Los Alamos
NATIONAL LABORATORY
EST. 1943

LA-UR-18-30292 (Accepted Manuscript)

On the memory attribution problem: A solution and case study using MPI

Gutierrez, Samuel Keith
Arnold, Dorian C.
Davis, Marion Kei
McCormick, Patrick Sean

Provided by the author(s) and the Los Alamos National Laboratory (2019-04-03).

To be published in: Concurrency and Computation: Practice and Experience

DOI to publisher's version: 10.1002/cpe.5159

Permalink to record: http://permalink.lanl.gov/object/view?what=info:lanl-repo/lareport/LA-UR-18-30292

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

RESEARCH LIBRARY

Received 9 October 2017; | Revised 23 August 2018; | Accepted
DOI: xxx/xxxx

On the Memory Attribution Problem: A Solution and Case Study Using MPI

Samuel K. Gutiérrez*[1,2] | Dorian C. Arnold[3] | Kei Davis[1] | Patrick McCormick[1]

[1]Computer, Computational, and Statistical Sciences Division, Los Alamos National Laboratory, New Mexico, United States
[2]Scalable Systems Laboratory, Department of Computer Science, University of New Mexico, New Mexico, United States
[3]Department of Math and Computer Science, Emory University, Georgia, United States

Correspondence
*Corresponding author, Email: samuel@lanl.gov

Summary

As parallel applications running on large-scale computing systems become increasingly memory-constrained, the ability to attribute memory usage to the various components of the application is becoming increasingly important. We present the design and implementation of *memnesia*, a novel memory usage profiler for parallel and distributed message-passing applications. Our approach captures both application- and message-passing-library-specific memory usage statistics from unmodified binaries dynamically linked to a message-passing communication library. Using micro-benchmarks and proxy applications, we evaluated our profiler across three Message Passing Interface (MPI) implementations and two hardware platforms. The results show that our approach and corresponding implementation can accurately quantify memory resource usage as a function of time, scale, communication workload, and software or hardware system architecture, clearly distinguishing between application and MPI library memory usage at a per-process level. With this new capability, we show that job size, communication workload, and hardware/software architecture influence peak runtime memory usage. In practice, this tool provides a potentially valuable source of information for application developers seeking to measure and optimize memory usage.

KEYWORDS:
HPC, MPI, memory utilization, profiling

1 | INTRODUCTION

In high-performance computing (HPC), parallel programming systems and applications are evolving to improve performance and energy efficiency, particularly as systems scale to higher degrees of intra- and inter-node parallelism. As the number of processing elements in these systems continues to grow, memory capacity to core ratios tend to remain constant or shrink. The data in Table 1 show the ratios of memory capacity to compute core count for the last ten top-ranked systems of the Top500 (1). Of these systems, only three have a *memory-to-core* ratio (in GB:core) of at least 2:1—the first was 4:1 in 2002 and the last was 2:1 in 2011—with over half of the remaining seven systems having less than 1 GB of memory per core. This decrease has been pushing many applications toward memory-capacity-bound computing regimes. In these cases, developers will increasingly rely on understanding how the supporting software infrastructure (i.e., operating system (OS), software libraries, and middleware) affects overall application memory efficiency along three major axes: runtime, job size (*scale*), and workload.

HPC application developers commonly couple the high-level application driver code, software components, that drive the use of lower-level parallel programming systems, with supporting software such as a message-passing library, resulting in a single executable after linking. Such couplings can make it difficult to accurately attribute an application's memory usage across the full set of software components. For example, we may not be able to accurately answer questions such as: *What is the message-passing library's contribution to my application's overall memory footprint?* In general, this *memory attribution problem* arises when an application developer cannot predict or evaluate during runtime where the

Top500 List	System	Core Count	Memory Capacity (GB)	Memory per Core (GB)
June 2002	Earth Simulator	5,120	20,480	4.00
November 2004	Blue Gene/L	32,768	8,192	0.25
June 2008	Roadrunner	122,400	106,086	0.87
November 2009	Jaguar	298,592	598,016	2.00
November 2010	Tianhe-IA	186,368	229,376	1.23
June 2011	K Computer	705,024	1,410,048	2.00
June 2012	Sequoia	1,572,864	1,572,864	1.00
November 2012	Titan	560,640	710,144	1.27
June 2013	Tianhe-2	3,120,000	1,024,000	0.33
June 2016	Sunway TaihuLight	10,649,600	1,310,720	0.12

TABLE 1 Hardware statistics of the last 10 number one computer systems according to the Top500 by earliest date of first-place ranking.

available memory is used across the software stack comprising the application, software libraries, and supporting runtime architecture needed to enable the application at a given scale, under a given workload, and in a time- and space-sharing scheduled environment.

In summary, improving application memory efficiency is becoming increasingly important in the development, deployment, and upkeep of parallel and distributed programs, but is complicated by concurrent instances of coupled software components dynamically consuming memory resources over time. At the same time, there is a lack of parallel tools capable of extracting the relevant metrics to solve the memory attribution problem. In this work, we address the memory attribution problem in parallel and distributed message-passing software systems as follows.

1. We propose an approach for accurate per-process quantification of memory resource usage over time that is able to clearly distinguish between application and MPI library usage. Our experimental results show that job size, communication workload, and hardware/software architecture influence peak runtime memory usage.
2. We develop a corresponding open-source profiling library named *memnesia* (2) for applications using any implementation of the Message Passing Interface (MPI) (3). We develop this software with a specific goal in mind: once memory attribution is better understood, applications and MPI implementations will potentially be able to improve or maintain their memory utilization as they are developed and maintained.
3. We evaluate our profiler's runtime overhead and behavior using both micro-benchmarks and proxy applications, concluding with an analysis of memnesia's memory overhead and perturbation.

To the best of our knowledge this is the first work to both implement and evaluate such an approach for parallel and distributed software systems.

2 | BACKGROUND

In the first half of this section, we discuss application memory utilization in the context of parallel message-passing programs, and then go on to discuss techniques in parallel application analysis, where we summarize well-known taxonomies that categorize tools along four axes. The last half of this section describes key approaches, mechanisms, and system software infrastructure used by our memory usage profiler.

