SharP Unified Memory Allocator: An Intent-Based Memory Allocator for Extreme-Scale Systems

Ferrol Aderholdt[1(✉)], Manjunath Gorentla Venkata[1], and Zachary W. Parchman[2]

[1] Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
{aderholdtwf1,manjugv}@ornl.gov
[2] Tennessee Technological University, Cookeville, TN 38501, USA
zwparchman42@students.tntech.edu

Abstract. The pre-exascale systems will soon be deployed with a deep, complex memory hierarchy composed of many heterogeneous memories. This presents multiple challenges for users including: how to allocate data objects with locality between memories and devices for the various memories in these systems, which includes DRAM, High-bandwidth Memory (HBM), and non-volatile random access memory (NVRAM), and how to perform these allocations while providing portability for their application. Currently, the user can make use of multiple, disjoint libraries to allocate data objects on these memories. However, it is difficult to obtain locality between memories and devices when using libraries that are unaware of each other. This paper presents the *Unified Memory Allocator* (UMA) of the SHARed data-structure centric Programming abstraction (SharP) library, which provides a unified interface for memory allocations across DRAM, HBM, and NVRAM and is extensible to support future memory types. In addition, the SharP UMA allows for portability between systems by supporting both explicit and implicit, intent-based memory allocations. To demonstrate the ease of use of the SharP UMA, we have extended both *Open MPI* and *OpenSHMEM-X* to support SharP. We validate this work by evaluating the performance implications and intent-based approach with synthetic benchmarks as well as adaptations of the Graph500 benchmark.

F. Aderholdt—This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 533–545, 2018.
https://doi.org/10.1007/978-3-319-96983-1_38

1 Introduction

Many current extreme-scale systems have a deep, complex memory hierarchy composed of heterogeneous memories including DRAM and high-performance graphics DRAM. The memory hierarchy is becoming deeper with the inclusion of HBM and NVRAM in many current and soon to be deployed systems. With this trend of a deeper and more complex memory hierarchy continuing into the exascale era, it is important that users are able to achieve high-performance from each system executing their scientific or analytic applications.

Currently, for each memory and device in the system, there exists API calls and libraries capable of allocating data objects on their particular memories. These include libraries such as malloc in libc for DRAM, CudaMalloc in the CUDA library for HBM memory on Graphical Processing Units (GPUs), and the PMEM library for allocating memory on NVRAM. However, for each of these memories, the libraries are not knowledgeable of the other libraries in the system, which can create challenges for users who are attempting to obtain locality and affinity with their memory allocations.

These challenges and the deepening of the memory hierarchy have caused many in industry and the research community to develop new memory allocators capable of efficiently allocating memory on the newly included memories, such as the *memkind* allocator [3]. The memkind allocator is capable of allocating memory on both DRAM and HBM (i.e., MCDRAM on Intel KNL) and presents itself to the user as an extensible interface, which can support future heterogeneous memories. However, applications making use of *memkind* are not portable between systems as the allocations of data objects are completed in an explicit manner, which requires systems to have both identical memories and affinities between devices and memories.

To alleviate the *User* from needing architectural knowledge of the machine, with respect to allocating memory, the *User*'s intent could be captured and interpreted to perform the proper memory allocation. Capturing *User* intent is a challenging task. The question that needs to be answered is: *How do we abstract the system architecture from the user while still providing accurate memory allocations?* Abstracting the system while forcing the *User* to know latency and bandwidth characteristics of the underlying memory accomplishes little unless these are used as thresholds for acceptable performance.

This paper presents a higher-level approach to solving this challenge with the UMA of the SharP library [14]. The UMA is a unified memory allocator abstracting the memories of the system and the allocators for those memories. This is achieved through an internal, extensible interface that utilizes the excellent memory allocators for memories such as DRAM, HBM, and NVRAM including the memkind, CudaMalloc, and PMEM allocators. This allows the user to leverage existing allocators while having SharP coordinate memory allocations and provide data locality and affinity for the *User*.

The allocator is presented to the *User* through a single interface that abstracts the memories from the *User* such that the *User* can perform memory allocations with high-level *Hints* and *Constraints* that describe the user's intent,

enabling intent-based memory allocations. In addition to high-level *Hints* and *Constraints*, users with expert knowledge of the system may explicitly declare the memory their data is to be allocated on as a constraint to the SharP UMA.

