Prospects and challenges of virtual machine migration in HPC*

Simon Pickartz1 | Carsten Clauss2 | Jens Breitbart3 | Stefan Lankes1 | Antonello Monti1

1Institute for Automation of Complex Power Systems, RWTH Aachen University, Aachen, Germany
2ParTec Cluster Competence Center GmbH, Munich, Germany
3Bosch Chassis Systems Control, Stuttgart, Germany

Correspondence
Simon Pickartz, Institute for Automation of Complex Power Systems, RWTH Aachen University, Mathieustraße 10, 52074 Aachen, Germany.
Email: spickartz@eonerc.rwth-aachen.de

Summary
The continuous growth of supercomputers is accompanied by increased complexity of the intra-node level and the interconnection topology. Consequently, the whole software stack ranging from the system software to the applications has to evolve, eg, by means of fault tolerance and support for the rising intra-node parallelism. Migration techniques are one means to address these challenges. On the one hand, they facilitate the maintenance process by enabling the evacuation of individual nodes during runtime, ie, the implementation of fault avoidance. On the other hand, they enable dynamic load balancing for an improvement of the system's efficiency. However, these prospects come along with certain challenges. On the process level, migration mechanisms have to resolve so-called residual dependencies to the source node, eg, the communication hardware.

On the job level, migrations affect the communication topology, which should be addressed by the communication stack, ie, the optimal communication path between a pair of processes might change after a migration. In this article, we explore migration mechanisms for HPC and discuss their prospects as well as the challenges. Furthermore, we present solutions enabling their efficient usage in this domain. Finally, we evaluate our prototype co-scheduler leveraging migration for workload optimization.

KEYWORDS
application migration, co-scheduling, HPC, IVShmem, MPI, topology awareness

1 INTRODUCTION

Current trends in High Performance Computing (HPC) result in supercomputers that increase in size for every generation. This increase is not only limited to the node counts but also affects the intra-node level leveraging multi-core CPUs. Already today, the fastest supercomputers exceed the ten million core mark. This development is not likely to change for future exascale systems that are expected to be built by hundreds of thousands of nodes and that feature core counts that are a magnitude larger than what is available in today's petascale machines.1 On the one hand, this growth demands for more and more complex communication topologies on the inter-node level, which have to be addressed by the software stack running on top. On the other hand, application developers are in charge of tuning their codes for the support of the rising intra-node level parallelism for the best possible exploitation of the available resources. Besides the increased system complexity, system failures will be one of the major issues of future exascale systems. Previous studies showed that the mean time to outage per processor is nearly constant over the past years,2 ie, the Mean Time Between Failure (MTBF) of the whole system will drastically decrease.3

Migrations of jobs or individual processes across nodes are one means to cope with the challenges discussed above. They enable a more efficient use of the available hardware in conjunction with co-scheduling techniques, ie, the execution of two or more applications with different resource demands on an overlapping set of nodes.4 Abandoning the concept of static schedules, jobs or parts thereof can be migrated during runtime across the cluster for the avoidance of load imbalances.5,6 This is especially interesting for legacy applications, which are not likely to be tuned to every new generation of supercomputers in many cases. Furthermore, migration mechanisms allow for an eased system maintenance by the evacuation

The copyright line for this article was changed on 6-July 2018 after original online publication.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2018 The Authors. Concurrency and Computation: Practice and Experience Published by John Wiley & Sons Ltd.
of individual nodes. This abandons the need for a scheduling of maintenance slots and rather facilitates flexible ad hoc services. If this technique is coupled with a failure prediction mechanism, resiliency can be improved by the evacuation of nodes that are expected to fail in the near future.

An example of how process migration facilitates the efficient implementation of maintenance events is presented in Figure 1. It examines the occurrence of a service event during the execution of mpiBLAST (cf Section 3.2), which was started with 64 processes on two 32-core systems based on Intel’s IvyBridge architecture. In the absence of Checkpoint/Restart (C/R) or migration support, a job abort is necessary with a full restart after the maintenance (cf Figure 1A). This results in an excessive waste of resources, as the time span up to the completion of the maintenance of Node 1 does not add to the efficient usage of the cluster. Although the penalty caused by such a situation is highly dependent on the exact point in time and the runtime of the particular job, the example gives an impression on possible consequences.

This penalty can be limited by leveraging C/R mechanisms, as these are used for the prevention of full application restarts due to maintenance or failures. However, these mechanisms usually stop all processes of the job even if there are still nodes available for computation (cf Figure 1B). In this example, Node 0 idles until the maintenance of Node 1 is completed, yet the application’s execution time is reduced by around 16%. However, this is a best-case assumption. Firstly, we assume an ideal C/R implementation that instantaneously suspends all running processes and restarts them at no cost. Usually, checkpoints are written to a global file system that may become a bottleneck at great process counts. Secondly, we expect support for checkpoints on-demand, i.e., often, checkpoints are taken periodically resulting in additional overhead due to the time gap between the last checkpoint interval and the actual maintenance.

The last example (cf Figure 1C) depicts how the overall execution time can be further reduced by the exploitation of migration support, e.g., the mechanism presented in Section 4. Instead of suspending all processes of the running job, those from Node 1 are temporarily migrated to Node 0. Although this results in an overbooking of this node, a reduction of the application’s execution time by around 13% compared to the ideal C/R mechanism is possible in this particular scenario. Even though the service of Node 1 lasted for 20 min, the runtime increases by only 8.5 min compared to the job’s execution without maintenance (cf Figure 1D).

1.1 Challenges

Such dynamic systems impose novel challenges that have an impact on the whole software stack. It starts at the process level and the question as to which data structures and Operating System (OS)-related entities have to be moved along for guaranteeing a continuing execution. Processes usually depend on resources on the source node, i.e., so-called residual dependencies such as local file descriptors, communication channels, and hardware-related resources. As the dynamic behavior of the system should be application transparent to a great extent, the middleware is responsible for the resolution of these dependencies prior to the migration. Alternatively, they have to be moved along with the migrated process, e.g., Virtual Machine (VM) migration follows this approach by an encapsulation of all dependencies within an isolated environment.

However, this only applies to resources that can be virtualized efficiently, e.g., resources that are independent of the underlying hardware. Usually, OS-bypass networks are employed for the interconnection of the individual cluster nodes in HPC (cf Section 3.1). These often entail hardware-dependent identifiers that are used for the referencing of the devices within the software layer. In such cases, an encapsulation of the process is not sufficient for its gracious migration, but these dependencies have to be resolved beforehand.
Furthermore, migrations result in dynamically changing communication topologies. Processes that were placed on distinct nodes might share the same node after a migration, or vice versa. Therefore, a different interconnect should be favored for the most efficient communication, i.e., in this case, the processes should leverage the shared physical memory which was not an option before. The communication layer should take such a behavior into account and adapt accordingly.

1.2 Contribution

This work builds upon our research in the field of application migration in HPC and co-scheduling techniques enabled by virtualization using VMs. In doing so, we present the following substantial contributions:

1. a thorough evaluation of our virtualization-aware Message Passing Interface (MPI) library.
2. a locality-aware extension with support for dynamically changing topologies, and
3. a prototype co-scheduler leveraging VM migration.

At this point, it should be noted that the presented techniques and mechanisms are independent from the underlying virtualization solution to a great extent, e.g., OS containers could be leveraged for the isolation of the HPC processes as well. In the presented work, we leverage system-level virtualization based on VMs as this provides us with a stable and efficient migration mechanism. In previous works, we could show that the overhead of VMs is negligibly small and outweighed by the advantages that come with this high degree of isolation.

The article is structured as follows: we start with a discussion on related work and background knowledge in Sections 2 and 3, respectively. The Shutdown/Reconnect (S/R) protocol enabling the seamless migration of processes using OS-bypass networks is presented in Section 4, followed by an introduction to our locality-aware extension of the communication stack. Before concluding the article in Section 7, we present our co-scheduler in Section 6, which is named "Poor Man’s Co-Scheduler (poncos)."

2 RELATED WORK

Migration as a means for resiliency has been studied in the past. All these works focus on process-level migration. As mentioned before, this involves the problem of residual dependencies that have to be resolved prior to the migration. Apart from the communication-related resources, these dependencies comprise other resources such as open file descriptors or child processes. Virtualization techniques based on VMs or OS containers mitigate this situation by an encapsulation of all dependencies within an isolated environment.

