Enabling Composite Applications through an Asynchronous Shared Memory Interface

Douglas Otstott* dotst001@fiu.edu
Noah Evans† noah.evans@gmail.com
Latchesar Ionkov† lionkov@lanl.gov
Ming Zhao* mzhao@fiu.edu
Michael Lang† mlang@lanl.gov

*School of Computing and Information Sciences
Florida International University
Miami, FL, USA

†Ultrascale Systems Research Center
Los Alamos National Laboratory
Los Alamos, NM, USA

***Abstract*—In this work we address the growing need for mechanisms for intranode application composition. We provide a novel shared memory interface that allows composite applications, two or more coupled applications, to share internal data structures without blocking. This allows independent progress of the applications such that they can proceed in a parallel, overlapped fashion. Composite applications using in-node shared memory can reduce the amount of data to be communicated between nodes, allowing data reduction or analytics to be performed locally and in parallel. To validate our approach we implemented our solution in Linux and used two proxy-applications to demonstrate how applications can be coupled and compare the performance to a traditional solution. We also compared the impact of composite applications to the performance of their unmodified versions. Our solution incurs small overhead in HPC linux environments and significantly outperforms preexisting approaches.**

***Keywords*-shared memory; composite applications; operating systems; memory management; checkpoint**

I. Introduction

As HPC systems increase in scale, scientific workflows utilizing them are becoming increasingly more complex. It is no longer enough for these systems to support time-sharing of nodes for computations and post-processing of simulation results. One of the key drivers for this change is the explosion of data produced by large-scale simulations. It is becoming increasingly difficult to move the data from the supercomputer to permanent storage and back again for additional processing steps.

Composite applications consist of two or more distinct applications coupled with scripting or glue-code to produce an aggregated macro-code resulting in greater functionality by absorbing additional processing steps. Such examples of coupled applications include the Community Earth System Model (CESM) [1], simulation paired with uncertainty quantification (UQ) or visualization, and multi-scale physics.

Composite applications are currently being developed in an ad-hoc manner with little system support. The driving need is to quickly couple existing applications with a small amount of glue-code to tie them together. This promotes replication of work, leads to fragile frameworks which do not scale, and can result in large amounts of data having to be moved off node.

Our novel approach to remedy the lack of system support for application composition is an asynchronous mechanism which would allow consistent data to be accessed within a node without interrupting the application. By leveraging copy-on-write (COW) and virtual memory mappings, applications can share data while preventing changes in one application's data set from immediately being reflected in the other. However, applications can still push and pull changes between processes when it becomes necessary.

To demonstrate this concept we developed a modification to the Linux shared memory system. An early prototype – Transparent Consistent Asynchronous Shared Memory (TCASM) – was described in a previous workshop paper [2]. In this work we further develop the prototype shared memory system and demonstrate how it can be used to create loosely coupled composite applications.

For our evaluation, we focus on two specific mini-applications and couple them with a realistic non-blocking checkpoint application and a simple analytics application to prove the benefits of this approach. We focus on application-specific checkpointing since it is the HPC community's key resilience mechanism and it is significantly impacted by the cost of data movement.

II. Background and Related Work

A. Need for Application Composition

As system sizes increase from petascale to exascale the data associated with large simulations is exploding. The memory footprint is huge, 32-64 petabytes [3]. This amount of data is difficult to move to and from persistent storage which is traditionally a parallel file system in HPC.

Currently in coupled physics, applications run sequentially, passing data from one package to another, with each package running on the whole allocation in turn. In future exascale architectures it is expected that some of these simulations will run in parallel rather than sequentially. One way to run these simulations would be to share data on the node. This has the advantage of reducing power, communication and may use the provided parallelism of current and future processors with large core counts more efficiently.

B. Related Application Coupling Techniques

Previous methods used for application composition included: directly linking in co-applications, handing off data through the use of a file on a shared file system, message passing, and standard shared memory mechanisms.

