Evaluating the Impact of High-Bandwidth Memory on MPI Communications

Giuseppe Congiu[*], Pavan Balaji
Mathematics and Computer Science Division
Argonne National Laboratory
email: gcongiu@anl.gov, balaji@anl.gov

***Abstract*—Modern high-end computing clusters are becoming increasingly heterogeneous and include accelerators such as general-purpose GPUs and many integrated core processors. These components are often equipped with high-bandwidth on-package memory (HBM) offering higher aggregated bandwidth than standard DRAM technology has but substantially less capacity.**

In this paper we carry out a fine-grained evaluation of HBM usage in MPI using Knights Landing Multi-Channel DRAM (MCDRAM). Our contribution is twofold. First, we analyze the performance of point-to-point and remote memory access with HBM; Second, we introduce capacity constraints and consider the impact that the MPI library has on the total memory budget. Our analysis shows that although MCDRAM can improve MPI communication performance, this improvement comes at the cost of higher memory usage. Since HBM is a scarce resource, we also provide a list of recommendations that can help users with diverse budgetary requirements for memory decide what MPI objects to place in MCDRAM in order to achieve the best performance possible with their codes.

***Index Terms*—Heterogeneous Memory, Message Passing Interface, High-Performance Computing, High-Bandwidth Memory, KNL MCDRAM**

I. Introduction

High-end computing clusters are becoming increasingly heterogeneous and include a variety of compute components, ranging from standard CPUs to highly parallel accelerators. Such increasing heterogeneity is driven by the need to better exploit data-level parallelism in existing and future codes and to account for a paradigm shift in high-performance computing (HPC) workloads that are moving from simulations to more complex workflows involving data ingestion, generation, and analysis, for example, deep learning applications [1].

The massive parallelism offered by accelerators exacerbates the performance gap between computation and memory, with the memory subsystem that has to supply (absorb) data into (from) an ever-larger number of cores concurrently. For this reason the memory architecture of HPC systems is transitioning from homogeneous configurations, characterized by single DRAM technology, to heterogeneous configurations comprising standard DRAM as well as on-package high-bandwidth memory (HBM) based on 3D stacked DRAM technology [2]. Future architectures will likely include a range of diverse memory components such as HBMs and byte-addressable nonvolatile memories.

Accelerators that integrate HBMs include the Intel Knights Landing (KNL) processors [3], the second generation of the Xeon Phi Many Integrated Core architecture, and the NVIDIA Tesla P100 [4]. Memory heterogeneity exposed by these accelerators introduces further complexity that has to be taken into account by programmers when striving for high performance in their codes. In fact, HBMs are limited in capacity compared with DRAM memory, and not all applications can equally benefit from them; overall, performance strongly depends on the way data is laid out and accessed.

Previous studies [5] [6] [7] have shown that the performance of bandwidth-bound applications can be improved whenever the HBM's capacity is sufficient to accommodate sensitive data structures. If memory requirements cannot be fully satisfied, the user has to decide what data placement strategy to adopt in order to achieve the best results. All these studies, however, lack a fine-grained analysis of the effects that HBMs have on communication libraries such as MPI. Indeed, MPI internally allocates and manages memory objects for a variety of reasons, including efficient intranode communication support. In this case the way the library places its internal objects in memory becomes relevant from a performance standpoint. If on the one hand appropriate placement of MPI objects in HBM can boost the library performance, on the other hand this carves into the available memory budget, reducing the amount of space accessible to the user's data and potentially degrading the overall performance.

In this paper, to fill the gap left in the literature, we study the impact that HBMs have on MPI communications and corresponding memory usage. Our contributions to the state of the art are as follows.

- We prototype HBM support into MPI for different types of intranode communication mechanisms, including point-to-point (pt2pt) and remote memory access (RMA). We use a combination of microbenchmarks and miniapplications to evaluate the effect of HBM on performance for different memory placement strategies of MPI internal objects.
- Based on our findings, we compile a list of recommendations for programmers that will be useful for further tuning their codes through optimal configuration of the MPI library. Our recommendations depend on both the analysis of performance-critical memory objects, along with their placement in physical memory, and the resulting memory usage requirements. This last aspect is particularly important because it can also affect the application's overall performance.

In our study we use the Intel KNL processor, available in the Joint Laboratory for System Evaluation KNL cluster [8] at Argonne National Laboratory, and the MPICH [9] implementation of the MPI standard, developed at Argonne.

The rest of the paper is organized as follows. Section II provides general background information about both the MPICH internals and the KNL on-package HBM called Multi-Channel DRAM (MCDRAM), including how this is detected by the operating system and exposed to the programmer. Section III describes the prototyping of HBM into MPICH. Section IV evaluates our prototype performance with a mix of microbenchmarks and miniapplications and provides a list of recommendations for optimally configuring MPICH. Section V provides information about related work. Section VI presents our conclusions.

II. Background

This section gives information about the intranode communication mechanism used by MPICH and the memory architecture of the Intel KNL processor, focusing on the offered on-package HBM configurations. The section also briefly describes how memory heterogeneity is exposed to users and managed in Linux-based systems.

A. Shared-Memory Communication in MPICH

MPICH is a widely portable, high-performance implementation of the MPI standard and is at the base of many other MPI implementations (also known as MPICH derivatives). MPICH design follows a layered approach, with different levels of abstraction that encapsulate distinct functionalities and services. A high-level architectural view of the library is presented in Figure 1.

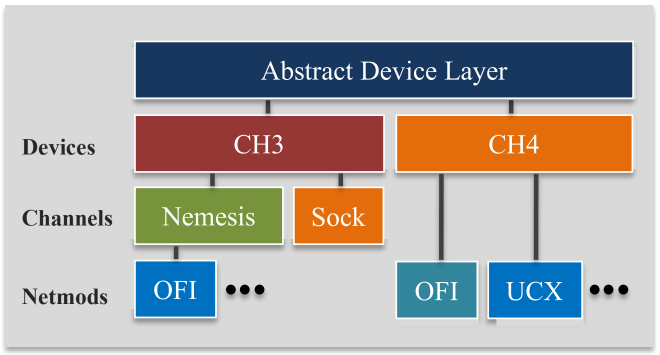


Fig. 1: MPICH architectural diagram

The *abstract device layer* at the top of the stack is responsible for decoupling the high-level MPI interface from the lower-level communication infrastructure. This layer also offers functionalities commonly needed by the underlying software modules. MPICH currently supports two implementations of the abstract device interface exposed by the top layer: one called *CH3* and the other called *CH4*.

