Portable Topology-Aware MPI-I/O

Rob Latham
Math. and Computer Science Div.
Argonne National Laboratory
Lemont, IL, USA
robl@mcs.anl.gov

Leonardo Bautista-Gomez
Barcelona Supercomputing Center
Barcelona, Spain
leonardo.bautista@bsc.es

Pavan Balaji
Math. and Computer Science Div.
Argonne National Laboratory
Lemont, IL, USA
balaji@anl.gov

***Abstract*—Recent advances in storage devices are opening new opportunities in high-performance computing (HPC). Technologies such as solid-state drives (SSD) and non-volatile memories (NVM) are becoming increasingly popular because of the important gains they can represent for HPC. Indeed, novel architectures with deeper storage hierarchies populated with SSDs and/or NVM offer new ways to improve applications' performance. For instance, fast multilevel checkpointing or in-situ data analysis are some of the techniques that can be greatly improved thanks to these new technologies. However, optimizations made for one system can impose performance costs in another machine due to topology differences. To take advantage of increasingly complex systems, we propose extensions to MPI enabling codes to determine which nodes of a system share common features. Our approach provides a portable mechanism for resource discovery. It also lays the foundation for additional optimizations in checkpointing and in ROMIO. In this paper we present the design and implementation of such a feature and test it with multiple benchmarks. Our results demonstrate the benefits of this portable resource discovery functionality.**

***Keywords*-MPI; communicators; topology; discovery; storage; portability; SSD; NVM; burst buffers**

I. Introduction

While supercomputers continue to deliver increasingly greater computational power, the associated storage systems lag behind in terms of available bandwidth. Supercomputer architects have designed more complex storage hierarchies to bridge the storage-CPU performance gap. These storage hierarchies include solid-state drives (SSDs) and nonvolatile memory (NVM), either local to a compute node or hosted on an intermediate proxy node. While these technologies offer new opportunities for performance improvements, they also pose a challenge: applications need a way to discover what, if any, storage hierarchy is in place.

In particular, the storage hierarchy can have an important impact on performance for large-scale scientific applications, and any gain in I/O can have large benefits for the application. Therefore, future exascale applications will need to make efficient use of all the storage levels in the system. A one-size-fits-all approach will not be apropriate: the storage architecture and characterisitcs of one machine might be completely different from other machines. In this context, the portability of scientific codes will depend on the capabilities of the system middleware and runtime to discover the different storage levels available in a portable fashion. Programming libraries, such as MPI, should offer features and tools that would allow scientists to efficiently take advantage of these new devices in an intuitive way. Thus, the MPI standard must adapt to this changing landscape and integrate the new architectural changes.

In this paper, we propose an MPI extension that allows users to portably discover which MPI processes share particular storage levels. For instance, if a system is equipped with SSDs on the compute nodes, this new MPI feature would allow users to create communicators consisting of all MPI processes that have access to the same local SSD. Those MPI communicators would then allow efficient communication and local optimization for HPC codes. While this example could be achieved by other means such as looking for the hostname, those other options are site-specific: porting to a new platform would require adjusting the "topology investigation" component. In addition, more complex architectures might be designed, such as systems with local storage shared by multiple compute nodes or local storage that can be reconfigured dynamically. Our proposed extension offloads the topology investigation to the MPI implementation and site-specific defaults. We have implemented this MPI extension and tested it with multiple benchmarks in different scenarios. Our results show that large performance improvements can be achieved when such a portable storage discovery is enabled.

I/O provides a compelling use case, but our approach could be extended to partition MPI processes into groups that share network resources, are physically adjacent, or otherwise share some kind of resource.

The contributions of this paper can be summarized as follows.

- We design and implement a portable way to discover the storage topology of the system.
- We verify the accuracy of the MPI topology-query functionality on a system with local storage on the compute nodes and global file system.
- We measure the performance gains obtained by leveraging multiple storage levels thanks to the proposed topology-query functionality.
- We model possible gains that could be achieved using this functionality in exascale systems for different storage system configurations.

In Section II we discuss the challenges of topology discovery, particularly with the introduction of an NVRAM layer to the storage hierarchy. Section III describes how we extended

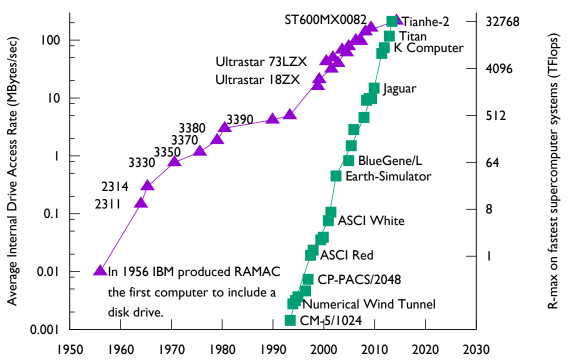

Fig. 1. Dramatic improvement in CPU performance (green, 80% per year) compared with storage bandwidth (purple, improving only 11% per year).

MPI-3 to meet these challenges. It also describes our approach and how it might be used in a real application. We prototyped this design in MPICH and modified two codes to make use of this new feature. As shown in Section IV, bypassing the parallel file system can greatly improve performance. Section V discusses related work, and Section VI summarizes our conclusions.

II. Background

HPC applications have been developed by using mainly two storage levels: the main memory and a globally visible parallel file system (PFS). The PFS gathers together thousands of hard disks for performance and reliability and presents a single unified namespace. In recent years, however, using the PFS as the only medium-term storage has been shown to impose a significant limitation on an application's performance [1]; and the performance gap between CPU and storage grows more dramatic with time. Figure 1 shows that as each new system comes on line, it provides the well-known doubling of compute power every 18 months. Storage *capacity* increases greatly as well, but storage *performance* improves only slightly. To close the performance gap, systems have begun to deploy devices in a deeper storage hierarchy.

Placing hard disks in the compute nodes as an intermediary step between the main memory and the PFS has been tried in the past, but traditional hard disks pose significant reliability challenges [2]. Because of the nature of hard disks and their moving components, relying on disk drives has made compute nodes prone to failures and has introduced instability in the whole system.

