On the Road to DiPOSH: Adventures in High-Performance OpenSHMEM

Camille Coti[1](✉) and Allen D. Malony[2]

[1] LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité, Villetaneuse, France
camille.coti@lipn.univ-paris13.fr
[2] University of Oregon, Eugene, USA
malony@cs.uoregon.edu

Abstract. Future HPC programming systems must address the challenge of how to integrate shared and distributed memory parallelism. The growing number of server cores argues in favor of shared memory multithreading at the node level, but makes interfacing with distributed communication libraries more problematic. Alternatively, implementing rich message passing libraries to run across codes can be cumbersome and inefficient. The paper describes an attempt to address the challenge with OpenSHMEM, where a lean API makes for a high-performance shared memory operation and communication semantics maps directly to fast networking hardware. DiPOSH is our initial attempt to implement OpenSHMEM with these objectives. Starting with our node-level POSH design, we leveraged MPI one-sided support to get initial internode functionality. The paper reports our progress. To our pleasant surprise, we discovered a natural and compatible integration of OpenSHMEM and MPI, in contrast to what is found in MPI+X hybrids today.

Keywords: OpenSHMEM · Distributed run-time system · One-sided communication

1 Introduction

The trend of increasing core counts of shared memory servers that make up the nodes of scalable high-performance computing (HPC) systems raise questions of how parallel applications should be programmed in the future. Distributed programming models based on message passing are effective for internode parallelism, but their runtime implementation can be less efficient for intranode parallelism. This put pressure on these programming paradigms to be used in hybrid forms. For instance, while *MPI everywhere* programs are perfectly reasonable for programming HPC machines, concerns for node-level performance argues for an *MPI+X* approach, where *X* is a shared memory programming methodology of choice. In doing so, conflicts can arise between the MPI and X runtime support, especially with respect to managing higher degrees of node-level parallelism.

© Springer Nature Switzerland AG 2020
R. Wyrzykowski et al. (Eds.): PPAM 2019, LNCS 12043, pp. 250–260, 2020.
https://doi.org/10.1007/978-3-030-43229-4_22

Alternatively, shared memory programming models are effective for intranode parallelism, but must be adapted to maintain a shared memory abstraction on distributed memories. Advances in low-latency, RDMA communication hardware make it possible to support (partitioned) global address space (P)GAS semantics with high-efficiency data transfer between nodes. For instance, the SHMEM interface and its OpenSHMEM standardization embody peer-to-peer one-sided *put* and *get* operations on distributed "shared" memory. While the abstraction is more compatible with shared memory programming, it was created originally from a perspective of internode interaction.

In this paper we consider the viability of OpenSHMEM as a unified parallel programming model for both intranode and internode parallelism. Our starting point is Coti's high-performance OpenSHMEM implementation for shared memory system called *Paris OpenSHMEM* (POSH for short) [6]. The goal of POSH is to deliver an OpenSHMEM implementation on a shared memory system that is both fast and lightweight as possible. Now we look to extend POSH for distributed HPC systems. The challenge in creating distributed POSH (DiPOSH for short) is first to support the OpenSHMEM API and second to optimize performance. For this reason, we take the strategy of layering DiPOSH on MPI one-side communication. This will give us a baseline to evaluate future enhancements. More importantly, it will expose any critical factors at the nexus between intranode and internode operation.

2 Related Works

Parallel programming approaches for evolving HPC systems must address the challenges of greater intranode concurrency, while connecting to powerful internode communication infrastructure. The MPI interface has dominated the message passing paradigm with important high-performance implementations available, including OpenMPI, MPICH, MVAPICH. However, MPI's generic semantics makes it more complex to implement for distributed communication and unnecessarily complicated at the node level. Efforts to integrate multithreaded shared memory programming (e.g., OpenMP) with MPI can suffer from mismatches in the runtime systems.

