Enabling Global MPI Process Addressing in MPI Applications

Jean-Baptiste BESNARD
jbbesnard@paratools.fr
ParaTools SAS
Bruyères-le-Châtel, France

Sameer SHENDE
Allen MALONY
sameer@paratools.com
malony@paratools.com
ParaTools Inc.
Eugene, USA

Julien JAEGER
Marc PERACHE
julien.jaeger@cea.fr
marc.perache@cea.fr
CEA, DAM, DIF
Bruyères-le-Châtel, France

ABSTRACT

Distributed software using MPI is now facing a complexity barrier. Indeed, given increasing intra-node parallelism, combined with the use of accelerator, programs' states are becoming more intricate. A given code must cover several cases, generating work for multiple devices. Model mixing generally leads to increasingly large programs and hinders performance portability. In this paper, we pose the question of software composition, trying to split jobs in multiple services. In doing so, we advocate it would be possible to depend on more suitable units while removing the need for extensive runtime stacking (MPI+X+Y). For this purpose, we discuss what MPI shall provide and what is currently available to enable such software composition. After pinpointing (1) process discovery and (2) Remote Procedure Calls (RPCs) as facilitators in such infrastructure, we focus solely on the first aspect. We introduce an overlay-network providing whole-machine inter-job, discovery, and wiring at the level of the MPI runtime. MPI process Unique IDentifiers (UIDs) are then covered as a Unique Resource Locator (URL) leveraged as support for job interaction in MPI, enabling a more horizontal usage of the MPI interface. Eventually, we present performance results for large-scale wiring-up exchanges, demonstrating gains over PMIx in cross-job configurations.

CCS CONCEPTS

• **Networks** → **Peer-to-peer protocols**; **Network experimentation**; • **Computer systems organization** → Fault-tolerant network topologies; **Interconnection architectures**.

KEYWORDS

MPI, Peer-to-peer, PMI, RPCs

ACM Reference Format:
Jean-Baptiste BESNARD, Sameer SHENDE, Allen MALONY, Julien JAEGER, and Marc PERACHE. 2022. Enabling Global MPI Process Addressing in MPI Applications. 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, 10 pages. https://doi.org/10.1145/3555819.3555829

EuroMPI/USA'22, September 26–28, 2022, Chattanooga, TN, USA

ACM ISBN 978-1-4503-9799-5/22/09…$15.00
https://doi.org/10.1145/3555819.3555829

1 INTRODUCTION

The Message Passing Interface (MPI) is crucial in the software stack of any supercomputer. Indeed, the need for improved processing requires the ability to coordinate multiple nodes. For this purpose, MPI has evolved over the years to respond to all the needs expressed by the High-Performance Computing (HPC) community. Its ability for evolution is no stranger in the long-lasting availability of MPI – recently celebrating its 25 years. Providing bare-metal performance on specific hardware while acting as a portable mediation layer now deeply embedded in scientific code-bases is a root cause of this state of things. Among others, MPI has evolved to provide improved non-blocking collectives[10], added support for shared-memory windows[9], and last but not least for MPI Sessions[11] – completely changing how MPI launches itself. Conjointly, the way of addressing the MPI process remained unchanged since the first version of the standard: MPI process ranks are identified by a 32 bits integers part of a given communicator. As far as communicators are concerned, they used to be relatively static, mostly a subset of the main `MPI_COMM_WORLD` communicator. However, sessions now enable the immediate creation of arbitrary communicators thanks to a new naming mechanism – the process-set (*pset*) name. These changes open wider connectivity opportunities between MPI processes and pose issues to runtimes that may now lack internal addressing capabilities, being tailored for the world model.

These recent MPI evolutions are responding to an increasing hardware pressure for composability. Indeed, MPI is now generally collocated with a shared-memory model such as OpenMP, spanning devices of different kinds (CPUs, GPUs). In this heterogeneous landscape, a static `MPI_COMM_WORLD` is insufficient: a program may be the composition of several jobs. In the first part of this paper, we motivate such application deployment that we call *service oriented HPC*. In particular, we identify process discovery and remote procedure calls as tools, respectively incomplete and lacking in MPI. We then outline the rich client-server model already available in MPI. Subsequently, we develop the requirements associated with inter-job wiring and focus on process discovery, which is not fully addressed by existing process management interfaces. For this purpose, we introduce our contribution leveraging an overlay network[16] connecting all MPI processes of a given user, providing *machine-wide* wiring and side-channel capabilities to MPI. This network is presented in the context of the MPC MPI[3, 18] runtime. By combining the overlay network and dedicated addressing capabilities (Unique IDentifiers for each process) we describe how we robustly build cross-job communicators. Moreover, we show how we leverage routed remote procedure calls (RPCs) to enable

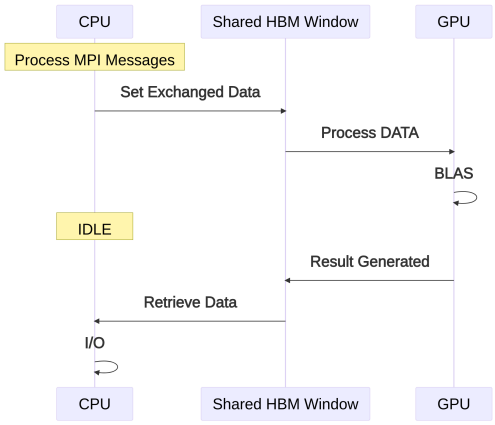

Figure 1: Illustration of GPU only computation with CPU only acting as coordinator for data-movements.

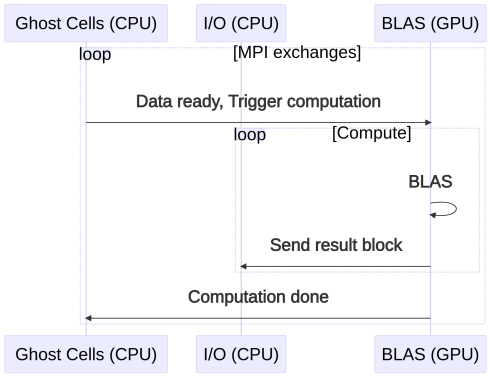

Figure 2: Computation of Figure 1 outlined in a service-oriented manner.

on-demand connectivity allowing our MPI runtime to start fully-disconnected (on the high-speed network side). Eventually, we compare our approach with PMIx[5] and demonstrate improvements in some configurations (sparse connectivity). We then conclude with future applications we envision for the overlay network.

