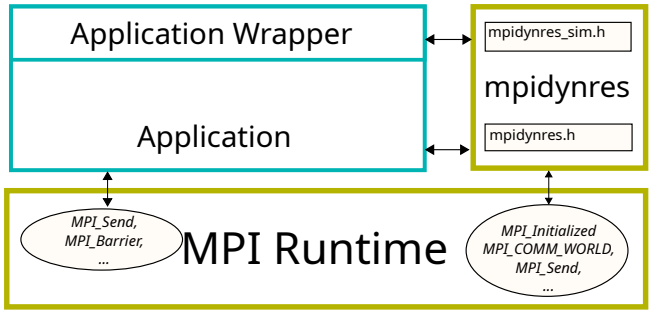

Fig. 7. The different components of an application using *libmpidynres*. Both the application and *libmpidynres* access the same MPI library. However, the application should derive most of its MPI objects from *libmpidynres*.

The library uses a communicator of fixed size to emulate an MPI Sessions environment with dynamic resources by using subsets of the fixed-size communicator. This is achieved by hiding and exposing the processes of the communicator to the application. Using this emulated environment, applications can use the MPI Sessions and dynamic resource management API described in Sec. 4, hence already explore and test these features even if the underlying MPI implementation and job scheduler do not support MPI Sessions and dynamic resource management. For sake of reproducibility and open science, the source code of libmpidynres is available on GitHub.[5].

5.1 libmpidynres as an Emulation Layer on top of MPI

libmpidynres is implemented as a C library that is used on top of an existing MPI library. This means that *libmpidynres* uses MPI calls internally for communication and management. From the application's point of view, it extends the available MPI API with additional functions.

Before the MPI Sessions environment becomes active, the application has to configure *libmpidynres* and initialize MPI. This part of the user application is called the *application wrapper*. The application wrapper then passes an entry point and a communicator for the emulated application to *libmpidynres*.

From there, *libmpidynres* manages the communicator's processes and runs the *emulated application* from the given entrypoint. The emulated application should only use MPI communicators and groups that are returned by *libmpidynres* or were derived from these. This ensures that *libmpidynres* has full control over the available processes. This architecture is illustrated in Fig. 7.

5.2 Emulated Process States

libmpidynres emulates dynamic resources on a communicator of fixed size (typically `MPI_COMM_WORLD`). This is achieved by selectively exposing a subset of the

[5] `https://github.com/boi4/libmpidynres`

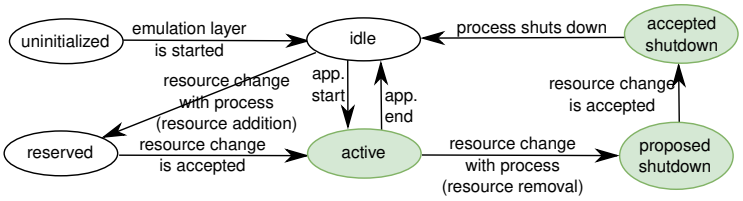

Fig. 8. The different states that an emulated process can be in, from *libmpidynres*' point of view. States where the application has control over the (OS-)process are highlighted in green.

communicator's processes to the application as its world process set. Consequently, the maximum number of processes that can be scheduled is limited by the size of the communicator used for emulation. Inactive processes are idling (in an `MPI_Recv` operation) and are waiting to be requested and then made available to the MPI application. However, unlike a full implementation inside a runtime, such processes cannot be made available to other, separate applications. Therefore, it is again important to stress that *libmpidynres* is only a proof-of-concept library for testing the interface and real support for dynamic resources has to be included in the various software components of the MPI stack.

The process of starting and stopping resources is quite complex and involves multiple temporary states a process can be in. These states are illustrated in Fig. 8. Note that these are the states from the library's point of view.

5.3 Resource Manager

libmpidynres uses a server-client model for managing process sets and resource change states. For that, the MPI process with rank zero of the libmpidynres MPI communicator acts as a dedicated *resource manager*.

This server-client approach avoids race conditions and assures a consistent state across all MPI ranks. However, the additional communication overhead leads to decreased performance especially with redundant requests from multiple ranks and to increased latency when doing API calls. However, the proof-of-concept, emulating nature of *libmpidynres* justifies this trade-off.