2.1 | Parallel Application Memory Utilization

Application memory utilization is concerned with application memory usage and often focuses on studying dynamic heap behavior. In this context, an application's *memory footprint*, the minimum memory capacity required to complete its calculation successfully, is the aggregate of the *application driver footprint* and each of the middleware and runtime library footprints. The application driver implements the numerical methods that underlie a particular system model or simulation, while the middleware and runtime services coordinate the execution of parallel (and potentially distributed) process instances. An application driver's footprint is primarily influenced by 1. its underlying numerical methods, 2. how those methods are implemented (e.g., data structures, parallelization strategies), and 3. the size and fidelity of its computational domain. Message-passing libraries such as Open MPI (4) and MPICH (5) are examples of message-passing middleware. Like the application drivers they support, they consume memory to maintain their internal state, which is primarily influenced by how they are driven with respect to job size (e.g., the size of `MPI_COMM_WORLD`) and communication workload.

2.2 | Parallel Application Analysis

Parallel and distributed tools that provide insight into application behavior are important for the development, deployment, and upkeep of parallel programs. Developing such tools is challenging because data collection and analysis is usually distributed across a set of computational resources, requiring that their resulting outputs be aggregated for further analysis by the end-user. Tools may be categorized by

- *Functionality* (correctness or performance): Correctness tools aid in identifying application (algorithmic) correctness bugs, whereas performance tools aid in identifying performance bugs.
- *Instrumentation methodology* (dynamic or static): Dynamic tools generally operate on unmodified application binaries and use facilities such as *ptrace* (6) to observe and control the execution of application processes. In contrast, static tools insert instrumentation instructions such as *probes* into applications during preprocessing-, compilation-, or link-time transformations.
- *Measurement methodology* (event tracing, event sampling): Event tracing gathers data by activating a set of instrumentation probes at every event associated with a trace, for example, *function interposing*, whereas sampling-based measurements are typically interrupt-driven and provide only a statistical view of application behavior, e.g., program counter sampling.
- *Interactivity* (online, offline): Online analysis tools are interactive and are meant to be influenced at run time by an end user during data collection and analysis phases. Offline analysis tools, in contrast, are generally more static, meaning that the tool is started with and runs alongside an application until termination, then tool data are written, post-processed, and finally analyzed by other programs. This approach, while popular in practice because of its relative simplicity, tends to scale poorly because of high data storage and analysis costs (7).

2.3 | Intercepting Application Behavior

Function interposition is a powerful technique used to insert arbitrary code between a caller and its intended callee (8, 9). For compiled languages this is typically achieved by *function symbol overloading*, where a duplicate function definition is introduced into an application such that the duplicate entry's symbol is resolved ahead of the intended callee's, with the consequence that its code is executed instead. This technique is well known and widely used to instrument dynamically linked libraries because probes can be introduced into unmodified binaries via the runtime loader, which is typically achieved by using `LD_PRELOAD`.

The MPI profiling interface (PMPI) provides a straightforward and portable mechanism for intercepting all MPI-defined functions (3). Specifically, the MPI specification requires that libraries provide an alternate entry point, achieved through a *name shift*, which can be used for tooling purposes. Listing 1 shows an example of how a tool might intercept application calls to `MPI_Barrier()` using PMPI and ultimately function interposing.

LISTING 1 MPI profiling interface example.

```
// For MPI_Comm type definition.
#include "mpi.h"

int MPI_Barrier(MPI_Comm comm) {
    // Tool code before barrier.
    ...
    // Execute MPI barrier.
    int rc = PMPI_Barrier(comm);
    // Tool code after barrier.
    ...
    return rc;
}
```

2.4 | Collecting Process/System Information

The *proc pseudo file system* (*procfs*) offers a convenient interface for obtaining information about and influencing the state of a running OS kernel (10). *procfs* provides user-space access to kernel-maintained state by exposing a file-based access semantics to the structure hierarchy it maintains (*directories* and *files*). Obtaining information about current OS state, including that of active processes, is accomplished by opening and parsing *files* located below *procfs*'s mount point (typically `/proc`). In many cases the content of these special files is generated dynamically to provide an updated view of the operating system's state.

In Linux, `/proc/[pid]/smaps` (*smaps*) shows memory consumption for each of the process's mappings (10). Each *smaps* entry can be thought of as having two pieces: a *header* and a *body*. The header contains address range occupancy, access permission, and (if applicable) backing store information, while the body contains memory map statistics, including resident set size (RSS) and proportional set size (PSS). The RSS represents

how much of the mapping is currently *resident* in RAM, including shared pages. In contrast, PSS represents a process's share of the mapping, meaning that, for example, if a process has 100 private pages and additionally shares 100 more with another process, then its PSS is 150 (i.e., $100 + 100/2$). A process's RSS and PSS will change during run time and are both influenced by the amount of process-driven memory pressure exerted on the system.

3 | METHODS IN MEMORY UTILIZATION ANALYSIS

In this section, we begin with an examination of related work in *memory utilization analysis*, describing how contemporary approaches address the previously described memory attribution problem. We then describe our approach and its corresponding open-source implementation, memnesia.

3.0.1 | Heap Profiling and Memory Map Analysis

Heap profiling identifies and gathers statistics about call paths containing dynamic memory management calls, for example, `malloc()` and `free()`. Notable heap profilers include Valgrind Massif (11), Google heap profiler (GHP) (12), and memP (13). GHP and memP work by using `LD_PRELOAD` and function overloading of memory management calls. This approach to heap profiling has limitations: 1. it does not work on statically linked binaries, 2. it does not allow a user to distinguish between memory pages mapped into a process's address space and memory pages that are resident in physical memory, 3. it multiply counts shared memory pages and does not allow a user to determine which cooperating process *owns* the page, and 4. it does not allow a user to distinguish application driver memory usage from runtime/middleware memory usage.

Memory map analysis collects information about a running application by inspecting application-specific entries in *procfs*. This approach is appealing for a variety of reasons. First, it is relatively straightforward to implement, avoiding complications brought by user-level interception of memory management functions (e.g., some memory management calls cannot be traced, for example when `glibc` calls `__mmap()` (14)) or virtualization (e.g., processes run in isolated virtual environments do not adequately emulate OS-specific management schemes regarding shared pages). Second, it can provide a more holistic view, when compared to heap profiling alone, into important features that ultimately impact a process's actual memory footprint, namely the size and count of private/shared pages and their occupancy in RAM. Finally, it is language-agnostic and therefore readily applicable to any running process. As an example, smem (15) is a standalone tool capable of generating a variety of whole-system memory usage reports based on the PSS metric. Like memnesia, smem uses memory map analysis of *smaps* for usage reporting but is not a parallel tracing tool.