2 EVOLVING CONSTRAINTS

High-Performance Computing is going through important evolution in terms of hardware. The tendency for hybridization is confirming itself: hardware will be fragmented from now on in special units[8, 15, 22]. In such diverse environment, programming a single program dedicated to multiple computing units is not trivial and can lead to inefficiencies. Indeed, if a single portion of the code is not fully scaling, for example due to fork-join overheads, the whole application is subject to *partial speedup bounding*[4]. Therefore, if alternating between node-level and inter-node parallelism was difficult for MPI+X; it will be close to impossible in MPI+X+Y. Our take on this challenge is core specialization instead of trying to map a given code to an overly complex machine topology. Rather of a single program taking care of the whole computation, alternating between linear algebra and Inputs/Outputs – we propose to move individual processing to the most suitable units while being *serviced* to the other processes.

2.1 Service Oriented HPC

As illustrated in Figure 1, one may consider BLAS on the GPU and I/O burst buffer taking advantage of the node's main memory while preserving computational matrices in a shared-window mapped to the node-local High-Bandwidth Memory (HBM) tying both the GPU and the CPU to a single common address space. Using this configuration, one can see that the CPU is idle during the linear algebra phase. It is of course possible to enhance this code. Practically splitting the work between the two devices would need a mix of tasks and certainly, OpenMP targets[6] creating nested parallelism regions. It means a single program has to encompass multiple devices and therefore stack multiple states to accommodate the various devices. If we now consider the alternate *service-oriented* approach for Figure 2, the control flow is more consistent. Indeed, instead of having control going back and forth between the various computing units, arrows are now closer to *tasks* carrying both data and a given processing order on these data. Moreover, in this illustrative example, we now have two services located on the CPU, I/O and MPI Ghost-Cell exchanges. These two services are simpler than the original code, which had to maintain multiple states. They expect a given order, trigger their operation, respectively ghost-cells exchanges and I/O storage, and then report for completion. Besides, the machine's scheduler can be sufficient to enable resource sharing between these two components without having to explicit alternated control flows unlike in Figure 1. In HPC, this horizontal scaling of applications is not common, in-situ[7], ad-hoc services[25] and work-flow oriented schedulers such as Flux[1] are pioneering these original configurations. More practically, looking at how the Internet is structured and able to scale, we have a convincing analogy of what we try to bring in this initial example.

2.2 Summary

Overall, coordinating simpler components instead of encompassing all the cases in an omniscient code leads to immediate code simplifications. First, in terms of components that can do only one thing, and more importantly the global state is now decoupled. Components interact in point-to-point *without remote state assumption*. Moreover, a given *service* could be shared between multiple jobs to increase its load. This aspect is at the core of this paper as our contribution focuses on enabling *machine-wide* MPI process discovery and wiring. As far as communications are concerned, doing so with MPI as of today is perfectible. Indeed, MPI is mostly relying, for good performance reasons, on two-sided messages which necessarily lead to a known remote state. Of course, one may consider looping over receives using `MPI_ANY_SOURCE` to process incoming data but it would make poor use of MPI's potential. Another approach would be to rely on one-sided operations using target notification thanks to a flag constantly monitored by the target code but this might not be portable to all hardware (GPUs, ...) and would waste computing cycles due to the active waiting in user

space. In short, what is discussed in the watermark here to empower such scenarios is active messages backed up by the MPI runtime for efficiency and portability. Such a paradigm using, for example, the Mercury RPC framework[21, 24] or eRPC[14] is already gaining traction in massively parallel task engines and I/O frameworks[25].

3 COMPOSITION IN MPI

One of our main goals in this paper is to enhance the composability of MPI programs enabling cross-job MPI process interactions. However, it would be inexact to ignore what the standard already proposes. Indeed, MPI has a rich client-server model enabling two groups of processes to discover themselves and exchange data using regular MPI messages. There are three main approaches, (1) gathering processes through an intermediate substrate (file-descriptor or Port) and (2) spawning a new set of processes to interact with or (3) relying on Multiple-Program Multiple-Data launch. These are presented here mostly as related-work, contextualizing what MPI is already capable of.

3.1 MPI Connect/Accept

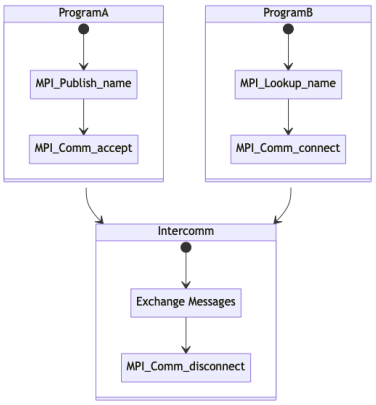

Figure 3: Sample client-server MPI sequence diagram using Connect/Accept

As depicted in Figure 3, process discovery is done using a port in MPI. A port can be published by a given process and then queried by another. Such port establishes a common ground to build an inter-communicator between the two sets of processes using `MPI_Comm_connect` and `MPI_Comm_accept` – yielding a new inter-communicator. Unlike their counterpart, intra-communicators, inter-communicators are bipartite structures enabling two groups of processes to communicate. In this case, messages' ranks are relative to the remote communicator group. Besides, collective communications have another semantic. Alternatively, `MPI_Comm_join` enables two processes to join their groups thanks to a common file-descriptor. It has a similar outcome as previously described connect/accept despite using a different substrate and limiting interaction with a single remote process.

As far as the scoping of the service name is concerned, it is implementation-dependent and varies from a single job to the whole machine depending on support. It can then be cumbersome to write portable code using this semantic. Consequently, the two methods covered in upcoming sections gained more traction. In this paper, we implement a dedicated overlay network to guarantee this scoping is machine-wide for a given user – making connect/accept more portable.

3.2 Spawning Processes

The `MPI_Comm_spawn` procedure is another way of enabling composition in MPI. This call leverages the infrastructure to allocate a new set of processes within the limit of the universe size, which is implementation-dependent. When a given process is spawned with its command line and given the number of slots, the resulting configuration is also an inter-communicator, which can be retrieved by invoking `MPI_Comm_get_parent` on the child processes. This approach typically relies either on the batch-manager or pre-allocated resources to provide extra on demand processes. It has the advantage of controlling process startup: necessarily creating a notion of inheritance between jobs (same user). In practice, the Process Management Interface (PMI) instructs the batch manager to allocate new processes with a spawn command. Then, the wiring is a special case of Connect/Accept with implicitly shared parameters.