In the HPC context, usually, C/R mechanisms are regarded for an improvement of fault tolerance and resiliency. Such mechanisms can be seen as a generalization of process migration using global consistency models entailing expensive synchronization. For example, Distributed Multi-Threaded Checkpointing (DMTCP) is a framework for user-level checkpointing with application-transparent support for C/R of applications using InfiniBand (IB). This solution realizes a cluster-wide synchronization of all processes belonging to the same job, which is required for a globally consistent checkpoint. However, the migration of single processes or process groups makes do with a more relaxed consistency model considering the affected connections only.

Co-scheduling is hardly applied to HPC systems as of today. On the one hand, this is not attractive for the community with current pricing models based on core hours. On the other hand, this approach demands for an understanding of the applications’ resource utilization and their mutual influence. Only then can it be motivated from the compute centers’ point of view for an optimization of the system workload. Although different proposals exist that use prediction models for efficient co-scheduling, they are usually based on empirical slow-down analyses. In contrast, we propose a scheduling scheme based on an online analysis of the applications’ resource consumption. Therefore, our solution neither requires an offline analysis nor relies on a knowledge base for the scheduling decisions.

Recently, Zhang et al have proposed Slurm-V, a framework for virtualized HPC clouds. They leverage Inter-VM Shared-Memory (IVShmem) and Single Root I/O Virtualization (SR-IOV) for the support of virtual clusters based on the SLURM resource manager. The framework comes with support for different execution models ranging from exclusive allocations to the sharing of nodes for concurrent jobs. Although this allows for co-scheduling of applications, the framework does not take any scheduling decisions based on their runtime behavior. The VMs, once scheduled on the cluster nodes, remain at their location until the respective jobs terminate. Currently, migrations are not supported, which would enable load balancing or an improvement of resiliency by means of proactive fault tolerance. With our co-scheduler poncos (cf Section 6), we follow a different approach. Instead of creating customized VMs for each job, we only leverage their strong isolation as a means for a graceful application migration. Therefore, a re-use of the existing VMs is possible, avoiding VM starts and shutdowns between the application runs.

In general, the topics discussed in this paper tackle issues that are relevant in a cloud computing environment as well, e.g., the noisy-neighbor problem describes performance degradations caused by resource sharing with other virtualized environments hosted on the same physical nodes. This is the result of resource overbooking, which is a common practice in today’s data centers, but not in HPC. In cloud computing environments, this problem is mitigated by an optimization of the oversubscription level. In contrast, co-scheduling in HPC does not oversubscribe the available
resources but rather aims at the sharing of compute nodes by applications with distinct resource demands. However, there are still inherently shared resources such as the last-level cache if there is no according hardware support for its exclusive assignment to CPU cores, eg, Intel’s Cache Allocation Technology.®

3 | BACKGROUND

This section discusses relevant background for the scope of this work. It starts with an overview of the hardware layer and a discussion of pitfalls when migrating VMs with attached passthrough devices. Furthermore, the test system is introduced, which has been used for the evaluation. The second part gives an overview of the software stack and presents the test applications that served for the evaluation.

3.1 | Hardware

Commonly, high-performance networks employed in HPC clusters bypass the OS layer for a reduction of latencies and an improvement of the throughput. IB is a pervasive technology that manages the connection state information directly within the hardware. Although it provides reliable communication channels, packet losses induced by migrations cannot be recovered. This is due to the connection state information that cannot be extracted by the software and, therefore, cannot be migrated between different Host Channel Adapters (HCAs).® The HCA would be incapable of determining which packets were delivered successfully and which had to be re-transmitted.

3.1.1 | Virtualization

Virtualization abstracts the computing environment from the actual hardware platform by hiding particular hardware characteristics and solving certain dependencies in the software. There are several approaches for doing so, and the virtualization may be located at different levels in the hardware/software stack, eg, on the system level, the OS level, or the process level. As already mentioned, in this article, we leverage VMs for process migration, but container-based virtualization on the OS level could also be conceivable for doing so. However, irrespectively of the virtualization’s location, particularly in a message-passing environment, the incorporation of communication facilities plays a crucial role. Hence, for instance, Input/Output (I/O) Virtualization enables the efficient use of I/O devices within VMs. In contrast to the virtualization of these devices, passthrough techniques such as Intel’s Virtualization Technology for Directed I/O (VT-d) extensions allow for a direct control of the device by the guest. This requires a pinning of the whole guest memory in the physical host memory, ie, the hypervisor has no information on where and when the guest’s memory is accessed by the hardware and, therefore, demand paging cannot be applied. Since this technique, by itself, only allows for the passthrough of one device to a single VM at a time, the SR-IOV specification was introduced for enabling the native sharing of one device among multiple VMs. However, both techniques impose location-dependent resources to the guests, which exacerbates their migration. Therefore, all passthrough devices have to be detached beforehand.

3.1.2 | The test system

The test system for the evaluation of the presented works is a four-node Non-Uniform Memory Access (NUMA) cluster. Each node possesses two sockets with eight physical cores, exposing 32 virtual cores in total. Two of the nodes are equipped with Intel SandyBridge CPUs (E5-2650) clocked at 2 GHz, whereas the others are supplied with newer generation Intel IvyBridge CPUs (E5-2650 v2) being clocked at 2.6 GHz. The interconnection network is based on Mellanox ConnectX-3 VPI technology. Therefore, the nodes are equipped with two-port HCAs implementing the PCIe 3.0 standard. These provide a theoretical peak throughput of 56 Gbit/s. Furthermore, the HCAs comply with the SR-IOV supporting up to 16 guests.

The nodes run the same setup employing a 3.10.0 Linux kernel of a standard CentOS 7.3 installation and use the OpenFabrics Enterprise Distribution (OFED) Stack in version 3.4-2.0.0.0 provided by Mellanox. All benchmarks and applications were compiled by using the GCC in version 6.2.0. For a reduction of the residual dependencies arising from migrations on the process level, we leveraged VM migration based on Kernel-based Virtual Machine (KVM). This provides us with a stable migration support while adding little overhead to native execution.

3.2 | Software

This section starts with a discussion of the pscom library and Nahanni—the basis for our modifications to the communication stack. Then, the benchmarks and applications used for the evaluation are presented.
The pscom library

The pscom library is the lower-level communication substrate of ParaStation MPI, a fully MPI-3-compliant MPICH-based MPI implementation. It is especially designed for the employment in HPC systems and integrates into MPICH by implementing the ADI interface. A variety of interconnects prevailing in the HPC domain is supported by means of plugins that facilitate the integration of new interfaces. A plugin has to provide bidirectional point-to-point communication channels for the specific interconnect. However, it can rely on higher-level facilities that are implemented on a hardware-independent layer within the library. The pscom comes with an on-demand mechanism implementing a lazy connect approach irrespectively of the underlying hardware. This improves the scalability of MPI sessions for thousands of ranks. Therefore, the actual connection establishment between two peers is delayed until the first write or read attempt occurs on that connection. Only then does the pscom checks and chooses the available plugins by a predefined priority/fallback scheme that favors those paths that promise faster communication.

Nahanni

Nahanni is a mechanism enabling IVShmem communication between co-located VMs. It consists of three parts: a shared-memory region on the host, a new virtual device called ivshmem as part of QEMU’s virtual hardware support, and a guest driver. The shared-memory region is created by using the POSIX API and, therefore, copes without modifications to the host OS, its modules, or KVM. Furthermore, the modifications to QEMU are included in the upstream sources since version 0.13 in 2010. The guest driver is kept small by leveraging the User-Space I/O (UIO) framework for the configuration of the ivshmem device. This appears as a PCI device to the guest including a configuration space with Base Address Registers (BARs) and device memory. In doing so, it supports both synchronization over shared memory, eg, via spinlocks, and via Message Signaled Interrupts (MSIs).

Applications and benchmarks

These comprise a self-written MPI benchmark as well as applications and application benchmarks common in the HPC domain. The self-written benchmark determines point-to-point latency and bandwidth by the exchange of messages between two processes in a PingPong pattern. For the evaluation of the locality-aware MPI layer (cf Section 5.2), we used the Bcast benchmark from the Intel MPI Benchmarks (IMB). Furthermore, we used the following application benchmarks and applications for the analysis of the presented concepts in real-world scenarios.