6 Case Study

To evaluate the proposed interface and *libmpidynres*, we implement an application example that is based on a loop where work is distributed among all processes in each iteration. In the following, we describe this example application.

Let us first look at the initialization part of the application given in lines 1-10 in Fig. 9. When an MPI process is started, it needs to set up and gain information about its environment. For that, the application initializes an MPI Session using

```
1    session = MPI_Session_init()
2    psets = MPI_Session_get_psets(session)
3    if "mpi://world" in psets:
4      main_pset = "mpi://world"; cur_iter = 0
5    else:
6      info = MPI_Session_get_info(session)
7      main_pset = info.get("main_pset")
8      cur_iter = info.get("cur_iter")
9    group = MPI_Group_from_sessions_pset(main_pset)
10   comm = MPI_Comm_create_from_group(group)
11   for (; cur_iter < N; cur_iter++): /* MAIN LOOP */
12     rc_type, rc_set = MPIDYNRES_RC_get()
13     if (rc_type != NONE):
14       if (rc_type == ADDITION):
15         main_pset = pset_create_op(UNION, main_pset, rc_set)
16       if (rc_type == REMOVAL):
17         main_pset = pset_create_op(DIFF, main_pset, rc_set)
18         psets = MPI_Session_get_psets(session)
19         if main_pset not in psets: break
20       MPIDYNRES_RC_accept({"cur_iter": cur_iter,
21         "main_pset": main_pset})
22       group = MPI_Group_from_sessions_pset(main_pset)
23       comm = MPI_Comm_create_from_group(group)
24       ...
25     do_work()
```

Fig. 9. Pseudo code showing using the proposed interface to successfully query and adapt to dynamic resources changes.

`MPI_Session_init`. Furthermore, the application queries its process sets using `MPI_Session_get_psets`. If the process is part of the "mpi://world" process set, the process was started together with the start of the application. If it is not part of the process set, the process was started because of a resource change and has to query some information to successfully join the application. In this example, it queries the current loop iteration and the process set that should be used for communication from its MPI session (the information was passed with an Info object when the resource change was accepted).

Once a communicator is created from the `main_pset`, the main loop is started. This is shown in lines 11-25 of Fig. 9. The application queries for resource changes at the beginning of each loop iteration. When dealing with resource changes, the application follows the strategy from Fig. 5. This means that the application tries to have all of its available resources grouped together in one process set, the "main process set". If a resource change adds new resources, the *union* process set operation is used to create a new "main process set". If a resource change removes existing resources, the *difference* process set operation is used instead. When the application accepts a resource change using `MPIDYNRES_RC_accept`, some information (the main process set name and the current loop iteration) are passed to newly started processes.

Using this system, the application is able to handle and adapt to resource changes while constantly having a valid MPI communicator. A concrete C implementation of this application was tested and evaluated using different scheduling algorithms and different communicator sizes. The application could successfully finish all of its loop iterations without any crashes or race conditions in the application or *libmpidynres*. More *libmpidynres* examples can be found on GitHub.[6].

7 Conclusion

In this work, we presented an interface that uses new MPI Sessions concepts to handle dynamically varying resources. The interface uses process sets to express resource differences that will be applied to the application. We implemented an emulation layer that allows applications to use the new interface. This makes prototyping of malleable applications with the proposed interface possible, even without MPI providing support for this, yet. Furthermore, using an example application built on top of this emulation layer, we have validated that using this interface, applications are capable of dealing with resource changes.

Regarding future work, one of the next steps is an extension of the prototype with the implementation of different parallel programming patterns (beside the *loop pattern*) and combine them with the interface proposed in this work. While the current interface is quite general and therefore may be useful for other programming patterns, it still provides some global changes that may affect all processes of the application. For more distributed programming patterns, a less global approach is needed where the application can group its own processes and tell the runtime that only certain groups should be affected by resource changes.

Another interesting area to apply this new interface to are existing tools for dynamic computing. For example, tools like *p4est* and *PETSc* can help with automating parts of the load balancing process in dynamic mesh refinement applications [2, 1]. Integrating dynamic resources into these tools is currently work-in-progress and could abstract the dynamic resources away from the library user and ease the creation of scalable parallel applications.

Besides many other future research aspects, we like to finally point out the problem of scheduling, which will require disruptive algorithms to cope with runtime-varying resources.

