Heterogeneous Memory and Arena-Based Heap Allocation

Sean Williams
New Mexico Consortium
swilliams@newmexicoconsortium.org

Latchesar Ionkov
Los Alamos National Laboratory
lionkov@lanl.gov

Michael Lang
Los Alamos National Laboratory
mlang@lanl.gov

Jason Lee
Los Alamos National Laboratory
jasonlee@lanl.gov

ABSTRACT

Nonuniform Memory Access (NUMA) will likely continue to be the chief abstraction used to expose heterogeneous memory. One major problem with using NUMA in this way is, the assignment of memory to devices, mediated by the hardware and Linux OS, is only resolved to page granularity. That is, pages, not allocations, are explicitly assigned to memory devices. This is particularly troublesome if one wants to migrate data between devices: since only pages can be migrated, other data allocated on the same pages will be migrated as well, and it isn't easy to tell what data will be swept along to the target device. We propose a solution to this problem based on repurposing arena-based heap management to keep locality among related data structures that are used together, and discuss our work on such a heap manager.

ACM Reference format:
Sean Williams, Latchesar Ionkov, Michael Lang, and Jason Lee. 2018. Heterogeneous Memory and Arena-Based Heap Allocation. In *Proceedings of MCHPC'18: Workshop on Memory Centric High Performance Computing, Dallas, TX, USA, November 11, 2018 (MCHPC'18),* 5 pages.
DOI: 10.1145/3286475.3286568

1 INTRODUCTION

There have been some steps recently to incorporate heterogeneous memory into high-performance computing. Most prominently, the now-defunct Intel Knights Landing [4] included integrated higher-bandwidth memory. Likewise, Nvidia is working to unify CPU and GPU memory with its NVLink[8] fabric and include higher-bandwidth memory, and Intel and others are bringing non-volatile memory to DIMM slots in order to have a higher-capacity, lower-performance option that is integrated into the memory address space.

The Intel Knights Landing exposes its high-bandwidth memory to the user as a NUMA node as do the IBM CORAL [7] systems. This makes perfect sense, since NUMA is a preexisting facility for bridging the gap between physical and virtual memory—in principle, virtual memory removes the need to care about physical devices. Thus, we expect that NUMA will continue to be used as the first-order abstraction for heterogeneous memory. In the past NUMA distance has just been representative of the "hop" distance between a CPU and a memory node. With high-bandwidth memory the NUMA distance is being used as a way to differentiate types of heterogeneous memory. Nevertheless, both multisocket machines (the traditional NUMA use case) and heterogeneous memory reassert the importance of knowing and specifying a physical home for a page of memory.

In Linux, users can interact with NUMA-informed placement via memory policies. The chief policies are, attach this page to the closest NUMA node; try to attach this page to a specific node, and if it's full, attach to the closest node; and attach this page to a specific node, and fail if it's full. This situation opens up many both practical and theoretical problems of varying importance and tractability—there's a large conceptual gap between these simple policies and the actual use cases of heterogeneous memory. This paper is about one of the simpler, more practical ones: that all these policies operate at page granularity.

The canonical bridge between pages and data structures is the heap manager, which dices up pages in such a way as to balance efficient utilization of memory with the performance of the allocator. But part of the point of heterogeneous memory is the fundamental trade-off between speed and capacity, so one would expect optimal use of such memory systems to involve a lot of churn. On the other hand, since memory can only be moved between NUMA nodes in page-sized chunks, an obvious problem arises: what's good for the data structure may not be good for the page. If a page contains both "hot" and "cool" data (i.e., data that would benefit from residing in high-performance memory, along with data that wouldn't), then it's never clear what the right choice is: should the page be on a high-performance device, wasting some of this precious resource, or should it be on a normal-performance device, to make more efficient use of limited space?

There's a potential answer in the concept of *arenas*, in which a heap manager maintains multiple heaps. This was traditionally done to reduce lock contention: heaps require substantial bookkeeping, and the associated data structures can need updating when servicing `malloc` and `free` calls. Having one or more heaps per thread can reduce or eliminate contention for each heap's bookkeeping data.