In contrast, the OpenSHMEM interface is very simple and straightforward to implement. Its remote memory access semantics is equally natural for targeting intranode and internode parallelism. In fact, MPI's one-sided communication support [8] is all that we needed to develop DiPOSH's remote functionality. This approach is attractive, because it takes advantage of already existing communication routines, along with the rest of MPI's infrastructure, such as optimized collective communications. However, it involves a thick software stack and required some additional synchronization to implement OpenSHMEM's communication model.

Certainly, there are several other (P)GAS programming systems actively being pursued, such as CoArray Fortran, Chapel, UPC/UPC++, Global Arrays, and HPX. As a library, DiPOSH target a lower-level OpenSHMEM API, increasing its portability and interoperability with other software.

3 Architecture

The core idea behind POSH is to implement the OpenSHMEM communication interface with a minimal API-to-network software stack, thereby minimizing the software overhead. As shown in [11,12], traversing the software stack can have a significant overhead on performance-critical communication networks. However, in order to be portable across parallel machines, the communication library needs to be able to use several types of networks. In this section, we are describing how DiPOSH handles the different types of communication channels.

3.1 Shared Heap

In the OpenSHMEM memory model, the memory of each process is made of two parts: its private memory, which only it can access, and a shared heap, that can be accessed in read and write mode by all the other processes. In POSH, this shared heap is implemented by a segment of shared memory. Each process on a node owns a segment of shared memory, which is accessed by all the other processes on the node, enabling straightforward communications locally.

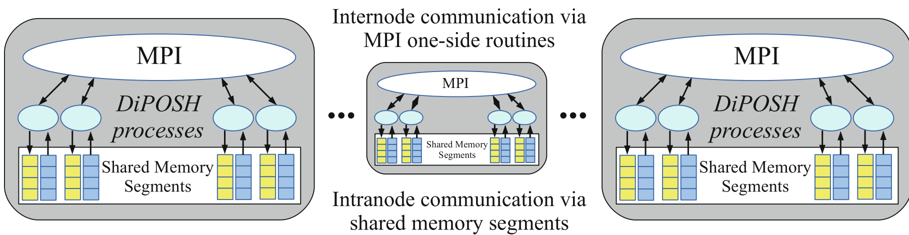


Fig. 1. High-level DiPOSH design illustrating intranode communication through shared segments and internode communication via MPI.

In DiPOSH, communications are also performed using this segment of shared memory: the remote memory access routines read and write data from and into the remote process's heap. Since the shared heaps are *symmetric* (*i.e.,* processes allocate the same space on their own heap), memory locations can be addressed within this segment of shared memory by using their offset from the beginning of the segment.

An example is presented in Fig. 1. Processes within a given node can communicate with each other using these segments of shared memory and processes located on different nodes use another communication channel, for instance, MPI one-sided routines.

In the particular case of MPI's one-sided communications, these routines use a *window* to perform one-sided communications. When the application is initialized, each process creates a window and associates the beginning of the segment

of shared memory as its base address. Therefore, one-sided communications handle directly data in the shared heap, making both communication channels (local and MPI) compatible. Moreover, since, unlike OpenSHMEM, MPI's one-sided communications are asynchronous and non-blocking, completion of the communication is ensured by lock and unlock operations on the window.

3.2 Network Portability

Using the appropriate communication channel to reach another process is a highly critical point of the design of DiPOSH. Indeed, choosing the right function and calling it is in the critical path of the software stack and therefore, needs to be handled carefully in the aim of minimizing the software overhead of the communication library.