This work makes the following contributions:

- We classify and design higher-level abstractions for *User*s to perform memory allocations on multiple memory types in the system while enabling data locality and affinity, which will reduce data movement.
- We design and implement the SharP UMA based on these higher-level abstractions and demonstrate their ease of use by extending both Open MPI and OpenSHMEM-X [1] to make use of this memory allocator.
- We demonstrate the effectiveness of this allocator with synthetic micro-benchmarks on multiple systems, demonstrating the portability of the approach, as well as porting the Graph500 benchmark to make use of our extended Open MPI and OpenSHMEM-X.

2 Related Work

There are two main areas of research related to this work. The first is the area of memory allocation, which has been thoroughly studied over many years for both single node and distributed allocations. The second area focuses on programming models that also use similar abstractions to provide portable memory allocation across various memories within the system. We will first discuss the area of memory allocators and then the abstractions enabling portable memory allocations and memory usage.

There have been many memory allocators developed over the past several years focused on providing simple interfaces for users to allocate memory. The majority of the earlier memory allocators such as Doug Lea's dlmalloc [11], GNU's malloc (ptmalloc) [8], Jason Evans' malloc (jemalloc) [6], and others [2]. In each of these allocators, the primary focus is on allocation performance and the reduction of fragmentation, as well as the elimination of false-sharing, through interfaces that leveraged arenas or thread specific memory pools inside their implementations. This allowed for thread-based allocations that remained lock-free resulting in higher-performance within the critical paths of execution and a reduction in false-sharing. Because of jemalloc's ability to perform fast allocations, it has been leveraged by other allocators such as the memkind [3] and PMEM memory allocators [9]. The memkind allocator is an extensible memory allocator that is designed to provide memory allocations on DRAM and HBM for the Intel Xeon Phi Knights Landing. It accomplishes this by providing interfaces for the user to create allocators for each memory kind in the system. If there are memories other than DRAM and HBM, the user must manually implement the underlying functionality for those memories. The PMEM memory allocator focuses specifically on persistent memory and provides multiple methods of allocating on these memories including: (i) memory-mapping a file in NVRAM and using jemalloc to provide memory allocations of that memory, (ii) treating the memory as a data object allowing the user to modify the object as they see fit

throughout execution, and (iii) giving the user a direct interface to treat the memory as if its virtual scratch memory with jemalloc.

With respect to abstracting the memories of the system and allowing a user to allocate memory in a portable fashion, there are multiple works focused on these areas including UNITY [10] and kokkos [5]. UNITY is a library that abstracts the memories of the system from the user allowing the user to consider only their data structures. The abstraction is done so the data objects allocated by the user are placed in memory and moved automatically based on usage and need. Kokkos similarly handles data placement for the user based on "traits" of memory, which are declared by the user in order to allocate memory appropriately.

Based on these works, the SharP UMA is different from the above works by not only abstracting the memories of the system like UNITY and kokkos, but also allow the user the place data in memories based on their intent. More clearly, the user can implicitly and explicitly allocate memory using the SharP UMA as well as being able to optimize their algorithms and data placement beyond the capabilities that are provided by our library.

3 Capturing User Intent

Many of the current extreme-scale systems are composed of CPUs, compute accelerators, and high-performing NICs. These architectures, while delivering high-performance, are often dissimilar to other systems with different affinities between devices, memories present, etc. With these differing architectures, it is difficult for research scientists to produce high-performing, portable implementations of their scientific algorithms because the implementation will have to be optimized for each system. With the increasing complexity of the memory hierarchy, the changes necessary to optimize an application will grow.

Capturing the user's intent could serve to lessen the changes required for an application moving from system to system and increase productivity for the application developer. The challenge of capturing user intent is determining the required granularity to provide a sufficient amount of performance portability. While the performance characteristics and programming of particular accelerators may require changes to an application when not using programming models such as OpenMP or OpenACC, we argue that users should not need to modify their application when moving from system to system due to architectural differences with respect to affinities and memory types. This is especially true as we move to systems with an increased heterogeneity of devices.