Existing examples of application coupling are similar to our solution, though not implemented directly in the OS. In-transit coupling of applications involves writing intermediate results to local storage where they are analyzed on the way to permanent storage [4] – in this case applications are linked by temporary files as is done in many-task frameworks like Falkon [5]. The ADIOS infrastructure [6] allows applications to be coupled by either using files, or with memory regions, but in case of the latter, it uses RDMA to transfer data between applications. In another work the authors rely on custom scripting to couple a quantum Monte Carlo code with Qwalk, a code to calculate the total energy, on a BlueGene/L system [7].

All of these examples could benefit from TCASM. TCASM provides a shared memory interface with the following features: asynchronous sharing, simple interface, process protection, attaching or detaching to a shared region at any time, reduced memory footprints, and data set versioning for scientific applications.

C. Other Related Work

Previous OS based methods to share memory between processes include System-V or POSIX shared memory [8] and tmpfs [9] - which allows shared files in a RAM disk. They do not provide a mechanism to ensure that the data is consistent and they require additional knowledge of all processes that share the data in order to synchronize accesses to it.

Knem [10] is specifically designed for interprocess/intra-node memory sharing for point-to-point communication and collective operations, as seen in MPICH2. The Boost [11] libraries have an interprocess section which includes message queuing, memory sharing and file locking functionality. Xpmem [12] is an ongoing project that facilitates inter-process memory sharing by allowing processes to map the virtual address space of different processes running on the same node. But all of these require synchronization between processes or a multi-buffer framework constructed from these mechanisms. Additionally, none of these support TCASM's copy-on-write functionality to simplify application programming.

III. Design and Implementation

A. TCASM Architectural Design

To explain our design we will use a simple yet typical coupled application example, a producer application and an observer application. The producer performs calculations on a data set with a certain set of computational steps which are then repeated for the next iteration. At the end of each set of steps the data is in a consistent state. The other process, or processes, called *observers*, can be started at anytime during the application's execution. Observers needs read access to a consistent state of the data that does not change until they are done processing it. They may work at a different rate than the producer and always need to access the most recent consistent state of the data.

Our goal with TCASM is to provide a copy-on-write (COW) shared memory region, mapped by both the producer and observer(s). In this way, the operating system mediates data changes to present consistent data to the sharing processes and reduces the amount of complexity in sharing data among processes.

The simplest explanation of the TCASM approach is that it provides a shared memory interface that avoids application synchronization by letting the operating system intervene when collisions on the data would occur. The operating system then COWs the data to avoid these collisions. TCASM tries to preserve the asynchronous aspect of multi-buffer approaches and minimize the memory and coordination required among the producer and observers. The basic premise is to avoid data duplication by sharing the pages that contain unmodified data among the multiple versions of the data set in memory.

In the worst case, if all pages in the data set are modified every time and each observer is advancing at a unique rate, the memory overhead will be equal to the number of observers times the size of the shared region, as in a multi-buffer implementation. However, our solution will still retain the advantages of unsynchronized access due to the operating system's management of COW data. Also the granularity of changes monitored between the two processes are reduced to the size of a page.

Our main goal is to provide a clean and simple interface for publishing data between distinct processes. We achieve this by modifying the two existing shared memory functions in Linux, `mmap` and `msync`. The implementation of TCASM under Linux is detailed in previous paper [2].

Figure 1 depicts the interaction between producer and observer and the shared memory region. Initially, the producer and observer's virtual memory pages point to an unmodified shared memory file. On the far right of the diagram, the producer is modifying a page, so a new page is allocated and mapped in the producer's virtual memory as per the standard COW procedure. On the left, the producer has published its changes, so the original pages must be preserved in the observer's address space. Pages are copied out of the shared region and the observer's virtual address space is updated to reflect this new state. This allows observers to advance at different rates than the producer and one another since they always keep their own versions of the data.