CH3 provides two different communication subsystems, or *channels*. The first is based on sockets, while the second is a more efficient implementation called *Nemesis* [10] [11]. Nemesis is highly optimized for intranode communication but also supports internode communication over multiple network fabrics using additional software modules called *netmods*. For intranode communication Nemesis uses lock-free message queues. Every process has two queues, a free queue (*freeQ*) and a receive queue (*recvQ*), each containing elements called *cells*, preallocated in a shared-memory region by the abstract device layer at library initialization.

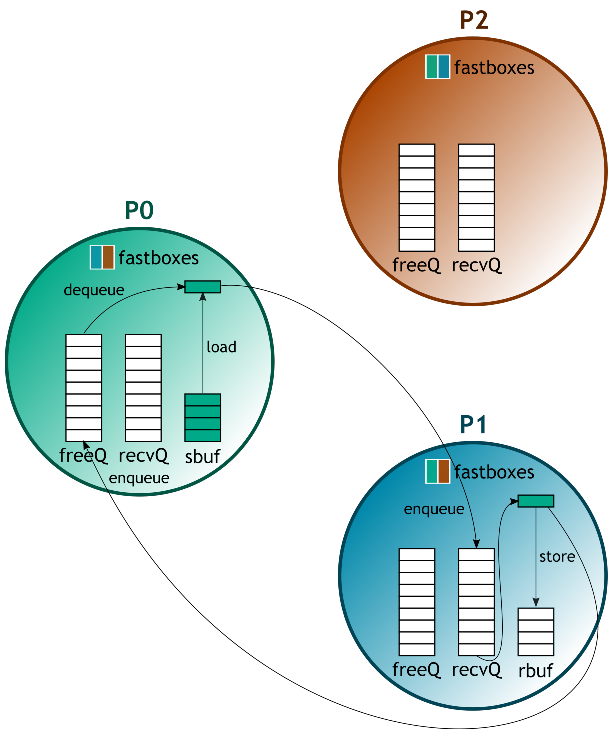


Fig. 2: Nemesis message queues mechanism

When a process needs to send a message to another process, it dequeues a cell from its freeQ, copies data from the send buffer into the cell, and enqueues it in the recvQ of the other process. The receiving process checks its recvQ to find the new message, dequeues the cell, copies the content into the receive buffer, and enqueues it back into the freeQ of the sending process. This process is graphically described in Figure 2. For internode communications the mechanism is similar, but in this case the role of the remote process is played by the local netmod. In this paper we are interested in studying MPI shared-memory communication performance; hence, we disregard the internode case.

The message queue mechanism is highly portable; however, enqueues and dequeues introduce additional latency in the communication. Although this might not be a problem for long messages, it is deleterious for short ones, in which latency is important. For this reason MPICH also provides a mechanism to bypass the message queue for short messages; this mechanism is called *fastbox*. Instead of accessing the message queue, a process first checks whether the fastbox is available. If it is, the process directly copies data into the fastbox and sets a flag to signal the receiver that a new message is ready. The receiver always checks the fastbox before accessing the recvQ. If the fastbox's flag is set, it copies the data from the fastbox into the receive buffer and resets the flag. Whenever the fastbox is not available, the sender falls back to the message queue. The implementation allocates $N-1$ fastboxes to every process, where N is the number of total processes in the node. Since the memory footprint of fastbox elements in each node grows quadratically with the number of processes, fastbox elements have to be kept small (equivalent in size to a single cell element, that is, 64 KB).

Fastboxes and message queues are used for short messages, that is, shorter than a predefined *Eager* threshold. For such messages the sender can directly enqueue cells into the other process recvQ and return to computation right away, without waiting for the message to be actually copied into the receiver's buffer (*Eager* protocol). For long messages, however, in order to avoid exhausting all the cells in the freeQ, Nemesis uses a different communication protocol called *Rendezvous*. In this case the sender cannot start transferring data to the receiver until this has posted a matching receive operation. Furthermore, for the intranode case considered here, the sender uses fastboxes and message queues only for the initial handshaking, while for data transfer MPICH allocates a different shared-memory area called *copy buffer*.

Message queues in Nemesis work both for point-to-point and RMA operations. For intranode RMA communications Nemesis can directly use shared memory, bypassing the message queues and reducing the number of memory copies from two to one. This can be done when the window in MPI is created by using either `MPI_Win_allocate` or `MPI_Win_allocate_shared`. When these functions are employed, MPICH can allocate a new shared-memory region for the exposed window and directly perform copies from the origin to the target buffer.

So far we have discussed the communication infrastructure implemented by CH3, CH4 is the new communication device provided by MPICH to replace the older CH3 implementation. CH4 can take advantage of improved communication functionalities built inside recent network cards to offload to hardware some of the functions that CH3 implements in software. Features of this sort include the possibility to bypass message queues and give the network hardware direct access to user buffers, completely avoiding any memory copy in internode operations. Another feature offered is native support for noncontiguous memory layouts (described through MPI derived datatypes) that also reduces the number of memory copies.

In terms of point-to-point intranode communication, CH4 supports the same message queue mechanism described for Nemesis; however, it currently lacks fastbox support. Similarly, RMA operations cannot automatically use shared memory unless the RMA window is allocated explicitly with `MPI_Win_allocate_shared`. For this reason, in this paper we focus exclusively on the CH3 Nemesis channel and ignore CH4.

B. Intel KNL Memory Architecture

Knights Landing [3] is the codename of the second generation of the Xeon Phi Many Integrated Core architecture initially launched by Intel in 2012. KNL processors can pack up to 72 cores in one chip. Cores are organized in 36 tiles and connected through a network-on-chip using a mesh topology. Each tile integrates an L2 cache of 1 MB, shared among the pair of cores; four vector processing units, two per core; and a caching home agent managing cache coherency. KNL also features a new type of 3D stacked, high-bandwidth, on-package memory called Multi-Channel DRAM. MCDRAM is limited in capacity (up to 16 GB) but can sustain up to ∼400 GBps of aggregated bandwidth, compared with the ∼90 GBps of traditional DRAM technology. Figure 3 shows the high-level architectural diagram of a KNL system.