3.3 MPMD Programs

Up to MPI 4.0, The most common manner used to start multiple programs in the same MPI environment is the Multiple Program Multiple Data (MPMD) launch facility. It means that different binaries are passed to the `mpiexec` command using the non-mandatory ":" separator. Implementation is transparent to the MPI library: these processes end up, given the world-model, in `MPI_COMM_WORLD` with a linear MPI process ranking – just as if there was a single binary at play. Consequently, MPMD applications must be aware of their collocated nature to build the corresponding communicators. MPI provides an attribute `MPI_APPNUM` on `MPI_COMM_WORLD`, it can be used as the split value to create a communicator for each application. With some extra logic, sizes and ranges can be exchanged for the various MPMD programs to enable later computation. Besides, in complement of the aforementioned world-model, MPI 4.0 brought the new Session model which allows processes to start disconnected. With Sessions, communicators for the various applications can then be started from a given process-set, for example, app://ocean.

The relative simplicity of MPMD in terms of implementation combined with its portability made this model of composition prominent in terms of MPI usage.

3.4 Summary

In the light of previous discussions, we have seen that the MPI standard already supports inter-job wiring. It can be achieved, first between separate jobs using Connect/Accept (or a file-descriptor) as the most general technique. Otherwise, a given program may spawn its processes. However, it is of limited use in the case of planable resources (e.g. side-support service), meaning that the most suitable model is MPMD – launching two distinct programs in the same communication domain. Yet, in the later case, the service instance would be limited to a single job. Connect/Accept then seems to be a good avenue to achieve more dynamic configurations

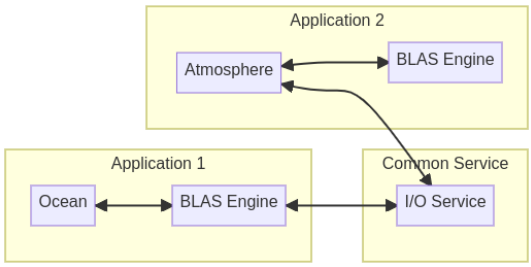

Figure 4: Example of two applications sharing the same I/O backend

such as the one presented in Figure 4. Such configuration is already used in today's systems: I/O services like GekkoFS[25] provide such support thanks to the Mercury RPC engine[21, 24], enabling multiple applications to share the same ad-hoc file-system. The questions is then how can MPI reliably expose such facility? Why current support despite its diversity is not leveraged? For example dynamic processes were judged as the most useless feature of MPI in a recent survey[2, 12].

3.5 Related Work

In this paper, we try to make the process-discovery mechanism more portable in MPI; this while outlining what MPI needs to provide such a facility. In particular, we propose a dedicated process discovery architecture fully integrated into the MPI library, enabling portable process discovery indifferently from the underlying batch system or PMI version. On this aspect, it is crucial to underline that PMI(x)[5] is evolving as a standard[1] and that it is close to providing the same facilities – particularly considering the tool attach example. However, as we further outline, our interface relies on dedicated remote-procedure calls with a simple registration mechanism. In addition, we provide reliable Unique IDentifiers (UIDs) generation handled by MPI across jobs. We believe that this enables more opportunities for MPI to rely on an alternative networking facility (overlay network) to generate out-of-band messages. This state of things is outlined in the rest of this paper but also in the future work, backing up the interest of keeping some of these facilities in the MPI vicinity. Besides, this does not exclude that some of this support is ultimately provided by PMIx as the standard evolves. There are similarities between the overlay network and the Distributed Virtual Machine (DVM) in the reference PMIX implementation[2] featuring an "overlay runtime". In our case, we focus on (1) set discovery (size, command, and UIDs) and (2) RPCs between these processes – nothing more. The DVM itself acts as a complete scheduler handling allocations and implementing PMIx in a portable manner over various schedulers having varying PMI(x) support. Indeed, the PMI ABI is still evolving and leads to challenges in contexts such as containers[13, 26] requiring the PMIx runtime to act as a mediation layer with a potentially unknown host system and PMI version.

[1] https://pmix.org
[2] https://openpmix.github.io/

Our work falls in the gray area of process management for MPI. Recently MPI has provided extended features to gather and structure process groups: MPI Sessions. Meanwhile, the support to build such groups remains implementation-dependent. In most cases, one uses the PMI as the de-facto standard for wiring up. Yet, PMIx is still evolving and becoming complex in some aspects, possibly due to the standardization constraints: being exhaustive and future-proof. Not all systems immediately support PMIx: it leads to a dependency between the support infrastructure and some MPI features. In this paper, we outline the interest of having (1) UIDs and (2) RPCs, both missing in PMIx, as an alternative to Put/Get which are sensitive to remote states. We also show that with a relatively simple interface and routing, one could ensure inter-job connectivity mostly independent from PMIx.

4 CONTRIBUTION

In the rest of this paper, we introduce an MPI out-of-band control message interface relying on routed messages over an overlay network. This facility enables the MPC runtime to start fully-disconnected and supports extended service discovery and subsequent wiring between all the jobs of a given user. To do so, we introduce the notion of process unique identifier (UID) at the machine level, outlining how we construct such value. We also present the overlay network and its routed RPC interface. Currently, this interface supports on-demand connections, liveliness assessment, connectivity dumps, and process discovery. The interest of this approach is to enable cross-job interaction in MPI, (1) discovering remote peers and (2) sending routed RPCs to them in order to establish high-performance links independently from the remote state (enabling client-server model).

5 OVERLAY NETWORK

In this section, we present the overlay network providing the core features of this paper, on-demand connectivity, and process discovery. We start by mentioning the previous implementation in MPC. Then we describe how this support was externalized and extended in a distinct component. On the implementation side, we motivate the need for unique identifiers, introducing our generation mechanism and describing how it propagates in MPI matching handles' internal IDs. On the connectivity side, we comment on the wiring-up protocol and the retained topology inspired by peer-to-peer networks. Eventually, we describe the corresponding application programming interface based on simple remote procedure calls (RPCs) with timeout support.