Fortunately, new storage devices are populating the market that offer new reliability-performance trade-offs. In particular, SSD and NVRAM are good candidates for bridging the gap between compute node memory and the PFS, providing a durable staging area for application data. How such devices will be placed in the system is still unknown, however. Envisioned topologies include each node having its own local storage; nodes in a compute rack sharing rack-local storage; or an I/O proxy node containing storage available to its proxied nodes [3]. In any case, a single configuration is unlikely to be the best fit for all possible applications. Therefore, applications will require flexibility if they wish to run efficiently across all future HPC designs.

A. Data Staging in HPC

Several I/O tasks require efficient use of all the available storage devices. One of the most demanding workloads for the file system is checkpointing. When a checkpoint is done in a coordinated fashion, the stress on the I/O storage is huge because of the large volumes of data that need to be stored reliably. This requirement can impose a large overhead on the application if checkpoints have to be taken at high frequency, which is what is expected at extreme scale because of the increase in the expected rate of faults. Moreover, as the failure rate grows with the component count, the I/O overhead grows with the global memory size. Thus, checkpoints can take several tens of minutes, and the checkpoint frequency is expected to be around one hour [4], leading to a scenario where the system spends more time writing data in the PFS than computing.

In order to relieve this pressure on the PFS, multilevel checkpointing has been proposed and implemented [5], [6], [7]. Multilevel checkpoints use local storage devices at high frequency and move checkpoint data to the file system at low frequency. This strategy can be combined in the compute nodes with data protection techniques such as data replication or erasure codes. For such techniques to work well, however, having a clear and accurate knowledge of the storage levels in the system is essential. Indeed, data replication and erasure codes will not work if the redundant data is stored in the same local device as is the original data. This need for data dispersion can be achieved in a portable fashion only if the runtime offers portable ways to query the storage topology of the machine.

In addition to checkpointing, HPC applications have become more complex as they have incorporated part of the data analysis in the application. The reason for this recent trend is the amount of time that is wasted moving data back and forth to the PFS. Thus, in situ analysis and visualization are becoming a major optimization strategy for extreme-scale applications. Clearly, data used for analysis and visualization has to be staged in some intermediary storage until the analysis/visualization process can consume it. This process again can be optimized if one possesses a good knowledge of the storage topology of the machine—hence the importance of such portable topology queries.

B. Striving for Portability

HPC users attempting to efficiently use all storage levels will need to know which processes share the same storage components. Furthermore, since many scientists run their applications on multiple machines and in many cases the same application is used by multiple scientists in different institutions around the world on different supercomputers, the discovery of a system's topology should be portable, Several portability efforts have been undertaken in the past. For instance, MPI-3 introduced the MPI_COMM_SPLIT_TYPE routine to split an MPI communicator based on colors (to partition the processes) and keys (to order the processes in a specific partition). One of the "types" available in MPI-3 is MPI_COMM_TYPE_SHARED, which offers the possibility of splitting a communicator in groups of processes that can share a memory region. Unfortunately, shared memory does not necessarily translate into groups of processes that share the same storage; other processes might not share any memory region but might share the same local storage.

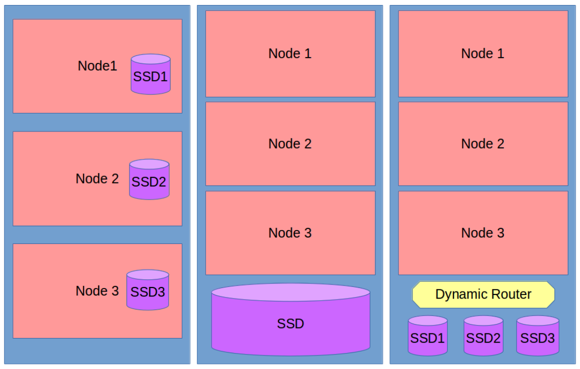

Fig. 2. Three storage topologies for extreme-scale systems. The first architecture has local storage in each compute node. The second one has local storage shared among multiple nodes (burst buffer). The third one has the same number of storage devices as nodes in a blade, but they are connected through a dynamic router in which any storage device can be mapped to any node.

Figure 2 shows three possible storage topologies. On the left, each node has its own SSD. All processes running on that node can access the SSD. Another option is depicted in the middle of the figure, showing intermediary storage located in a *per drawer* basis. All the nodes in a drawer share the same local storage, but they do not share any memory region. On the right is a system that puts multiple storage devices behind a router, which could be reconfigured dynamically after the failure of a node (but not of its previously designated storage device). These are just some examples of a much larger spectrum of storage topologies. Clearly needed are new MPI features that allow for a portable way to discover storage topologies in supercomputers.

III. Motivation and Design

A portable resource discovery feature can be implemented in multiple ways, each having different overhead-accuracy trade-offs. In this section we discuss several design options and possible extensions.

A. *Extending MPI*

The MPI-3 standard introduced a routine `MPI_COMM_SPLIT_TYPE` that provides additional ways to partition a given communicator. The older routine `MPI_COMM_SPLIT` also partitions a communicator but relies on the caller to provide the "color" (to facilitate grouping of processes) and "key" (to facilitate ordering within newly constructed groups) parameters. `COMM_SPLIT` works well for application-driven communicator creation. When using that routine to establish communicators based on some hardware feature, however, callers may not have a sufficiently detailed understanding of a machine's topology to correctly select the "color" parameter. `COMM_SPLIT_TYPE` provides a more machine-oriented way to create communicators.

```
MPI_Info_create(&info);
MPI_Info_set(info, "nbhd_common_dirname",
    "/path/to/NVRAM");
MPI_Comm_split_type(MPI_COMM_WORLD,
    MPIX_COMM_TYPE_NEIGHBORHOOD, 0, info, &comm);
```

Fig. 3. Example of the new flag for split_type. The info key "nbhd_common_dirname" holds as its value the location of a directory that may be available to one, some, or all processes.