With this method, the producer task incurs a COW overhead for each page that it modifies. However, we argue that the overhead of COW will be overshadowed by the lack of explicit synchronization required to manually copy the data. In addition, an expensive copy operation can be prevented if the producer does not modify all pages in the shared region, giving an advantage to the COW mechanism over unconditionally copying the entire shared range of pages in a multi-buffer

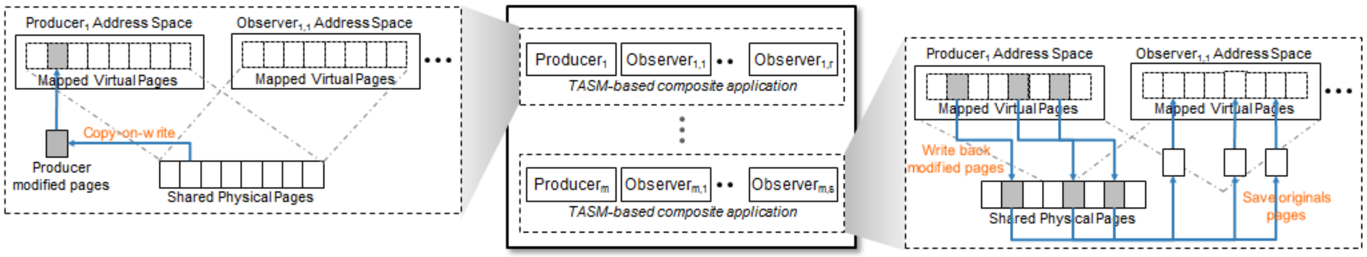

Fig. 1: TCASM Architecture

solution. Furthermore, this implementation provides significant decoupling between producers and observers. The only shared structure is the data itself and the producer makes its progress continuously without coordinating with observers which are created or destroyed independently.

B. TCASM Application Interface

Using TCASM requires some simple application modifications. The data required by any observer needs to be placed in shared regions. The code needs to be modified to publish the new versions at points where data is consistent, by calling `msync`. Simple application code would look as follows:

```
# application (producer)
mmap (filename,data)
Enter Timing loop {
Do work ()
call msync() # to publish data version
}
```

```
# co application (observer)
Enter Processing loop {
mmap (filename,data) # to get new data
Do work ()
call unmap()
}
```

For an MPI runtime, mapped filenames are generated to align with MPI ranks such that the observers can deduce the names from information they already have. A similar scheme can be implemented for other runtimes.

The application using TCASM to share its data also needs to provide some metadata to hold application-specific descriptions of the structures to be shared. Information such as the number of regions and their respective sizes, the variables and their types, data versions, and important iterators from the application, may be required in order to allow the observers to interpret the data. For example, a checkpoint co-application would need all of the information that is required to restart the application from a checkpoint such as the input deck and iterator values. For an analytics observer the extents and offsets of the data structures of interest for the calculation would need to be included. Any constants can be included here as well. Using a defined data description standard such as HDF5 or netCDF could easily be supported.

The initiating application has to provide this data set for the observing applications. The application also needs to allocate the data structures for publishing using `mmap`. We developed

Operation	Latency
`msync`	0.98 microseconds/page
`memcpy`	0.63 microseconds/page
COW	2.71 microseconds/page

TABLE I

a C based FORTRAN library, and a custom allocator for C++ for this reason.

There are several FORTRAN wrapper functions to facilitate the system calls and pointer arithmetic necessary to instantiate data from shared memory. Since the interface is general, developers who wish to use this interface need only allocate c-type pointers (a data type in FORTRAN), and pass them to one of the functions in the C library. These pointers can then be used to allocate data types in FORTRAN.

For C++ the best solution was to implement a custom memory allocator which would use `mmap` to create objects. This way, any critical data could simply be initialized using the shared memory allocator in place of the standard allocator.

IV. Evaluation

A. Methodology

To evaluate the performance and applicability of our approach we employ custom micro-benchmarks to look at overheads and to compare various sharing mechanisms. We then use proxy applications to investigate the performance benefits of TCASM in two specific use cases; coupled with a checkpoint utility co-application and an analytics co-application.

B. Performance Overhead

In this section, we report the observed performance depredation incurred by TCASM's *Copy-On-Write (COW)* and *msync* mechanisms using an in house benchmark application. Results collected are summarized in Table I. Latency of calls to memcpy are included for comparison. Time values presented are aggregate times for contiguous data sets, represented as time per page.