5.1 Legacy On-Demand in MPC

Before we elaborate on the new features, it is interesting to give details on what was previously present inside MPC for the on-demand mechanism. Indeed, MPC already had support for routed messages, which means that if a given process received a message for a remote process: it would send it over using a simple distance metric. For this purpose, and as illustrated in Figure 5, when MPC started up, it relied on the Process Management Interface (PMI) to start a set of *static* routes according to a configurable topology (ring, k-mesh, k-torus). As the distance metric retained was simply the absolute one-dimensional distance, MPC enforced a "ring" in all topologies

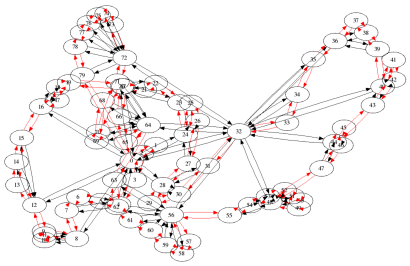

Figure 5: MPC connectivity dump after the MPI_Init barrier (8-tree). In red static routes (ring) in black routes created on-demand thanks to routed "control-messages".

to avoid local routing minimums. Over this mechanism, we relied on a feature internally called control messages; it consisted of small messages routed through the network. These were used for on-demand, RDMA emulation, and internal driver negotiation (RDMA buffer resizing). However, they suffered from several limitations. First, only eager messages were routed due to lack of matching – prescribing the rendez-vous protocol. Second, due to high-speed networks it was cumbersome to do error checking in terms of connectivity, unlike what is handled portably by the operating system with a connected protocol such as TCP. Eventually, in MPC the process identifier was the PMI rank, leading to overlap between multiple jobs due to its linear nature – preventing any form of inter-job connect/accept.

5.2 Networking Topology

To make this feature more reliable, we externalized it over a dedicated out-of-band network using TCP. Thanks to this connected protocol, node failures are easier to detect than over a high-performance network. Indeed, the Operating System (OS) provides support for handling remote failure (potentially trough internal timeout mechanism) and no additional code is needed to handle it at implementation level. Achieving such support over unreliable protocols and/or devices implementing OS bypass requires much more careful handling and is harder to maintain on multiple kinds of fabrics whereas TCP is widely available.

Moreover, instead of using direct connections, we implemented a topology inspired by peer-to-peer networks using the kademlia topology[17]. A given process connects to the following process rank with a power of two strides. Such a network has the propriety of providing a logarithmic distance, an aspect of interest in our case. Moreover, kademlia defines several mitigations for the liveliness of peers as encountered in a peer-to-peer network.

As far as the implementation is concerned, we start a listening TCP socket accepting inbound connections inside each MPI process. Then we store its Internet Protocol (IP) address inside the Process Management Interface (PMI). Each process proceeds to launch the kademlia topology using the keys from the PMI[3]. Once this phase is done, the job can now run fully without using the PMI as the network is now transitively connected. In this process, the root process (PMI rank 0) has a special role. Indeed, it is also in charge of joining the other jobs. For this purpose, we define the notion of launch process-set. A launch process-set is a group of processes gathered in a given job (matching an allocation one-to-one). It is important to note that MPI Sessions have brought in MPI the notion of process-set, which is also a group of MPI processes with a given label (URL). In our case, given our current implementation these two concepts are only partially interleaved as MPC runs in thread-based and therefore our sets are made of UNIX processes, not MPI processes. Yet, in the process-based case, the launch process-set can be exposed as a process-set as envisioned by the MPI Standard due to the identity between UNIX processes (what is launched by the batch-manager) and MPI Processes what communicates in MPI. In the rest of this paper we use the notion of process-set but this small dichotomy caused by MPC's peculiarities with sets from actual MPI Sessions remains.

Each set has a unique identifier computed by the runtime. To do so, we rely on a shared file system to acquire a lock file. This is done by trial and error by trying to create a file in a dedicated directory with a random UID. In prevision for machine-wide multi-user sets, this UID is composed partially of the UNIX user-id (over 20 Most Significant Bits (MSB)) with a random part for the local ID (remaining 12 bits). If the creation succeeds and the content matches the expected value (to avoid possible races), the set is attributed to the given UID. The advantage of such an approach is that listing all process sets is simply doing a listing of the files present in this directory. Besides, as far as the cleanup of this directory is concerned, jobs delete their files when leaving, and for the specific case of crashed jobs, any job failing to contact a given process-set root (which corresponding IP address is inside the file) simply deletes the corresponding file after multiple connection attempts. We then have a kademlia sparse topology inside the job, and between job roots, we have a directory structure enabling inter-job discovery. By carefully placing this directory in the user's home with correct access rights, we can ensure (1) the security of the IP and (2) password protection of the various server endpoints thanks to login information embedded in the set file. This allows a given process to potentially reach processes from other jobs. The routing protocol is very simple. If the destination process set is different, then the message is first sent to the root process of the local set. Inside the root process, a connection is initiated to the destination root in the remote set, and the message is forwarded. Inside sets, the routing uses the kademlia metric to reach the ranks. Thanks to this nested routing, any process is now able to exchange routed messages to other processes on the machine, independent from their respective jobs. Whether the destination process is in the same set can be queried solely from the UID – as detailed in the following section.

5.3 Unique Identifiers (UIDs)

PMIx is lacking of the notion of unique identifier, enabling machine-wide connectivity, such descriptor has to be reconstructed from PMI ranks and name-spaces identifiers. As we further develop, we designed our overlay network to achieve such support, providing compact *addresses* to each MPI process. This leads to a behavior mimicking one of Unique Resource Locators (URLs). A group of process is then given a unique name (URL) which is then addressable to target underlying process ranks as sub-resources (leading to an

[3]We have plans to implement an independent bootstrapping using the root-file in the users-home for cases where the PMI is not available.

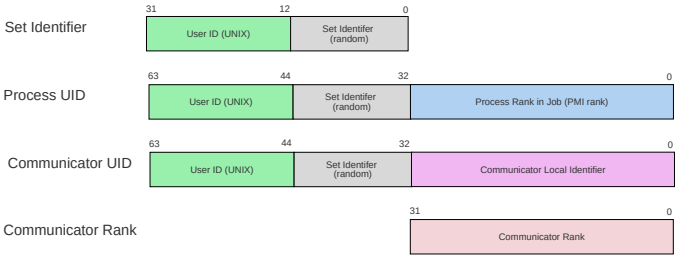

Figure 6: Illustration of UID nesting in MPC.

URI). The direct consequence is MPC's ability to contact arbitrary processes in the `MPI_UNIVERSE`. We even implemented universe point-to-point MPI messages which can take a UID instead of an MPI process rank and communicator. UIDs are built simply by changing all references to the process ID (the actual UNIX process, not the MPI one, MPC being thread-based) inside MPC from 32 bits to 64 bits. As presented in Figure 6, we inserted the set identifier as constructed in the previous section on the 32 MSB. The remaining 32 bits are the regular linear rank provided by the PMI. The result, transitively to the unique nature of the set identifier is the process rank being unique on the whole machine – becoming a UID. Moreover, having a 64 bits process rank is also having the 32 bits set. Consequently, this rank alone is sufficient for end-to-end routing permitting its use as an URI (Unique Resource Identifier).