The MPI standard already provides a parameter "split_type": `MPI_COMM_TYPE_SHARED` splits a communicator such that all processes in the new communicators have access to shared memory. We have introduced a new type, `MPIX_COMM_TYPE_NEIGHBORHOOD`, that splits a communicator based on certain hardware characteristics. These characteristics could be accessing a common file system, which we implemented, or sharing a common network topology feature. We use the "info" parameter, already part of the standard routine, to provide additional information for the library to make its neighborhood-splitting decisions. Figure 3 demonstrates the way one might call this routine in C.

In this work we have implemented and evaluated the info key "nbhd_common_dirname." An MPI implementation will determine whether the directory associated with this key is common to one or more MPI processes. The implementation will then set up one or more MPI communicators consisting of processes that share the given directory. Implementations could define behavior for other info keys. We propose that an MPI implementation or platform that does not support a specific info key will return an error and set up new communicators as MPI_COMM_NULL. An alternative approach for handling unknown info keys or not providing any info key, which we imagine would be evaluated as part of the standardization process, could have the routine ignore any unknown info key and set up single processes communicators for all callers.

Using the info key to pass in additional parameters has several benefits. An MPI info object associates string-based keys with string-based values. The implementation is free to ignore any key it does not understand. An MPI implementation can introduce new keys without requiring any header file changes or introducing any binary incompatibility. The info key also provides an opportunity for a deployment to provide site-specific tuning. In ROMIO, a config file can provide a set of default hints suitable for a given environment. One can imagine using such a mechanism to further simplify the

```
if (rank == 0)
    construct unique file name
MPI_Bcast(testdirname, PATH_MAX, MPI_BYTE, 0, comm);
ret = mkdir(testdirname, S_IRWXU);
if (ret == -1 && errno != EEXIST) goto fn_fail;
open(filename,O_CREAT,S_IRUSR|S_IWUSR);

MPI_Barrier(comm);
/* each process has created a file in a M-way shared
 * directory (where M in  the range [1-nprocs] ).
 * now see else can see this directory */
if ((dir = opendir(testdirname)) == NULL)
    goto fn_fail;
while ( (entry = readdir(dir)) != NULL) {
    if (strcmp(entry->d_name, ".") == 0) continue;
    if (strcmp(entry->d_name, "..") == 0) continue;
    ranks[j++] = atoi(entry->d_name);
}

MPI_Comm_group(comm, &comm_group);
MPI_Group_incl(comm_group, j, ranks, &newgroup);
MPI_Comm_create(comm, newgroup, newcomm);
MPI_Group_free(&newgroup);
MPI_Group_free(&comm_group);
```

Fig. 4. Code for the exhaustive storage topology algorithm (omitting, e.g., error checking and string manipulation).

resource-locating step.

1) Aggressive storage probing: For most applications, splitting communicators happens outside the critical path: an application will set up a communicator and then commence communication. We can therefore probe the underlying hardware topology fairly aggressively in order to discover how processes are placed on the machine, without too much concern for how such probing will disturb the calling application. We suggest two approaches: one slow and thorough, the other a fast approach that might misreport some configurations but stresses the underlying storage far less.

We provide in Figure 4 a stripped-down C code showing the essence of the slow algorithm to determine which processes share a particular file system. The algorithm starts with every MPI process creating a file corresponding to its MPI rank in an agreed-upon (e.g., via info key) test directory. Next, each process gets a listing of files in the output directory. If, for example, the directory contains the files "1," "10," "1024," and "52345," those ranks are the ones to which this directory in question is common. We can then create new MPI groups consisting of ranks sharing access to the directory and can immediately promote those groups to communicators.

Arguably, this approach poses significant challenges with large numbers of MPI processes. Should the target directory reside on the globally visible parallel file system, creating one file per process could pose a metadata strain on the PFS. It would be able, however, to accurately report the topology of hierarchical storage systems where the common storage is shared at the rack or service-node level. Another drawback to this approach is the lack of metadata operations in the MPI standard. Our current implementation relies on POSIX creation and directory reading routines. While many storage devices today do provide a POSIX API, some architectures are likely to provide non-POSIX interfaces to storage devices (e.g., keyval or object-based).

```
    MPIR_Get_node_id(comm, rank, &id);

    /* - Create file on one processor
     * - pick a processor outside the "on this node" group
     * - if that processor can see the file, then assume the
     *   file is visible to all groups.
     */

    if (rank == 0) {
            /* omitted: select 'challenge_rank' of process
             * not on this node */
    }
    mpi_errno = MPI_Bcast(&challenge_rank, 1, MPI_INT, 0, comm);

    /* now start poking file system:*/

    /* Use a single short message to force check after
     * create: ordering is a little odd in case we are
     * creating and checking on the same rank  */

    if (rank == challenge_rank) {
        MPI_Irecv(NULL, 0, MPI_BYTE, 0, 0, comm, &check_req);
    }

    if (rank == 0) {
        mpi_errno = MPI_File_open(MPI_COMM_SELF, filename,
                        MPI_MODE_CREATE | MPI_MODE_EXCL |
                        MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
        MPI_File_close(&fh);
        /* the check for file has to happen after file
         * created. only need one process, though, not a
         * full barrier */
        MPI_Send(NULL, 0, MPI_BYTE, challenge_rank, 0, comm);
    }

    if (rank == challenge_rank) {
        MPI_Wait(&check_req, MPI_STATUS_IGNORE);

        /* too bad there's no ADIO equivalent of access: we'll
         * have to open/close the file instead */

        mpi_errno = MPI_File_open(MPI_COMM_SELF, filename,
                        MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
        if (mpi_errno == MPI_SUCCESS) {
            globally_visible = 1;
            MPI_File_close(&fh);
        } else {
            /* do not report error up to caller.
             * merely testing the presence of the file */
            mpi_errno = MPI_SUCCESS;
            globally_visible = 0;
        }
    }
    MPI_Bcast(&globally_visible, 1, MPI_INT,
        challenge_rank, comm);

    if (globally_visible) {
        MPI_Comm_dup(comm, newcomm);
    }
    else {
        MPI_Comm_split(comm, id, key, newcomm);
    }
}
```

Fig. 5. Heuristic for detecting common topologies quickly.