A major drawback to the current situation, where pages are tied to NUMA nodes, is that it can be hard to assess the value of transferring a page to or from a high-performance memory device. Our proposal is that, rather than using arenas for the purpose of reducing lock contention, arenas be used to group data structures that should "travel together." Under this proposal, the assessment is not, "is it a good idea to transfer this data structure"? but, "is it a good idea to transfer this whole arena"? The remainder of this paper

MCHPC'18, Dallas, TX, USA

978-1-4503-6113-2/18/11…$15.00
DOI: 10.1145/3286475.3286568

will be devoted to discussing various aspects, and pros and cons, of this approach. Additionally, we provide some use cases for our implementation and results from experiments.

2 APPROACH

2.1 Homogeneity

NUMA was designed for multisocket machines, so it is based on an assumption that the only difference between memory devices is latency. Thus, the default NUMA policy, in which allocations are preferentially located on the memory node nearest the allocating processor, makes perfect sense. This point is even baked into the name, nonuniform memory *access*: the assumption is that memory devices are homogeneous, and only their access is different.

With heterogeneous memory, it is typically the case that both the characteristics (e.g., performance) and the access of memory devices will be different. This makes it difficult to determine what the right allocation policy is and, whether there actually is a right allocation policy. Heterogeneity makes the problem of allocation placement far more difficult than the merely nonuniform-access situation.

In essence, we can reduce this problem to three components: mapping allocations to pages, mapping pages to policies, and mapping policies to devices.

2.2 Allocations → Pages: Arenas

At the operating system level, memory is only available to page granularity, via the `mmap` system call. As a consequence, memory can only be attached to devices (i.e., NUMA nodes) at page granularity. Of course, most data structure sizes are not multiples of a page size (4 KiB, 2 MiB, etc.), so we use heap managers (i.e., `malloc`) to pack data onto pages in a reasonably efficient way. But this is another situation where a traditional assumption of homogeneity becomes pernicious.

But what relationship should be established between pages, allocations, and devices? Intel's `memkind` [3] library allows one to create heaps associated with particular "kinds" of memory, e.g., high-bandwidth. The basic idea of `memkind`, that it's necessary to address the problem of matching allocations to heterogeneous pages using an arena-based heap allocator, is sound.

What role should arenas serve in solving this problem? In `memkind`, the intention seems to be that arenas enumerate the kinds of memory, so that one has a default arena, a high-bandwidth arena, a high-capacity arena, and so on.

As will be discussed in Section 2.3, we believe that the ability to move allocations between devices is an important capability. How does one move allocations? The `memcpy` function puts the allocation at a new address, so preexisting pointers to the allocation will continue to point to its old address. On the other hand, the `migrate_pages` system call preserves pointers, but as the name suggests, it only operates on whole pages.

This has the obvious consequence that, if one migrates an allocation, `migrate_pages` moves all pages containing that allocation. Since the exact behavior of `malloc` is hard to predict, this could cause arbitrarily bad data movement to occur: Imagine a data structure was involved in an intensive computation and was on high-performance memory, but that computation is over. In principle, these data should be moved back to normal memory. But it is unclear what else occupies that page and if there is other data on it that are still needed in high-performance memory.

The key dynamic here is, since data must be migrated in page-sized chunks, that arenas should be deployed based on assumptions about the coherence of data structures. If the programmer believes data structures A and B are typically operated on together, then they should be placed on the same page. If data structure C has nothing to do with A or B, then C should not be placed on a page with them.

What this means in practice is, the proposed API allows one to define any number of arenas, and select which arena any particular data structure is placed on. When one wants to migrate data to a new device, one migrates its entire arena. This has the effect of ensuring that migration occurs only with data structures that should migrate together, and excluding data structures that should not be swept along for the ride.

2.3 Pages → Policies: Modeling

The big problem of the previous section is, how does one decide when to move an arena, and to which device? This opens up a vast configuration space, which some "hero programmers" may use for obsessive optimization, but most people won't want to bother. One could imagine treating memory placement as a sort of numerical optimization problem, i.e., one could imagine constructing a model of where each allocation should reside as a function of time.