This UID in MPC acts as an URI enabling the resolution of any UNIX process. However, as MPC is a thread-based MPI, we left the MPI rank unchanged, still 32 bits. First, it is not an opaque type in MPI leaving no freedom to change it. And second, because it would not have been so useful. Indeed, as outlined by MPI sessions, everything is encapsulated as you always need a communicator to communicate. And transitively, if the communicator identifier is unique, the communication can be sand-boxed to a given group, enabling precise MPI process rank resolution. This is how MPI sessions operate as they do not define an endpoint. The process is such as having "internally" a rank given by the PMI which typically maps linear ranks to UNIX processes. And it is only when the communicator is created from the group that the sessions end up with an endpoint – thanks to the internal communicator ID part of the matching logic. The sessions handle has no distributed encapsulation role, and does not define an endpoint, its internal ID distinguishes between local sibling handles to implement error-handling – intermixing sessions handles is forbidden. As a consequence, two distinct MPI Sessions instantiating `mpi://WORLD`, would not be able to do a connect/accept between each other as doing so with intersecting communicators is forbidden (here they are even equal in terms of underlying group). The reason is that the support group of the communicator refers to the actual process rank of the MPI process and that in all cases even if hosting a hundred sessions, a process only has one "internal" rank in use inside its group descriptor. Conversely, if the MPI session would create a new endpoint (with its own UID), communication and grouping including between local endpoints would be feasible.

Getting back to MPC to enable this encapsulation from the endpoint (UNIX process in our case) to the communicator, we had to update communicator ID generation. Indeed, to support connect/accept we have to ensure all communicator IDs were different. To do so, we simply added the set identifier at its beginning, promoting the ID to 64 bits as shown in Figure 6. This way, the intercom resulting from a connect/accept can be built directly from the group representation as serialized from the remote process – a communicator having a support group relying on UIDs (set and rank encoded on 64 bits). Moreover, if the two communicators are merged, the corresponding UIDs can be directly merged leading to a group spanning two jobs without any renumbering or indirection. The MPI process rank is then indexing members of the support group. Moreover, this has the advantage of factorizing the code for on-demand connectivity inside and between jobs. Indeed, the UID and the corresponding routed-RPC can be used for all processes on the machine thanks to a common name-space. The Remote Procedure Calls open the way for one-sided connectivity to remote processes whereas PMI Put/Get is only limited to two-sided connectivity.

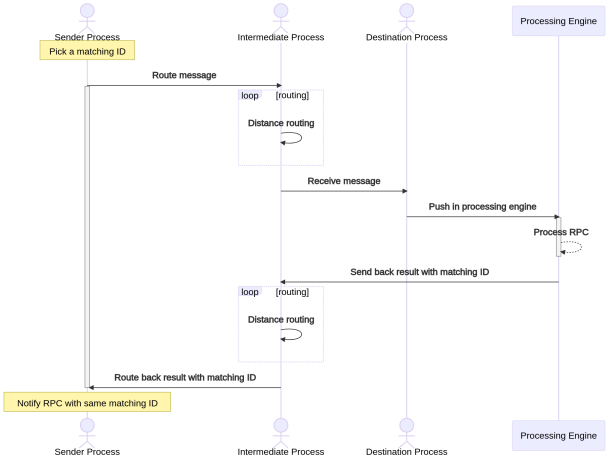

Figure 7: Sequence diagram for routed RPCs on the overlay network.

To route Remote-Procedure Calls (RPCs), we rely on a simple protocol shown in Figure 7. First, each RPC is given a unique local identifier thanks to a linearly increasing 64 bits value. This value is sent alongside the message to enable response matching. Both outgoing and incoming messages share the same routing with a different message type (request and response). On the receiver side, messages to be processed are pushed in a list for asynchronous processing by a helper thread. This enables the decoupling of the routing engine by delaying processing and avoiding deadlocks. Sender-side, the process waits for a response with the expected matching identifier. To support failed and unreachable processes, a timeout is implemented during the wait for the response. This timeout is set to 30 seconds to avoid false triggers. Routed messages have a Time-To-Live (TTL) to avoid infinite loops in the routing network given possible node failure. Overall, the overlay network (1) routes RPC messages and their responses and (2) handles potential failures on the network thanks to timeouts.

6 OVERLAY COMMANDS AND UTILITIES

Thanks to this RPC mechanism, we implement several query commands allowing us to perform remote operations while retrieving potential return values. Indeed, due to their one-sided nature RPCs allow processing to be launched on the remote node unlike Puts and Gets which necessarily require a form of symmetry in the operations – this is particularly helpful when connecting to a remote MPI process. In MPC we implemented the following RPC operations to support both process discovery and dynamic connectivity:

- **Request set info:** for a given set retrieve information on peers (count and identifiers);
- **Ping:** send a message to compute roundtrip time (used for liveliness assessment and benchmark);
- **On demand:** invoke remote callback to perform on-demand connections;
- **Connectivity:** dump the overlay network topology for the given UID;
- **Communicator Information:** request serialization of the group of a given remote communicator;
- **Service registration:** implement service lookup and registration over the overlay network.

In complement, a simple interface allows sets and UIDs manipulation, for example, to retrieve a given set. As far as RPCs are concerned, they are exposed through dedicated functions and rely on a common RPC infrastructure, implementing the wiring protocol. The runtime can also register arbitrary RPCs to implement side-channel control messages, these are used, for example, to implement emulated RDMAs.

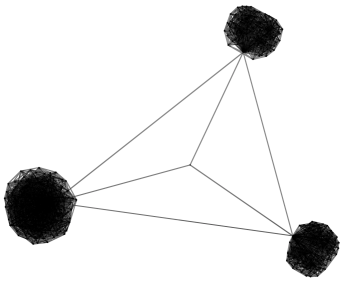

Figure 8: Overlay network topology for three commands involving 512 processes plus `mpc_set` (dot in the middle connected to the root of each job to run connectivity commands).

```
$ mpc_sets sets
-------------------------------------------------------
| SET ID : 4129295386      | SET ID : 3289965661      |
| SIZE   : 512             | SIZE   : 512             |
| CMD    :  ./IMB-MPI1     | CMD    :  ./IMB-MPI1     |
-------------------------------------------------------
| SET ID : 1837669245      | SET ID : 394267674       |
| SIZE   : 512             | SIZE   : 1               |
| CMD    :  ./IMB-MPI1     | CMD    :  mpc_sets sets  |
-------------------------------------------------------
```

Listing 1: Sample output from the "mpc_sets sets" command.