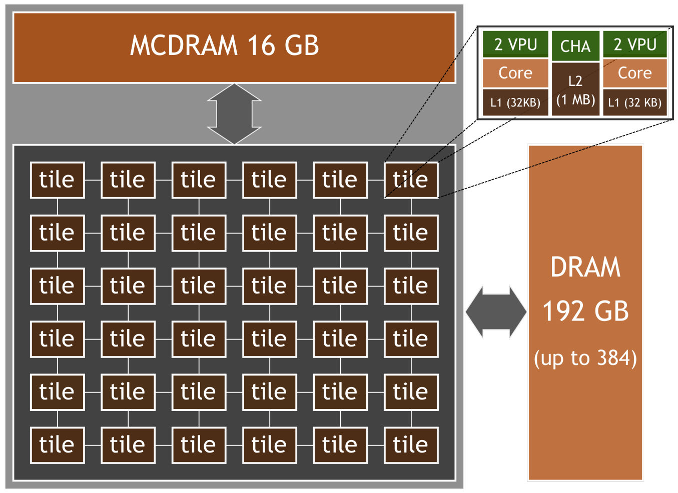


Fig. 3: KNL architectural diagram

The MCDRAM can be configured to work in three modes (memory modes).

- **Cache**: The MCDRAM is totally transparent to the user and managed by the hardware as an additional, directly mapped L3 cache. This configuration is the default and also the least flexible since it gives no control on data placement in physical memory. For this reason it is suitable for existing applications that are not directly concerned with heterogeneous memory systems but can still take advantage of some of the new features to improve performance;
- **Flat**: The MCDRAM extends the address space of conventional DRAM, meaning that the user can directly access it using operating system support or appropriate user space libraries. This mode offers the highest level of control and can potentially give the biggest performance improvements if used intelligently;
- **Hybrid**: Part of the MCDRAM is managed by the hardware as L3 cache, and the remaining part is exposed directly to the user. This mode offers a combination of automatic caching and user-defined memory placement strategy.

We focus on the flat mode to selectively move MPICH internal memory objects to MCDRAM. We chose this configuration over the cache and hybrid modes because we can avoid uncontrolled data movement from the system, thus performing a fine-grained evaluation of the impact that high-bandwidth memory has on MPI communication performance.

C. Heterogeneous Memory in Linux

In flat mode the MCDRAM is detected by Linux as a separate NUMA node with no associated compute resources. This approach is taken so that, by default, any allocation will be directed to DRAM. Therefore, even in flat mode, existing applications will continue to work undisturbed with standard DRAM memory. In order to place allocations into MCDRAM, the operating system has to be explicitly instructed by the user through the `mbind` system call, as follows.

*mbind(**void** *addr, **unsigned long** len, **int** mode, **unsigned long** *nodemask, **unsigned long** maxnode, **unsigned** flags)*

The call takes the starting address (*addr*) and length (*len*) of the virtual address space allocation that the memory binding policy will be applied to; the mode describing the type of memory binding to be performed (e.g., “bind” to a specific NUMA node or “interleave” across all the specified NUMA nodes); the nodemask specifying which NUMA node(s) the policy will apply to; and the number of such NUMA nodes (*maxnode*). A mode flag (*flags*) defining the behavior of the operating system at the time physical memory then is allocated (e.g., whether the binding is “strict” and the allocation should fail if it cannot be satisfied by MCDRAM or if it should failover to DRAM instead).

In order to use `mbind`, memory has to be allocated through the `mmap` system call. This means that all allocations must be aligned and a multiple of the page size. For small buffers that are frequently allocated and freed, this approach is not efficient because it causes wastage of memory and incurs the overhead of a system call every time new memory is requested. For this reason Intel provides a heap management library offering `malloc`-like functionalities called *memkind* [12]. With memkind, users can allocate memory in MCDRAM using the `hbw_malloc` function. Similarly to malloc, this function allocates a larger area of virtual memory using mmap, binds it to MCDRAM using mbind, and then partitions it into smaller regions that are eventually returned to the caller. A similar but more flexible framework called *hexe* has been proposed by Oden and Balaji [5].

Unfortunately none of the aforementioned libraries provides shared-memory allocation support, and thus they are not useful for the objectives of this work.

III. Heterogeneous Shared Memory in MPICH

MPICH supports shared memory using both POSIX and SystemV interfaces and can select the most appropriate one depending on the host system. In POSIX, shared memory can be allocated by creating an inode object through `shm_open` and then passing the returned descriptor to the mmap system call along with the `MAP_SHARED` flag, at which point mmap allocates memory, associating it with the corresponding inode. The inode becomes the handle that processes use to access the shared-memory region and map it to their private address space. Unlike normal files, however, the file-mapped memory is not backed up by any block in the file system. When SystemV is used, shared memory is allocated in a similar way, but the shared object is created by using a different mechanism.

Regardless of the interface used to allocate shared memory, the returned virtual address can be passed to mbind, which then applies the desired policy. Afterwards, every process can access the shared-memory object and map it to its private address space, also inheriting the binding policy previously set.

In MPICH, message queues’ elements (cells) are allocated in shared memory automatically when the library is initialized. For shared-memory windows, however, MPI provides `MPI_Win_allocate` and `MPI_Win_allocate_shared`. These interfaces take an additional *info* object that can be exploited to pass performance hints to the implementation. To enable heterogeneous memory support for the shared-memory objects described in Section II-A, one needs only to set the appropriate memory-binding policy for the returned virtual address either directly using “mbind” or indirectly using a third-party library that uses mbind underneath. For fastboxes, cells, and copy buffers we have no alternative and thus use additional *CVAR*s to customize the runtime behavior of the library. For RMA windows we have two options: to use CVARs or to use the info object previously mentioned. In our implementation we support both CVARs and the info object, with CVARs overriding user-defined hints.