Consider the form of such a model: $f(\mathbf{A},t,\mathbf{I}) = (\cdots)$, where $\mathbf{A}$ is the set of allocations the program will make, t is time (i.e., the number of instructions previously executed), and $\mathbf{I}$ is an input deck. What is the cardinality of $\mathbf{A}$ for a particular program? In other words, how many allocations does a particular program make? We could simplify this question even further to, how many allocations are made by a call to `malloc` within a loop or recursive function? The answer to that question is well known; unfortunately, the answer is $\perp$, i.e., the value of undecidable computation. Thus, the cardinality of $\mathbf{A}$, the set of allocations made by a program, cannot be known in general, so this modeling exercise would seem to be off to a rough start.

If we instead pose our model in terms of arenas, then this problem goes away: the number of arenas is explicitly chosen by the programmer, so the cardinality of the set of arenas is not dependent on any particular run of the program, much less on all possible runs. Under this arrangement, therefore, we can indeed *pose* a model of memory placement by allowing the programmer to give the problem known bounds.

Notably, this present work only addresses the decidability of $\mathbf{A}$, which makes it possible to pose models. Whether those models will be any good likely hinges on the decidability of f overall, which is a tall order. We can certainly bound this problem by using knowledge of our HPC applications. By categorizing HPC applications we can come up with a set of ad-hoc methods that will improve performance of HPC applications in most cases.

2.4 Policies → Devices: Orderings

Intel's `memkind` allows one to allocate memory on "kinds" of devices, and one kind is high-bandwidth memory. How do its authors decide what constitutes a high-bandwidth NUMA node? Let us begin with a different question: what is NUMA distance?

Remember that NUMA was developed for homogeneous memory that has heterogeneous access. NUMA distance is, not surprisingly, a relative measure of latency. Among other things, this assumes homogeneous capacity, so it always makes sense to choose the lowest-latency memory—there are no trade-offs here.[1]

High-bandwidth memory does represent a trade-off: we can deduce this from the fact that one has even bothered to differentiate them, i.e., to make normal memory in addition to high-bandwidth memory. If high-bandwidth memory were strictly better, then system designers would do away with normal memory. The ordinary trade-off is capacity, so the designers were faced with a conundrum: if high-bandwidth memory is given a low NUMA distance, then allocations will default to it, and bandwidth-insensitive data will fill it up. Special memory requires special allocation, so high-bandwidth memory is given a high NUMA distance.

This means, in order to allocate high-bandwidth memory, one needs to know it's there. Which gets us back around to `memkind`: how does it know that a system has high-bandwidth memory? If you dig into the library, you will eventually find an inline assembly block with `cpuid`, followed by some bitwise operations on magic numbers. These identify whether the CPU family is Intel Knights Landing, and if it is, then high-bandwidth allocations are bound to the NUMA node with the magic distance of 31. Other architectures are not supported—though in Intel's defense, heterogeneous memory is still pretty exotic.

This discussion was intended to motivate the following one: how do we decide the properties of memory devices? Memory policy is the current hammer, and NUMA is the current nail, so the question for today is, how could we categorize NUMA nodes? This doesn't quite capture the reality of the situation, since we don't exactly need categories, but we need orderings. That is, if we want memory on a high-bandwidth device, well, there could be several devices with different bandwidths, some of which are high and some of which are not. The better question, therefore, is, how do we construct a bandwidth ranking of devices?

The first answer that will jump into most people's heads is empiricism. Bandwidth is measurable, so take measurements. This approach is problematic, as the central tenet of empirical approaches is that "incidental" observations are representative of a general phenomenon. This makes testing methodology an important issue, since (e.g.) bandwidth measurements are only meaningful if the testing protocol is representative of the conditions under which the memory will be used, size of access and stride of memory comes into play.

A more compelling answer would be to extend NUMA and/or ACPI to include standardized results (e.g., from well-conducted tests) of various memory metrics for the different NUMA nodes. Such a standard is well outside the scope of this paper, though it is something we would be interested in working on, given enough interest and collaborators.

A final, simple approach would be to require administrators of high-performance computers to maintain a configuration file listing standardized characteristics of the NUMA nodes. This is tractable (in principle) because the population of heterogeneous-memory high-performance computers in the world will likely remain small, and such a configuration only needs to be written at initial setup and following major upgrades. It would then also be up to the system administration to oversee a testing protocol, or else to rely on specifications or similar material from the hardware vendors.

In any case, this section remains speculative because of the simple fact that heterogeneous memory is itself largely speculative at this time. All we can do, therefore, is speculate about what the future may hold.