C. Comparison to Other Sharing Methods

In these experiments we test microbenchmarks using TCASM, mmap, an in-line function call (func) and composition using a shared file (file). Producer and observer overheads were compared against the same benchmark running in isolation using a variety of composition mechanisms. Mmap and xpmem were implemented with basic producer-consumer queues and all were compared over increasing data sizes. Figure 2 shows overhead of the composition mechanisms. The missing points in the TCASM data are due to noise in the data.

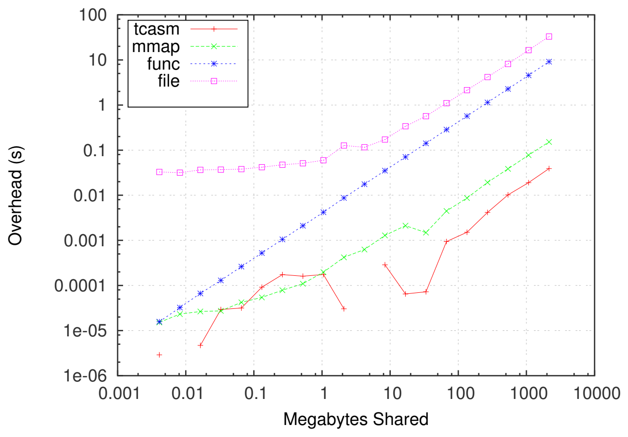

Fig. 2: Comparison of sharing composition mechanisms relative to a producer not sharing any data

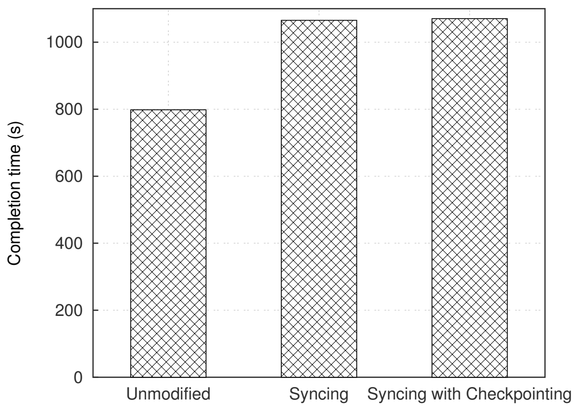

Fig. 3: MiniFE Performance: Unmodified compared to TCASM syncing data every iteration and TCASM syncing every iteration with a co-process copying the data off node.

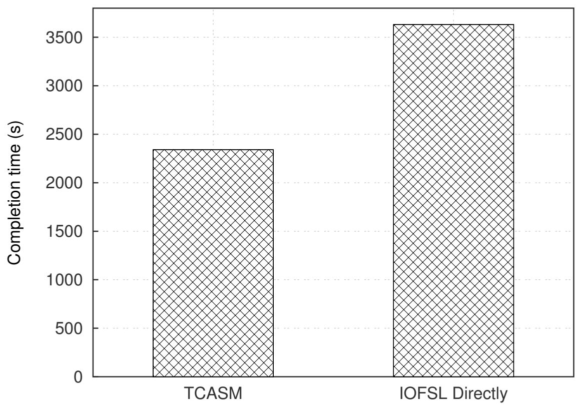

Fig. 4: Runtime of SNAP's TCASM Implementation with Checkpointing Observers compared to direct IOFSL Checkpointing)

To further evaluate our implementation we also modified two MPI based mini-apps (SNAP [13] and MiniFE [14]) to instantiate critical data structures with `mmap` and publish consistent data sets at regular intervals using `msync`. These mini-apps are often used to evaluate how real scientific applications will perform on new systems.

We evaluated both SNAP and MiniFE's TCASM implementations against their unmodified implementations under various parameters.