For memory-binding policy selection we do not use mbind directly because it is not portable. Instead we use the *hwloc* hardware locality library, which also provides support for memory binding through `hwloc_set_area_membind`. Moreover, recent versions of hwloc have added support to detect MCDRAM in KNL systems [13]. MPICH already uses hwloc for binding processes to cpusets, and thus we can reuse most of the existing infrastructure, extending it with heterogeneous memory. Hwloc defines memory- and process-binding policies as well as additional flags to control the operating system behavior in case the requested allocation cannot be satisfied by the specified memory. For the point-to-point objects (i.e., fastboxes, cells, and copy buffers) we bind MPICH objects to a single MCDRAM node and enforce the policy as “strict” to make sure that there is no fallback to DRAM. For RMA operations we leave full control to the user.[1]

[1]Additional information on hwloc_set_area_membind can be found at https://www.open-mpi.org/projects/hwloc/doc/hwloc-v2.0.0-a4.pdf.

IV. Evaluation

This section presents the evaluation of KNL MCDRAM with MPI communications and the corresponding memory usage analysis. We start by briefly describing the hardware environment and the benchmarks we used. Our study is conducted on a single node of the Joint Laboratory for System Evaluation KNL cluster at Argonne National Laboratory. KNL nodes in the cluster are equipped with Intel Xeon Phi 7210 processors, each running 64 cores clocked at 1.3 GHz. Processors also have a 16 GB on-package HBM and 192 GB of external DRAM.

We start evaluating performance in terms of both bandwidth and latency using a combination of different microbenchmarks. These allow us to assess MCDRAM peak performance and characterize its possible impact on MPI communications. We then introduce capacity constraints and analyze the impact that different MPI objects have on the available MCDRAM memory budget. Based on the results, we compile a list of recommendations that can help the user prioritize objects in MCDRAM.

We consider a stencil code that well represents the communication pattern of a wide range of HPC codes; and using our previous considerations, we measure possible runtime improvements deriving from different MCDRAM capacity constraints.

For all our experiments we use the KNL system in *quadrant* and *flat* mode. In this configuration only two NUMA nodes are available: one containing the cpuset and DRAM memory (Numa 0) and one containing MCDRAM (Numa 1). Additionally, to minimize variability of results related to process migration, we bind every MPI process to a unique physical core among the 64 available using the `-bind-to=core` option of `mpirun`.

A. STREAM

Before proceeding with the analysis of MPI performance we start with a preliminary characterization of MCDRAM[2] bandwidth profile. To do so, we use the OpenMP version of the *STREAM* [14] benchmark. In our configuration the benchmark runs with a variable number of OpenMP threads, ranging from 1 to 64, each transferring from a minimum of 64 K to a maximum of 4 M *doubles*, that is, from a minimum of 512 KB to a maximum of 32 MB. Performance is measured by running the *STREAM Copy* test 10 times and taking the average bandwidth.

Figures 4a and 4b show the results for DRAM and MCDRAM, respectively. In single-thread tests the measured bandwidth is similar for the two memories, with $\sim$16 GB/s for DRAM and $\sim$18 GB/s for MCDRAM. After increasing the number of threads beyond 8, however, MCDRAM outperforms DRAM, with a peak of $\sim$380 GB/s when using all 64 available cores; DRAM, on the other hand, can sustain only up to $\sim$82 GB/s, that is, $\sim$4.5$\times$ less than MCDRAM, as also reported by previous studies [5] [6].

[2]In this case we have used *numactl –membind=1* to move all the application's memory to MCDRAM, as done by previous works.

B. Remote Memory Access

For the evaluation of MPI RMA operations we use the OSU Micro-Benchmarks suite. We are interested in the aggregated bandwidth and in the overall latency when all the KNL cores are actively participating in the communication. The OSU suite provides multibandwidth and multilatency tests for pt2pt communication but not for RMA. For this reason we have modified the *osu_put_bw*, *osu_put_latency osu_get_bw*, and *osu_get_latency* to support multiprocess RMA, using the same communication schema used for pt2pt tests (i.e., pairs of processes do RMA using `MPI_Put` and `MPI_Get`).[3] For each test we use different memory placement configurations of the RMA memory window, as reported in Table I.

TABLE I: RMA memory placement configurations. **O** indicates origin buffer, and **T** indicates target buffer; lowercase indicates DRAM placement, and uppercase indicates MCDRAM placement.

	OrigBuf	***TargetBuf***
o/t	DRAM	DRAM
o/T	DRAM	MCDRAM

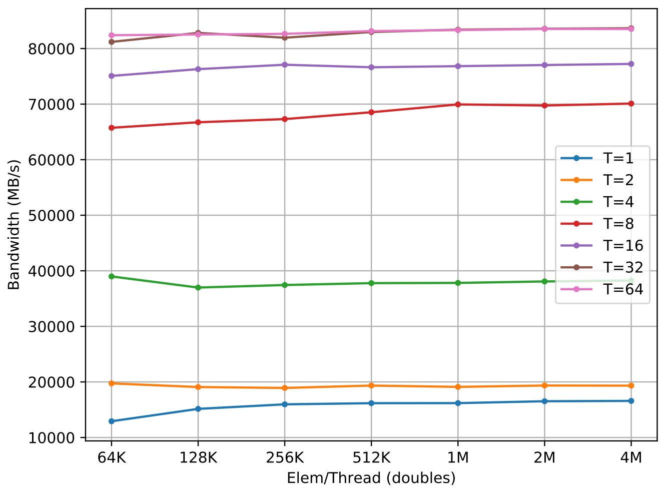
Figure 5 shows bandwidth performance for multi-`MPI_Put` (5a) and `MPI_Get` (5b) tests when 32 pair of processes are used. (We limit our analysis to 64 processes because, as shown for the STREAM benchmark, MCDRAM bandwidth can be saturated only when all the KNL cores are active.) The two tests are similar: puts load data from the user buffer (*origin*) and store it in the shared-memory window buffer (*target*), and gets load data from the shared-memory window buffer and store it to the user buffer.

As one can infer from the results, the most performance-critical object for RMA operations is the store buffer (i.e., the target buffer in put operations and the origin buffer in get operations). In fact, placing this in MCDRAM can boost the bandwidth of put operations up to 105%. For get operations the store buffer is the origin, while target becomes the load buffer. In this case placing the load buffer in MCDRAM gives only moderate improvements compared with put, around 36–41% for messages in the range of 8 KB to 128 KB and around 13% for longer messages.