The NAS Parallel Benchmarks (NPB) is a suite of different computing kernels that are commonly used by large-scale fluid dynamics applications. It offers varying problem classes suiting different cluster sizes. Its MPI implementation contains eight different benchmarks comprising five computing kernels and three so-called pseudo applications. Especially interesting for communication-related evaluation are the two kernels Fourier Transform (FT) and Conjugate Gradients (CG).

mpiBLAST is an MPI-only parallel version of the original Basic Local Alignment Search Tool (BLAST) algorithm from computational biology. We use a slightly modified version of mpiBLAST 1.6.0 in which we removed all sleep() function calls that were supposed to prevent busy waiting. On our test system, this resulted in a performance increase of about a factor of 2. Due to its embarrassingly parallel nature using a nested master-worker structure, mpiBLAST allows for perfect scaling across tens of thousands of compute cores.

LAMA is an open-source C++ library for numerical linear algebra. It ships with a standard implementation of a CG solver using a hybrid OpenMP/MPI programming model. As the involved data structures do not fit into CPU caches, the performance is fundamentally limited by the main memory bandwidth and the inter-core/inter-node bandwidth for reduction operations. Thus, the CG solver obtains the best performance with using just a few cores.

A PROTOCOL ENABLING THE MIGRATION OF MPI PROCESSES

The migration of processes that are part of a distributed application is a challenging task. One of the reasons is the issue of so-called residual dependencies, i.e., dependencies to the source node which are part of the process state. Even if these dependencies are encapsulated, eg, by an isolation of the process within a container or VM, one has to cope with location-dependent resources when it comes to the migration of HPC processes. Commonly, HPC systems employ OS-bypass networks such as IB for the inter-node communication among the processes. As the name suggests, these networks allow for a direct interaction with the underlying hardware without any intervention by the OS.

Therefore, migrations require some mechanism eliminating these dependencies. The following section introduces our approach enabling the migration of MPI processes in OS-bypass networks. Details on the implementation and a thorough evaluation can be found in the works of Pickartz et al and in the authors’ previous work, respectively. The source code has been made available via GitHub.
4.1 Design

In contrast to traditional data centers or cloud computing environments, HPC usually focuses on the exploitation of the system’s peak performance. Therefore, we impose a set of requirements that have to be fulfilled by a migration mechanism to be viable for this domain.

**Avoidance of any runtime overhead.** The system should not be negatively affected in the absence of migrations.

**Minimization of the migration costs.** The proposed mechanism adds to the total migration costs of the underlying migration mechanism, eg, VM migration. The additional overhead should be kept as small as possible.

**Application transparency.** Users will only accept the software if the changes to legacy applications are minimal for the support of the novel technology. Therefore, we require a basic mode that gets along without application support.

**Transport agnosticism.** For portability concerns, the proposed mechanism should be independent from the underlying hardware as far as possible. However, certain assumptions can be made, which are met by common HPC systems, eg, reliable communication channels between the processes.

One approach for the realization of an application-transparent migration mechanism is a virtualization of the location-dependent resources. Thereby, all resources related to the source node, eg, hardware identifiers, are mapped to their virtual counterpart, which is then provided to the application layer. However, such an approach contradicts our requirement of runtime overhead avoidance on the one hand and transport agnosticism on the other hand. The virtualization layer often introduces an additional overhead depending on the semantic gap to the underlying hardware. In any case, this should be avoided in HPC to the extent possible. Works done by Cao et al and Ma et al have revealed a certain impact of such approaches. Furthermore, the solution based on the virtualization of location-dependent resources has to be implemented for each transport individually, eg, for IB, a wrapper around the verbs interface would be required. If one wanted to leverage the Performance Scaled Messaging (PSM) API for the Omni-Path interconnect, a re-implementation of basically the same logic would be necessary.

For these reasons, we chose to follow a different path by actively releasing communication-related resources prior to the migration instead of their virtualization. Therefore, we introduce six states into the communication stack that have to be passed cyclically by the process: (1) Migration Inactive, (2) Migration Requested, (3) Migration Preparing, (4) Migration Allowed, (5) Migration Finished, and (6) Resuming Plugins (cf Figure 2).

In the initial Migration Inactive state, all processes perform their normal execution as if the migration feature was disabled. This state allows for the realization of our first requirement of minimal runtime overhead. Although the protocol itself can be implemented as a pure in-band solution leveraging the existing communication channels between the processes, it has to be triggered from the outside world, eg, by sending a message on a Message Queue Telemetry Transport (MQTT) channel the processes are subscribed to, as done in our implementation. In the following, the entity deciding when migration shall occur will be called the migration framework.

On the receipt of such a message, all affected processes enter the Migration Requested state. This will be detected by the migration logic on the entry to the communication layer, eg, when the application layer issues a send or receive request. The next state, the Migration Preparing state,
corresponds to the actual execution of the Shutdown/Reconnect (S/R) protocol discussed in the following section. To meet the requirement of minimal additional migration costs, the protocol is only triggered for connections marked as non-migratable. This is a connection property that can vary in accordance with the underlying migration mechanism, e.g., in a virtual setup where each VM potentially holds more than one process, intra-VM shared-memory connections can be preserved.

Once all non-migratable connections have been shut down, the processes enter the Migration Allowed state and give feedback to the migration framework. Currently, the processes wait in a busy loop for the completion of the underlying migration mechanism and with that for the state change to Migration Finished. However, this state could be used for a temporary replacement of non-migratable connection types to connection types that could be migrated alongside the processes, e.g., TCP/IP connections are known to survive the live migration of virtual machines.\textsuperscript{51} As soon as the migration terminates, the processes again receive a signal from the migration framework and perform the according state change. This triggers the re-establishment of all connections that have been shut down prior to the migration. Therefore, the Resuming Plugins state is entered and finally left when the migration mechanism becomes completely inactive again.

### 4.1.1 | The Shutdown/Reconnect protocol

This refers to State 3 (Migration Preparing) of the migration states discussed above. Although the states have to be passed by all migrated processes with activated S/R support, the execution of the S/R protocol is only performed for non-migratable connections. This drastically reduces the migration overhead. Thereby, we only need to guarantee a local consistency on a per-connection basis, i.e., the whole migration is transparent to processes that are neither subject to a migration nor connected to other processes that are relocated within the cluster.

Figure 2B considers the execution of the S/R for a single connection; however, one should keep in mind that this can be performed concurrently for all non-migratable connections as we assume no dependencies between the individual channels. The S/R protocol is initiated with a shutdown request token sent by the process to be migrated to its peer. For the avoidance of further write attempts by the application layer on that connection, we introduce a write suspend flag postponing such endeavors. By using this mechanism, we meet the requirement of application transparency at the same time. Since the whole S/R logic can be implemented on the communication layer, if at all, the discussed behavior results in a delayed execution of send requests by the application layer. Although we assume reliable connections, there must be no data in-flight before closing the channel. Therefore, we keep it open for reading until the token is passed back as shutdown response by the peer process. This behavior can be leveraged by the communication layer for a reduction of the communication latency, i.e., channels closed for reading or writing do not need to be considered by a progress engine. As soon as the token arrives at the migrated process, the according channel can be closed for reading as well as by setting the respective read suspend flag. Once this state is valid for all non-migratable connections, the migration framework can be informed (cf Section 4.1).

Although we assume bidirectional communication channels, the protocol can be easily mapped onto unidirectional channels. Therefore, the peer processes have to initiate the same protocol for the counterpart on the receipt of a shutdown request.

Vital to the successful migration is the exchange of the shutdown request token in a PingPong manner, which drains the respective connection on both ends. We impose certain requirements that have to be met by the communication library and the underlying transport for our protocol to work, as follows. (1) The library has to provide some kind of queuing facility for those send/receive requests following a connection shutdown. This should be the case for most libraries in the HPC context which provide message tagging, e.g., as done by the MPI standard.\textsuperscript{52} (2) We assume that the underlying transport delivers messages in the same order as they have been transmitted and that the sender is informed about the completion of the operation indicating that the message is eventually received at the other end. Both properties are supported by IB and should be met by most common transports, e.g., the TCP/IP protocol running on top of Ethernet fulfills these demands as well. These requirements allow for the realization of a local consistency on the connection basis, i.e., having a consistent cut prior to the migration.

After the migration, two things need to be done: (1) the release of the read/write suspend flags and (2) reconnection of the communication channels. The effort that is necessary for the reconnection depends on the facilities provided by the communication layer. For our implementation, we could leverage the existing on-demand connection mechanism of the pscom library (cf Section 3.2.1). This comes with the advantage that only those channels are re-connected not before they are required for communication again. However, any mechanism is applicable allocating new resources on the target node, e.g., the mechanism that has been used for the initial connection establishment on the source node. At this point, another advantage of the S/R approach over a virtualization of location-dependent resources becomes apparent. The re-establishment of the communication channels allows for an evaluation of locality information choosing the best communication path for the updated topology, e.g., in case of process migration, two processes could potentially use shared-memory communication that should be favored over any alternative. An example of how we exploit the fact that two VMs reside on the same physical host after a migration is presented in Section 5.