3.0.2 | Middleware Attribution of Memory Usage

As previously described, determining how much memory the message-passing library consumes is challenging and becoming increasingly important in the development, upkeep, and deployment of parallel programs. Current approaches for MPI library memory attribution generally can be categorized as library-specific instrumentation or benchmark-driven library analysis. An example of the former, *craymem*, can be found in Cray's implementation of MPICH (16), where through environmental controls, internal memory monitoring statistics can be accessed via textual output (either to a terminal or a file.). Such library-specific approaches are implementation-dependent and often provide coarse-grained output. For example, the output

```
# MPICH_MEMORY: Max memory allocated by malloc: 466088 bytes by rank 0
# MPICH_MEMORY: Min memory allocated by malloc: 464792 bytes by rank 1
# MPICH_MEMORY: Max memory allocated by mmap: 24704 bytes by rank 0
# MPICH_MEMORY: Min memory allocated by mmap: 24704 bytes by rank 0
# MPICH_MEMORY: Max memory allocated by shmget: 37232464 bytes by rank 0
# MPICH_MEMORY: Min memory allocated by shmget: 0 bytes by rank 1
# [0] Max memory allocated by malloc: 466088 bytes
# [0] Max memory allocated by mmap: 24704 bytes
# [0] Max memory allocated by shmget: 37232464 bytes
```

does not allow a user to determine when during the program's execution memory usage *high-water marks*—the maximum recorded values—were reached or whether these maxima were transient or sustained for long periods of time.

As an example of the latter, mpimemu (17) provides benchmark-driven memory attribution for MPI implementations. mpimemu is an MPI program with built-in memory map monitoring that works by sampling `/proc/self/status` and `/proc/meminfo`, while also imposing a scalable communication workload on the system. Runtime memory attribution is approximated by calculating usage deltas between samples collected during its execution and those collected before the MPI library was initialized. This approach works for understanding coarse-grained application and workload features captured in the given benchmarks, but does not provide any insight into how a given MPI library's memory usage is affected when driven by a specific application or set of applications.

	Language Independent	Linkage Agnostic	Residency Aware	Residency Aware (Proportional)	Component Aware	Temporal
GHP	–	–	–	–	–	–
memP	–	–	–	–	–	–
Massif	–	–	–	–	–	–
smem	✓	✓	✓	✓	–	–
craymem	✓	✓	–	–	–	–
mpimemu	–	✓	✓	–	–	–
memnesia	✓	–	✓	✓	✓	✓

TABLE 2 Tools and their respective attributes.

3.1 | Our Approach

We present an event-driven-analysis approach for accurately capturing both application- and message-passing-library-specific memory usage of parallel and distributed message-passing programs. As shown in Table 2, our approach overcomes virtually all of the shortcomings of previous approaches. While our approach generalizes to any message-passing system, our reference C++ implementation, memnesia, relies on OS support for certain *procfs* features (i.e., *smaps*) and C entry points into the MPI library.

3.1.1 | Application Instrumentation and Data Collection

We implement our application instrumentation as a runtime system compiled into a shared library that is loaded into target application binaries at startup via the runtime loader. In practice this is accomplished via environmental controls: before application startup, `LD_PRELOAD` is set to include the memnesia runtime, then the application is launched as usual. The application drives data collection through its use of MPI. Each entry point into the message passing library becomes an instrumentation point at which we execute tool code between the caller and the intended callee using function interposition. At each instrumentation point, the memnesia runtime places *calipers*—a pair of instrumentation probes—around the target function such that *smaps* data are collected immediately before and after callee execution, as shown in Listing 2. Tool data are stored in per-process, in-memory caches maintained by the memnesia runtime through parallel data aggregation. On program completion, memory analysis data are written to disk, as shown in Figure 1.

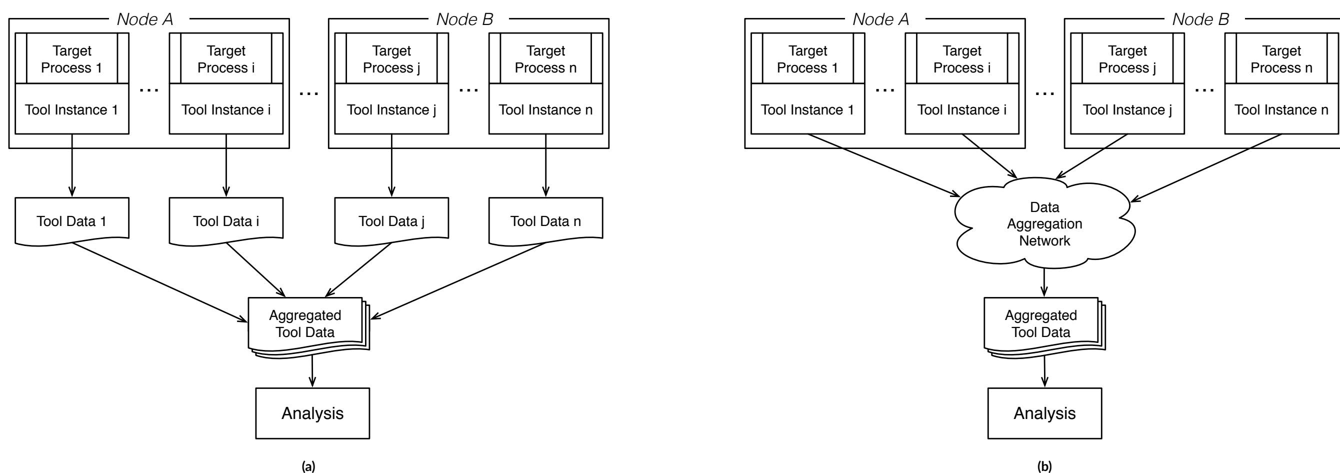


FIGURE 1 On the left, a typical offline tool architecture where analysis probes start with the application and remain in place for the entirety of the application's execution. After all analysis data are written, they are then read, aggregated, and finally analyzed by a separate tool. On the right, the tool architecture we adopted, which bears many similarities to its counterpart on the left. The key difference is that tool data aggregation is parallelized using the job's resources (using MPI).

memnesia trace data are stored in a straightforward on-disk representation made up of records containing three fields:

LISTING 2 A code snippet showing memnesia instrumentation of `MPI_Barrier()`.

```
int MPI_Barrier(MPI_Comm comm) {
    int rc = MPI_ERR_UNKNOWN;
    {
        // Constructor collects /proc/self/smaps sample.
        memnesia_scoped_caliper caliper(MEMNESIA_FUNC);
        // Execute barrier on behalf of the application.
        rc = PMPI_Barrier(comm);
    }
    // caliper's destructor collects another smaps sample.
    return rc;
}
```

- *trigger* (8-bit integer): name (ID) of the function triggering data collection.
- *time* (float): data collection time relative to when the message-passing library was initialized.
- *usage* (float): observed memory usage that is calculated by summing the constituent PSS entries in *smaps*, while ignoring those associated with our instrumentation library—an enhancement included to improve the accuracy of our reported statistics.