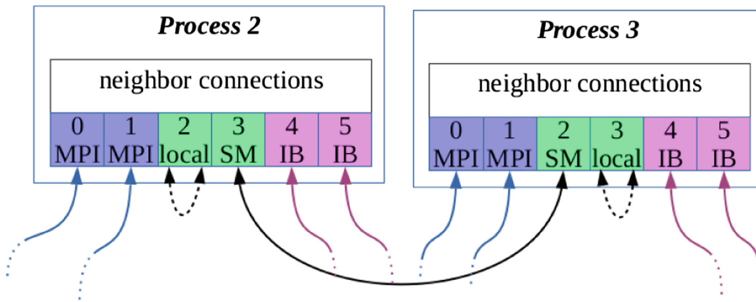


Fig. 2. Data structure handling how processes select the communication channel to be used with each other process. In the table of neighbors, for each other process there is a set of function pointers pointing to the appropriate communication routines.

We chose to make this decision once, when the communication library is initialized. All the processes exchange their contact information in an *allgather-like* collective operation. Then, each process knows how it can reach every other process and determine which communication channel it will use when they need to communicate with the other process.

Each process maintains a data structure that keeps information on their *neighbors* (*i.e.,* the other processes). This data structure also contains function pointers to the communication routines that correspond to the communication channel that will be used to communicate with this process. This organization is represented in Fig. 2.

It was measured in [1] that calling a function from a function pointer has an acceptable cost, compared to other call implementations (close to a direct call). Moreover, data structure requires about 1 kB per neighbor on a 64 bit architecture, which is not significant regarding the memory footprint of parallel scientific applications, even at large scale.

3.3 Cohabitation with Other Models

As mentioned earlier in this paper, combinations of programming models is an attractive solution to program extreme-scale machines. Therefore, it is necessary for parallel execution environment to be compatible with each other. The OpenSHMEM specification v1.4 mentions a few examples of OpenSHMEM and MPI compatibility in some implementations (annex D).

DiPOSH is *fully compatible* with MPI. In its current implementation, its run-time environment is written in MPI, which means that the coordination between the OpenSHMEM processing units (*e.g.*, communication of their contact information) is made using MPI calls. As a consequence, when the OpenSHMEM application is initialized by `start_pes()`, the OpenSHMEM library calls `MPI_Init`. In future versions, DiPOSH might use another run-time environment in order to be independent of MPI and avoid requiring having an MPI implementation on the system. However, MPI is common enough to make having it installed on a parallel machine a very weak assumption.

An excerpt of a program using both MPI and OpenSHMEM and supported by DiPOSH is given in Fig. 3. We can see that MPI and OpenSHMEM communication routines can be mixed in the program, for the programmer to use whichever paradigm fits better its need for each communication.

```
start_pes( 0 );
rank = shmem_my_pe();
value = (int*)shmalloc( sizeof( int ) );
/* ... do stuff ... */
if( 0 == rank )
    shmem_int_put( value, &result, 1, 1 );
MPI_Barrier( MPI_COMM_WORLD );
/* ... do stuff ... */
if( 0 == rank )
    MPI_Send( &number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );
if( 1 == rank )
    MPI_Recv( &number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &stat );
/* ... do stuff ... */
MPI_Allgather( &number, &result, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD );
```

Fig. 3. Excerpt of a program using both OpenSHMEM and MPI.

In addition to being an interesting feature for application developers, being able to use MPI in OpenSHMEM programs can also be useful for supporting tools, that can take advantage of some MPI-specific collective operations (such as reductions using user-defined operations and datatypes) to aggregate data.

3.4 Profiling

Profiling OpenSHMEM applications can be done by TAU without any specific tool interface [10]. DiPOSH can provide more low-level information, such as

profiling information on the communication channels, for the user to be able to tune the communication library.

TAU's measurement model give access to a multitude of possible performance data about the execution, from hardware counters to MPI information from MPI_T. Interestingly, MPI_T also gives the possibility to modify some parameters. For instance, OpenMPI provides several *levels* of parameters: end-user, application user, developer. Some of these parameters can be used in *write* mode by the application in order to tune the library at run-time. For instance, the maximum size of a message sent in eager mode, the size of a shared memory segment... can be modified through the MPI_T interface. Moreover, TAU can keep track of the memory usage in the shared heaps.