2) A more scalable heuristic: We provide a lower-overhead heuristic in Figure 5. After grouping MPI processes by node, one process on one group creates the test file. We then select a different group. If a "challenge rank" process from that group can see the file created by the first group, we assume that all processes from all groups can see that file (e.g., the common directory is a parallel file system available to all processes). If not, we assume that the file is only node-visible (e.g., an NVRAM block device mounted somewhere on that node). We have the "checking" process wait on a zero-byte send from the "create" process to prevent the "checking" process from racing ahead of the "create" process. We have omitted error handling in this code fragment, but the implementation in MPICH does report errors back to the caller.

This heuristic will detect common "node-local" storage topologies but will not generate an ideal collection of communicators over topologies where storage is local to a subset of nodes. For example, if a burst buffer resides at an intermediate node shared with a specific rack of the machine, this heuristic would flag the topology as "not globally visible": the resulting communicators would span one node, not all processes in the rack.

While not foolproof, this heuristic matches the common use cases and requires only two file system operations (creation and access), both of which we implement with MPI-IO routines. Avoiding direct calls to POSIX helps with portability: in the case of non-POSIX storage interfaces, the MPI-IO implementation might have specialized routines as in ROMIO's ADIO layer [8].

B. Use Cases

The linear growth of file systems cannot always cope with the exponential growth of supercomputers and sometimes becomes a bottleneck. This situation has led to a series of strategies specifically designed to avoid data transfers to the file system. All these techniques rely on local storage to alleviate the stress on the file system and reduce overhead. The arrival of new storage technologies, such as NVRAM and 3DRAM, offers a promising landscape for the development and enhancement of those techniques that leverage local storage efficiently,

A clear example of such techniques is multilevel checkpointing, which consists of checkpointing in local storage at high frequency and performing file system checkpoints at much lower frequency. Local checkpointing is useful for tolerating soft errors, but it cannot recover from node crashes. Therefore, local checkpointing is usually accompanied by either erasure codes or checkpoint replication. These tasks can be offloaded to dedicated service processes that have access to the checkpoint files written by the application processes. Here is where a portable way to find shared storages is of great benefit because it allows users to construct groups where some (or most) processes produce data and write it in a local storage, and the other processes consume it. With multilevel checkpointing, the fault-tolerant dedicated processes read the local checkpoint data to encode it or replicate it, in order to tolerate node crashes.

Another use case that is gaining popularity as we approach exascale is the implementation of workflows. Scientific simulations executed in supercomputers produce a vast amount of data that needs to be processed for analysis and/or visualization. Traditionally such a task was done on one or just a few nodes, but doing so has become increasingly difficult as the size of the postprocessing data keeps growing. Therefore, in situ postprocessing has become the rule rather than the exception. Several workflow frameworks have been proposed recently [9], [10] to take advantage of local storage and extra computing resources. Several workflow frameworks split the global MPI communicator so that some processes participate in the simulation and others in the postprocessing. The postprocessing processes should have access to the local storage where the simulation data has been stored. MPIX_COMM_TYPE_NEIGHBORHOOD offers a portable way of implementing such partitioning.

We note that in some architectures, local storage is not equivalent to intranode storage. In fact, multiple nodes (e.g., all nodes in a blade) could share some common local storage. In such a case even if some node in the blade goes down, other nodes could potentially access the local checkpoint files written by the failed node. Storage shared among nodes opens new possibilities such as taking local checkpoints that can tolerate node crashes without the need for erasure codes or checkpoint replication. A third use case involves generalizing the checkpointing problem to all I/O. Earlier work has looked at log-oriented approaches for write-intensive workloads [11], [12]. One could extend these approaches to use any topology information provided by the MPI implementation. These logging approaches have the problem of generating one file per MPI process. Grouping processes into subsets sharing a common storage device presents a natural way to reduce the number of logs generated.

IV. Experimental Evaluation

We demonstrate the benefit of these topology query extensions with a microbenchmark and two general cases. In one, we use topology information to produce node-local checkpoints. In another, we use topology information to select an efficient resource for inter-process data exchange.

A. Basic Performance

We first performed a simple evaluation to quantify the cost and scalability of our topology discovery. This benchmark measures the time to call MPI_COMM_SPLIT_TYPE in three cases: provide no hint information, interrogate the provided directory exhaustively, and interrogate the provided directory with our heuristic. If a name is not given, the routine returns a NULL communicator. The difference in times between the two configurations represents the overhead of topology discovery. In common use, communicator creation is a one-time setup cost and is unlikely to be critical to performance, but quantifying the cost of our approach is still important.

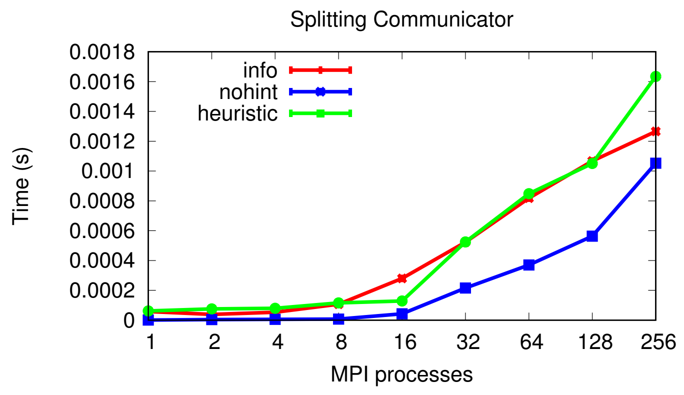

Fig. 6. Cost associated with topology discovery. Lower is better. For comparison, the "nohint" case does no topology discovery and returns an empty communicator.