From those data, component-level metrics can be obtained readily: total application memory usage $m(t)$ (that of the application driver and MPI library) at time t is equal to the *smaps* usage u reported at that point, i.e., $m(t) = u$. MPI library usage $\hat{m}(t)$ at time t is determined by summing all preceding usage deltas (Equation 1 and Figure 3)—the intuition is that there is a causal relationship between MPI library calls and any observed usage deltas (positive or negative), since the MPI library was the only software component executing between data collection points. With these values, an application driver's memory can trivially be calculated as the difference between total memory usage and MPI library memory usage.

$$\hat{m}(t_j) = \sum_{i=0}^{i<\lfloor j+1/2 \rfloor} \Delta_i, \; \Delta_i = u_{2i+1} - u_{2i} \tag{1}$$

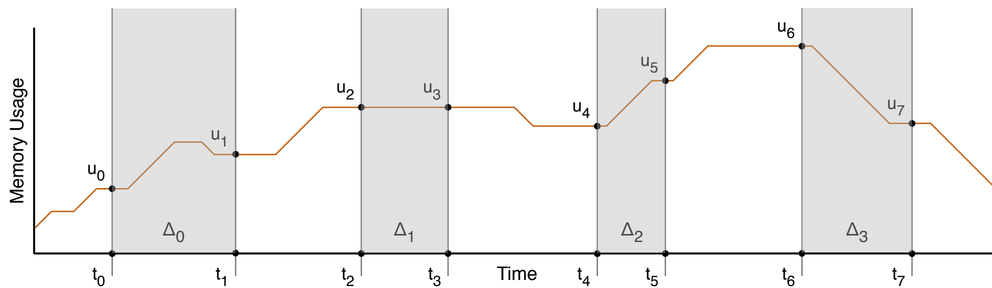


FIGURE 3 Illustration depicting single-process memory usage and data collection points.

In summary, our approach overcomes virtually all the shortcomings of previous methods, though our current reference implementation has limitations: 1. memnesia requires OS support for certain *procfs* features and C entry points into the MPI library, accessed through dynamic linkage, and 2. PSS reporting for applications that use *hugepages* (18, 19) is not currently supported.

4 | APPLICATION DRIVERS: MICRO-BENCHMARKS AND PROXY APPLICATIONS

Computational benchmarks are commonly used to assess and compare the performance of various workloads on differing software (e.g., library, OS) and hardware (e.g., processor, network) configurations. Individual benchmarks may be designed to exhibit a minimal set of behaviors to enable precise characterization of specific hardware or software mechanisms—so-called *micro-benchmarks*. At the other end of the spectrum, the net performance characteristics of whole real-world applications on a range of inputs may be the atomic units of observation. While the latter can calibrate expectations for the applications tested, such benchmarking may be expensive in terms of resources consumed and time to result and may produce results not generalizable to other inputs or applications. As a middle ground, *proxy applications* have become established as useful tools. The proxy is intended to be a software construction that is somehow representative of a larger application (or some components of a larger

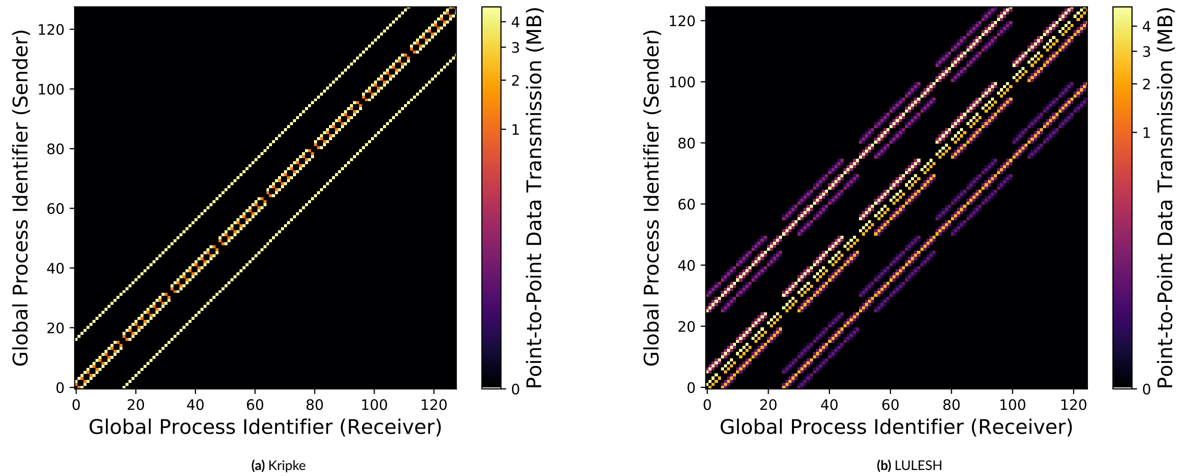


FIGURE 4 Visualization of point-to-point communication structure formed by each proxy application used in this study. Colors are mapped to data transfer totals between MPI processes (send/receive pairs) using point-to-point communication operations.

application) in terms of algorithmic structure (perhaps for the purpose of rapid prototyping) or computational and communication behavior (for benchmarking). For this work, we use two well-known proxy applications with memory usage and communication patterns typical to many HPC programs: LULESH (20) and Kripke (21).

4.1 | Micro-Benchmarks

Trivial and Alltoall: Micro-benchmarks that are meant to represent extreme ends of the *in-band* (i.e., application-driven) communication spectrum. Our *Trivial* benchmark calls `MPI_Init()` and then immediately calls `MPI_Finalize()`, thereby representing the most trivial of all MPI applications, i.e., one with no communication. We study the trivial case to understand an MPI library's minimum required memory footprint for parallel application lash-up. *Alltoall*, by contrast, is meant to represent applications that impose the most stressful (from the MPI library's perspective) communication workload: an all-to-all communication pattern where data are exchanged between every pair of processes. In particular, this program executes `MPI_Alltoall()` over `MPI_COMM_WORLD` in a loop, alternating between per-process message sizes of 2 kB and 4 MB. For each iteration of the loop, new communication buffers are allocated before the all-to-all data exchange and then freed after its completion. This communication workload is used to study the memory efficiency of run-time metadata structures with regard to memory registration and connection management.