In general, there are two levels of granularity that could be used to capture user-intent. These levels include (i) lower-level characteristics and (ii) higher-level generalizations of the components of the system.

For (i), the lower-level characteristics of the memories used for the storage of data objects may include performance characteristics or device traits. This can be demonstrated by having the user specify that a particular data object should be allocated on a memory with a particular access latency or bandwidth. However, this requires the user to have a relatively high understanding of the memory technologies available in the system. Additionally, using specific constraints

on the latency and bandwidth of memories makes the assumption that memory technologies and their performance will be relatively static. Improving the latency and bandwidth characteristics of memory types could cause previously assumed values to be incorrect, resulting in an incorrectly behaving application or a failure at runtime.

In (ii), a higher-level granularity further abstracts the system allowing the user to know little about the underlying memory other than its general properties. For example, the user may wish to use HBM on a GPU for their computation, but not know the specific latency and bandwidth measurements of the HBM. By using a high-level hint, the user would still be able to ensure an allocation on the proper memory. However, high-level abstractions of the memory types can produce incorrect results without the coupling of multiple hints to help describe affinities to devices or other memory types that may be used. Using the same example of HBM on a GPU, the user can specify that they wish to allocate memory on the GPU that is also close to the executing Processing Element (PE) by combining hints (i.e., a hint for HBM and locality to the PE).

4 SharP Unified Memory Allocator

Based on the discussion in Sect. 3 and to support the emerging architectures in extreme-scale systems while providing high-performance and portability, we have designed an interface to make use of high-level *Hints* and *Constraints* to capture user intent while providing support for memory allocations on various memories including DRAM, HBM, and NVRAM. In this section, we will discuss both the capturing of user intent by our unified interface as well as the mapping from the intent to the underlying allocators.

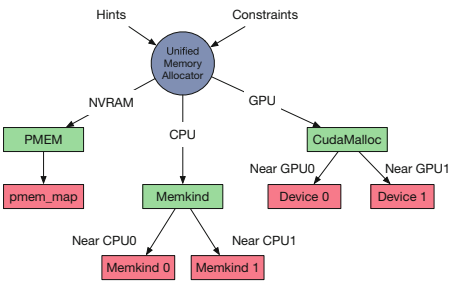


Fig. 1. Memory allocation with the SharP UMA.

4.1 Unified Memory Allocator's Interface

To provide a useful interface for the *User*, both the system and the allocators used for the system are abstracted. This abstraction is accomplished by capturing user intent at a high-level with *Hints* and *Constraints* and mapping these

correctly to the memories that will use them. To abstract the system and represent many possible intents the user may have, we provide several *Hints* specific to areas such as data (i) usage, (ii) accessibility, and (iii) resilience.

1. **Usage:** To capture user intent for data usage, we provide various hints related to usage based on computation. This includes computation on the *Central Processing Unit* (CPU) and compute accelerators such as GPUs. In addition, locality is another aspect of usage that may be described such as allocation of data objects near the PE and near the NIC.
2. **Accessibility:** While providing usage hints allows us to narrow a mapping of intent to a memory, it does not complete it. Coupling usage with accessibility, which describes the properties of the memory with respect to its accessibility by PEs within a job, we are capable of better defining a mapping. Examples of accessibility include memories that are accessible only within a node, between nodes, and across jobs.
3. **Resilience:** Resilience is captured from the user and their intent based on persistence. This allows the user to declare specific data objects need to be allocated such that the data objects can persist through catastrophic failures.

The list of hints and constraints can be used individually, where a single hint is satisfactory for an accurate description of usage, access, or resilience, or the user can compose the hints and form more complex descriptions. For example, describing the level of resilience provided may be difficult for users. While only persistence may be used to describe the users intent, memory placement is important. For instance, if the user wished to describe that memory should be persistent but backed by the parallel filesystem rather than NVRAM, then the persistence hint is not satisfactory. However, when adding access hints, as the parallel filesystem will be accessible between jobs, it can be used for persistent data objects. This mapping is the greatest challenge for this type of unified interface.