Acknowledgments: 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 Union's Horizon 2020 research and innovation programme and Belgium, France, Germany, Switzerland.

References

1. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., et. al: PETSc Web page. `https://www.mcs.anl.gov/petsc` (2019), `https://www.mcs.anl.gov/petsc`

[6] `https://github.com/boi4/libmpidynres/tree/master/examples`

2. Burstedde, C., Wilcox, L.C., Ghattas, O.: p4est: Scalable Alg. for Parallel Adaptive Mesh Ref. on Forests of Octrees. SIAM J. Sci. Comput. **33**(3), 1103–1133 (2011)
3. Comprés, I., Mo-Hellenbrand, A., Gerndt, M., Bungartz, H.J.: Infrastructure and API Ext. for Elastic Execution of MPI Applications. Proc. 23rd EuroMPI (2016)
4. El Maghraoui, K., Desell, T.J., Szymanski, B.K., Varela, C.A.: Dynamic Malleability in Iterative MPI Applications. In: Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07). pp. 591–598 (2007)
5. Gamell, M., Katz, D.S., Kolla, H., Chen, J., Klasky, S., Parashar, M.: Exploring Automatic, Online Failure Recovery for Scient. Appl. at Extreme Scales. In: SC '14: Proc. of the Int. Conf. for HPC, Netw., St. and Anal. pp. 895–906 (2014)
6. Gupta, A., Acun, B., Sarood, O., Kalé, L.V.: Towards realizing the potential of malleable jobs. In: 2014 21st Int. Conf. on HPC (HiPC). pp. 1–10 (2014)
7. Holmes, D., Mohror, K., Grant, R.E., Skjellum, A., Schulz, M., Bland, W., Squyres, J.M.: MPI Sessions: Leveraging runtime infrastructure to increase scalability of appl. at exascale. ACM Int. Conf. Proceeding Ser. **25-28-Sept**, 121–129 (2016)
8. Iserte, S., Mayo, R., Quintana-Ortí, E.S., Peña, A.J.: DMRlib: Easy-Coding and Eff. Res. Mgm for Job Malleab. IEEE Transact. on Comp. **70**(9), 1443–1457 (2021)
9. Laguna, I., Richards, D., Gamblin, T., Schulz, M., De Supinski, B., Mohror, K., Pritchard, H.: Evaluating and Extending User-Level Fault Tolerance in MPI Applications. Int. J. of HPC Applications **30**(3), 305–319 (2016)
10. Lemarinier, P., Hasanov, K., Venugopal, S., Katrinis, K.: Architecting Malleable MPI Applications for Priority-Driven Adaptive Scheduling. In: Proceedings of the 23rd European MPI Users' Group Meeting. p. 74–81. EuroMPI 2016, Association for Computing Machinery, New York, NY, USA (2016)
11. Martín, G., Marinescu, M.C., Singh, D.E., Carretero, J.: FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013 Parallel Processing. pp. 138–149. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
12. Prabhakaran, S., Neumann, M., Rinke, S., Wolf, F., Gupta, A., Kale, L.V.: A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In: 2015 IEEE Int. Par. and Dist. Proc. Symposium. pp. 429–438 (2015)
13. Schreiber, M., Riesinger, C., Neckel, T., Bungartz, H.J., Breuer, A.: Invasive Comp. Balancing for Appl. with Shared and Hybrid Par. Int. J. Parallel Program. (2015)
14. Sudarsan, R., Ribbens, C.J.: ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. In: 2007 International Conference on Parallel Processing (ICPP 2007). pp. 44–44 (2007)
15. Teich, J., Henkel, J., Herkersdorf, A., Schmitt-Landsiedel, D., Schröder-Preikschat, W., Snelting, G.: Invasive Computing: An Overview, pp. 241–268. Springer New York, New York, NY (2011)
16. The MPI Forum: MPI: A Message-Passing Interface Standard Version 4.0
17. The MPI Forum: MPI: A Message-Passing Interface Standard Ver. 2.2 (09 2009)
18. The MPI Forum: MPI: A Message-Passing Interface Std. Ver. 4.0 (Draft) (11 2020)
19. Zafari, A.: Advances in task-based parallel programming for distributed memory architectures (PhD dissertation) (2018)