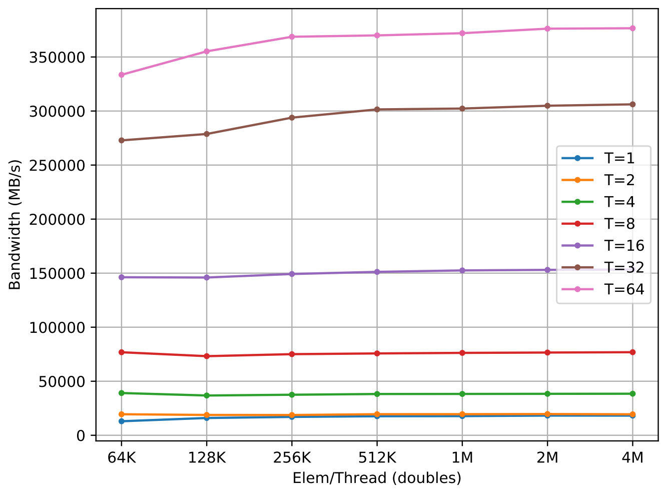
Table II reports multilatency test results for 32 pairs of processes. Latency figures further confirm the importance of placing the store buffer in MCDRAM. Indeed, for put operations, latency can be reduced significantly, to over 60% of the baseline configuration (o/t). For get operations the reduction is moderate and does not go beyond $\sim$12% of the baseline.

In the RMA tests we did not consider the placement of the origin buffer in MCDRAM for get operations because the origin buffer is normally allocated by the user. The target buffer, on the other hand, can be under the direct control of the MPI implementation whenever `MPI_Win_allocate` or

[3]The multiprocess RMA microbenchmarks code, as well as the pt2pt code used for the other experiments, can be found at https://github.com/gcongiu/osu_micro_benchmark_mcdram.git.

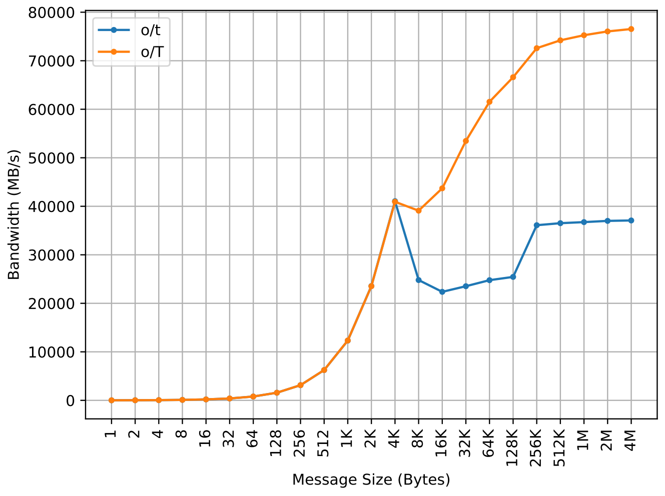


(a)

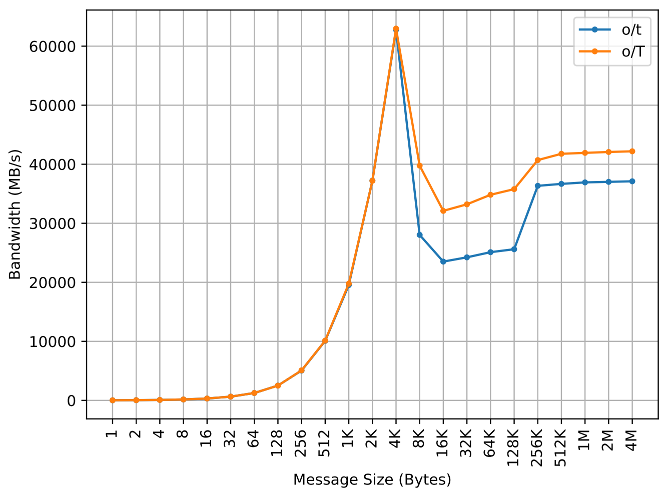


(b)

Fig. 4: *STREAM Copy* test aggregated bandwidth values for DRAM (4a) and MCDRAM (4b) with varying number of threads (1, 2, 4, 8, 16, 32, and 64)



(a)



(b)

Fig. 5: osu_put_mbw (5a) and osu_get_mbw (5b) benchmark results for 32 pairs

`MPI_Win_allocate_shared` is used to create the shared-memory window. For this reason, users should take advantage of MPI RMA functions to create and manage memory windows so that the implementation can appropriately place the corresponding buffers into the most suitable memory.

C. Point-to-Point

For the evaluation of MPI point-to-point operations we use the *osu_mbw_mr* and *osu_multi_lat* tests. These tests perform parallel communication, through `MPI_Isend` and `MPI_Recv`, across multiple pairs of processes. For each number of pairs we use the memory placement configurations reported in Table III. The selected placement configurations allow us to carry out a fine-grained performance analysis of the memory subsystem for the most relevant memory objects in the MPICH implementation.

Figure 6 shows bandwidth performance for 32 pairs of processes. The results indicate that for short messages (for which MPICH uses the Eager protocol) placing fastboxes (“F/c/cb”), cells (“f/C/cb”) or both (“F/C/cb”) in MCDRAM gives better performance over the baseline configuration (“f/c/cb”), up to ∼50% and ∼54%, respectively (performance of “f/C/cb” and “F/C/cb” is aligned). For this particular test we also notice that cells perform better than fastboxes when moved to MCDRAM. The reason is that the *osu_mbw_mr* test issues multiple send/recv operations for the same data. Hence, only the first message will use fastbox, while the following messages will use cells (i.e., message queues). After analyzing

TABLE II: RMA multilatency test results (in microseconds) for the considered memory placement configurations. The table also reports the percentage latency reduction against the baseline, namely, "o/t."