[1] In fact, there is one trade-off: one can theoretically get higher bandwidth and higher latency by interleaving memory across all NUMA nodes. This is the intention behind the "interleave" kernel policy, but it isn't used much in practice.

2.5 Shared Memory Arenas

The heap memory managers, as implemented at the moment, allow for memory waste due to fragmentation. Although the modern heap managers can be configured to aggressively return free pages to the operating system, the task is further complicated by the more complex data structures the managers use. In addition to the multiple arenas they create, the memory is further split into bins that try to group data of the same size, and extents that belong to the same bin.

In computers with uniform memory, while fragmentation is wasteful, the problem can usually be fixed by installing more DRAM. For high performance memory, like HBM, the size is usually fixed and memory waste becomes a bigger issue.

In the general case of computer use, the memory waste is considered a normal result of a hard optimization problem. Each process has its own resources and the processes generally don't trust each other. In the HPC environment the biggest memory users on a server usually belong to the same job, and while they might run as separate processes (for example, the normal case for MPI applications), they implicitly trust each other. This trust can be used for better memory utilization by creating a shared memory heap manager that can be used across multiple processes.

The proposed API with its arenas fit well with the implementation of a shared memory heap manager. The developer can choose which data to be in private arenas, handled by the local heap manager, and which to be shared with other processes from the same job.

In addition to the benefits of using shared memory heap managers, there are also some drawbacks. Sharing arenas might cause worse memory allocation performance due to lock contention. Data from multiple processes will be interspersed in the same arena, which will require careful handling of data movement. Bugs like buffer overflows can be harder to detect, as the bug could originate from a different process than the one it appears in.

3 IMPLEMENTATION

3.1 Arena Implementation

The arena API is implemented as part of the SICM [6] project. Currently it restricts an arena memory to belong to a single NUMA node. The main functions of the API are:

```
sicm_arena sicm_arena_create(size_t maxsize,
      sicm_device *dev)
int sicm_arena_set_device(sicm_arena sa,
      sicm_device *dev)
void *sicm_arena_alloc(sicm_arena sa, size_t sz)
void *sicm_realloc(void *ptr, size_t sz)
void sicm_free(void *ptr)
sicm_arena sicm_arena_lookup(void *ptr)
```

Upon arena creation, it is assigned to the specified `sicm_device`. If requested, the arena can be moved to another device by using the `sicm_arena_set_device` function. A maximum size of the arena can be specified. If the user tries to allocate more memory than that size, `sicm_arena_alloc` will return `NULL` pointer.

The arenas functionality is implemented as an extension to the jemalloc [5] arenas. The SICM library provides custom hooks to jemalloc that ensure the arena's extents use pages from the appropriate NUMA memory. They also keep track of the extents so the pages can be migrated to another node if requested. The page migration is implemented by using the `mbind` Linux system call. If not all extents for an arena can be migrated, `sicm_arena_set_device` returns an error.

3.2 Integration with Existing Middleware

The interface described in the preceding text is targeted for HPC runtimes and libraries. Advanced programmers could use it directly with in applications we see enabling access to heterogeneous memory for common runtimes such as OpenMP, MPI and Global Arrays, Legion, Charm++, etc. We have initial implementations for MPI, Global Arrays and are evaluating functionality for OpenMP.

Global Arrays[1] is an ideal candidate for integration with our API, having been written with both NUMA and shared memory in mind. What it does not have is the ability to actively choose memory devices or create arenas to use for its allocations. Our API will allow for Global Arrays to do so, which in turn will allow for better performance through better placement of data. Currently, Global Arrays has been modified so that its calls to `shm_open` are redirected to SICM. No selection of memory device is done beyond selecting the first device. The shared memory arenas used for Global Arrays depend on the pthreads support for shared memory mutexes.

Data placement in the context of MPI communications is of paramount importance to achieve high performance: high-performance data transfers over the network is only efficient if the data can be placed correctly in the memory hierarchy and ultimately efficiently accessible by MPI ranks or threads that need it. Unfortunately, the MPI standard and MPI implementations do not provide any mean or interface for the placement of data in complex memory hierarchies. In addition, the community agrees that it is beneficial to provide mechanisms to application developers so that they can express the intent related to the data transfer in order to better select where to store the data once the MPI operation completes.