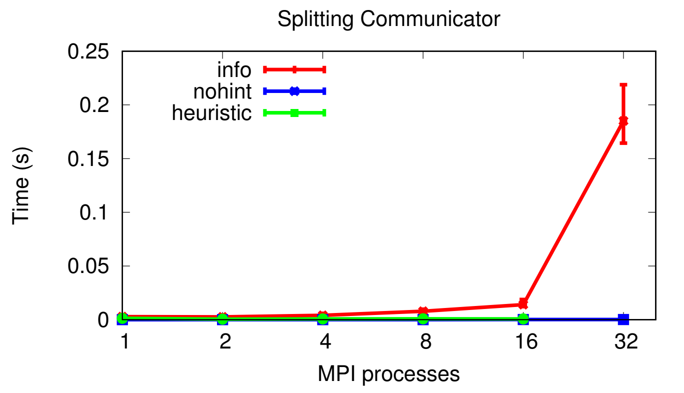

Fig. 7. Massive overhead incurred by aggressive algorithm when applied to a GPFS file system; compare the heuristic. Observe the y axis is several orders of magnitude larger in this case.

We used the Blues machine, hosted at Argonnne's Laboratory Computing Resource Center. Blues is a Linux cluster whose "batch" queue contains nodes with 16 Sandy Bridge Xeon E5-2670 2.6 GHz processors. The compute nodes access a GPFS file system and have no local storage aside from a ramdisk. In these experiments we treat the ramdisk on `/scratch` as a stand-in for NVRAM or SSD storage.

In the microbenchmark we ran each configuration 100 times and report the median with 95% confidence intervals. Figure 6 shows the performance difference between the heuristic and exhaustive approaches. In this experiment only 16 MPI processes per node are interrogating a node-local file system, limiting the overhead.

The evaluation looks much different when we apply these algorithms to a shared parallel file system. To answer just how high the overhead of the exhaustive algorithm can be, we applied our evaluation to the GPFS parallel file system shared across all of the Blues cluster. GPFS is tuned for large parallel data transfers, but like virtually all parallel file systems it is not well suited to a storm of metadata operations. The results in Figure 7 show that even at modest scale the heuristic approach, with its small number of storage operations, performs far better than the exhaustive approach.

B. IOR Benchmark

Our next experiment aims to demonstrate that using this MPI extension, one can tap the full potential of multiple storage levels even in unconventional architectures. This test was done on a cluster with over 900 system-on-chip nodes. Each node has two cores based on ARM technology running at 1.7 GHz and 4 GB of ECC-less low-power DRAM. The nodes are connected together by an InfiniBand network and to a Lustre file system. The nodes also have a local microSD storage that can be used as a `scratch` directory. While microSD storage is not common in high-end HPC platforms, the proposed topology discovery feature discovers which processes have access to the same storage resources, regardless of the underlying technology. In this cluster one has no direct way of knowing which MPI processes share the same microSD storage; thus one would need to implement a storage discovery scheme manually that is unlikely to be portable to other systems.

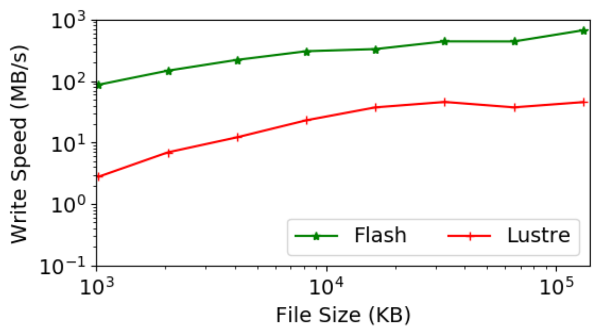

Fig. 8. Write speed measured with IOR for different file sizes from 1 MB to 128 MB. Both x and y axes use logarithmic scales.

We evaluated the write performance of both storage levels, local and global. We used IOR [13], a benchmark specially designed to accurately measure the speed of parallel file systems and other storage devices in HPC systems. We focus on the write speed because this is what will dominate the time to completion of applications that use the PFS at high frequency. In particular, long executions writing checkpoint files in the PFS.

Figure 8 shows the write speed on both storage levels for different file sizes, going from 1 MB to 128 MB. We observe that for all file sizes writing in the local flash storage is about one order of magnitude faster than writing in the file system. This result is expected because it is well known that sharing resources such as the PFS leads to congestion while moving large volumes of data.

Next, we set up an I/O benchmark that performs a series of write operations and some partial computation. The execution time is largely dominated by the I/O time. We set up two scenarios. In the first scenario, I/O operations were performed on the Lustre PFS. In the second scenario, we split the

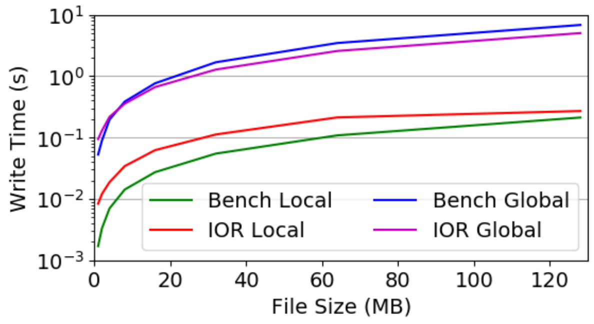

Fig. 9. Execution time of IOR vs our synthetic benchmark for different file sizes. The y axis uses a logarithmic scale.

communicator based on processes with access to the same `/scratch` directory, and we performed the writes in the shared storage. We compare the performance of our synthetic benchmark with the performance of the IOR benchmark to demonstrate the relevance of the IOR results and how they relate to more complex benchmarks.

Figure 9 shows that both the synthetic benchmark and IOR behave similarly. The execution time changes by over an order of magnitude from the first scenario to the second one that leverages the local storage while leveraging the intermediary storage levels through COMM_SPLIT_TYPE. As expected, the times are not identical since the I/O benchmark performs slightly more computing work. Nonetheless, they show the same behavior because they are both limited by the I/O bandwidth of the corresponding storage space. While the benefits of local vs global storage are known, what is important to note is that the code using COMM_SPLIT_TYPE is completely portable and could be executed in a system without any intermediary storage. This feature opens the possibility for users to have one single master code that can be executed in different architectures and leverage intermediary storages when available.