### 4.2 | Evaluation

The following sections present an evaluation of the SR protocol. Firstly, the runtime behavior is analyzed by both microbenchmarks and application benchmarks. Secondly, the protocol’s scalability is investigated in a migration scenario.
4.2.1 Runtime overhead

The runtime overhead should be avoided as far as possible in accordance with the requirements stated above, i.e., applications that are not subject to a migration should not be affected by the implemented mechanism. Although the actual migration logic is only triggered if the process changes its state to Migration Requested, the progress engine has to observe the state variable regularly. In doing so, the implications on the critical path of the communication stack are essential for the application’s performance. For an estimation thereof, we performed a comparison between our implementation with enabled S/R support and the upstream sources§ (cf Figure 3).

The throughput results were obtained by executing our self-written benchmark (cf Section 3.2.3) determining the point-to-point latency and bandwidth. One of the processes resided on each of the two IvyBridge hosts respectively and IB was used as interconnect. The curves reveal that our implementation does not add any overhead on a wide range of message sizes. The same holds for the communication latency obtained by the exchange of zero-byte messages. Here, we see a difference neither in the average obtained from 106 runs nor in the standard deviation. A similar picture emerges regarding an overhead analysis using the NPB suite (cf Figure 3B). We used the MPI implementation of the computing kernels with 64 processes on all four cluster nodes and chose problem Class C. Each kernel was executed in a loop for 30 min, and the individual results were averaged afterwards. The overhead is negligibly small, and we only see small fluctuations around the zero line.

The preceding benchmark results show that the S/R protocol does not have a noticeable impact on the runtime behavior of the MPI applications. This is an important characteristic if we assume that actual migrations happen rarely during the course of an application. For an assessment of the migration performance, we therefore performed a further set of experiments determining the impact of the protocol on the actual migration time. The results of our study are presented in the following section.

4.2.2 Scalability

This parameter strikes the migration time itself. The time required for the shutdown and the later reconnect should be kept as small as possible. Since this has a direct dependency to the amount of non-migratable connections, i.e., for each of them, the S/R protocol has to be executed, we performed a migration analysis with varying process and connection counts, respectively (cf Figure 4).

We started this scalability study by the execution of an MPI benchmark exhibiting an all-to-all communication pattern, i.e., N MPI ranks continuously exchange zero-byte messages among each other. This benchmark was executed within two VMs possessing 16 GiB of guest-virtual memory, and the MPI processes were distributed evenly between them. One VM resided on one of the SandyBridge hosts, whereas the other was migrated between the two IvyBridge hosts back and forth. This procedure was repeated 15 times to obtain stable results. Consequently, each migration round requires the execution of the S/R protocol for \( \frac{N}{2} \) connections per process, i.e., a total of \( \left( \frac{N}{2} \right)^2 \) connections. It should be kept in mind that the shared-memory connections between processes running within the same VM can be preserved.

Although we can observe a linear growth of the time to shut down with respect to the connection count, it constitutes only an overhead of 0.5%-14% for 4 to 900 connections. Breaking this down to the per-connection overhead, we obtain a shutdown time of only 2 ms. The described scenario constitutes the worst case for a migration. Commonly, there are only few direct connections among the MPI processes of a single job, e.g., collectives are often mapped onto tree structures, resulting in a drastic reduction of affected connections. For an impression of how the S/R performs on the migration of a real-world application, we used mpiBLAST as an example (cf Figure 4A). This was started within four different VMs running on all cluster nodes using different process counts. Again, one VM was migrated back and forth between the two IvyBridge hosts. At the

---

§https://github.com/ParaStation/pscom
maximum, 326 ms was required for the shutdown of all non-migratable connections in an MPI session with 64 processes, i.e., eight processes were migrated. Compared to the overall migration time, this constitutes an overhead of less than 5%.

### 5 | Locality-Aware MPI Layer

Locality awareness and the thereby arising topology awareness has been around since the early 2000s with the emergence of Symmetric Multi-processing (SMP) systems. On the one hand, the communication layer should take the locality information of the processes into account, e.g., processes residing on the same physical host should leverage shared-memory communication channels for best performance, whereas inter-node communication is usually realized by means of high-speed interconnects such as IB. On the other hand, the communication topology plays a major role for the applications' performance. This aspect is two-fold: (1) the user-level application may provide the communication stack with information on the communication pattern, e.g., the MPI standard provides a means for the adjustment of the process placement by the definition of a virtual topology; and (2) the communication layer itself can take the locality information into account, e.g., for an optimization of collective communication.

In the context of virtualization, these aspects become relevant again. First of all, the communication layer has to distinguish between isolation domains residing on the same physical host and those running on distinct hosts, i.e., the former should leverage the shared physical memory for communication to obtain the best performance. In Section 5.1, we detail how this can be realized for system-level virtualization based on VMs by means of IVShmem. Secondly, the communication layer has to cope with potentially dynamic network topologies as migrations of VMs during runtime, e.g., for load balancing purposes, have a direct impact on the network topology of the affected jobs. This behavior differs from the static SMP awareness as discussed above. In doing so, two VMs originally placed on different nodes may now reside on the same machine and should therefore be able to dynamically adapt the communication paths accordingly. In Section 5.2, we illustrate how such a dynamic adaptation can be conducted especially with respect to collective communication patterns.

At this point, it should be emphasized that the topology dynamics we consider here is just rooted in the altering of process locations due to migrations, whereas the number of processes within a connected MPI session always stays the same. Yet, the MPI standard allows for session growth (and even for subsequent shrinkage) in accordance with its Dynamic Process Model. However, the challenges for locality awareness are here not much higher than for the static case since this mechanism is triggered from the application (most notably via MPI_Comm_spawn()).

### 5.1 | Intra-host inter-VM communication

In previous works, we investigated the impact of virtualization on applications running on a single physical node. Therefore, we executed different MPI applications within multiple VMs by using a non-virtualization-aware MPI library. In doing so, with a rising VM count, more and more process pairs had to communicate over the local IB HCA instead of using shared-memory channels. We found that especially communication-intensive applications suffer from performance penalties of up to 26% compared to the execution within a single VM. As a consequence, HPC systems leveraging virtualization should choose a VM granularity as coarse as possible. However, this limits the flexibility of a load balancer, which increases with finer granularities. Therefore, we designed a communication layer providing support for IVShmem communication, which is discussed in the following.

A shared-memory connection between co-located VMs might contradict with their strong isolation principle at first glance. In particular, from the security's perspective, this is an aspect worth discussing. However, at this point, we have to trade off security against performance, and the latter is commonly of major interest in the HPC domain. Furthermore, our solution only affects the implementation of the communication facility within
the MPI library. For the protection of the applications against each other in co-scheduling scenarios, it is possible to pass distinct memory segments
to VMs of different jobs, i.e., a VM only sees the memory segment that is necessary for the communication with processes of the same job residing
within other VMs.

5.1.1 The pscom IVShmem plugin
The pscom IVShmem plugin enables support for intra-host inter-VM communication within the communication library. A comprehensive discus-
sion of the plugin can be found in the work of Pickartz et al.13 It mainly consists of two parts: (1) the upper layer acting as the interface to the
hardware-agnostic part of the pscom library and (2) the lower layer that uses the Nahanni device for the management of the shared-memory region.
The main task of the upper layer is the implementation of the handshake mechanism that has to be provided by each plugin. This is the crucial com-
ponent of the plugin that allows two processes to determine whether communication is possible over this plugin or not. For making this decision, a
common determinator is necessary indicating their co-locality. Since the virtualization layer imposes a strong isolation of the VMs, the single com-
mon feature is exactly the shared-memory segment provided by Nahanni. Therefore, we use a UUID at a predefined offset within that segment that
has to be unique across the whole cluster. If the UUIDs seen by the processes match, they may assume to be located on the same physical node and
establish a communication channel over this memory segment.

Although the locality detection is dependent on the leveraged virtualization techniques, it has to be solved in any case to allow for the best
possible communication performance between co-located isolated domains. For example, in case of OS containers, a similar approach could be fol-
lowed by using a mounted memory segment; however, in this case, it would be probably more sophisticated to retrieve this information from the
sys-filesystem.

5.1.2 Evaluation of the plugin
Here, we determined the basic communication performance in terms of throughput and latency (cf Figure 5). In doing so, we measured the results
of our self-written benchmark for the following four scenarios.