We implemented a command-line interface called `mpc_set` to offer process-set listing capabilities. This command enables the dumping of the overlay network topology while listing all UIDs involved, set sizes, and associated commands. It is implemented by joining the overlay network as a regular set; providing the ability to emit commands to the various other sets present on the machine. The topology dump of Figure 8 is done by sending a connectivity command to all members of all sets from the `mpc_set` instance. In this output, the command line program can be seen in the middle as connected to the three jobs wired with the kademlia topology. As shown in Listing 1, a textual output is also available to list currently running sets and their respective commands.

7 PERFORMANCE EVALUATION

In this section, we evaluate the performance of our overlay network at scale, first assessing its scalability. Second, we perform some routed RPCs benchmarking, and eventually, we compare this throughput to PMIx on the same system. Following tests were done on a prototype cluster. It features bi-socket AMD Rome CPUs (AMD EPYC 7H12 64-Core Processor) for a total of 128 cores per node (256 threads), and the interconnect consists of Bull Exascale Interconnect (BXI) V1.3 adapters. The cluster runs on slurm 20.11.8 and relies on PMIx 3.1.5 (gitedebb24) without UCX support[19] as we run on Portals 4 hardware.

7.1 Topology Scalability

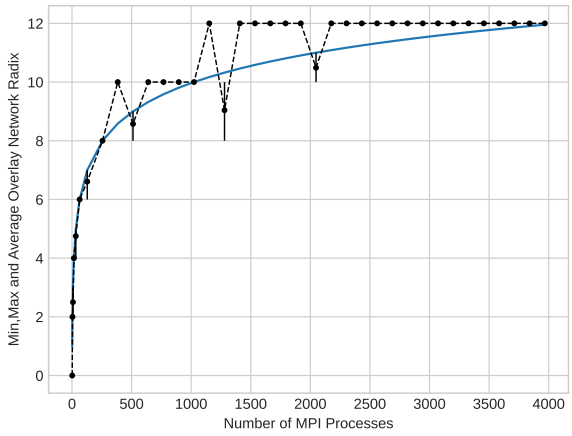

Figure 9: Minimum, Maximum and average (error-bars) overlay network connectivity degree for various scales.

In Figure 9, we present the evolution of the overlay network vertex degree in function of the number of involved MPI processes. As expected, this degree follows a base two logarithm. Overall, this shows that the overlay network practically limits connectivity requirements. It has an expected maximum of 32 connections per process for the maximum number of ranks supported by MPI due to the integer limitation of the rank encoding only up to two billions (2,147,483,647).

7.2 Wiring-up Performance

It is not trivial to fairly compare PMIx and our RPC approach for wiring-up. Indeed, PMI is generally based on a collective to perform the dissemination. Therefore, `Puts` are emitted, committed, and eventually, a disseminating `Fence` propagates them to the various nodes. After this step, subsequent `Get` will be able to retrieve the given keys and associated values. Whereas, for routed RPCs, no collective is involved. For most networks, the process willing to connect sends a message with associated connection information as a parameter to the remote. Once the message arrives, the remote back-connects to the initiator. This process ensures the remote is connected when the RPC returns at the initiator side. Thus, connectivity information are sent when initiating a connection without performing a collective.

To compare data exchanges, we devised a simple benchmark trying to do equivalent data exchanges using the two interfaces. This benchmark measures the time between two PMI Barriers. In the PMIx (V3) benchmark, each process rank does the following:

(1) PMI Fence (non-disseminating)
(2) **Timer Start**
(3) Put
(4) Commit
(5) PMI Fence (disseminating or not)
(6) Get a subset of Values
(7) PMI Fence (non-disseminating)
(8) **Timer End**

For the routed RPC case on the overlay network we do the following, trying to achieve approaching data-exchanges with the given interface:

(1) PMI Fence (non-disseminating)
(2) **Timer Start**
(3) Callback RPC Connection info to a subset of Processes
(4) PMI Fence (non-disseminating)
(5) **Timer End**

The PMI fences used in both cases allow for comparing the two approaches which do not share the same synchronous semantic – they are expected to produce the same overhead in both approaches. On the data-exchange side, each node retrieves (or sends) its connection info to a given number of processes mimicking wiring up. We then propose the overall per-connection cost as simple figure of merit. For example, if n processes spent *a total of* T seconds in the timed section to individually wire-up k processes, we can deduce that average job-wide cost of a given connection C_c is such as $C_c = \frac{T}{n*k}$.

Figure 10 presents a comparison of these two test-cases up to 4096 cores by doing a ratio of the average job-wide cost. Lower than 100% means MPC is faster, and conversely higher is slower. The measure is done with different connectivity ratios from 1% (with at least two connections) up to 100% meaning fully-connected. The routed RPCs tend to be more efficient (up to 20 times faster) for lower connectivity rates up to 20%. Opposingly, for higher connectivity, routed RPCs are inefficient. The reason for this lies in the implementation differences. PMIX does the data-dissemination inside a collective operation (analogous to Allgather), whereas the overlay network emits one single RPC per on-demand, leading to a quadratic complexity. It is then clear that the overlay network does not take advantage of aggregation opportunities to reduce the number of operations. However, thanks to the logarithmic diameter of the routing network, the number of connections remains constant, reducing the networking overhead. For these reasons, routed RPCs are more efficient when the collective cost is not amortized by the number of participants (here endpoints to connect to). Moreover, in our measurements, we saw an important performance improvement in PMIx over 2500 processes, reasons for this abrupt variation is still to be explained.

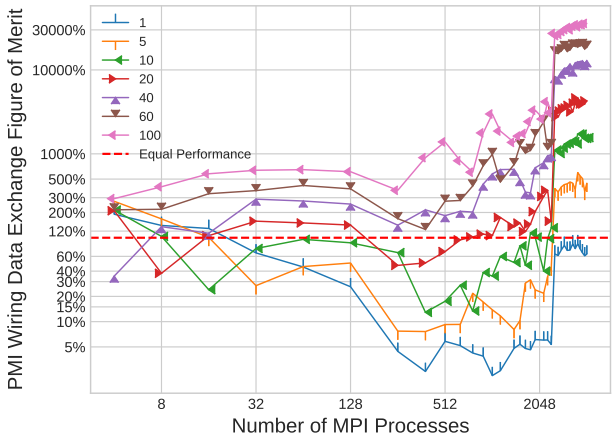

Figure 10: Ratio of MPC overlay's figure of merit C_c over PMIx's for our Put/Get testcase from 0 to 4096 processes with a disseminating fence. Both axes are in logscale due to the large dynamic (base 10 vertically and 2 horizontally).