C. I/O Kernel Benchmarks

Checkpointing is one of the most I/O-intensive operations for HPC applications. Thus it is a good example to show how the proposed MPI extension can help us leverage multiple storage levels efficiently and portably. To demonstrate this benefit of topology awareness, we consider an application running on a Linux cluster where each compute node has local storage and also has access to a fast parallel file system. Instead of checkpointing to a shared canonical file on the globally accessible parallel file system, we instead checkpoint locally. Creating one local file per node avoids the parallel file system's locking and coordination overhead and generates fewer files than does a model with one file per MPI process.

The HACC-IO kernel[1] mimics the I/O behavior of the HACC cosmology simulation code [14]. The benchmark does no simulation itself but merely writes and reads 9 variables associated with N particles.

[1]http://git.mcs.anl.gov/HACC-IO.git/

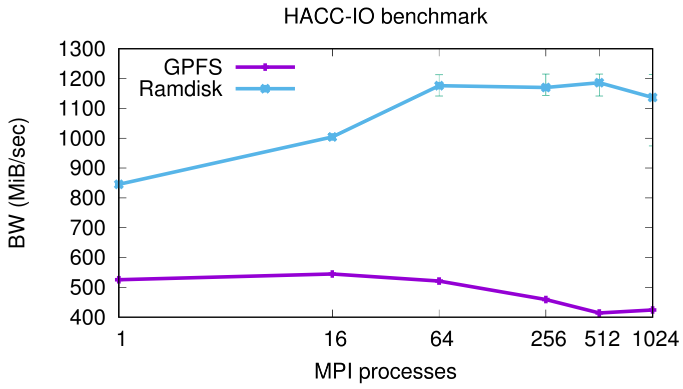

Fig. 10. After the benchmark acquires a local-storage communicator, it uses this communicator to collectively write checkpoints locally. By avoiding the overhead of the parallel file system, these local checkpoints complete quickly.

As with the microbenchmark earlier, we ran this experiment on the Blues cluster. We configured MPICH to use the `libfabric` driver and used the PSM provider for libfabric.

In this test we compared two approaches for saving and restoring a checkpoint. The traditional way performs collective I/O to a shared file on the GPFS file system. In this approach every process opens the checkpoint file with `COMM_WORLD` for a communicator. The "topology-aware" way splits the WORLD communicator based on which processes have access to the `/scratch` ramdisk. These sub communicators then each collectively write a checkpoint to the common-to-them `/scratch` file system.

As shown in Figure 10, checkpointing locally provides performance benefits. GPFS provides several consistency and coordination features that MPI-IO does not require. In this weak-scaling benchmark in which each process writes out 4,023,224 particles, processes must obtain file system write locks. Even with MPI-IO optimizations, HACC-IO incurs several overheads when writing to the parallel file system. By grouping processes into those that can write to the local ramdisk, we can greatly improve performance. While this approach has been known for a long time, most current solutions to leverage local storage must use architecture-specific hooks to discover the storage topology. Solutions such as *gethostbyname* are not portable, and not all machines have the same storage topology.

The parallel file system does provide several advantages. The checkpoint file ends up in a canonical form suitable, for example, for visualization or other workflow components. Additionally, the parallel file system provides some measure of durability. We have not investigated whether approaches to stage out and stage in these fragmented checkpoints are slower than the time gained by checkpointing locally.

Next, we fixed the number of MPI processes and increased the amount of data for each process. The data checkpointed is then analyzed by another process to detect trends and anomalies. This is a classic example of scientific workflows including data analytics and data visualization. Note that