Based on these constraints, we extended Open MPI to allow users to allocate memory by providing hints while using the existing `MPI_Mem_alloc` function. Our extension consists in the implementation of a new mpool component in the OPAL layer that interfaces with SICM.

Practically, application developers can express hints through the info structure that is passed in when using `MPI_Mem_alloc`. These hints can then be used to allocate memory arenas using SICM. At the moment, such hints includes specifying the need for high bandwidth memories or standard main memory but can easily be extended to other types of memory and others types of requirements from users. Finally, this approach has the huge benefit from not requiring any modifications to the MPI standards or Open MPI interfaces, while still giving control over memory allocation and data placement to users.

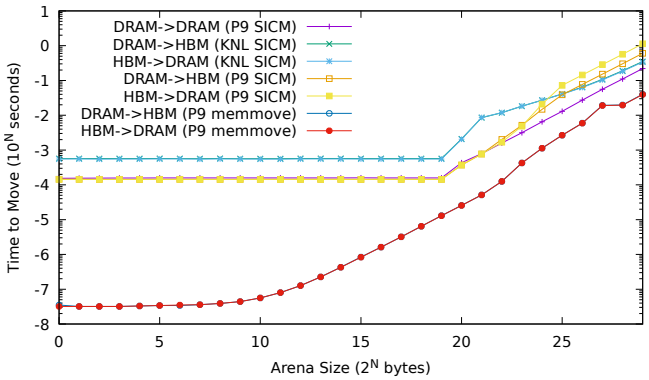

Figure 1: Time To Move Memory Between Devices

4 EXPERIMENTAL STUDY

4.1 Moving Memory Between Devices

Figure 1 shows comparison of the time to move an arena between different devices, depending on the size of the arena. We ran the experiments on two architectures that support heterogeneous memory. The IBM CORAL Power 9 machines have six active NUMA nodes, four with DRAM and two with the high bandwidth memory located on the Nvidia GPU. The Knight's Landing machines have two NUMA nodes, one with DRAM and one with high bandwidth memory. We tested moving arenas from one memory type to another. The DRAM-to-DRAM values show the movement from one DRAM NUMA node to another. Because the KNL machines have only one DRAM node, there is no DRAM-to-DRAM plot for it. Each datapoint represents the average time taken to move an arena with size 2^N bytes. For comparison, we also show the time it takes to move the data with `memmove`.

The results show that data migration is expensive. Moving 0.5 GB of data takes approximately a second. Therefore, arena migration to high performance memory makes sense only if the computational kernel that is using it runs long enough to amortize the cost of migration. Arena migration is much slower than moving the data with `memmove`. The main basis for the slowdown is migration uses the Linux kernel to transport the data to pages of different memory device while preserving the same addresses, while `memmove` doesn't. The initial analysis of the kernel shows that the code for the migration is not optimized for the task, and palpable future work would be to improve the speed of the arena migration.

4.2 VPIC

In order to quantify the usefulness of our API in real world applications, we compared using our API with using `malloc(3)` and `numactl(8)` in VPIC[2], a particle-in-cell simulation code for modeling kinetic plasmas in one, two, or three spatial dimensions.

Figure 2 shows the results of running VPIC with `malloc(3)` running normally, under the influence of `numactl(8) --preferred`, and replaced with our API. The rank count of 64 was chosen to be

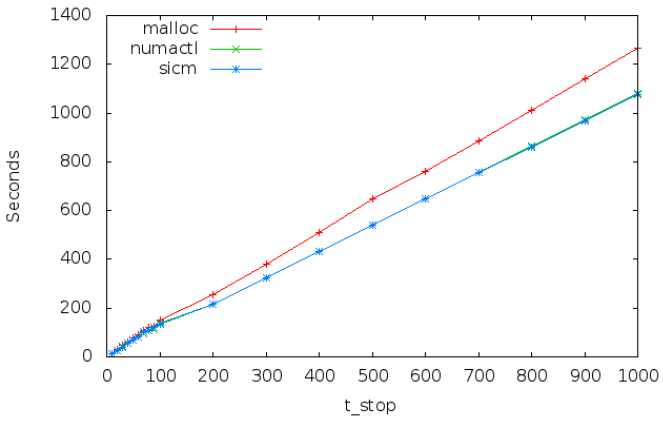

Figure 2: VPIC runtimes with different t_stop values, using `malloc(3)`, `numactl(8)`, and SICM. (64 ranks, nppc=25)