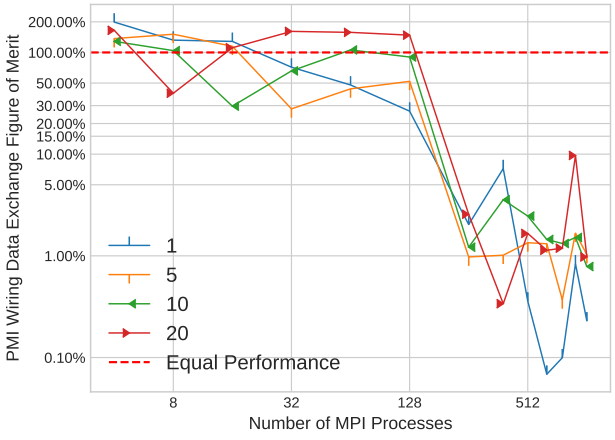

Figure 11: Ratio of MPC overlay's figure of merit C_c over PMIx's for our Put/Get testcase from 0 to 1024 processes without a disseminating fence. Both axes are in logscale due to the large dynamic (base 10 vertically and 2 horizontally).

As PMIx supports retrieving values without a disseminating fence[4], we repeated our measurements in this updated configuration in Figure 11. In this updated test, the runtime then has to fetch each value individually inside the respective servers. One can see that PMIx takes advantage of an SHM memory segment as up to 128 processes (node size) performance is steady. However, when going out of the node, the performance quickly decreases for all connectivity ratios. This, up to a point where the runtime encounters some issues (certainly due to request contention) – preventing us to run up to 4096 processes. Interestingly, this new configuration which is not how PMIx is to be used for performance [5], outlines how PMIx could take advantage of our routed RPC approach to improve its point-to-point resolution support. Indeed, in the context of horizontal connectivity the point-to-point pattern may become prevalent as disseminating necessarily means there is a underlying group to be used (practically processes from the same allocation). In the case of completely unrelated jobs, this synchronous model does not hold anymore – justifying our interest for point-to-point queries.

Eventually, to correctly contrast these results, it is important to mention that we solely measured the data-exchange phase and that the overall picture is larger. Indeed, setting up the overlay network has a cost by itself and unlike PMIx which can potentially factorize it, being external to the job (Slurm or Orted DVM), our approach repeats it at each launch. It means that optimally the overlay should be either initially sparser or externalized in system daemons to provide the most efficient launch time which is an important factor. As of now, the overlay is created synchronously at the start, we are working on connecting lazily to limit its creation cost and dynamically adapting the network diameter over time.

7.3 RPC Latency

When using the overlay network for control messages, RPCs are routed which means that each process has privileged neighbors as per the underlying routing topology. As we use a kademlia-inspired topology, a given process is connected to the following powers of two in the network. As per Figure 9, with 3200 MPI processes the overlay degree is twelve. It means, by looking at Figure 12 twelve nodes are one hop away. It can be seen that despite routing being involved the logarithmic diameter of the kademlia network provides bounded latency and while running over 3200 processes (or 25 Nodes), the worst average is 1.29 milliseconds, and the global average is 0.65 millisecond. These values are of course higher than what could be expected using the high-performance network, however, they remain acceptably low. Indeed, the choice for TCP here is a clear trade-off for portability over performance, this support network is required on all target systems as side-channel independent from the fabric type used for MPI messaging.

8 CONCLUSION

As outlined in Section 3, we motivated that MPI should consider the composition of programs as a key element to handle the increasing pressure from the underlying hardware. This supposes two things, (1) efficient and reliable job/process discovery as we covered in this paper, and (2) stateless client-server patterns (RPCs) which are not yet available in MPI[6]. Focusing on service discovery aspects, we introduced an overlay network in charge of providing wiring-up capabilities to MPI programs, not only inside but also between jobs, this thanks to a routing mechanism. This enables job discovery at the machine level for a given user. Its main purpose is to guarantee the availability of the publish/lookup mechanism independent of the underlying batch manager or system support. We have shown how routed RPCs over the overlay network were able to improve over the PMIx wiring substrate under some conditions (limited connectivity).

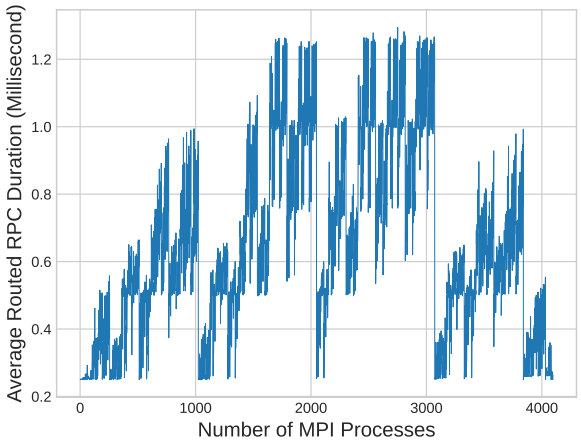

Figure 12: RPC latency (roundtrip) from process 0 to all processes when running over 4096 processes (32 Nodes). Values are averaged 500 times to outline underlying topology.

Moreover, we discussed how Unique IDentifiers for MPI processes would make connect/accept more robust thanks to a direct encoding of communicator support groups internal to the MPI runtime. Having an endpoint semantic inside of sessions would enable such connectivity without the risk of an internal endpoint clash. More generally, extending the meaning of the rank, or providing an abstraction matching the semantics of an URI would improve MPI's service support by providing out-of-job connectivity to MPI processes. As of now, MPI sessions opened two ways to creating communicators from labels (which are practically URL). However, the standard forbids communication between such sessions in its 4.0 iteration. Whether the addressable element should be the Group or the MPI Process itself is an open question. Yet we see it as crucial to enable what we described as *composition* patterns to overcome the complexity barrier faced by HPC programs through divide and conquer. In this work, we have shown that with a simple abstraction and limited dependencies (only TCP) an MPI runtime could embed this support to enable its portability. Second, we demonstrated that PMIx could take advantage of routed RPCs to enhance its performance when not relying on disseminating fences, for example, when addressing remote process groups.

[4]We still call a fence to ensure values are Put before the Get.
[5]A disseminating fence is historically the standard way, it can also be non-blocking
[6]MPI Connect/Accept supposes a remote state for disconnection as all remote communicator members have to connect and disconnect collectively.

Overall, we consider that enabling MPI to operate horizontally in a more dynamic environment, including persistent service is one of the main challenge the standard has to face in the near future. Indeed, it now has competitors which do feature bare-metal performance, seamlessly leveraging high-speed interconnects. Given these infrastructure are much more suitable for composition, we think that they will first be collocated with MPI and then they may progressively replace it as the associated interface and runtimes reach enough momentum. Sessions did set the base for addressing but support for operations, discovery, persistence *between* theses sessions is becoming increasingly needed.