To support the mapping between memories and hints, we first abstract the physical memories of the system and enumerate their capabilities to be stored internally. We similarly compose the *Hints* and *Constraints* provided by the user into an enumerated element. Thus, we are capable of determining mapping by creating a list of matching enumerations between the memories and user intent. After the mapping is completed, an allocator object is returned to the user, which allows the user to allocate and free memory on the list of memories satisfying their request. Explicit allocations are accomplished through the same interface with explicit *Hints* (e.g., HINT_DRAM0, HINT_HBM0, HINT_HBM1, etc.).

The resulting interface can be seen in Listing 1.1 and a demonstration of memory allocation with the SharP UMA in Fig. 1.

```
typedef struct sharp_allocator_info_params {
    sharp_hint_t      allocator_hints;
    sharp_constraints_t allocator_constraints;
} sharp_allocator_info_params_t;

sharp_allocator_obj_t * sharp_allocator_init_obj(sharp_allocator_info_params_t * params);

void * sharp_allocator_alloc(sharp_allocator_obj_t * allocator, size_t size);

void * sharp_allocator_alloc_memalign(sharp_allocator_obj_t * allocator,
                                      size_t size,
                                      int alignment);

int sharp_allocator_free(sharp_allocator_obj_t * allocator, void * buffer);
```

Listing 1.1: Intent-based Interface for SharP's UMA

5 Extending Existing Programming Model Implementations

To demonstrate the ease of leveraging SharP for allocating memory, we have extended two popular programming model implementations: (i) Open MPI and (ii) OpenSHMEM-X. In both cases, we extended the implementation to provide the functionality of SharP to the *User*. However, in the case of OpenSHMEM, the programming model, rather than just the implementation, had to be extended to support the memory allocator. In this section, we will describe the modifications we made to support SharP in both Open MPI and OpenSHMEM-X.

5.1 Extending Open MPI

In an effort to demonstrate the utility of the SharP UMA, we extended the Open MPI implementation to support intent-based memory allocations on hierarchical and heterogeneous memories. To do this, we leveraged the `MPI_Alloc_mem` functionality available in the Message Passing Interface (MPI) specification. From the specification, `MPI_Alloc_mem` allocates memory for the user with an effort in allocating efficient memory for *Remote Direct Memory Access* (RDMA) operations [7]. This allows the user to allocate data objects on memories regardless of whether the usage is purely local (i.e., local computation) or remote (i.e., point-to-point and one-sided communication).

To extend the functionality of `MPI_Alloc_mem` to support the SharP UMA, we made use of its *info objects*. The info object in MPI is an object containing key-value pairs, which are parsed by functions like `MPI_Alloc_mem` with the information contained in the object being used to provide extra functionality. This allowed us to extend the function to support the *Hints* and *Constraints* mentioned in Sect. 3. This allows Open MPI to allocate data objects based on user intent across heterogeneous memories.

Unfortunately, the interface for the SharP UMA will generate an allocator object based on the user's *Hints* and *Constraints*. This presents a challenge as only Open MPI will have access to the object, which means each call to `MPI_Alloc_mem` will generate a new allocator object and can increase overhead if

placed in critical sections. In order to reduce this overhead, we added a caching mechanism that caches the most recent allocator objects for future memory allocations. This reduces the overhead as allocator objects only need to be generated if the *Hints* and *Constraints* change between allocations.

Freeing allocated memory is accomplished by making use of `MPI_Free_mem`, which only takes in a pointer to an allocated data object. In order to correctly free the memory, we keep track of allocated memories in a list that is traversed to determine if the memory is from SharP. If it is, then SharP will free the memory.

5.2 Extending OpenSHMEM-X

Unlike the extension of Open MPI, which leveraged interfaces already present in the MPI specification, OpenSHMEM uses a different memory model. In OpenSHMEM, memory is allocated on DRAM in the symmetric heap, which is a memory heap where all PEs allocate data objects with a symmetric address. This means, new interfaces must be created such that OpenSHMEM may support the heterogeneous memories present in many extreme-scale systems.