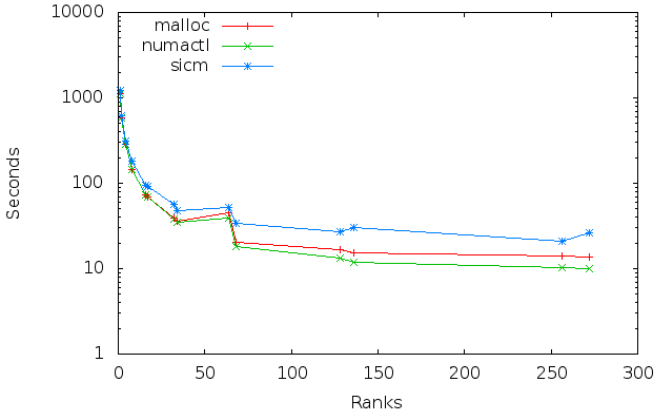

Figure 3: VPIC runtimes with different number Of ranks, using `malloc(3)`, `numactl(8)`, and SICM (with spilling).

small enough to run quickly, while large enough to represent a real problem. The value nppc was set to 25 in order to allow for the entirety of VPIC allocations to reside in high bandwidth memory. Using `malloc(3)` normally results in the longest runtimes of each run of VPIC. Using `numactl(8)` and our API results in lower runtimes. However, our SICM API results in runtimes that are slightly faster than with `numactl(8)`.

Figure 3 shows the results of running VPIC with provided fixed input decks. In these runs, the VPIC allocations were not always able to fit into high bandwidth memory, so a simple spilling function was added into VPIC to use DRAM arenas once high bandwidth memory was exhausted (once DRAM was chosen to be used, high bandwidth memory was not used again during a run). The results show that our API has higher overhead than both malloc(3) and numactl(8) when spilling is required.

5 CONCLUSION

Heterogeneity always presents a serious problem for computer scientists' preference for elegance, and heterogeneous memory is shaping up to be no different. Broadly speaking, we see the problem of heterogeneous memory as consisting of three major parts: controlling how allocations end up on pages, deciding how pages end up under policies, and specifying how policies correspond to actual devices.

The first problem, putting allocations on pages, we addressed through a redefining the meaning of allocator arenas. Under this scheme, we assume that data will move between devices as the program runs, and we give programmers a tool to handle the unintended consequences of page migration. We then argued that this view of arenas also dampens some of the undecidability of the problem of modeling the behavior of computer programs. Finally, we discussed the problems of interpreting memory policies, and presented a few solutions.

We described initial use of the SICM API, and showed preliminary results for micro-benchmarks and an application, VPIC.

Together, we believe this represents a complete "middleware" package for heterogeneous memory, so that we will be poised to tackle its issues once a major heterogeneous-memory supercomputer is built.

As future work, we are planning to extend the API to support arenas on multiple NUMA nodes, as well as a way to specify that an arena can be placed on any available memory, because its data is not in active use at the moment. Also, an asynchronous version of the arena migration will help improving the overall performance of the applications. An important task to look into is improving the performance of the `mbind` Linux implementation and closing the performance gap between `memmove` and the system call.

6 ACKNOWLEDGEMENT

The Simplified Interface to Complex Memory (SICM) is 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.

REFERENCES

[1] 2018. Global Arrays. (2018). https://github.com/GlobalArrays/ga
[2] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson. 2008. 0.374 Pflop/s Trillion-particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 63, 11 pages. http://dl.acm.org/citation.cfm?id=1413370.1413435
[3] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. 2015. *memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies.* Technical Report. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States).
[4] George Chrysos. 2014. Intel® Xeon Phi coprocessor-the architecture. *Intel Whitepaper* 176 (2014).
[5] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDCan Conference, Ottawa, Canada.*
[6] LANL. 2018. SICM – Simplified Interface to Complex Memory. (2018). https://github.com/lanl/SICM
[7] LLNL. 2018. CORAL/Sierra System. (2018). https://asc.llnl.gov/coral-info
[8] NVidia. 2018. NVidia NVLink High-Speed Interconnect. (2018). https://www.nvidia.com/en-us/data-center/nvlink/