9 FUTURE WORK

In this paper we focused ourselves on process discovery and wiring. We think that the overlay network in the context of MPI has much more to offer. This section, quickly list various ideas that we consider as future work.

Fault tolerance: TCP's connected nature facilitates the implementation of a reliable "ping" commands able to put up with failed processes. The overlay-network can then be used as a portable support for implementing fault-tolerance mechanisms;

Routed collectives: some programs do expensive collectives punctually at the start and then rely on a relatively sparse communication matrix, but the given endpoints usually remain connected. If such punctual collective could be routed it would avoid connecting many processes for nothing;

Out of band: as the overlay network is based on TCP it can easily accommodate for transient clients. This opens the way for a wide range of MPI support tools, from monitoring MPI using the tools interface[20, 23] to sending commands to the application (steering, specific queries, ...);

Hybrid topology: we also want to allow routing over the high-speed network endpoints, practically reviving the control messages previously present in MPC (section 5.1). With the advantage of relying on more efficient hardware, potentially increasing RPC throughput while reducing opportunistically the network diameter.

REFERENCES

[1] Dong H Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Helgi I Ingólfsson, Joseph Koning, Tapasya Patki, Thomas RW Scogland, et al. 2020. Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems* 110 (2020), 202–213.

[2] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. 2020. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience* 32, 3 (2020), e4851.

[3] Jean-Baptiste Besnard, Julien Adam, Sameer Shende, Marc Pérache, Patrick Carribault, Julien Jaeger, and Allen D Maloney. 2016. Introducing task-containers as an alternative to runtime-stacking. In *Proceedings of the 23rd European MPI Users' Group Meeting*. 51–63.

[4] Jean-Baptiste Besnard, Allen D Malony, Sameer Shende, Marc Pérache, Patrick Carribault, and Julien Jaeger. 2017. Towards a Better Expressiveness of the Speedup Metric in MPI Context. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 251–260.

[5] Ralph H Castain, Joshua Hursey, Aurelien Bouteiller, and David Solt. 2018. Pmix: process management for exascale environments. *Parallel Comput.* 79 (2018), 9–29.

[6] Bronis R de Supinski, Thomas RW Scogland, Alejandro Duran, Michael Klemm, Sergi Mateo Bellido, Stephen L Olivier, Christian Terboven, and Timothy G Mattson. 2018. The ongoing evolution of openmp. *Proc. IEEE* 106, 11 (2018), 2004–2019.

[7] Matthieu Dorier, Matthieu Dreher, Tom Peterka, Justin M Wozniak, Gabriel Antoniu, and Bruno Raffin. 2015. Lessons learned from building in situ coupling frameworks. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. 19–24.

[8] Michael A Heroux, Rajeev Thakur, Lois McInnes, Jeffrey S Vetter, Xiaoye Sherry Li, James Aherns, Todd Munson, and Kathryn Mohror. 2020. *ECP software technology capability assessment report.* Technical Report. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).

[9] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. 2013. MPI+ MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing* 95, 12 (2013), 1121–1136.

[10] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Implementation and performance analysis of non-blocking collective operations for MPI. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 1–10.

[11] Daniel Holmes, Kathryn Mohror, Ryan E Grant, Anthony Skjellum, Martin Schulz, Wesley Bland, and Jeffrey M Squyres. 2016. Mpi sessions: Leveraging runtime infrastructure to increase scalability of applications at exascale. In *Proceedings of the 23rd European MPI Users' Group Meeting*. 121–129.

[12] Atsushi Hori, Emmanuel Jeannot, George Bosilca, Takahiro Ogura, Balazs Gerofi, Jie Yin, and Yutaka Ishikawa. 2021. An international survey on MPI users. *Parallel Comput.* 108 (2021), 102853.

[13] Joshua Hursey. 2020. Design considerations for building and running containerized mpi applications. In *2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE, 35–44.

[14] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter {RPCs} can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16.

[15] A. Kreuzer, E. Suarez, N. Eicker, and Th. Lippert (Eds.). 2021. *Porting applications to a Modular Supercomputer - Experiences from the DEEP-EST project.* Schriften des Forschungszentrums Jülich IAS Series, Vol. 48. Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag, Jülich. 254 pages. https://juser.fz-juelich.de/record/905738

[16] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. 2005. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials* 7, 2 (2005), 72–93.

[17] Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*. Springer, 53–65.

[18] Marc Pérache, Patrick Carribault, and Hervé Jourdren. 2009. MPC-MPI: An MPI implementation reducing the overall memory consumption. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 94–103.

[19] Artem Y Polyakov, Boris I Karasev, Joshua Hursey, Joshua Ladd, Mikhail Brinskii, and Elena Shipunova. 2019. A performance analysis and optimization of PMIx-based HPC software stacks. In *Proceedings of the 26th European MPI Users' Group Meeting*. 1–10.

[20] Srinivasan Ramesh, Aurèle Mahéo, Sameer Shende, Allen D Malony, Hari Subramoni, Amit Ruhela, and Dhabaleswar K DK Panda. 2018. MPI performance engineering with the MPI tool interface: the integration of MVAPICH and TAU. *Parallel Comput.* 77 (2018), 19–37.

[21] Robert B Ross, George Amvrosiadis, Philip Carns, Charles D Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K Gutierrez, Robert Latham, et al. 2020. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology* 35, 1 (2020), 121–144.

[22] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, et al. 2020. Co-design for a64fx manycore processor and" fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[23] Martin Schulz, Marc-André Hermanns, Michael Knobloch, Kathryn Mohror, Nathan T Hjelm, Bengisu Elis, Karlo Kraljic, and Dai Yang. 2021. The MPI Tool Interfaces: Past, Present, and Future—Capabilities and Prospects. In *Tools for High Performance Computing 2018/2019*. Springer, 55–83.

[24] Jerome Soumagne, Dries Kimpe, Judicael A Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert B Ross. 2013. Mercury: Enabling remote procedure call for high-performance computing.. In *CLUSTER*. 1–8.

[25] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. 2018. Gekkofs-a temporary distributed file system for hpc applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 319–324.

[26] VÌctor Sande Veiga, Manuel Simon, Abdulrahman Azab, Carlos Fernandez, Giuseppa Muscianisi, Giuseppe Fiameni, and Simone Marocchi. 2019. Evaluation and benchmarking of singularity mpi containers on eu research e-infrastructure. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE, 1–10.