Multiple Bandwidth/Message Rate: The OSU multiple bandwidth/message rate test measures aggregate uni-directional bandwidth between multiple pairs of processes using MPI (22). The purpose of this micro-benchmark is to quantify achieved bandwidth and message rates between a configurable number of processes concurrently imposing a communication workload on a system.

4.2 | Structured Grids: Kripke

Kripke is a proxy application developed at Lawrence Livermore National Laboratory, designed to be a proxy for a fully functional discrete-ordinates (Sn) 3D deterministic particle transport code (21). Figure 4a shows the point-to-point communication structure formed by this application.

4.3 | Unstructured Grids: LULESH

LULESH, the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics code, is designed to be a proxy for a fully functional Lagrangian hydrodynamics application on unstructured grids (20). It is in fact a family of implementations specialized to various programming models and hardware platforms and expressed in multiple programming languages. Our work uses the C++ MPI port (23). Figure 4b shows the communication structure formed by this proxy application.

	Trinitite	Snow
Model	Cray XC40	Penguin Computing
# Nodes	100	368
OS	Cray Linux Environment	TOSS/CHAOS (24) (Linux)
CPU	$2\times$ 16-core Intel Haswell E5–2698 v3 2.3 GHz	$2\times$ 18-core Intel Xeon E5–2695 v4 2.1 GHz
RAM	128 GB	128 GB
Network	Aries NICs (25), Dragonfly topology (26)	Intel OmniPath
MPI	Cray MPICH 7.6.2	Open MPI 1.10.5, MVAPICH2 2.2
Compiler	Intel 17.0.4	GCC 5.3.0

TABLE 3 An overview of hardware and software used for this study.

5 | RESULTS

In this section, we present and discuss our results gathered using our profiling and analysis infrastructure. We first discuss our tool's capabilities and the resulting insight into how memory is allocated as a function of run time, scale, and workload. Further, we show how memnesia is able to capture features particular to a workload instance, namely those related to data structure management, message protocol, and communication pattern, all at a per-process and per-software-component level—a capability that is not readily available today through other means.

5.1 | Experimental Setup

Performance results were gathered from the Trinitite and Snow systems located at Los Alamos National Laboratory, detailed in Table 3. Data were collected during regular operating hours, so the systems were servicing other workloads alongside, but in isolation from, ours. For each study in this section, experiments were executed in succession on a single set of dedicated hardware resources. Our experiments used weak scaling such that each process was given a fixed problem size.

5.2 | Memory Usage Timelines

Figure 5 shows per-process memory usage for two proxy applications running atop different MPI implementations on Snow. Results shown are from 100-cycle runs of LULESH (96^3 elements per process) and 50-cycle runs of Kripke (16^3 zones per process)—small-scale configurations meant to showcase our tool's analysis and reporting capabilities. The left column shows the evolution of MPI library memory usage (in isolation from the application driver's) over time and highlights how different communication substrates and workloads, shown in Figure 4, influence usage features. Similarly, the right column shows total memory usage (i.e., application and MPI library). Here we can see that both applications share a similar memory usage pattern: simulation state data dominate overall usage and remain relatively constant throughout their execution. Figure 6 shows aggregate (i.e., *summed*) memory usage reports for three workloads run at 216 processes on Trinitite, where for LULESH and Kripke we use the same per-process configurations as before. Notice that our tool can capture application-specific data structure management features, for example, the regular oscillatory behavior exhibited in our Alltoall benchmark. We omit aggregate memory usage plots at other processor counts because the weak-scaled simulation state data dominates overall memory usage, so additional plots would look similar to the ones provided, only differing by some process scaling factor.

5.3 | Peak Memory Usage

In this section, we study how job scale, workload, and runtime implementation influence per-process peak memory usage for MPI libraries and, whenever possible, compare results gathered from Cray MPICH's internal usage monitoring (craymem) to results reported by memnesia using two different *smaps* metrics: RSS and PSS. Unless otherwise noted, memnesia's reporting is based on proportional set size. We compare RSS and PSS metrics to highlight the differences between the two because of memory page sharing. RSS tends to be a more pessimistic, and oftentimes inaccurate, metric because memory usage attributed to shared pages is counted multiple times; the multiplier in our case is the number of MPI processes sharing pages on a node. Table 4 shows peak memory usage averaged over MPI processes for four different workloads; the reported error for each entry represents the standard deviation across the peak memory usage reported for each process. Since craymem reports three