- Native-SHM: shared-memory communication on the host
- Native-IB: communication using the local IB HCA on the host
- VM-IB: communication between two processes in co-located VMs using IB with SR-IOV
- VM-IVShmem: communication between two processes in co-located VMs via IVShmem

The throughput results reveal a performance almost identical to that of Native-SHM. For small message sizes, we can observe a little overhead,
whereas VM-IVShmem is slightly faster than Native-SHM for larger messages that still fit into the last-level cache. However, the performance differences
are negligibly small and might stem from different implementations of the communication channels within the respective plugins. Essentially,
the IVShmem plugin provides a performance benefit of up to 40% over VM-IB. Regarding the latency results, we see a similar picture (cf Figure 5B).
The point-to-point latency is reduced by around 67% for zero-byte MPI messages when choosing VM-IVShmem over VM-IB while having a variance
that is only at 56% of the variance we see for VM-IB. Compared to Native-SHM, we can observe a slight increase in latency and variance. However,
with around 0.10 µs to 0.14 µs, respectively, this is negligibly small. It should be noted that we did not put much effort into the optimization of the
IVShmem plugin, e.g., a tuning of parameters such as the send/receive buffer size should reveal potential performance improvements.

![Graph](image-url)  
**FIGURE 5** Microbenchmark analysis of the IVShmem support within pscom: (A) throughput and (B) latency results captured by performing a
PingPong pattern between two processes. The latencies are given in µs.
5.2 Topology awareness

From the perspective of an MPI implementation, topology awareness means, in the first instance, that the library is capable of dealing efficiently with network-related characteristics or peculiarities such as heterogeneity and diversity. For example, multiple concurrently available communication technologies, protocols, and/or routing schemes allow (but also oblige) the MPI library to select the most promising communication path between the source and the destination—preferably by taking also parameters like message lengths and communication patterns into account.

A very common example for such a case is a cluster consisting of multiple SMP nodes: while the communication within such nodes should be conducted via the shared memory (particularly via IVShmem in case of inter-VM communication), the inter-node communication has to be routed via the interlinking network. Mechanisms for locality detection, as explained in the previous section for IVShmem, can be utilized for the realization of such an appropriate path selection on a point-to-point basis.

However, especially with respect to collective communication patterns, e.g., barrier or broadcast operations, things become more complicated. In such cases, further parameters such as the amount of involved processes and their location come additionally into play. This subsection starts with an overview of existing approaches for overcoming these issues for the case of static topologies, whereas the subsequent paragraphs will introduce recent advancements made by us with respect to dynamically changing topologies.

5.2.1 Static topology awareness for collective communication

This can be conducted during the initialization phase of the MPI session. Here, basically, two different approaches are feasible: (1) either some higher-layer instance, e.g., the process manager that has a global view onto the process distribution within the physical topology, can be queried; or (2) the topology has to be scanned through the MPI layer on the network level. Relying on information provided by the process manager seems to be obvious since this instance actually decides about the placement of processes onto the nodes. However, this information is merely based on the processes’ node affiliation because network-related topology information is usually not available at that level. Therefore, a network scan on the MPI level may yield more reliable information and may allow for a more fine-grained categorization of the communication paths than just intra-node and inter-node links. On the downside, this approach is potentially more cost intensive as a truly global view requires a check of all possible connections between all process pairs.

Either of the approaches has a weighted (and commonly undirected) graph as a result. This can be stored as an adjacency matrix or in terms of multistage tables representing different communication classes. The latter is especially useful for hierarchical topologies. Although the optimal embedding of a collective communication pattern, e.g., a broadcast operation, into an arbitrary topology graph is an NP-hard problem, there exist some quite simple heuristics that can be used to optimize the respective mapping. This is especially true for hierarchical topologies, e.g., a two-tier hierarchy as formed by an SMP cluster, the following rules could be applied for an improvement of the performance of collective operations.

1. Every sender-receiver path used by an algorithm contains, at most, one inter-node hop.
2. No data item travels multiple times to the same node.

The first rule limits the impact of the inter-node latency to one hop, whereas the second rule saves inter-node bandwidth that would be required for the transmission of the same data in parallel to the same node (but to different processes, of course). Figure 6 shows an example of a simple broadcast pattern where in image (A), these rules are violated, whereas in image (B), the pattern is accordingly optimized.

5.2.2 Dynamic topology awareness for collective communication

This entails the crucial issue of the coherent detection of a changed topology across the affected processes; in the fully connected case, this corresponds to all processes of an MPI session. This problem is two-fold: (1) the respective mechanism has to ensure the propagation of the topology update to all affected processes and (2) the related consequences in terms of changed communication patterns have to be applied coherently.

![Figure 6](image-url) A simple broadcast pattern with (image A) a non-topology-aware approach and (image B) an approach taking the locality information into account. The dashed arrows indicate paths where Rules 1 and 2 are broken.
Otherwise, only some processes may already use an adapted pattern while the remaining still use the old pattern, which certainly leads to mismatches and deadlocks. Figure 7 illustrates this issue: here, Process 1 migrates in the course of the broadcast operation from Node 1 to Node 0. The resulting question is: Do processes 0, 1, and 2 coherently recognize that the pattern is to be changed?

At this point, it should be emphasized that it is not an issue if the older pattern is still used after the migration—it would merely be no longer optimized—but it has then to be ensured that it is further used on all processes involved in the collective operation. Therefore, a solution could be the introduction of a synchronization point, e.g., at the beginning of a collective operation where the pattern to be used is negotiated. However, such a frequent negotiation would, on the one hand, add further overhead to each call of a collective operation. On the other hand, the MPI semantics allow collective functions to complete as soon as the caller’s participation in the pattern is finished. This contradicts with a synchronized negotiation as proposed above.

5.2.3 Topology-aware extension to ParaStation MPI

This is part of our contributions for the study of the challenges as well as the performance improvements that may result from such dynamic topology awareness. In doing so, we tackle the synchronization problem—at least in the first instance—at the application level. Therefore, we propose an additional experimental collective MPI function (namely, MPIX_Comm_refresh(), see below) that is available for explicit and frequent calls from the application level at appropriate points in code or time.

The reason for choosing this approach as a starting point is that the realization of a coherent detection of an altered topology is quite hard from within the MPI layer. Despite the fact that in a fully connected MPI session, every process eventually notices that one (or more) of its peers has been migrated, the problem of reacting upon such an event coherently still exists. This is because the individual process cannot determine without a global operation whether all the other peers have already detected the changed topology as well, and, as already discussed above, such an additional global operation under the hood of every collective communication function would impair the overall performance.

On the downside, proposing a new user-level function for realizing this global operation just passes the buck to the application programmer—most probably resulting in an underuse or even in an overuse of this function. However, at this point, further solutions to this problem may become apparent: on the one hand, a quite natural approach would be to perform the required additional synchronization step frequently but not every time a collective MPI function is called. That way, the negative performance impact can be curbed, of course, at the expense of a delayed adaptation of the collective communication patterns to the changed topology. On the other hand, some kind of a piggyback mechanism could be envisaged that delivers the needed coherence information alongside the common MPI payload. Although the implementation of such an elaborated mechanism is currently under our investigation, it is not part of this article so that we will focus here on the former approach.

The actual topology awareness of ParaStation MPI comes in terms of optimized collectives for SMP clusters from its upper MPICH layer. This allows for a propagation of locality information in terms of Node-IDs by lower communication layers. This can be leveraged for a mapping of collective operations within the upper MPICH layer onto point-to-point communication patterns that exploit the fast node-internal communication paths. Therefore, ParaStation MPI checks all pscom connections whether they are related to a plugin that is dedicated to node-local communication. Since these checks are done across all processes of an MPI session and due to the fact that connection types are bidirectionally symmetric, appropriate Node-IDs representing SMP domains can easily be determined and communicated.

In the static case, these checks are done during the MPI initialization phase, and the resulting Node-IDs are then stored in a table that covers all processes of the MPI session and that is kept constant over the session’s lifetime. For doing so, all processes of the session have to be connected to each other already during the initialization. This implies a deactivation of the pscom on-demand feature.

---

1 This is due to the S/R mechanism as explained in Section 4.
2 Therefore, we do not actually intend this function to become part of the MPI standard, but rather want to use it here as a starting point for the technical discussion.
3 These are the layers beneath the so-called ADI3 interface.
In the dynamic case, the above-mentioned `MPIX_Comm_refresh()` function has to be called either explicitly from the application level or implicitly from within the MPI layer. As this function is collective over the respective MPI communicator, it can re-trigger the pscm connection checks and gather the related results in a new Node-ID table associated with this communicator. For this purpose, all connections have to be effectively re-established after migration, which, in turn, is also checked and triggered within the refresh function. All subsequent collective operations on the communicator will then be performed in accordance with the updated information.