An example of profiling information about NUMA (NUMA hits, misses, and so on) is given in the performance section of this paper, in Subsect. 4.3.

4 Performance

We have run preliminary performance evaluations of DiPOSH on the Grid'5000 platform [4], using the Grimoire cluster in Nancy. It is made of 8 nodes, each of which featuring two 8-core Intel Xeon E5-2630v3 CPUs, 4 10 Gb Ethernet NICs and a 56 Gbps Infiniband network interconnection. The operating system deployed on the nodes is a Debian 9.8 with a Linux kernel 4.9.0. All the code was compiled using g++ 6.3.0 with -O3 optimization flag, and DiPOSH was linked against some Boost's libraries version 1.62 and OpenMPI 2.0.1.

4.1 Communication Performance

We evaluated the communication time using `shmem_char_put()` and `shmem_char_get()` operations on buffers of variable sizes. Each communication time was measured using `clock_gettime()` and run 20 times, except for small buffers (less than 10^4 bytes), that were run 200 000 times, since they are more subject to various noises on the system.

We measured the latency and the throughput on a single node and compared between when processes are bound to the same socket (`--map-by core` in OpenMPI) and on two different sockets (`--map-by socket` in OpenMPI). The latencies are given in Fig. 4 and the throughputs are given in Fig. 5.

As expected, communications are slightly faster when both processes are executed on the same socket. Also as expected, the performance of the direct shared memory implementation (called POSH SM in the captions) is significantly faster than the implementation using MPI-3 RDMA calls. This can be explained by the fact that implementing on top of MPI involves communicating through a thick software stack (see the aforementioned notion of software overhead), but more importantly, implementing the semantics of OpenSHMEM's one-sided communications using MPI-3's one-sided operations involves additional operations (`MPI_Win_lock()` and `MPI_Win_unlock()`).

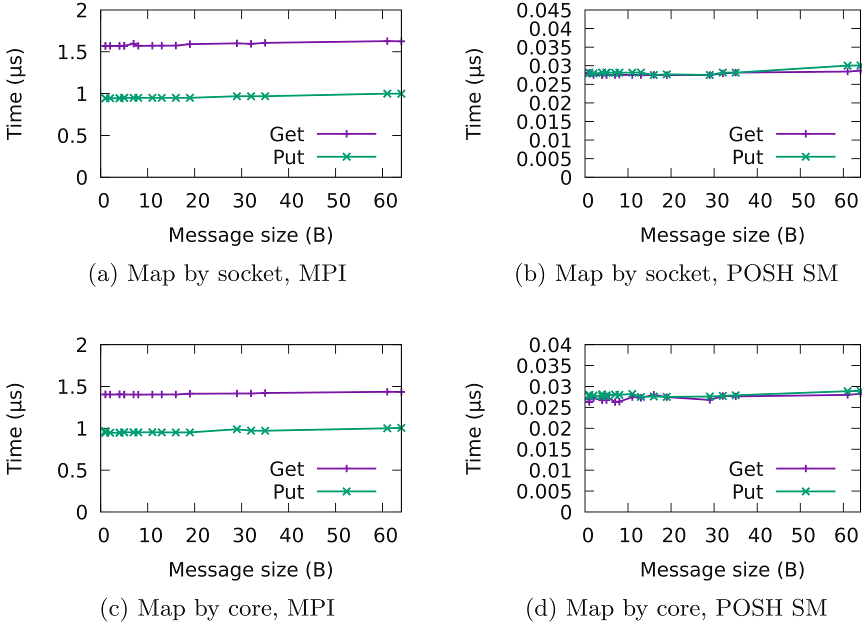
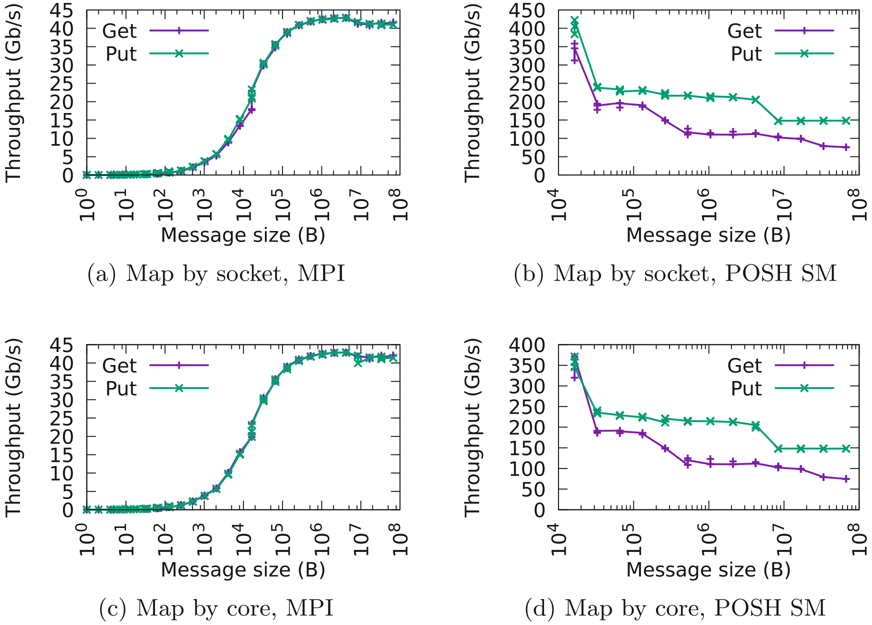