To provide the necessary support in OpenSHMEM, we created a set of new interfaces that both create a heap on a particular memory and allow future symmetric memory allocations on these memories. For simplicity, the addressing in these generated heaps are asymmetric. The new interfaces are as follows:

- `shmemx_hhm_create`: Creates a new heterogeneous memory region for future memory allocations. This interface takes the *Hints* and *Constraints* from the user along with a size parameter defining how large the heap should be. This will generate an allocator object, which is stored internally and associated with the memory region, which we refer to as a partition similar to Cray SHMEM [12].
- `shmemx_partition_malloc`: Allocates memory on the newly created partition, which interfaces with the allocator object from the SharP UMA. There are similar allocation interfaces for `realloc`, aligned memory allocations, and freeing memory, and, for brevity, they are not listed here.

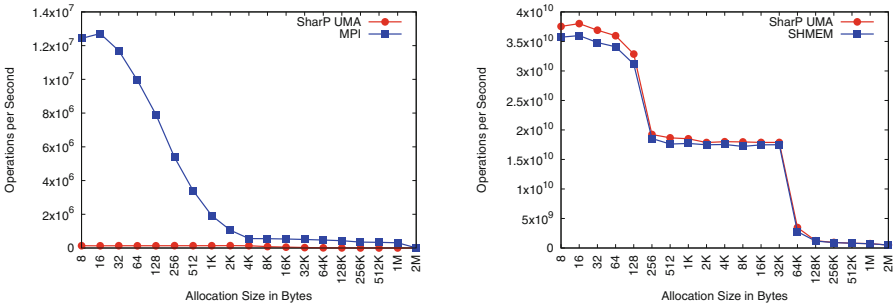
6 Experimental Evaluation

To evaluate this work, we will validate both the performance characteristics of the allocator and the correctness of the allocator's ability to provide intent-based allocations. To show the performance characteristics of the allocator, we will only measure the overhead of performing allocations as the allocator's ability to handle fragmentation and other characteristics are already known as the SharP UMA is leveraging known allocators. To validate the correctness, we make use of micro-benchmarks to measure the bandwidth and message rate of one-sided *Put* operations. We also study the overhead of using the extended programming model implementations from Sect. 5 with applications by porting and evaluating the Graph500 benchmark.

The testbeds we used for the evaluation include multiple systems at ORNL and the *Oak Ridge Leadership Computing Facility* (OLCF). These include Turing, a small 16 node cluster comprised of two Intel Xeon processors, 128 GB of RAM, and a ConnectX-4 EDR interconnect per node, and Rhea, a 512 node cluster similarly comprised of two Intel Xeon processors, 128 GB of RAM, and a Connect-X 3 FDR interconnect per node. These two systems are very similar in composition but have separate affinities with respect to the NIC, which should give us a good understanding of the ability of intent-based memory allocation.

6.1 Performance

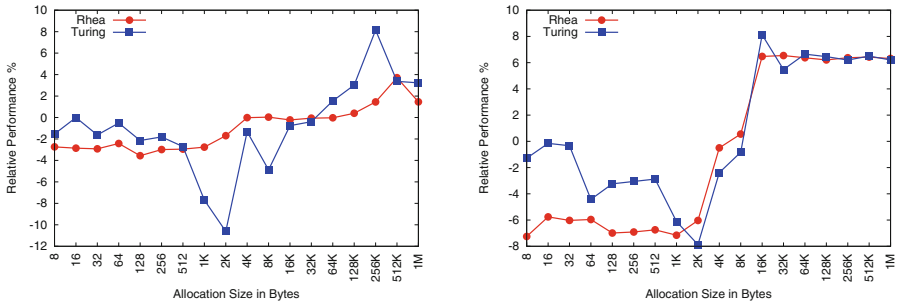
To determine the overhead of the SharP UMA interface for both Open MPI and OpenSHMEM-X, we will perform a series of memory allocations and frees with 70% of the operations being allocations and 30% being free operations. This is completed on increasing sizes of allocations from 8 byte allocations up to 2 MB huge page allocations with the evaluation of each size being comprised of 20,000 operations. For this benchmark, we made use of the Turing cluster. The results of this benchmark can be seen in Fig. 2.