For an automated update, i.e., a transparent usage of the refresh function, the user can set a certain environment variable (namely, `PSP_AUTO_COMM_REFRESH`) to a value that defines the number of collective function calls on each communicator that should pass before the next collective update. That way, the user may choose a good tradeoff between overhead and benefit for the automated adaptation suited for the particular application. However, as an adaption is only reasonable if the communicator is used quite frequently for collective operations and since migrations should not happen that frequently, a ballpark figure could be 100 function calls—limiting the impact of the related adaptation overhead to 1%.

5.2.4 Evaluation

The evaluation of the potential performance improvements of the topology awareness as well as the overhead of the automated communicator update uses the broadcast (`MPI_Bcast`) algorithms as they are provided by the upper MPICH layers of ParaStation MPI. In principle, MPICH features the following three different patterns for a broadcast and selects the most promising one on the basis of message length and process count:

- a binomial tree algorithm for short messages (< 12 KiB) and/or for small process counts (< 8 ranks),
- a Scatter-based algorithm with a subsequent Allgather based on recursive doubling for mid-size messages, and
- a Scatter-based algorithm with a following Allgather based on a ring pattern for large messages (> 512 KiB).

If SMP awareness is enabled, MPICH performs one of these patterns at first in a strict inter-node style for forwarding the message to all involved nodes,** followed by the subsequent intra-node distribution via the shared-memory channels. This differentiation between inter-node and intra-node peers is done via shadow communicators that are built by means of the Node-IDs, which, in turn, are derived from the pscm connection types.

Figure 8 shows the performance results that we have measured with the IMB for the `Bcast` pattern. Curve ——— represents the latencies for enabled SMP awareness, i.e., considering both IVShmem and common shared-memory channels, whereas —— shows the behavior without this awareness. As one can see, especially for mid-size messages and the related communication algorithm, the adaptation of the pattern to the actual topology exposes to be beneficial. However, for the dynamic case, where we may assume that the actual topology has just recently changed to the current one, this adaptation comes at the expense of a refreshed communicator. For gauging also this overhead, we have again applied the broadcast pattern of the IMB for the already adapted case, but with different auto-refresh frequencies (cf Figure 9): 0 = no refresh (——); 1 = refresh upon every communicator usage (——); 10/100 = refresh just upon every tenth and every hundredth communicator usage, respectively (—— ———).

As expected, the communicator update within every call results in a drastic increase of the broadcast’s latency. However, less frequent calls of the refresh mechanism do not have a considerable impact on the performance. Depending on the application’s behavior and the expected amount of migrations, values between 10 and 100 are reasonable for the update frequency.

**It should be noted that if the root of the broadcast is not a so-called node-root rank, an additional intra-node point-to-point communication between the broadcast root and the local node-root has to be performed beforehand.
FIGURE 9  Overhead generated by the automatic update of the MPI communicator for different refresh rates and the Bcast operation. This was executed for small-size messages among 64 processes residing in 64 VMs distributed across all four cluster nodes

6  | PONCOS: A CO-SCHEDULER FOR VIRTUALIZED CLUSTERS

So far, we have discussed challenges that arise from migrations in HPC environments. On the one hand, a migration using isolated environments such as VMs circumvents the problem of residual dependencies to a large extent (cf Section 1.1). On the other hand, such an isolation demands for support for dynamically changing topologies within the communication layer (cf Section 5).

By proposing the S/R protocol as well as locality awareness within the MPI layer, we provide solutions to some issues discussed before. As co-scheduling is one of our major motivations for a consideration of migration in the HPC context, we have implemented a prototype co-scheduler. It targets at an optimization of the systems’ utilization. In doing so, it takes the applications’ main memory bandwidth utilization into account for taking load balancing decisions.

6.1  | Design and implementation

We have presented the first release of the poncos in previous works. This performs a co-scheduling of HPC applications by taking scheduling decisions based on the applications’ main memory bandwidth utilization. As there are no hardware counters in the CPUs of our test systems, which allow for a bandwidth estimation, poncos leverages libDistGen instead. This library estimates the portion of the available memory bandwidth on a set of exclusively assigned cores. Therefore, it compares the currently available bandwidth ($B_{\text{current}}$) to values that have been captured in an initialization phase on the idle system ($B_{\text{max}}$). Thus, the estimated usage of the running application can be computed as follows:

$$B_{\text{job}} = 1 - \frac{B_{\text{current}}}{B_{\text{max}}}.$$  

$B_{\text{job}}$ should not be treated directly as a percentage since the maximum value it may attain is not equal to 1. This is because the CPU will never starve a core but distribute the available bandwidth equally among all cores. A detailed discussion on this topic can be found in the work of Breitbart et al, but in general, a higher number corresponds to a high memory bandwidth utilization whereas a value of 0 means that the application essentially issues no main memory access. It should be noted that a scheduling based on the applications’ main memory bandwidth utilization requires a short delay between each scheduled job. Expressive values can only be obtained after an initialization phase, which may vary from application to application. For now, we rely on a timer that fits our test applications. However, an automatic detection of application phases could be implemented by using hardware performance counters.

6.1.1  | Two-app scheduler

The two-app scheduler was the only scheduling algorithm implemented in the previous release of poncos. This divides the cluster into two halves. Each half embraces all nodes but only half of the cores respectively, ie, we use two slots per node. Therefore, we span these slots over half NUMA domains (cf Figure 10) to allow memory-bound applications to profit from the higher main memory bandwidth when using multiple NUMA domains within each node. In contrast, compute-bound or communication-bound applications should be mostly insensitive to the actual intra-node pinning. As a result of this method, no more than two jobs are executed at a time. The scheduling algorithm used within this approach is presented in Algorithm 1.

††https://github.com/lrr-tum/libdistgen
FIGURE 10 The mapping of slots used by poncos onto a single node. Each slot spans over two half NUMA domains for the provision of maximal main memory bandwidth

Algorithm 1 The two-app scheduler of poncos

\begin{algorithm}
\begin{algorithmic}
\ForAll{$\text{job}_{\text{cur}}$ in $\text{job\_queue}$}
\State $\text{WAITFORFREE SLOT}$
\State $\text{START}(\text{job}_{\text{cur}})$
\State $\text{WAITFORINITIALIZATION}(\text{job}_{\text{cur}})$
\If{$\text{running\_jobs} = 2$}
\State $\text{SUSPEND}(\text{job}_{\text{prev}})$
\EndIf
\State $\text{B}_{\text{job}_{\text{cur}}} \leftarrow \text{MEMBW\_UTIL}(\text{job}_{\text{cur}})$
\If{$\text{running\_jobs} = 2$}
\State $\text{RESUME}(\text{job}_{\text{prev}})$
\If{$\text{B}_{\text{job}_{\text{cur}}} + \text{B}_{\text{job}_{\text{prev}}} > \text{threshold}$}
\State $\text{SUSPEND}(\text{job}_{\text{cur}})$
\State $\text{WAITFOR}(\text{job}_{\text{prev}})$
\State $\text{RESUME}(\text{job}_{\text{cur}})$
\EndIf
\EndIf
\State $\text{job}_{\text{prev}} \leftarrow \text{job}_{\text{cur}}$
\State $\text{B}_{\text{job}_{\text{prev}}} \leftarrow \text{B}_{\text{job}_{\text{cur}}}$
\EndFor
\end{algorithmic}
\end{algorithm}

By using this approach, we could demonstrate the feasibility of co-scheduling based on an online analysis of the memory bandwidth utilization. However, this scheduler is limited in its flexibility as only two applications can be executed simultaneously, i.e., each application has to allocate all half-nodes available in the system. This limitation is not crucial for small systems. However, we expect only few applications to scale on all nodes of an exascale system and that co-scheduling is one means to overcome the resulting underutilization.

6.1.2 Multi-app scheduler

The multi-app scheduler forgoes this limitation by accepting allocations of fewer nodes than available in the cluster per job. Therefore, we introduce a slightly modified algorithm (cf Algorithm 2). Furthermore, we implemented the notion of controllers that can be used by the different schedulers for the isolation of the co-scheduled jobs within the nodes. For now, we have implemented a cgroup controller as well as a VM controller. The former leverages Linux control groups for the isolation of jobs sharing a node. The VM controller uses system VMs based on QEMU/KVM and comes with migration support.