Fig. 4. Latency between two processes on the same node, over POSH's shared memory communication channel vs over MPI one-sided communications.

We have also measured the communication performance between two nodes, here using the implementation using MPI one-sided communications. The latency and the bandwidth are given in Fig. 6. It is interesting to see that, in spite of the harmfulness of the transposition between MPI-3's RDMA communication model and OpenSHMEM's communication model, the throughput is close to the announced bandwidth of the network used (56 Gbps).

4.2 Parallel Matrix-Matrix Multiplication

We implemented a parallel matrix-matrix multiplication using Cannon's algorithm. The initial local submatrices are placed in the shared heap, for other processes to fetch them using *get()* communications, hence placing their local copy in the private memory. Therefore, the local computations themselves are made on private memory. We used the DiPOSH communication channel on top of MPI-3 RDMA calls. The performance is represented in Fig. 7 and we can see that the OpenSHMEM model allowed us to write a straightforward implementation of the algorithm and DiPOSH provided good parallel performance.



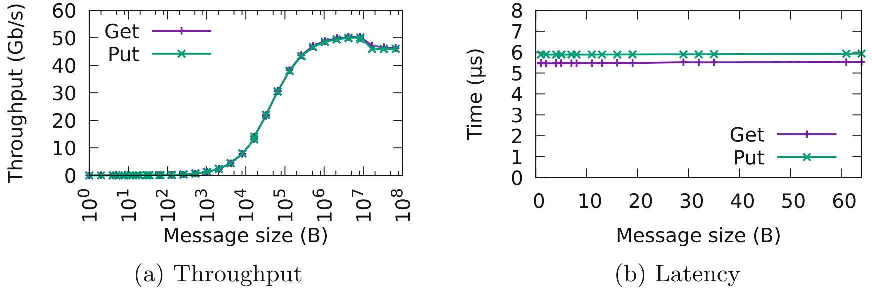
(a) Map by socket, MPI

(b) Map by socket, POSH SM

(c) Map by core, MPI

(d) Map by core, POSH SM

Fig. 5. Throughput between two processes on the same node, over POSH's shared memory communication channel vs over MPI one-sided communications.