D. Checkpointing Use Case

We also identified use cases that demonstrate the viability of the shared memory interface enabled by TCASM. First we investigate an application with embedded checkpoint functionality decomposed into a separate simulation application and a checkpoint utility co-application. We designed the checkpointing system to use TCASM to share critical data between the producer and observer processes which would then forward the data to a storage server where it would be persisted in the file system. The data can later be used to restart a job, in case of a failure. To reflect production HPC environments, the observer incorporates a typical I/O forwarding layer to allow data to be staged on remote nodes. Specifically, we consider the IOFSL [15] forwarding library, a scalable I/O forwarding framework consisting of a library for the client and a server that resides on a remote node(s). The observer processes use the IOFSL [15] forwarding library to forward the shared data to a remote storage server. For comparison, we also modified SNAP to use IOFSL directly without employing the TCASM observers.

Figure 3 shows the run times of three flavors of MiniFE (from left to right): 1) the original implementation, 2) an implementation utilizing the custom memory allocator and syncing shared memory at regular intervals, and 3) the TCASM functional implementation (syncing at regular intervals) sharing the node with TCASM observers designed to persist the data necessary to fully restart MiniFE from its last consistent state. TCASM allocation and regular syncing account for the 34% run time overhead while the presence of observers adds nothing to execution time.

Similar to Figure 3, Figure 4 reflects the performance difference between the TCASM implementation of SNAP running alongside the TCASM checkpointing observer, and SNAP modified to do its own checkpointing after each iteration. Clearly, there are significant performance gains from leveraging shared memory for data sharing with the TCASM implementation finishing in about 64% of the total run time of direct checkpointing implementation. This particular experiment was done on a a relatively small data set (500MB per process across 32 MPI processes).

Finally, to demonstrate TCASM's effectiveness at scale, several experiments were conducted on 202 cluster nodes using the PRObE system [16]. Figure 5 illustrates the relative performance of 3 implementations of SNAP. The first, is the original implementation of SNAP with 400 ranks running across 200 nodes with one head node reserved for synchronization. The second is the same set up with the TCASM version of SNAP syncing every time it reaches a consistent

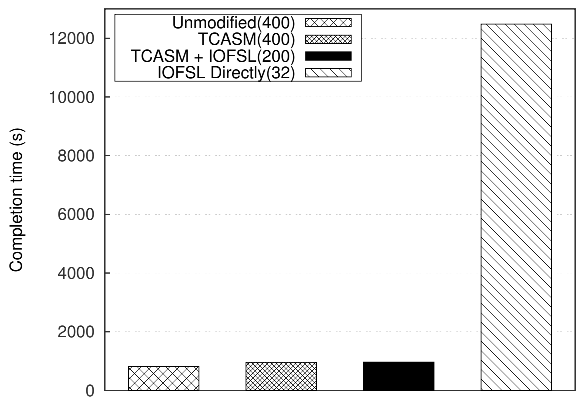

Fig. 5: Performance of SNAP, scaled on 200 and 400 compute nodes

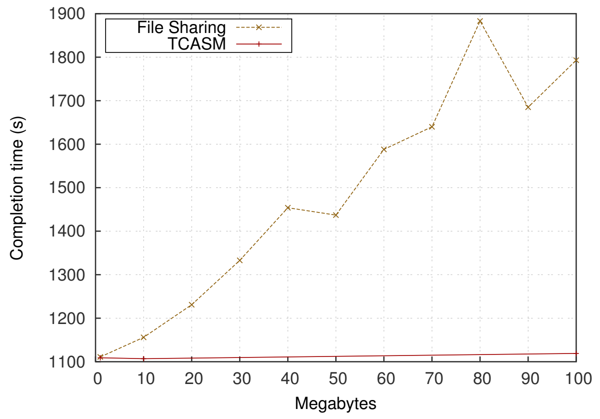

Fig. 7: Runtime of SNAP using the file system to share data as compared to TCASM

state, with and without a set of observers syncing to the final node, which is reserved for persistent data storage. As the first three bars in Figure 5 illustrate the difference is minuscule. Since `msync` is a local operation, only affecting the memory on the node its called on, its overhead is related to how many processes are calling it, and how much data is being synced. In this particular setup, the cost of two ranks calling msync on 1 Gigabyte simultaneously is negligible. The final bar represents the same setup with an implementation of SNAP that writes to the storage server directly. However, in this experiment, only 32 of the 400 processes are syncing during the course of the experiment. Here, the real benefit of TCASM checkpointing becomes evident.