Yet, the multi-app scheduler assumes two slots per node comprising one half of each NUMA domain, respectively. As the slots opposing those allocated by a job may be used by different applications at the same time, we need to keep track of the machine usage per slot. This information can then be used to suspend those jobs using the slots opposing the current job (cf Lines 5 and 6 in Algorithm 2). In Algorithm 1, it is sufficient to take the main memory bandwidth utilization of a single node in place of the whole cluster. This is possible as we can assume a homogeneous utilization across the system for the two-app case. Now that the machine usage varies among the cluster nodes, i.e., not all nodes are shared by the same job pairs, we need to compute the job’s utilization based on the individual values obtained for the allocated slots. If this exceeds a predefined threshold, there are two options: (1) the job is suspended such as done in the two-app scheduler and has to wait until the cluster allocation changes (cf Lines 19-21 in Algorithm 2), i.e., a job terminates; (2) if supported by the controller, the scheduler tries to relieve the overloaded nodes by swapping the affected slots with those of underutilized nodes (cf Lines 12-17 in Algorithm 2). An alternative to swapping applications is killing and restarting the job, which may
make sense especially if an application was just started; however, not all applications or batch scheduler job scripts cope well with such an approach, especially if they modify the local file system.

For an assessment of the potential introduced by co-scheduling, we regarded a simple job queue comprising four jobs: two mpiBLAST jobs followed by two jobs executing LAMA's CG solver. Both mpiBLAST jobs request 16 processes, i.e., two slots, and query the DNA of a fruit fly (Drosophila melanogaster). The query was created by using $13 \times 10^3$ sequences from itself. In contrast, the LAMA jobs requested four processes and four threads per process, respectively, resulting in two required slots as well. We chose this LAMA configuration for the avoidance of any NUMA effects arising from the execution of fewer LAMA processes than available NUMA domains. The CG solver was applied on a sparse matrix generated by using LAMA's matrix generator. It has a size of $2000 \times 2000$ elements and is filled with a 2D five-point stencil.

Figure 11 presents a comparison of the different scheduling approaches that come with poncos. As the test system comprises two different node types (cf Section 3.1.2), we limited the CPU frequency of the IvyBridge nodes to the 2 GHz of the SandyBridge systems and used the same memory modules in all four nodes.

The first scenario (cf Figure 11A) corresponds to an exclusive assignment of the jobs to the cluster nodes, which is used by most of the larger compute centers. In this case, the overall execution time of around 25 min is mainly determined by the two LAMA jobs due to their memory-bound characteristics. The second scenario (cf Figure 11B) uses the multi-app scheduler described above in conjunction with the cgrouper controller. One can easily identify the initialization delay prior to the measurement of the job’s main memory bandwidth utilization. As mpiBLAST is a compute-bound application, poncos chooses a co-scheduling on the first two cluster nodes. In contrast, LAMA’s CG solver, which already saturates the main memory bandwidth with a subset of the available cores, has to be handled differently. After the initialization phase of the second LAMA job, poncos recognizes an overload of Nodes 2 and 3. However, as the cgrouper controller does not allow for a load balancing by means of migration support, it suspends the job until the first LAMA instance terminates. Only then is it resumed, resulting in an overall execution time of the regarded job queue that even exceeds that of the exclusive case by around 4%.

As opposed to this, the multi-app scheduler allows for a drastic reduction of the execution time when leveraging a controller with migration support (cf Figure 11C). For a minimization of the migration costs, we only migrate the guest memory and register state between the hosts. The virtual disk image is located in the network storage and accessible from all nodes. Large disk images composed of multiple libraries increase the VM startup time.
time and the migration time since part of the image may have to be re-read by the new host. However, the image size itself does not directly limit migration. In general, the migration time is typically dominated by the time it takes to transfer the memory used by the HPC application.

The multi-app scheduler initially behaves similar to the previous scenario, i.e., both mpiBLAST jobs are co-scheduled on Nodes 0 and 1 in accordance with the values obtained from libDistGen. However, on the detection of the overload when scheduling job four onto Nodes 2 and 3, poncos can now resolve this by swapping the slots of the second LAMA job with slots from the first two nodes. In this case, both slots of the first mpiBLAST job are chosen as swap candidates. However, it should be noted that this choice was made by chance for two slots of the same job. Depending on the values obtained from libDistGen, the choice could have been made for either of the four slots on Nodes 0 and 1. After the migration, each LAMA job is co-scheduled with one of the mpiBLAST jobs, respectively, resulting in a reduction of the overall execution time of 23% compared to exclusive scheduling. As both affected jobs use more than a single cluster node, the S/R protocol is required in this case. It ensures the resolution of the location-dependent resources (cf Section 4), i.e., the open handles to the node-local IB adapter that has been made available via SR-IOV. We conduct the migration by using the local IB adapter for the data transfer, i.e., migration via Remote Direct Memory Access (RDMA). This results in an important performance improvement compared to the migration over Gigabit Ethernet. In the scenario depicted in Figure 11, we observe a migration time of around 15 s for the swap, i.e., the concurrent migration, of all four VMs.

For the sake of completeness, we included a comparison to a slightly modified job queue in which LAMA’s CG solver and mpiBLAST appear in alternating order (cf Figure 11D). In this case, the potential co-scheduling pairs are directly located on the same set of nodes abandoning the need for migrations or a suspending of jobs. This results in a reduction of the execution time by around 11% compared to the previous scenario. Apart from the time saving due to the missing migration, the cgroup scheduler potentially results in faster execution time as it does not introduce an additional virtualization layer. However, this comes at the cost of missing migration support, which introduces a strong dependency of the order of the job queue.

![Figure 11](image-url)
CONCLUSION

In this paper, we have investigated not only the prospects but also the challenges arising from migration techniques in the HPC context. In doing so, we discuss challenges that come with future exascale systems such as a drastically decreasing MTBF and load imbalances resulting from the increasing node and core counts. The migration of HPC jobs or parts thereof can be one means to overcome some of these issues. On the one hand, according mechanisms can be used for the evacuation of nodes that are likely to fail. On the other hand, load imbalances can be resolved by the migration of a portion of the processes in transparency to the application layer. However, migrations put further demands onto the communication layer if the dynamic behavior of the system should be transparent to the application layer.

Firstly, all residual dependencies to the source node of a migration have to be resolved. We meet this requirement by leveraging system-level virtualization based on VMs. However, the presented concepts and methodologies also apply to other mechanisms that may be used for the isolation of processes. For example, container-based virtualization may be an attractive alternative to VMs sacrificing their flexibility while reducing the overhead compared to native execution. In any case, the virtualization of performance-critical resources, eg, the high-performance interconnect, should be avoided. Therefore, we propose the S/R protocol (cf Section 4) that enables the seamless migration of MPI jobs using OS-bypass networks. In a scalability study, we could demonstrate its feasibility for up to 900 connections.

Secondly, the malleability resulting from dynamically changing topologies has to be taken into consideration by the session layer. Although topology awareness and locality awareness have to be addressed by static systems as well, the dynamic behavior imposes further requirements on these mechanisms. On the one hand, the communication layer shall always choose the communication path promising best performance, which may be affected by migrations. On the other hand, collective operations may profit from locality information for an optimization of the underlying communication pattern. With our locality-aware extension to ParaStation MPI, we address both issues: the IVShmem plugin for the pscom library (cf Section 5.1.1) allows for efficient intra-node inter-VM communication, whereas our extension of the MPI layer (cf Section 5.2.3) ensures topology awareness on the session layer. We could show that the communication between two co-located VMs via IVShmem offers near-native performance. Furthermore, the latency of broadcasts for mid-size messages could be reduced by up to 50%.

In the last part, we showcase the potential of co-scheduling with our prototype scheduler poncos in a multi-node setup. In the presented scenario, we could reduce the overall execution time of an example job queue by 23% when using co-scheduling in conjunction with VM migration compared to an exclusive assignment of the nodes. In doing so, we leverage the presented SR protocol for a resolution of the residual dependencies to the communication hardware. Yet, the concepts are independent from the underlying migration and isolation techniques. The S/R protocol is designed for communication-related resources and does not consider heterogeneity with respect to computational-related resources such as accelerators. However, the challenges are similar in this context, ie, ensuring a consistent state prior to the migration. This can, eg, be achieved by a virtualization of the accelerator resources and a migration of the accelerator images across physical devices at synchronization points. This way, even the migration across heterogeneous architectures is possible if frameworks such as OpenCL are used.