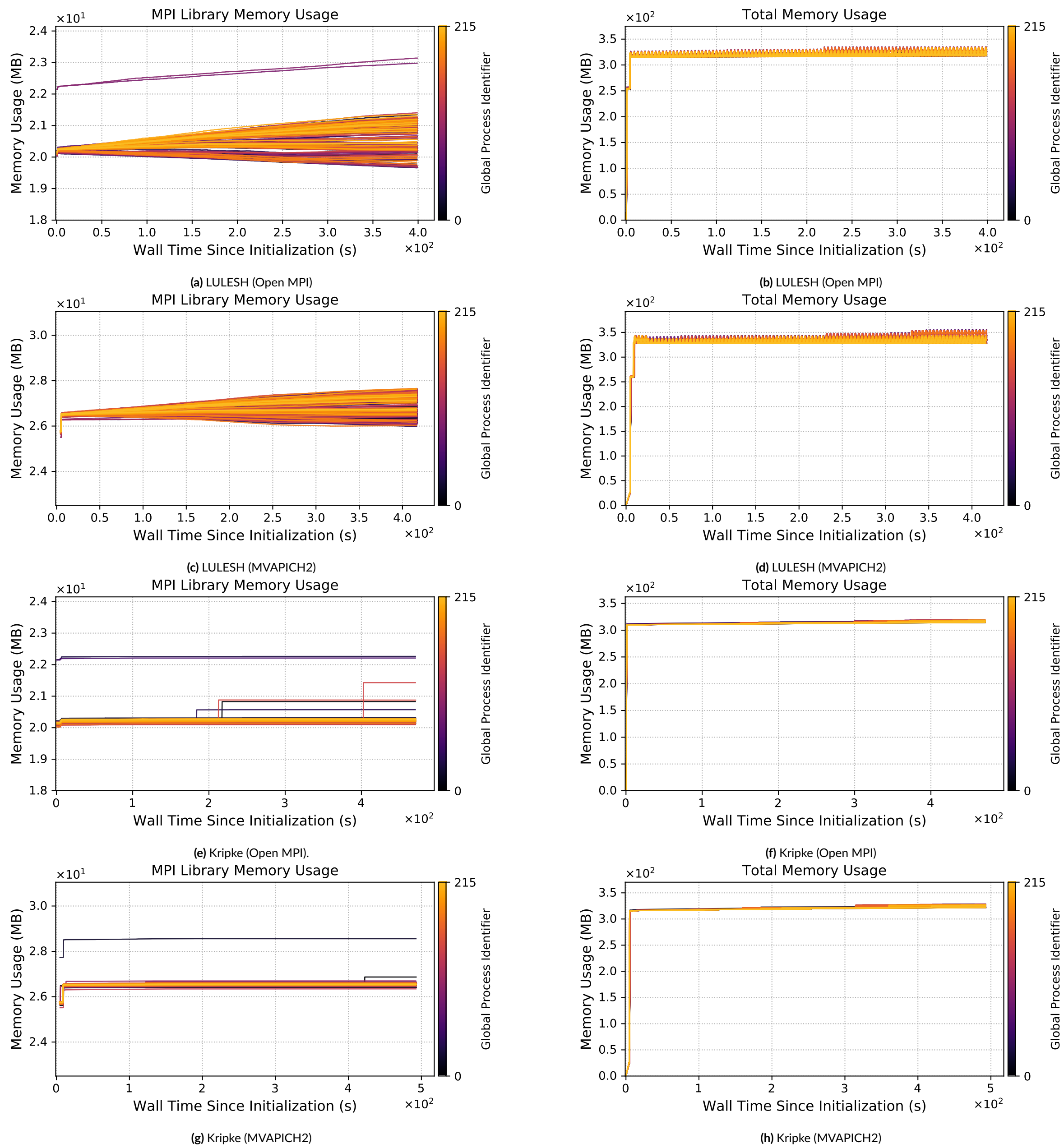
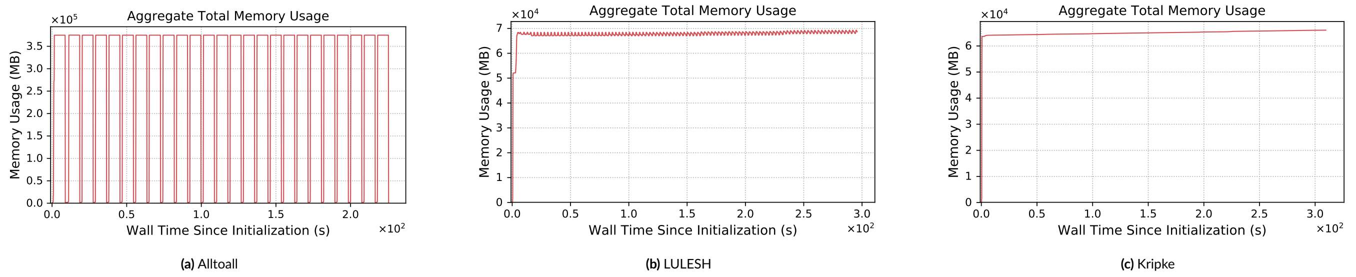


FIGURE 5 Tool output showing per-process memory usage over time. Colors are mapped to a process's `MPI_COMM_WORLD` rank.

memory usage components (`malloc()`, `mmap()`, and `shmget()`) without providing a corresponding time component (i.e., when they occurred relative to one another), we simply sum those values for reporting.

Our results show that job size, communication workload, and hardware/software architecture influence peak runtime memory usage, as indicated by craymem and memnesia (RSS and PSS) reporting (Table 4 and Table 5). Memory usage spikes observed at 64 processes are caused by crossing a compute node boundary—our experiments run a maximum of 32 processes per node. Of the workloads, Trivial achieves the lowest peak usage, while the proxy applications tend to yield the highest. The large standard deviations observed in the craymem data are due to large variations in per-node `shmget()` usage reporting: a single process reports non-zero usage, while the rest report zero. While valid, attributing shared-memory usage in this way is inaccurate for cooperative parallel workloads.



(a) Alltoall **(b)** LULESH **(c)** Kripke

FIGURE 6 memnesia timelines showing aggregate memory usage over time from 216-process (six-node) runs on Trinitite.

		Number of MPI Processes					
		1	8	27	64	125	216
Trivial	craymem	4.4 ± 0.0	4.9 ± 12.6	6.2 ± 29.4	19.1 ± 33.4	19.1 ± 32.7	23.4 ± 36.9
	memnesia-rss	2.2 ± 0.0	2.4 ± 0.0	2.6 ± 0.0	6.6 ± 0.2	6.7 ± 0.3	6.8 ± 0.4
	memnesia-pss	2.2 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	4.6 ± 0.2	4.7 ± 0.3	4.8 ± 0.4
Alltoall	craymem	4.4 ± 0.0	4.9 ± 12.6	6.4 ± 29.2	20.2 ± 33.3	20.2 ± 32.5	24.6 ± 36.7
	memnesia-rss	2.2 ± 0.0	3.0 ± 0.3	4.4 ± 0.0	12.7 ± 0.4	11.6 ± 0.5	11.2 ± 0.5
	memnesia-pss	2.2 ± 0.0	1.1 ± 0.1	2.7 ± 0.0	5.9 ± 0.2	6.0 ± 0.3	5.9 ± 0.4
Lulesh	craymem	4.4 ± 0.0	5.9 ± 12.6	7.3 ± 29.2	20.2 ± 33.3	20.2 ± 32.5	24.5 ± 36.7
	memnesia-rss	2.4 ± 0.0	5.0 ± 0.2	6.5 ± 0.8	10.4 ± 1.2	10.8 ± 0.8	11.1 ± 0.7
	memnesia-pss	2.4 ± 0.0	2.0 ± 0.1	2.3 ± 0.1	6.3 ± 0.7	6.9 ± 0.1	6.9 ± 0.1
Kripke	craymem	4.4 ± 0.0	5.9 ± 12.6	7.2 ± 29.2	20.1 ± 33.3	20.1 ± 32.5	24.4 ± 36.7
	memnesia-rss	2.4 ± 0.0	3.7 ± 0.1	4.3 ± 0.2	8.0 ± 0.2	7.9 ± 0.3	8.0 ± 0.4
	memnesia-pss	2.3 ± 0.0	1.9 ± 0.0	2.1 ± 0.1	5.6 ± 0.2	5.7 ± 0.3	5.8 ± 0.4

TABLE 4 Average reported peak memory consumption (in MB) on Trinitite.