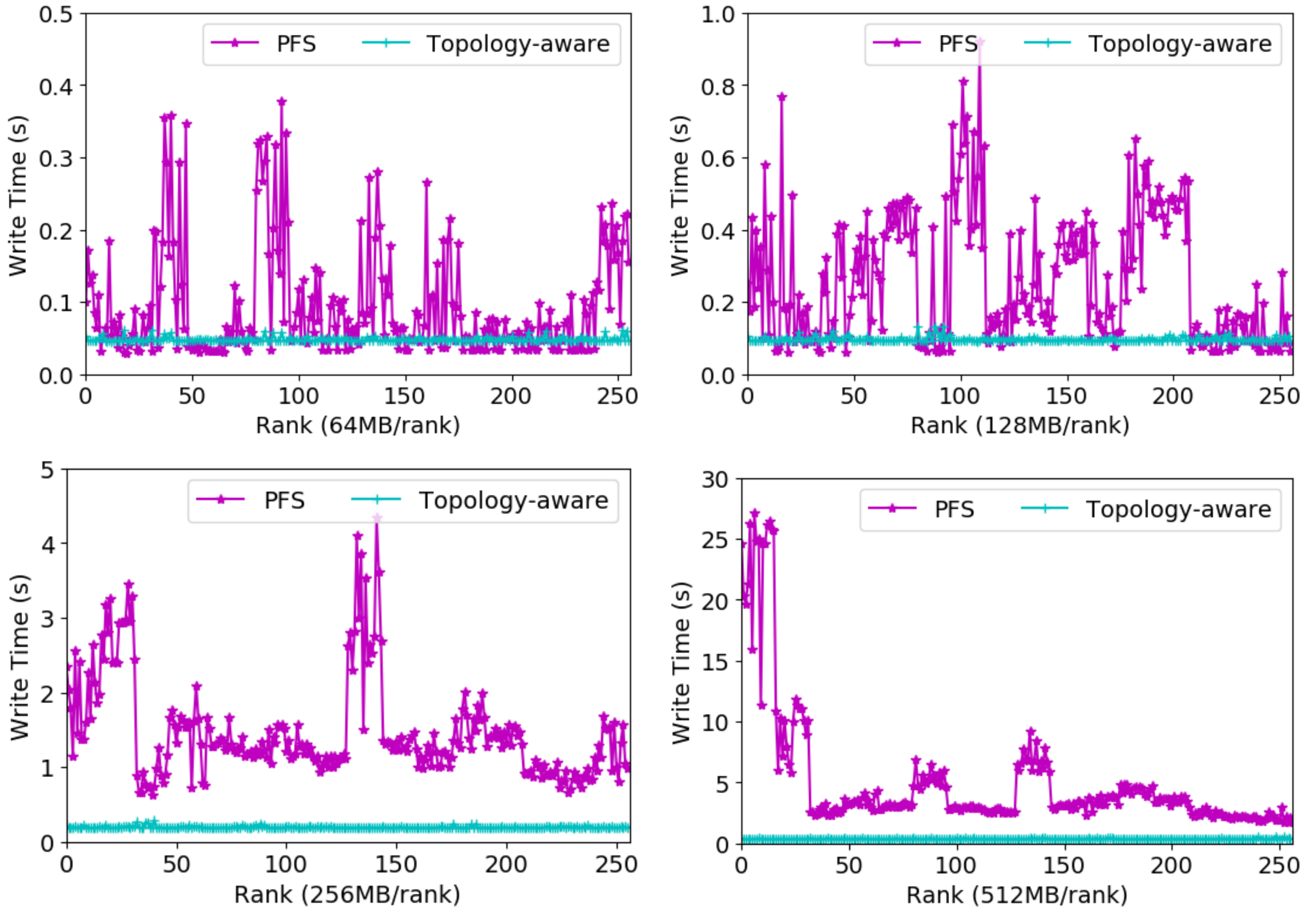

Fig. 11. Storage-topology-aware checkpointing vs global checkpointing.

reading the checkpoint data during runtime for data analysis increases the topology awareness requirements since the MPI process reading the data needs to have access to the same storage as do the processes producing the data. In other words, writing to local storage regardless of who else has access to it is not enough because there could be analytic processes without access to any checkpoint file, or checkpoint files that are not analyzed by any analytics process. Again, we set up two scenarios: one in which the data is written on the PFS and a second scenario in which the data is written in a shared local storage using the topology-aware MPI extension proposed. We measured the writing time for each rank in an execution with 256 MPI ranks. We performed this measurement for four data sizes, from 64 MB per rank to 512 MB per rank.

TABLE I
MEAN WRITING TIME ACROSS ALL RANKS

Ckpt. Size per Rank (MB)	PFS (s)	Topology-Aware (s)
64	0.094	0.048
128	0.262	0.096
256	1.474	0.193
512	5.053	0.385

The results are depicted in Figure 11. As we can see, writing in a global storage is several orders of magnitude more expensive than leveraging the intermediate storage levels through the topology-aware construct. Note that the y axis changes substantially as the size of the checkpoints increases. In fact, while writing 512 MB in the PFS, some processes take more than 25 seconds; in contrast, using the topology-aware feature of MPI, they can write two orders of magnitude faster. The reason is that the PFS is receiving a total of 128 GB of data whereas the local storage needs to handle only 8 GB of data. Table I shows the mean writing time across all MPI ranks for PFS-based checkpointing compared with using topology-aware storage. As we move toward exascale, with thousands of processes writing data for different purposes (e.g., checkpointing, analytics), these topology-aware techniques will become more critical to efficiently using those machines.

D. Dynamic Data Exchange I/O Benchmark

To further stress our experiments, we evaluated another I/O benchmark to simulate a much more dynamic scenario. Instead of writing always in local or global storage, we tested a synthetic benchmark in which all MPI processes exchange large volumes of data between them in a completely dynamic fashion. This case is more complex than simply reading data from the same processes during the entire execution. In the new setup, processes must decide whether they write the data

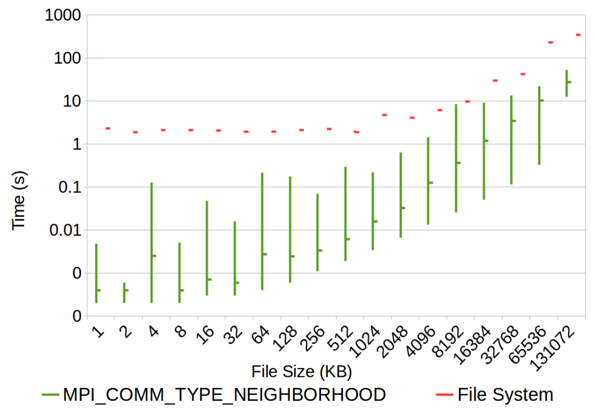

Fig. 12. Using a file system to exchange data might make sense in coupled codes or workflow situations. Exchanging data through a shared local file system is dramatically faster than doing so through the global parallel file system.

locally or globally depending on which process must *consume* the produced data. If the process consuming the data has access to the same storage as the process producing the data, then the data is written locally; otherwise it is written in the global PFS. The second scenario assumes that no topology-aware option exists; thus all processes have to perform all the writes in the global PFS. All writes correspond to chunks of data of the same size during the entire execution. We increase the block size for multiple executions. This workload is a much more dynamic behaviour that cannot be handled simply by the classic approach of globally writing in local SSD (as for multilevel checkpointing); instead, some blocks of data can be written locally while other have to be written globally, so that the consumer task can access those blocks.

Figure 12 shows the results of this evaluation for different block sizes. For each point, we plot the minimum, maximum, and mean of an execution with 256 processes producing data at the same rate. For the largest files size (i.e., 128 MB) the aggregate of all processes data reaches about 32 GB.

As we can observe, COMM_TYPE_NEIGHBORHOOD allows us to leverage all the hardware resources available, leading to orders of magnitude better performance. This difference in performance becomes critical for large file sizes, where leveraging the common local resources takes less than half a minute, whereas writing to the parallel file system takes over 5 minutes. This difference is due to the fact that hardware resources are used differently by the two cases, and this is directly related to the fact that such hardware is available when a portable way to discover it exists.

.