However, more research is necessary to explore how far these frameworks support a dynamic change of the accelerator during runtime, eg, a migration between nodes with different GPU generations. As part of future works, we further plan for a container-based virtualization support for poncos. This should result in a reduction of the overhead generated by the virtualization layer. Moreover, we intend an investigation of different scheduling goals with poncos such as a minimization of the power consumption.

ACKNOWLEDGMENT

This research and development was supported by the Federal Ministry of Education and Research (BMBF) under grant 01IH13004B (Project FAST).

ORCID

Simon Pickartz http://orcid.org/0000-0002-6316-6396

REFERENCES

1. Bergman K, Borkar S, Campbell D, et al. Exascale computing study: technology challenges in achieving exascale systems. Technical Report; 2008.
2. Cappello F. Fault tolerance in petascale/exascale systems: current knowledge, challenges and research opportunities. Int J High Perform Comput Appl. 2009;23(3):212-226.
3. Dongarra J, Héraut T, Robert Y. Fault tolerance techniques for high-performance computing. Paper presented at: Herault T, Robert Y, eds. Computer Communications and Networks. Cham, Switzerland: Springer; 2015.
4. Breitbart J, Weidendorfer J, Trinitis C. Case study on co-scheduling for HPC applications. Paper presented at: 2015 44th International Conference on Parallel Processing Workshops; 2015; Beijing, China.
5. Breitbart J, Pickartz S, Weidendorfer J, Monti A. Viability of Virtual Machines in HPC. Cham, Switzerland: Springer International Publishing; 2017;721-733. https://doi.org/10.1007/978-3-319-58943-5_58
6. Pickartz S, Lankes S, Monti A, Clauss C, Breitbart J. Application migration in HPC—A driver of the exascale era? Paper presented at: 2016 International Conference on High Performance Computing Simulation (HPCS); 2016; Innsbruck, Austria.
7. Pickartz S, Breitbart J, Clauss C, Lankes S, Monti A. Co-Scheduling of HPC Applications. Amsterdam, The Netherlands: IOS Press; 2017.
8. Milojčić DS, Doug里斯 F, Paindaveine Y, Wheeler R, Zhou S. Process migration. ACM Comput Surv. 2000;32(3):241-299.
9. Pickartz S, Clauss C, Lankes S, Krempel S, Moschty N, Monti A. Non-intrusive migration of MPI processes in OS-bypass networks. Paper presented at: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); 2016; Chicago, IL.
10. Pickartz S, Clauss C, Lankes S, Monti A. Enabling hierarchy-aware MPI collectives in dynamically changing topologies. Paper presented at: Proceedings of the 24th European MPI Users’ Group Meeting (EuroMPI’17); 2017; New York, NY.
11. Breitbart J, Weidendorfer J, Trinitis C. Automatic co-scheduling based on main memory bandwidth usage. Paper presented at: Proceedings of the 20th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP); 2016; Chicago, IL.
12. Breitbart J, Pickartz S, Weidendorfer J, Lankes S, Monti A. Dynamic co-scheduling driven by main memory bandwidth utilization. Paper presented at: 2017 IEEE International Conference on Cluster Computing (CLUSTER 2017); 2017; Honolulu, HI.
13. Pickartz S, Baude J, Lankes S, Monti. A Locality-Aware Communication Layer for Virtualized Clusters. Cham, Switzerland: Springer International Publishing; 2017. In press.
14. Pickartz S, Gad R, Lankes S, et al. Migration Techniques in HPC Environments. Cham, Switzerland: Springer International Publishing; 2014;486-497.
15. Duell J. The design and implementation of Berkeley Lab’s Linux checkpoint/restart. Technical Report; 2005.
16. Chakravorty S, Mendes C, Kalé L. Proactive fault tolerance in large systems. Paper presented at: HPCRI Workshop in Conjunction with HPCA; 2005; Phoenix, AZ.
17. Chakravorty S, Mendes CL, Kalé LV. Proactive Fault Tolerance in MPI Applications via Task Migration. Berlin, Germany: Springer-Verlag; 2006;485-496.
18. Wang C, Mueller F, Engelmann C, Scott SL. Proactive process-level live migration and back migration in HPC environments. J Parallel Distrib Comput. 2012;72(2):254-267.
19. Gao Q, Yu P, Huang W, Panda DK. Application-transparent checkpoint/restart for MPI programs over InfiniBand. Paper presented at: 2006 International Conference on Parallel Processing (ICPP06); 2006; Columbus, OH.
20. Hursey J, Squyres JM, Mattox TI, Lumdsaine A. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. Paper presented at: IEEE International Parallel and Distributed Processing Symposium; 2007; Rome, Italy.
21. Sankaran S, Squyres JM, Barrett B, et al. The LAM/MPI checkpoint/restart framework: system-initiated checkpointing. Int J High Perform Comput Appl. 2005;19(4):479-493.
22. Ansel J, Arya K, Cooperman G, DMTCP: Transparent checkpointing for cluster computations and the desktop. Paper presented at: 2009 IEEE International Symposium on Parallel Distributed Processing; 2009; Rome, Italy.
23. Cao J, Kerr G, Arya K, Cooperman G. Transparent checkpoint-restart over InfiniBand. Paper presented at: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC’14); 2014; Vancouver, Canada.
24. De Blanche A, Lundqvist T. Addressing characterization methods for memory contention aware co-scheduling. J Supercomput. 2015;71(4):1451-1483.
25. Mars J, Vachharajani N, HUndt R, Sofia ML. Contention aware execution: Online contention detection and response. Paper presented at: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization; 2010; Toronto, Canada.
26. Eklov D, Nikoleris N, Black-Schaffer D, Hagersten E. Bandwidth bandit: Quantitative characterization of memory contention. Paper presented at: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO); 2013; Shenzhen, China.
27. Zhang J, Lu X, Jose J, Li M, Shi R, Panda DKD. High performance MPI library over SR-IOV enabled InfiniBand clusters. Paper presented at: 21st International Conference on High Performance Computing (HPC); 2014; Panaji, India.
28. Zhang J, Lu X, Jose J, Shi R, Panda DKD. Can Inter-VM Shmem Benefit MPI Applications on SR-IOV Based Virtualized Infiniband Clusters? Cham, Switzerland: Springer International Publishing; 2014;342-353.
29. Zhang J, Lu X, Chakraborty S, Panda DKD. Slurm-V: Extending Slurm for Building Efficient HPC Cloud with SR-IOV and IVShmem. Cham, Switzerland: Springer International Publishing; 2016;349-362.
30. Yoo AB, Jette MA, Grondona M. Slurm: Simple linux utility for resource management. Paper presented at: Workshop on Job Scheduling Strategies for Parallel Processing; 2003; Berlin, Germany.
31. Budai P, Goldschmidt B. Performance Analysis of Cloud-Based Application. Berlin, Germany: Springer; 2014;476-483. https://doi.org/10.1007/978-3-662-43880-0_54
32. Householder R, Arnold S, Green R. On cloud-based oversubscription. Int J Eng Trends Technol. 2014;8. https://doi.org/10.14445/22315381/IJETT-V8P273
33. Breitbart D, Dubitzyzki Z, Epstein A, et al. An adaptive utilization accelerator for virtualized environments. Paper presented at: IEEE International Conference on Cloud Engineering; 2014; Boston, MA. https://doi.org/10.1109/ICCE.2014.63
34. Tomás L, Vázquez C, Tordsson J, Moreno G. Reducing noisy-neighbor impact with a fuzzy affinity-aware scheduler. Paper presented at: 2015 International Conference on Cloud and Autonomic Computing; 2015; Boston, MA. https://doi.org/10.1109/ICCAC.2015.14
35. Improving Real-Time Performance by Utilizing Cache Allocation Technology. Intel Corporation; 2015.
36. Huang W, Liu J, Koop M, Abali B, Panda D. Nomad: Migrating OS-bypass networks in virtual machines. Paper presented at: Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE’07); 2007; San Diego, CA.
37. Intel Virtualization Technology for Directed I/O. Intel Corporation; 2014.
38. Intel LAN Access Division. PCI-SIG SR-IOV Primer. Technical Report 2.5. Intel Corporation; 2011.
39. Guay WL, Reinemo SA, Johnsen BD, et al. Early experiences with live migration of SR-IOV enabled InfiniBand. J Parallel Distrib Comput. 2015;78:39-52.
40. Kivity A, Kamay Y, Laor D, Lublin U. kvm: the Linux virtual machine monitor. Paper presented at: Proceedings of the Linux Symposium, vol 1; 2007; Ottawa, Canada.