(a) Open MPI (b) OpenSHMEM-X

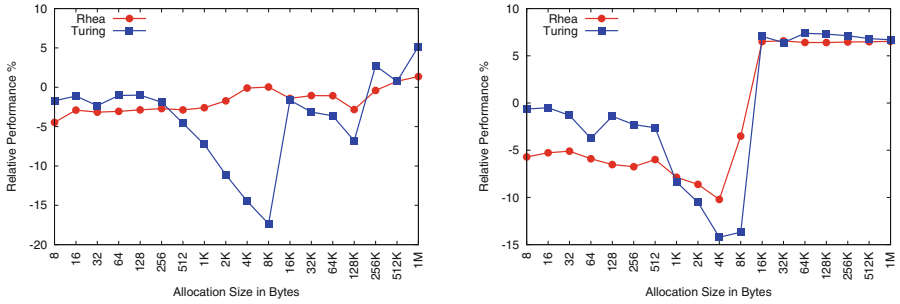
Fig. 2. Memory allocations and frees using the extended (a) Open MPI and (b) OpenSHMEM-X versions with the SharP UMA with 70% of operations being memory allocations. Higher is better.

The results for both the extended Open MPI and OpenSHMEM-X versions were as expected. For Open MPI, the performance of the extension with SharP UMA is very poor due to the constant checking of *Hints* and *Constraints* to determine if an appropriate allocator object has been created yet. However, the average time per memory allocation of a page size (i.e., 4 KB) and lower is roughly 7 µs, which means the extension is still useful so long as it used outside of critical paths. On the other hand, the extension of OpenSHMEM-X is more favorable with many allocation times being within 5% of the unmodified `shmem_malloc` timings. This is because the extension for OpenSHMEM-X does not require a check to determine if a new allocator object needs to be created. Instead, memory can be allocated from an already allocated pool.



(a) Open MPI (b) OpenSHMEM-X

Fig. 3. Bandwidth results on two systems with differing affinities to the NIC. Higher is better.



(a) Open MPI (b) OpenSHMEM-X

Fig. 4. Message rate results on two systems with differing affinities to the NIC. Higher is better.

6.2 Correctness

To validate the correctness, we will perform a series of micro-benchmarks in which we measure the bandwidth and message rate of *Put* operations on two systems, Rhea and Turing, which have different affinities between DRAM and the NIC. Thus, if we attempted to allocate memory near the NIC on one system using the SharP UMA, then we would expect for the memory to be allocated near the NIC on the other system without any code changes as the memory should be allocated based on user intent.

For bandwidth and message rate, we increased the message size from 8 bytes to 1 MB with 10,000 operations being completed for each message size. For the evaluation of each size, we took the median result. For both systems and both Open MPI and OpenSHMEM-X, the Unified Communication X (UCX) communication library [13] was used with short messages used for message sizes up to 128 bytes, buffered messages used for sizes between 128 bytes and 8 KB, and zero-copy used afterwards. The relative results for each test can be seen in Figs. 3

and 4. In both, the relative performance is normalized based on memory allocated near the calling PE with both communicating PEs being located without affinity to the NIC. This particular configuration was chosen as PEs without affinity to the NIC suffer from lower network performance than PEs near the NIC.

As expected, the relative results for both systems under test follow a similar path, with an increased amount of similarity when messages are large enough to make use of zero-copy. For Open MPI, the similarity is less pronounced as the overhead of buffered messages is quite large on the Turing cluster as compared to the Rhea cluster. This suggests that as applications move large amounts of data, the allocation of memory near the NIC is beneficial for PEs without affinity to the NIC.

6.3 Graph500

For the Graph500 benchmark, we made use of the one-sided MPI implementation and the OpenSHMEM adaptation [4]. In each, we modified the implementations to make use of the interfaces described in Sect. 5 and allocate memory near the PE and near the NIC. We used a scale of 16 and evaluated the strong scaling of the application up to 256 PEs on the Turing cluster. The relative results can be seen in Fig. 5, with the results normalized based on memory allocated near the PE.