V. Related Work

Several attempts have been made to introduce topology discovery inside the MPI standard. For instance, an MPI tool for discovering switch-level topologies in Ethernet clusters was proposed several years ago [15]; the work relied on the Simple Network Management Protocol to obtain topology information. Another attempt to discover network topology in HPC systems was the netloc project [16]. This project proposed a modular approach in order to support multiple network types and discovery methodologies; it also proposed representing the topology as a graph in order to support any network topology configuration. However, none of these works investigated the discovery of the storage topology—an important limitation since the storage hierarchy is getting deeper.

The OpenMPI team have also extended SPLIT_TYPE but taken a different approach. They extended the `split_type` parameter to allow other hardware options [17]. While such an approach is valid, we feel that it limits portability. MPI info objects have the helpful property that an implementation is free to ignore unknown or unrecognized keys. Extending SPLIT_TYPE via Info objects will maintain binary compatibility when an implementation provides a new topology discovery option. Furthermore, the topology detection routines provided by OpenMPI cover only CPU and network topology and do not address storage hierarchy.

Given the current popularity of new storage technologies, a large number of groups have been studying the most efficient ways of utilizing those. Some studies focus on how to take advantage of such technologies, in particular given that DRAM main memory accounts for about 40% of the nodes' power consumption [18]. Other research [19] has focused on improving I/O performance by using topology-aware data placement, even under congestion scenarios. While these works explore the most efficient ways of leveraging additional storage levels in HPC systems, they do not address the issue of topology discovery and portability. Other approaches, such as TreeMatch [20], have proposed techniques to map processes according to different constraints such as communication density in NUMA nodes, but they do not provide a portable way to discover storage topologies.

Large HPC laboratories across the world have expressed interest in integrating burst buffers in the compute nodes of their next-generation machines. Indeed, many researchers have analyzed the impact of such buffers in future supercomputers. The motivation is always closely related to the ideas discussed in Section II. Studies about the best scheduling approach based on I/O contention in burst-buffer-enabled machines [21] and the role of burst buffers in extreme-scale storage systems [1] are just a few examples of the work being done in this domain. Such work has focused on the benefits of new storage levels but has not addressed application-centric concerns such as portable topology discovery and the respective scalability challenges.

VI. Concluding Remarks

We have presented an extension to MPI-3 providing a portable approach for investigating the topology of a compute system. In a storage context this extension allows a program to determine which nodes share faster local devices. We have implemented this extension in MPICH, and we encourage readers to experiment with our proposed extension and report good or bad results.

We expect this extension to serve as the foundation for more sophisticated data management libraries. Several projects have addressed underwhelming parallel I/O performance by creating one file per MPI process. Splitting the global communicator into groups sharing a common local file system provides a natural way to reduce the number of files produced and can provide higher performance.

With topology discovery in place, we can consider a multilevel synchronization strategy. The standard FILE_SYNC routine pushes data from the client to permanent storage. With a portable way to explore topology, a third layer between local memory and permanent global storage becomes an option. A FILE_SYNC_LOCAL routine would take local memory and move it to persistent local storage. Later, either at program exit or with an explicit FILE_SYNC call, the library would take the local storage and transfer it to global storage.

Acknowledgments

We gratefully acknowledge the computing resources provided on Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the Role of Burst Buffers in Leadership-Class Storage Systems," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012, pp. 1–11.

[2] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, ser. FAST '07. Berkeley, CA, USA: USENIX Association, 2007, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267903.1267905

[3] Cray, "Datawarp user guide s-2558-5204," 2016.

[4] B. Schroeder and G. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Stems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[5] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 32:1–32:32. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063427

[6] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[7] J. S. Plank, K. Li, and M. A. Puening, "Diskless Checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[8] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-I/O interfaces," in *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, October 1996, pp. 180–187. [Online]. Available: http://www.mcs.anl.gov/ thakur/papers/adio.ps

[9] K. Maheshwari, J. M. Wozniak, H. Yang, D. S. Katz, M. Ripeanu, V. Zavala, and M. Wilde, "Evaluating storage systems for scientific data in the cloud," in *Proceedings of the 5th ACM Workshop on Scientific Cloud Computing*, ser. ScienceCloud '14. New York, NY, USA: ACM, 2014, pp. 33–40. [Online]. Available: http://doi.acm.org/10.1145/2608029.2608034

[10] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling Spark on HPC systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 97–110. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907310

[11] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of SC09*, Nov. 2009.

[12] D. Kimpe, R. Ross, S. Vandewalle, and S. Poedts, "Transparent log-based data storage in MPI-IO applications," in *Proceedings of the 14th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2007)*, Sep. 2007.

[13] "Interleaved or Random Benchmark," https://github.com/IOR-LANL/ior.

[14] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel, N. Frontiere, and Z. Lukić, "The universe at extreme scale: Multi-petaflop sky simulation on the BG/Q," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 4:1–4:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389002

[15] J. Lawrence and X. Yuan, "An MPI tool for Automatically Discovering the Switch Level Topologies of Ethernet clusters," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[16] B. Goglin, J. Hursey, and J. M. Squyres, "netloc: Towards a comprehensive view of the HPC system topology," in *2014 43rd International Conference on Parallel Processing Workshops*. IEEE, 2014, pp. 216–225.

[17] "Pull request, enabled COMM_TYPE_SPLIT dependent on locality," https://github.com/open-mpi/ompi/pull/326, accessed: 2016-05-09.

[18] J. S. Vetter and S. Mittal, "Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing," *Computing in Science & Engineering*, vol. 17, no. 2, pp. 73–82, 2015.

[19] F. Wang, S. Oral, S. Gupta, D. Tiwari, and S. S. Vazhkudai, "Improving Large-Scale Storage System Performance Via Topology-Aware and Balanced Data Placement," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2014, pp. 656–663.

[20] E. Jeannot, E. Meneses, G. Mercier, F. Tessier, and G. Zheng, "Communication and Topology-Aware Load Balancing in Charm++ with TreeMatch," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–8.

[21] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, "Scalable I/O-aware job scheduling for burst buffer enabled HPC clusters," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 69–80.