		Number of MPI Processes					
		1	8	27	64	125	216
Trivial	snow-ompi	7.6 ± 0.0	6.5 ± 0.0	6.4 ± 0.0	19.8 ± 0.0	19.9 ± 0.0	20.1 ± 0.0
	snow-mvapich2	7.9 ± 0.0	10.9 ± 0.1	11.0 ± 0.1	24.5 ± 0.0	25.0 ± 0.2	25.7 ± 0.0
	tt-mpich	2.2 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	4.6 ± 0.2	4.7 ± 0.3	4.8 ± 0.4
Alltoall	snow-ompi	7.6 ± 0.0	6.6 ± 0.0	6.5 ± 0.4	19.8 ± 0.0	19.9 ± 0.0	20.2 ± 0.2
	snow-mvapich2	8.0 ± 0.0	11.0 ± 0.1	11.0 ± 0.1	24.6 ± 0.3	25.0 ± 0.1	25.8 ± 0.0
	tt-mpich	2.2 ± 0.0	1.1 ± 0.1	2.7 ± 0.0	5.9 ± 0.2	6.0 ± 0.3	5.9 ± 0.4
Lulesh	snow-ompi	7.6 ± 0.0	7.1 ± 0.5	7.3 ± 0.7	20.6 ± 0.8	20.6 ± 0.6	20.8 ± 0.4
	snow-mvapich2	8.0 ± 0.0	12.3 ± 0.4	12.4 ± 0.7	26.0 ± 0.7	26.3 ± 0.6	27.0 ± 0.4
	tt-mpich	2.4 ± 0.0	2.0 ± 0.1	2.3 ± 0.1	6.3 ± 0.7	6.9 ± 0.1	6.9 ± 0.1
Kripke	snow-ompi	7.6 ± 0.0	6.6 ± 0.0	6.5 ± 0.0	19.8 ± 0.0	20.0 ± 0.3	20.3 ± 0.2
	snow-mvapich2	8.0 ± 0.0	11.8 ± 0.1	11.8 ± 0.0	25.3 ± 0.0	25.8 ± 0.1	26.6 ± 0.1
	tt-mpich	2.3 ± 0.0	1.9 ± 0.0	2.1 ± 0.1	5.6 ± 0.2	5.7 ± 0.3	5.8 ± 0.4

TABLE 5 Average peak memory consumption (in MB) on Trinitite (tt) and Snow as reported by memnesia.

5.4 | Tool-Induced Application Overhead

To quantify tool-induced overhead, we study how two performance metrics commonly used to assess message-passing systems, message rate and bandwidth, are affected while under memnesia supervision. This is accomplished by running the previously described multiple bandwidth/message rate micro-benchmark in both the presence and absence of memnesia instrumentation, where the latter serves as our performance baseline. Data were collected on Snow using Open MPI 1.10.5 over three different job sizes, plotted in Figure 7. We then conclude with an analysis of memnesia's memory overhead and application perturbation.

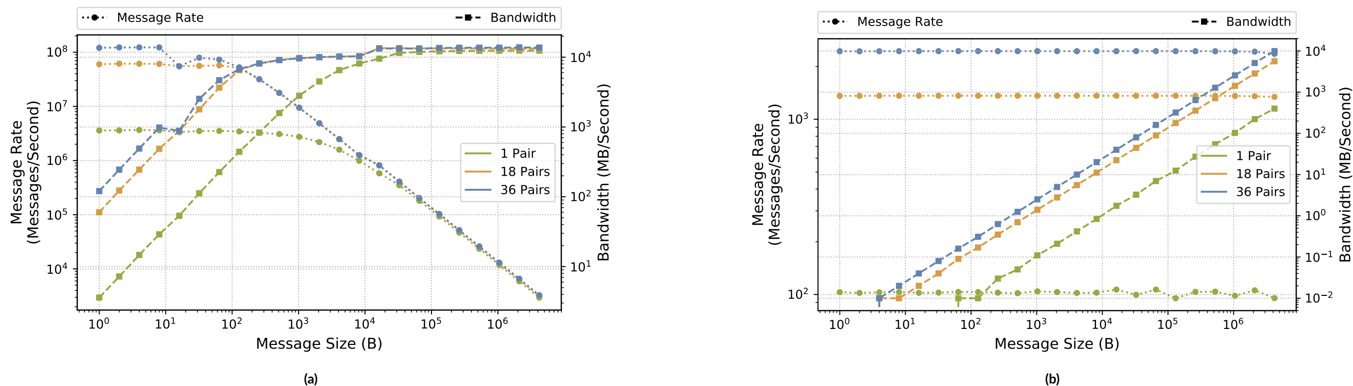


FIGURE 7 Results from the OSU multiple bandwidth/multiple message rate micro-benchmark, where the number of send/receive pairs vary. Figure a shows our performance baseline, while Figure b shows performance results with memnesia instrumentation enabled, both plotted using a log-log scale.

5.4.1 | Effects on Message Rate and Bandwidth

Across the board, memnesia overheads are most apparent at small message sizes, where its effect on operational latencies dominates messaging rates in messages per second (MPS). Notice that our performance baselines have typical messaging rate curves where small message transfers yield the highest rates (3.6×10^6, 6.0×10^7, and 1.2×10^8 MPS for 1 B payloads at 1, 18, and 36 send/receive pairs, respectively)—decreasing steadily from there as message size increases. In contrast, with memnesia supervision message rates appear to be capped and remain constant irrespective of message payload size, yielding message rates of approximately 1.0×10^2, 1.4×10^3, and 2.4×10^3 MPS across all payload sizes. This is caused by the collection of two *smaps* samples for each call into the MPI library, thereby increasing latency and therefore negatively affecting message rate.

Large-message bandwidth is least affected by the presence of memnesia instrumentation because increased operational latencies are amortized over the transfer of larger payloads. That is, once a transfer is initiated, memnesia instrumentation has no appreciable effect on transfer rate. This micro-benchmark represents a worst-case scenario; still, memnesia can be useful in practice even though relative differences shown here are large. Scientific applications tend not to be rate bound by small messages, which is the metric that is most severely degraded by the use of memnesia.

5.4.2 | Memory Overhead and Perturbation

As argued in Section 3, gathering accurate component-level memory usage statistics is difficult. Because our memory profiler is loaded into the target application binary at startup, it becomes an additional application component and is therefore subject to the memory attribution problem, so we describe its overheads analytically. The aggregate memory overhead of our current reference implementation can be calculated as follows. Given m MPI processes under memnesia supervision, let n_p be the total number of trace events triggered by process $p, 0 \leq p < m$, where n_p equals the total number of times p called into the MPI library. For each trace event, two records are collected and subsequently stored, as detailed in Section 3.1.1. So total tool-induced memory overhead given m processes is proportional to $2s\sum_{i=0}^{m-1} n_i$, where s is a constant representing the size of a single trace record in bytes. Each trace record contains four entries: an 8-bit identifier naming the function that triggered data collection, two double-precision floating-point values storing timing information (start time and duration), and a 64-bit integer storing memory usage in kilobytes. Assuming 64-bit double-precision floating-point values, $s = 25$ B.