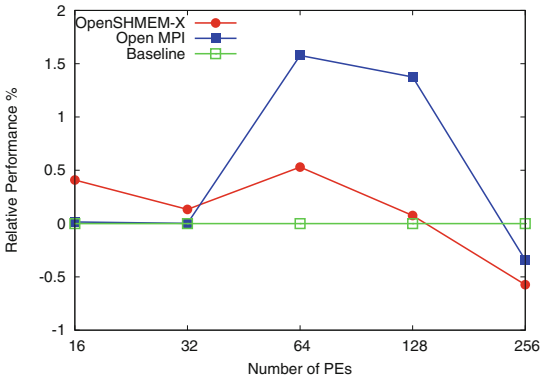


Fig. 5. Results of the Graph500 benchmark with results normalized based on memory allocated near the calling PE. Higher is better.

The results of this evaluation are relatively similar showing no significant improvement in performance by allocating memory near the NIC. However, the takeaway from these results is that we can now allocate the data objects used in Graph500 across any memory with affinities to different devices. In addition, the overhead of the allocator for Open MPI does not significantly impact the performance of the application, which is promising.

7 Conclusion

In this paper, we presented the SharP UMA, an intent-based memory allocator with a unified interface for allocating memory across DRAM, HBM, and NVRAM. We presented the interface of the SharP UMA in Sect. 4 and demonstrated its simplicity by extending well known programming model implementations, Open MPI and OpenSHMEM-X, to support the SharP UMA in Sect. 5. Additionally, we validated this work through an evaluation that examined the performance implications of the intent-based allocator and the correctness of the allocator while moving from system to system. We found the SharP UMA provides minimal overhead in OpenSHMEM-X, but does provide overhead for allocations of memory in Open MPI, which can be mitigated by not placing memory allocations in critical sections. We also showed the movement of our microbenchmarks between systems with differing device affinities without recompilation produced similar results, demonstrating the correctness of our implementation. In addition, we ported the Graph500 benchmark to make use of the extensions we made to Open MPI and OpenSHMEM-X and found relatively similar performance while having greater control of the allocated memory.

Acknowledgment. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

References

1. Baker, M., Aderholdt, F., Venkata, M.G., Shamis, P.: OpenSHMEM-UCX: evaluation of UCX for implementing openSHMEM programming model. In: Gorentla Venkata, M., Imam, N., Pophale, S., Mintz, T.M. (eds.) OpenSHMEM 2016. LNCS, vol. 10007, pp. 114–130. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50995-2_8
2. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: a scalable memory allocator for multithreaded applications. SIGPLAN Not. **35**(11), 117–128 (2000). http://doi.acm.org/10.1145/356989.357000
3. Cantalupo, C., Venkatesan, V., Hammond, J., Czurlyo, K., Hammond, S.D.: Memkind: an extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. Technical report, March 2015
4. D'Azevedo, E.F., Imam, N.: Graph 500 in OpenSHMEM. In: Gorentla Venkata, M., Shamis, P., Imam, N., Lopez, M.G. (eds.) OpenSHMEM 2014. LNCS, vol. 9397, pp. 154–163. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26428-8_10
5. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. http://www.sciencedirect.com/science/article/pii/S0743731514001257
6. Evans, J.: A scalable concurrent malloc (3) implementation for FreeBSD. In: Proceedings of the BSDCan Conference, Ottawa, Canada (2006)
7. Forum, M.P.: MPI: a message-passing interface standard. Technical report, Knoxville, TN, USA (1994)

8. Gloger, W.: Wolfram Gloger's malloc homepage. http://www.malloc.de/en
9. Intel: Intel NVM library. http://pmem.io/nvml/libpmem
10. Jones, T., et al.: Unity: unified memory and file space. In: Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017, pp. 6:1–6:8, ACM, New York (2017). http://doi.acm.org/10.1145/3095770.3095776
11. Lea, D., Gloger, W.: A memory allocator (1996)
12. Namashivayam, N., et al.: Symmetric memory partitions in openSHMEM: a case study with intel KNL. In: Gorentla Venkata, M., Imam, N., Pophale, S. (eds.) OpenSHMEM 2017. LNCS, vol. 10679, pp. 3–18. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73814-7_1
13. ORNL: UCX: Unified Communication X (2015). http://www.openucx.org
14. Venkata, M.G., Aderholdt, F., Parchman, Z.W.: Sharp: towards programming extreme-scale systems with hierarchical and heterogeneous memory. In: Proceedings of the 6th International Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications, August 2017