	MPI_Put			MPI_Get		
	o/t	**o/T**	↓ %	**o/t**	**o/T**	↓ %
0 B	49.192	49.142	-	48.998	49.069	-
1 B	112.389	175.953	-	111.547	79.760	-
2 B	48.571	48.041	-	48.105	47.906	-
4 B	47.795	47.903	-	47.747	47.491	-
8 B	48.107	48.282	-	48.029	47.907	-
16 B	47.985	48.653	-	48.044	47.610	-
32 B	48.320	48.572	-	48.243	47.845	-
64 B	49.186	48.680	-	48.850	48.619	-
128 B	49.039	48.319	-	48.607	48.503	-
256 B	48.855	47.925	-	48.333	48.251	-
512 B	48.804	47.925	-	48.171	48.130	-
1 KB	49.046	48.247	-	48.568	48.411	-
2 KB	48.870	47.934	-	48.553	48.468	-
4 KB	48.496	48.274	-	48.442	48.328	-
8 KB	48.983	48.893	-	48.787	48.652	-
16 KB	50.121	50.086	-	50.114	49.881	-
32 KB	51.545	50.953	-	51.702	50.540	-
64 KB	54.671	53.892	-	54.410	53.937	-
128 KB	71.826	66.411	7.5	72.506	65.100	10.0
256 KB	259.659	92.015	64.5	257.580	257.371	0.08
512 KB	545.213	252.521	53.6	545.257	506.752	7.06
1 MB	1039.163	558.357	46.2	1037.612	931.533	10.2
2 MB	1940.753	999.427	48.5	1939.089	1717.608	11.4
4 MB	3739.691	1857.870	50.3	3739.174	3294.915	11.8

TABLE III: Point-to-point memory placement configurations. **F** indicates fastbox, **C** indicates cells, and **CB** indicates copy buffer; ;owercase indicates DRAM placement, and uppercase indicates MCDRAM placement.

	Fastbox	***Cells***	***Copy Buffer***
f/c/cb	DRAM	DRAM	DRAM
F/c/cb	MCDRAM	DRAM	DRAM
f/C/cb	DRAM	MCDRAM	DRAM
F/C/cb	MCDRAM	MCDRAM	DRAM
f/c/CB	DRAM	DRAM	MCDRAM

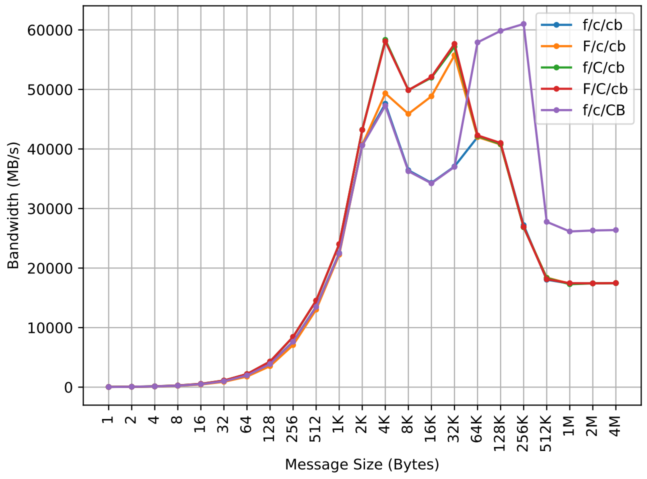


Fig. 6: osu_mbw_mr microbenchmark results for 32 pairs

the performance in more detail we found that about one-third of the messages use fastboxes, while for the remaining two-thirds the implementation falls back to message queues. For this reason we have more messages using cells, and placing these in MCDRAM can provide a larger improvement. For long messages (for which MPICH switches to the Rendezvous protocol) the performance-critical object becomes the copy buffer, and placing this in MCDRAM achieves the best performance. Fastboxes and cells are rarely used in practice, and in fact placing either of them in MCDRAM does not provide any benefit since the curves for "f/c/cb," "F/c/cb," "f/C/cb," and "F/C/cb" are all aligned, as shown in Figure 6.

Our analysis of MPI point-to-point communication concludes with the multilatency tests for 32 pairs of processes reported in Table IV. Similarly to the bandwidth tests, latency of short messages shows better performance when fastboxes are placed in MCDRAM (up to 36% reduction). Placing cells in MCDRAM gives practically no performance improvement over fastboxes because in this particular case these are unused (latency tests issue only one message at a time for each process and then measure the time it takes for the message to arrive to the other process; this message always goes to fastbox). For long messages ($\geq$ 64 KB) placing the copy buffer in MCDRAM can give up to 39% latency reduction.

D. Recommendations to Users

Based on our microbenchmarks analysis, we compile for each communication mechanism a list of recommendations that should help users optimize their codes with the MPICH library in different memory capacity constraint scenarios.

1) RMA: we recommend that users allocate a shared-memory window and corresponding buffer using either the `MPI_Win_allocate` or the `MPI_Win_allocate_shared` interface. In this way the library can optimize the memory allocation whenever the user needs it by placing the shared-memory buffer in MCDRAM. For put operations this is definitely the best strategy since the performance gain obtained by having the store buffer in MCDRAM is significant. For get operations having the load buffer in MCDRAM is still beneficial,

TABLE IV: Point-to-point multilatency test results (in microseconds) for the considered memory placement configurations. The table also reports the percentage latency reduction against the baseline, i.e., "f/c/cb."

	f/c/cb	F/c/cb	f/C/cb	f/c/CB	F (↓ %)	CB (↓ %)
0 B	0.841	0.827	0.843	0.838	1.664	-
1 B	0.891	0.883	0.892	0.892	0.897	-
2 B	0.901	0.893	0.902	0.902	0.887	-
4 B	0.938	0.928	0.942	0.940	1.066	-
8 B	0.951	0.940	0.952	0.950	1.156	-
16 B	0.981	0.971	0.983	0.980	1.019	-
32 B	1.192	1.191	1.194	1.191	0.083	-
64 B	1.192	1.191	1.197	1.191	0.083	-
128 B	1.238	1.237	1.240	1.236	0.080	-
256 B	1.252	1.252	1.257	1.252	0	-
512 B	1.609	1.509	1.615	1.609	6.215	-
1 KB	2.659	2.566	2.666	2.661	3.497	-
2 KB	1.728	1.722	1.734	1.727	0.347	-
4 KB	2.828	2.777	2.807	2.818	1.803	-
8 KB	5.377	4.639	5.430	5.376	13.72	-
16 KB	11.496	8.342	11.503	11.514	27.43	-
32 KB	23.682	15.068	23.745	23.702	36.37	-
64 KB	50.975	51.087	50.999	44.454	-	12.79
128 KB	116.144	118.372	117.863	89.120	-	23.26
256 KB	530.683	511.696	504.205	321.064	-	39.49
512 KB	984.598	982.653	984.732	663.066	-	32.65
1 MB	1920.218	1921.261	1916.377	1313.256	-	31.60
2 MB	3807.138	3805.567	3809.388	2558.192	-	32.80
4 MB	7567.653	7567.129	7577.482	5090.969	-	32.72

although the gain is more moderate compared with that using puts.