As previously described, our memory profiler is loaded into the target application binary at startup via the runtime loader. Consequently, memnesia's presence perturbs application heap behavior through its use of dynamic memory management (allocations and deallocations). For a single process, the primary unit of observation, the amount of tool-induced application perturbation is proportional to s times the number of trace records already collected by memnesia.

6 | DISCUSSION AND FUTURE WORK

Even though storage requirements for memnesia trace records are relatively small, structural improvements can be made to reduce their size, thereby decreasing overall tool-induced memory overhead and application perturbation. Because of the way our profiler is introduced into the application, tool-induced memory exhaustion manifests as an application runtime error that ultimately results in parallel job termination. The current implementation of memnesia requires calling `MPI_Finalize()` to flush memory usage statistics to disk for later analysis by other programs. This

requirement is potentially problematic for long-running MPI applications because the amount of memory consumed by the tool grows without bound. A straightforward solution to limit memnesia's memory usage might include the use of `MPI_Pcontrol()`, which allows for a standard, user-accessible interface for controlling when tool data checkpoints are performed (3). This capability, in turn, could allow for user-defined *analysis extents*, trace data collected over a user-defined time span, that may then be used to attribute memory usage to specific application phases.

7 | CONCLUSION

This work addresses the need for an easy to use, reasonably general, open source, and minimally intrusive tool for attributing dynamic memory usage to individual libraries comprising a distributed memory application. Our technique is able to capture features particular to a workload instance at a per-process and per-software-component level, a capability that is not readily available today through other means. The key techniques are function interposition and accurate memory map analysis. Our case study is MPI, addressing the growing need to understand and control the memory footprint of HPC applications on memory-constrained hardware. MPI already provides an interposition layer in the form of PMPI, obviating the need to create one for an arbitrary library via a mechanism such as `LD_PRELOAD`. Our results show that job size, communication workload, and hardware/software architecture influence peak runtime memory usage. As an example, our experimental results show that different popular MPI implementations exhibit different memory usage behaviors, and such information could influence the choice of MPI implementation by application developers or users, and could also be of use to both MPI implementers and application/library developers to guide memory-use optimization of their implementations.

ACKNOWLEDGMENTS

A special thanks goes to Robert F. Bird and the anonymous reviewers for their helpful comments. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the U.S. Department of Energy under contract DE-AC52–06NA25396.

References

[1] Top500.org . Top500 Supercomputing Sites http://www.top500.org; 2018.

[2] S. K. Gutiérrez . memnesia: MPI Memory Consumption Analysis Utilities. https://github.com/hpc/mpimemu/tree/master/memnesia; Los Alamos National Laboratory; 2017.

[3] Message Passing Interface Forum . MPI: A Message-Passing Interface Standard Version 3.1 http://www.mpi-forum.org; 2017.

[4] Gabriel Edgar, Fagg Graham E., Bosilca George, et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: 11th European PVM/MPI Users' Group Meeting; 2004; Budapest, Hungary.

[5] Gropp William, Lusk Ewing, Doss Nathan, Skjellum Anthony. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*. 1996; 22(6):789–828.

[6] *ptrace (2) Linux User's Manual.* 2018.

[7] Roth Philip C., Arnold Dorian C., Miller Barton. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing; 2003.

[8] Curry Timothy W., et al . Profiling and Tracing Dynamic Library Usage Via Interposition. In: :267–278 USENIX Summer; 1994.

[9] Swift Michael M., Bershad Brian N., Levy Henry M.. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems (TOCS)*. 2005;23(1):77–110.

[10] *proc (5) Linux User's Manual.* 2015.

[11] Nethercote Nicholas, Seward Julian. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: no. 6:89–100 ACM Sigplan Notices; 2007.

[12] Google . Google Performance Tools—gperftools https://github.com/gperftools/gperftools; 2017.

[13] Chambreau Chris. memP: Parallel Heap Profiling http://memp.sourceforge.net; 2010.

[14] Wyckoff Pete, Wu Jiesheng. Memory Registration Caching Correctness. In: no. 2:1008–1015 IEEE International Symposium on Cluster Computing and the Grid; 2005.

[15] The smem authors . smem memory reporting tool. https://www.selenic.com/smem; 2017.

[16] Pritchard Howard, Gorodetsky Igor, Buntinas Darius. A uGNI-Based MPICH2 Nemesis Network Module for the Cray XE. In: Proceedings of the European MPI Users' Group Meeting; 2011.

[17] S. K. Gutiérrez . A Memory Consumption Benchmark for MPI Implementations. https://github.com/hpc/mpimemu; Los Alamos National Laboratory; 2017.

[18] *libhugetlbfs (7) Linux User's Manual.* 2008.

[19] *ld.hugetlbfs (1) Linux User's Manual.* 2012.

[20] Karlin Ian, Keasler Jeff, Neely Rob. LULESH 2.0 Updates and Changes. *LLNL-TR-641973 Technical Report*. 2013;:1–9.

[21] Kunen Adam J., Bailey Teresa S., Brown Peter N.. *Kripke—A Massively Parallel Transport Mini-App*. United States. Department of Energy.; 2015.

[22] The Ohio State University . MVAPICH Benchmarks http://mvapich.cse.ohio-state.edu/benchmarks; 2017.

[23] Karlin Ian, Bhatele Abhinav, Chamberlain Bradford L., et al. LULESH Programming Model and Performance Ports Overview. *LLNL-TR-608824 Technical Report*. 2012;:1–17.

[24] Braby Ryan L, Garlick Jim E, Goldstone Robin J. Achieving Order Through CHAOS: the LLNL HPC Linux Cluster Experience. In: The 4th International Conference on Linux Clusters: The HPC Revolution; 2003.

[25] Alverson, Bob and Froese, Edwin and Kaplan, Larry and Roweth, Duncan . Cray XC Series Network. *Cray Inc., White Paper*. 2012; WP-Aries01–1112.

[26] Kim John, Dally Wiliam J, Scott Steve, Abts Dennis. Technology-Driven, Highly-Scalable Dragonfly Topology. In: IEEE Computer Society; 2008.