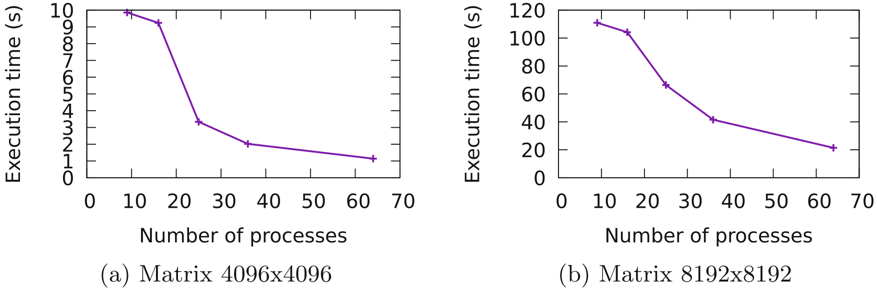
(a) Throughput

(b) Latency

Fig. 6. Communication performance between two nodes, DiPOSH over MPI.

4.3 Some Profiling Information

As described in Sect. 3.4, DiPOSH can extract low-level profiling information from the communication channels and provide them to TAU through user-defined events. We executed the parallel matrix-matrix multiplication on a single node using the POSH shared memory communication channel, and obtained NUMA statistics during the execution. The result displayed by TAU is given in Fig. 8. For instance, we can see, for this particular `shmem_get` call, how much time was spent reaching data on another NUMA node (`numa_foreign`). Such



(a) Matrix 4096x4096 (b) Matrix 8192x8192

Fig. 7. Scalability of a parallel matrix-matrix multiplication (Cannon's algorithm) implemented in OpenSHMEM, using DiPOSH over MPI.

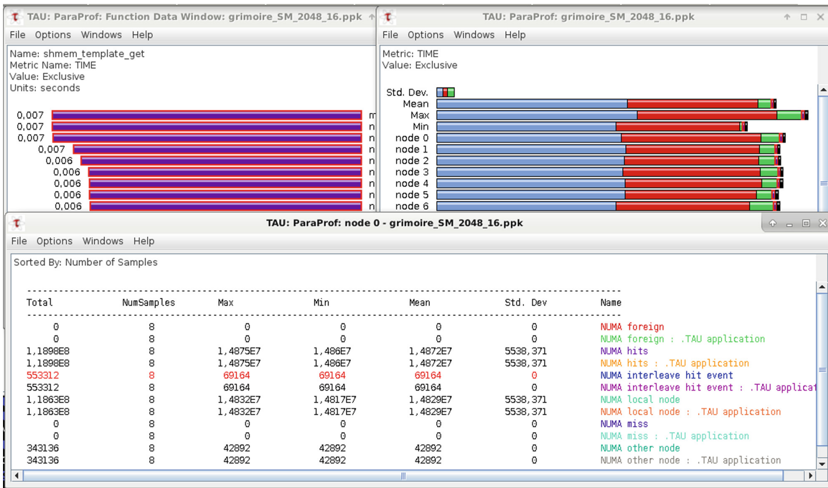


Fig. 8. Low-level profiling information: in addition to the usual function calls (in red, `MPI_Init()` and in purple, `shmem_*_get`), TAU displays information about the NUMA communications. (Color figure online)

information can be useful to understand how time is spent in communications and optimize data locality in order to minimize communication time.

5 Conclusion and Perspective

In this paper, we have presented the core design philosophy of DiPOSH, a high performance, distributed run-time environment and communication library implementing the OpenSHMEM specification. More specifically, DiPOSH focuses on taking advantage of the simple communication patterns of OpenSHMEM in order to implement a very thin software stack between the API and the network. Furthermore, it aims at being highly portable, able to take advantage of

high performance communication drivers, and delivering exceptional node-level efficiency with its POSH core.

In contrast to MPI+X hybrids, DiPOSH is completely compatible with MPI, making it possible to implement hybrid-like MPI+OpenSHMEM applications that can take advantage of the communication models of both libraries. Moreover, tools that support the parallel execution of an OpenSHMEM application can use these MPI communications, for instance, for global performance data aggregation and *in situ* analytic.