2) Pt2pt: MPICH uses two communication protocols, differentiating between short (Eager) and long (Rendezvous) messages. In this case, depending on the memory usage constraints imposed by the application, we may have to choose which among fastboxes, cells, and copy buffers should be placed in MCDRAM. To do so, we first need to analyze the memory requirements of different objects. The fastboxes' memory footprint is given by $N \times (N-1) \times 64\ KB$; the cells' memory footprint is given by $N \times 64\ cells/proc \times 64\ KB$; and the copy buffers' memory footprint is given by $N \times (N-1) \times 8 \times 32\ KB$. For all of these, N represents the number of processes in the node. Overall, if we consider 64 processes, the maximum memory requirement[4] of each object is, respectively, 252 MB, 256 MB, and 1008 MB. Thus, fastboxes and cells each account for about 1.5% of the total MCDRAM capacity, while copy buffers account for 6.15%. Since it is important that application-sensitive data structures be placed in the appropriate memory, MPICH should be memory conscious and leave as much memory as possible to applications' data. If we decide to limit MPICH MCDRAM usage to 1.5%, we can easily address performance of short messages by placing fastboxes in MCDRAM. As we have shown in this section, however, the best performance for bandwidth is obtained by placing both fastboxes and cells in MCDRAM. Thus, if we decide to dedicate the 3% of MCDRAM to MPICH objects, we can accommodate all the relevant short message objects and achieve the best performance possible. If we also want to optimize long messages, in the worst case we have to be ready to pay as much as over 9% of the available capacity. However, as we will show in the rest of the section through a miniapp example, the all-to-all communication scenario leading to maximum memory utilization is never encountered in practice. In fact, even collective calls like `MPI_Alltoall` use scalable algorithms that reduce the number of communication pairs [15].

[4]This is the worst case assuming that every process actively communicates with every other process in the node.

E. Stencil Code

The 2D stencil code we use for our evaluation represents a good example of a regular communication pattern found in many single program, multiple data applications. The code decomposes the input domain evenly across the available processes, which we set to 64 as the total number of cores in KNL (8 per direction). Each process then exchanges its halo regions with neighbors using `MPI_Isend` and `MPI_Irecv` and then `MPI_Wait` for communication to complete before doing another round of halo exchange. Thus, the actual number of communication pairs in this case is given by the sum of 6×6 inner processes communicating with 4 neighbors (north, south, east, and west), 6×4 side processes communicating with 3 neighbors, and 4 corner processes communicating with 2 neighbors; in total we have 224 pairs instead of $N \times (N-1)$. Since memory is allocated only when accessed (first touch), this reduces the footprint of fastboxes to $224 \times 64\ KB = 14\ MB$ and the footprint of copy buffers to $224 \times 8 \times 32\ KB = 56\ MB$, that is, 0.07% and 0.34% respectively.

Boundary regions are not directly passed to the MPI send function but are instead packed into contiguous buffers (`sbufnorth`, `sbufsouth`, `sbufwest`, and `sbufeast`) using `MPI_Pack`. Similarly, the MPI receive function does not directly store the received data in the main matrix but instead in a set of intermediate buffers (`rbufnorth`, `rbufsouth`, `rbufwest`, `rbufeast`). Eventually, data is moved from the intermediate buffers to the main matrix by using `MPI_Unpack`.

TABLE V: Stencil runtime test results (in seconds).

	2 KB	**4 KB**	**8 KB**	**16 KB**	**32 KB**	**64 KB**
f/cb	0.357267	0.730782	1.011214	0.618523	1.764798	4.101942
F/cb	0.348530	0.725480	0.913981	0.560735	1.588268	4.090613
f/CB	0.354700	0.728448	1.022707	0.615309	1.765437	3.856240
F (↓ %)	-	-	9.7	9.4	9.9	-
CB (↓ %)	-	-	-	-	-	6.1

We vary the size of the input domain from 32 MB (2,048 double points along each dimensions) to 32 GB (65,536 double points along each dimension) and use different memory placement configurations, as already done for the microbenchmarks study. Each input size is determined to allow a number of elements sufficiently large to be exchanged between processes so that we can verify previous microbenchmark results on a real use case. Thus, for the smallest domain of 32 MB each send/recv buffer is 2 KB, while for the largest domain of 32 GB each send/recv buffer is 64 KB.

Overall, for the smallest domain, each of the 64 processes requires 512 KB to store the results from the last iteration and 512 KB to store the results from the current iteration, which depends from the exchanged halo regions and the previous iteration, for a total of 1 MB. Additionally, each process needs to allocate space for the send/recv buffers for a total of 16 KB (2 KB for each of the four pack and four intermediate buffers). In total the stencil code requires slightly more than 64 MB of memory. For the largest domain this requirement goes up to 64 GB.

Thus, the maximum memory footprint of the stencil code exceeds the capacity of MCDRAM by a factor of more than 4. In this case the application might be already struggling to get the most sensitive data in MCDRAM, and the MPICH library should try to minimize its memory usage.

To validate the recommendations previously compiled, we selectively placed one MPICH object at the time in MCDRAM and evaluated the corresponding impact on the runtime. Figure 7 shows runtime results (in seconds) for different domain sizes. In this case we report the size of the send/recv buffers instead of the full domain.

Because every communication is completed by an `MPI_Wait` call, cells are never used, and all data is moved by using fastboxes. For this reason we have not included cells elements in our stencil study. In the figure we see that placing fastboxes and copy buffers in MCDRAM can reduce the runtime of both short and long messages.