E. In situ Analysis Use Case

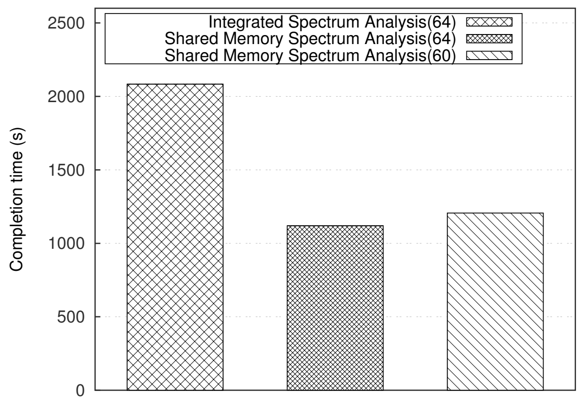

Fig. 6: Runtime of SNAP with an added Spectrum Analysis procedure added versus a TCASM implementation sharing necessary data to an analytical observer

A second use case for TCASM is in situ data analysis. For this case, we implemented a specialized TCASM observer to calculate the energy spectrum of a given SNAP experiment for each time step. At the end of each time step, SNAP publishes the data necessary to calculate the energy spectrum along with the necessary meta-data. An observer for each rank performs the necessary calculations on its local data set and calls an MPI collective operation to ascertain the global values. For testing purposes, this data was simply printed to standard output. To compare the performance of this implementation a module was added to SNAP to compute the energy spectrum within SNAP itself. The exact same code was used, with the exception of the initialized variables. In SNAP, the data is simply passed in or made available globally. In the observer, the data must be allocated and initialized from shared memory. The rest of the operations are identical.

Figure 6 shows the comparison between the two implementations, on a 64 core cluster node using process binding. Using TCASM for spectrum analysis cuts the execute time of SNAP in half. Again, in an attempt to isolate the computational cost and context switching noise between producers and consumers, the number of ranks was dropped to 60 and divided among the sockets. This resulted in slightly higher execution time. Even with the computational overhead of the analytical operations, it is still preferable to have more MPI processes rather than provide process isolation.

Figure 7 shows SNAP's completion time vs data size shared with two versions; 1) data shared with TCASM, flat scaling; 2) data shared on the filesystem, poor scaling. The data shared was for the spectrum calculation to show the benefit of TCASM over ad-hoc composition on the filesystem.

V. Conclusions and Future work

In this paper we presented TCASM, an asynchronous shared memory interface for coupling composite applications. We discussed the design of the implementation using the memory subsystem of Linux and showed the advantage of providing a simple and transparent interface that allows sharing of memory while minimizing the memory used and eliminating synchronization between coupled processes.

We provided interfaces for scientific applications which are predominately written in C, FORTRAN, or C++. These contributions allow our work to be easily used by application developers.

We also demonstrated real world use cases for TCASM by using two scientific mini applications. SNAP and MiniFE were modified to use shared memory and publish consistent states at regular intervals. The resulting data was then used by an "observer" to bleed it off to persistent storage servers via an IO forwarding layer. The presence of the observer was shown to have little impact on the performance of the application, while some overhead was attributed to the cost of repeated calls to `msync` and the overhead from copy-on-write. However, this overhead is small in comparison to traditional checkpointing techniques, where the application would have to temporarily halt execution while its data is copied to persistent storage. In this case we see a huge 11.96x improvement over traditional checkpointing.

In addition, we developed an analytic observer to calculate the energy spectrum of the flux array in the SNAP application. We showed a performance improvement of 1.86 times, again, due to the decoupling of the application computation and the analytics calculation which allows for their overlap. This approach would clearly be beneficial for many coupled application scenarios, such as visualization, uncertainty quantification, etc.