We have seen that DiPOSH can interface with the TAU measurement library in order to provide low-level profiling information about the communication channel. This can be used by application users in order to tune the library they are using, but it also opens perspective for a tooling interface that would exploit this information.

Future developments with DiPOSH include experimenting with high performance communication channels, for instance UCX [12], and KNEM for intranode communications [7], in order to be able to support a large number of networks while keeping the API-to-network path short. Moreover, we will work on providing fault tolerance capabilities, which is both necessary and challenging on exascale machines [5], particularly, for OpenSHMEM [9]. Fault tolerance can be achieved at system-level, for automatic fault-tolerance such as transparent checkpoint-restart [3] and at user-level, in order to provide the programmer with features that allow them to implement fault tolerant parallel applications, such as ULFM for MPI [2].

In addition to supporting a broad variety of networks, we are looking at a set of benchmarks and mini applications that can emphasize and stress the characteristics and choices of OpenSHMEM implementations. The next step will be to evaluate DiPOSH on large-scale supercomputers on these applications and benchmark its performance at fine grain.

We are currently working on supporting the complete OpenSHMEM interface and preparing a release. In the meantime, a partial support of the standard (providing at least point-to-point communications) can be found on a famous Git platform at the following URL: https://github.com/coti/POSH.

Acknowledgment. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

1. Barrett, B., Squyres, J.M., Lumsdaine, A., Graham, R.L., Bosilca, G.: Analysis of the component architecture overhead in open MPI. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 175–182. Springer, Heidelberg (2005). https://doi.org/10.1007/11557265_25
2. Bland, W., Bouteiller, A., Hérault, T., Hursey, J., Bosilca, G., Dongarra, J.J.: An evaluation of user-level failure mitigation support in MPI. Computing **95**(12), 1171–1184 (2013)

3. Butelle, F., Coti, C.: Distributed snapshot for rollback-recovery with one-sided communications. In: 2018 International Conference on High Performance Computing & Simulation (HPCS), pp. 614–620. IEEE (2018)
4. Cappello, F., et al.: Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In: SC 2005: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing CD, Seattle, Washington, USA, pp. 99–106. IEEE/ACM, November 2005
5. Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., Snir, M.: Toward exascale resilience: 2014 update. Supercomput. Front. Innov. **1**(1), 5–28 (2014)
6. Coti, C.: POSH: Paris OpenSHMEM: a high-performance OpenSHMEM implementation for shared memory systems. Procedia Comput. Sci. **29**, 2422–2431 (2014). 2014 International Conference on Computational Science (ICCS 2014)
7. Goglin, B., Moreaud, S.: KNEM: a generic and scalable kernel-assisted intranode MPI communication framework. J. Parallel Distrib. Comput. **73**(2), 176–188 (2013)
8. Hammond, J.R., Ghosh, S., Chapman, B.M.: Implementing OpenSHMEM using MPI-3 one-sided communication. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 44–58. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05215-1_4
9. Hao, P., et al.: Fault tolerance for OpenSHMEM. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, pp. 23:1–23:3. ACM, New York (2014)
10. Linford, J.C., Khuvis, S., Shende, S., Malony, A., Imam, N., Venkata, M.G.: Performance analysis of OpenSHMEM applications with TAU commander. In: Gorentla Venkata, M., Imam, N., Pophale, S. (eds.) OpenSHMEM 2017. LNCS, vol. 10679, pp. 161–179. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73814-7_11
11. Luo, M., Seager, K., Murthy, K.S., Archer, C.J., Sur, S., Hefty, S.: Early evaluation of scalable fabric interface for PGAS programming models. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, p. 1. ACM (2014)
12. Shamis, P., et al.: UCX: an open source framework for HPC network APIs and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 40–43. IEEE (2015)