Hobbes [17], a new exascale OS, is currently under design which supports composite applications. This work is currently being integrated into Hobbes as well as being made available to the Linux HPC community. We would also like to prove this method with large-scale applications such as the Community Earth System Model (CESM) [1] and other coupled physics applications. We see this work as a valuable building block to support dynamic run times for many-core processors.

Acknowledgments

We thank Hakan Akkan for his initial development, and Joe Zerr for his help with the SNAP mini-app in terms of analytics and checkpointing. This work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, and is supported by the U.S. Department of Energy DE-FC02-06ER25750 and in part by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE). http://www.nmc-probe.org/. The publication has been assigned the LANL identifier LA-UR-13-28104. Douglas Otstott and Ming Zhao are supported by the National Science Foundation grant CCF-0938045 and CAREER award CNS-125394 and the Department of Homeland Security grant 2010-ST-062-000039.

References

[1] P. R. Gent, G. Danabasoglu, L. J. Donner, M. M. Holland, E. C. Hunke, S. R. Jayne, D. M. Lawrence, R. B. Neale, P. J. Rasch, M. Vertenstein *et al.*, "The community climate system model version 4," *Journal of Climate*, vol. 24, no. 19, pp. 4973–4991, 2011.

[2] H. Akkan, L. Ionkov, and M. Lang, "Transparently consistent asynchronous shared memory," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2013, p. 6.

[3] G. Grider, "Exascale fsio/storage/viz/data analysis can we get there? can we afford to?" 2011.

[4] K. Moreland, R. Oldfield, P. Marion, S. Jourdain, N. Podhorszki, V. Vishwanath, N. Fabian, C. Docan, M. Parashar, M. Hereld *et al.*, "Examples of in transit visualization," in *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*. ACM, 2011, pp. 1–6.

[5] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a fast and light-weight task execution framework," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, no. 43. Reno, NV: ACM Press, November 2007. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362680

[6] N. Podhorszki, S. Klasky, Q. Liu, C. Docan, M. Parashar, H. Abbasi, J. Lofstead, K. Schwan, M. Wolf, F. Zheng *et al.*, "Plasma fusion code coupling using scalable i/o services and scientific workflows," in *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*. ACM, 2009, p. 8.

[7] A. Sayed and H. El-Shishiny, "Computational experience with nanomaterial science quantum monte carlo modeling on BlueGene/L," in *MEMS, NANO, and Smart Systems (ICMENS), 2009 Fifth International Conference on*. IEEE, 2009, pp. 213–217.

[8] J. Moran, "Sunos virtual memory implementation," in *Proceedings of the Spring 1988 European UNIX Users Group Conference*, 1988.

[9] P. Snyder, "tmpfs: A virtual memory file system," in *Proceedings of the Autumn 1990 EUUG Conference*, 1990, pp. 241–248.

[10] B. Goglin and S. Moreaud, "Knem: A generic and scalable kernel-assisted intra-node {MPI} communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176 – 188, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512002316

[11] "Boost C++ libaries," 2007. [Online]. Available: http://www.boost.org/

[12] "xpmem: Cross-process memory mapping," 2014. [Online]. Available: https://code.google.com/p/xpmem/

[13] J. Zerr and R. Baker, "Snap: Sn (discrete ordinates) application proxy - proxy description," 2013. [Online]. Available: https://github.com/losalamos/SNAP

[14] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratory, Technical Report SAND2009-5574, 2009.

[15] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable i/o forwarding framework for high-performance computing systems," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[16] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd, "Probe: A thousand-node experimental cluster for computer systems research," vol. 38, no. 3, June 2013. [Online]. Available: \url{https://www.usenix.org/publications/login/june-2013-volume-38-number-3/probe-thousand-node-experimental-cluster-computer}

[17] R. Brightwell, R. Oldfield, A. B. Maccabe, and D. E. Bernholdt, "Hobbes: Composition and virtualization as the foundations of an extreme-scale os/r," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2013, p. 8.