Table V summarizes the runtime results shown in the previous figure, along with the corresponding percentage reduction of fastboxes and copy buffers compared with the "f/c/cb" configuration. The table shows that for short messages placing fastboxes in MCDRAM can in fact reduce the runtime by about 10%. For long messages, the reduction is 6% when placing copy buffers in MCDRAM. Since we are not performing any computation in the stencil test, we have increased the number of iterations (i.e., communication rounds) for the shortest messages from 1,000 to 10,000. This explains why for 16 KB the runtime is higher than for 32 KB.

For domain sizes up to $16,384 \times 16,384$ points (i.e., 32 MB/process) the total memory footprint is about 4 GB, which requires only 25% of MCDRAM capacity. In this case our previous recommendations advise that both fastboxes and copy buffers be placed in MCDRAM. For domain sizes starting at $32,768 \times 32,768$ (i.e., 128 MB/process) the total memory footprint is over 16 GB, which already requires 100% of the MCDRAM capacity. In this case 0.34% capacity required for copy buffers might not be affordable, and our recommendation is to optimize only short messages (∼0.07% capacity), disregarding long ones.

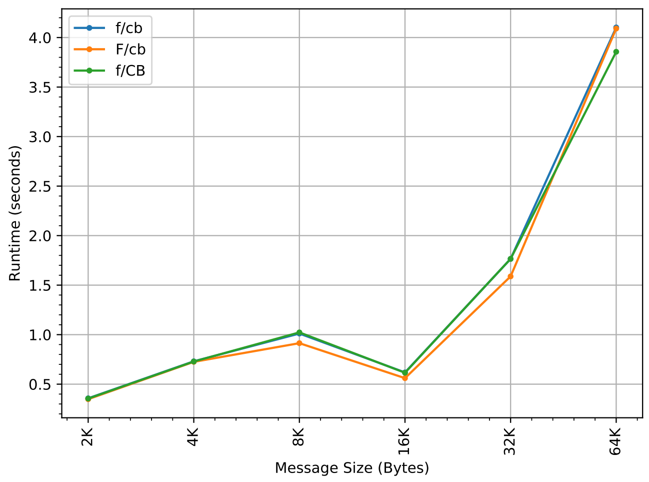


Fig. 7: Stencil with 64 processes

V. Related Work

High-bandwidth memories, such as KNL MCDRAM, tackle the memory wall problem and bring to the users unprecedented memory bandwidth. For this reason bandwidth-bound applications can significantly benefit from HBMs usage to improve their performance. In this sense many works have focused on extending existing codes with heterogeneous memory support and automatically or manually promoting sensitive data structures in HBM. Smith, Park, and Karypis [16] ported an unstructured sparse tensor decomposition application using canonical polydiac decomposition on KNL and, taking advantage of MCDRAM, obtained up to 1.8× speedup over a dual-socket Intel Xeon system with 44 cores. DeTar et al. [7] ported a lattice QCD code called MILC to KNL and obtained significant improvements when using MCDRAM in either cache or flat mode. Bo Peng et al. [6] carried out a detailed study of the impact of MCDRAM on a range of data analytics applications on KNL. They analyzed different memory modes, problem sizes, and threading levels and gave a set of guidelines for selecting the proper memory allocation based on the application's memory usage characteristics.

In all these works MCDRAM cache and flat modes give similar performance when the involved datasets fit in memory. If they do not fit, the user must select which datasets are more performance critical and move only these into MCDRAM or manually manage data movements between DRAM and MCDRAM. In this sense Perarnau et al. [17] evaluated the performance benefits of software-managed data migration strategies for user structures between DRAM and MCDRAM.

Another problem with HBMs is the lack of portable and fine-grained memory management interfaces offered by operating systems and user space libraries. As we have discussed, the Linux kernel allows access to heterogeneous memory through the "mbind" system call. However, mbind works only on allocations returned by the "mmap" system call, which is expensive and is limited to multiples of the page size. A partial solution for KNL is offered by the *memkind* library developed by Intel [12]; however, this is architecture specific. A more flexible approach is offered by Oden and Balaji with the *hexe* library [5]. Unfortunately none of these is both production ready and portable.

MPI implementations such as MPICH internally allocate and manage memory for a number of reasons, including shared-memory intranode communication. To the best of our knowledge currently no work has integrated HBM support in MPI and evaluated the corresponding performance implications on different communication mechanisms and memory usage scenarios.

VI. Conclusions

Memory heterogeneity is becoming increasingly important for achieving high performance on modern HPC systems. Many works therefore have focused on the effective use of HBMs to store data structures of bandwidth-bound applications, taking advantage of the higher aggregated bandwidth offered by this technology compared with standard DRAM. However, scientific codes also use additional communication libraries, such as MPI, for distributing work across processes both on the same node and across nodes.

In this paper we have integrated heterogeneous memory support in MPICH, giving users access to the full features of the memory subsystem using a familiar and standardized interface. Additionally, since HBMs are substantially more limited in capacity compared with DRAM, it is also important to make sure that such limited capacity is used intelligently by the MPI library.

To this extent we have evaluated the performance implications that HBMs have on different communication mechanisms in MPI, including point-to-point and RMA. Based on our observations we have provided a set of recommendations to help user decide how to prioritize MPICH objects in MCDRAM in different communication scenarios. If followed, our recommendations can help users planning for a memory placement strategy that makes effective utilization of the limited MCDRAM capacity and results in the best performance possible.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility.

References

[1] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 3. Atlanta, Georgia, USA: PMLR, 17–19 June 2013, pp. 1337–1345. [Online]. Available: http://proceedings.mlr.press/v28/coates13.html

[2] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *2008 International Symposium on Computer Architecture*, June 2008, pp. 453–464.

[3] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights Landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, March 2016.

[4] NVIDIA, "NVIDIA Tesla P100," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[5] L. Oden and P. Balaji, "Hexe: A toolkit for heterogeneous memory management," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2017, pp. 656–663.

[6] I. B. Peng, S. Markidis, E. Laure, G. Kestor, and R. Gioiosa, "Exploring application performance on emerging hybrid-memory supercomputers," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec. 2016, pp. 473–480.

[7] C. DeTar, D. Doerfler, S. Gottlieb, A. Jha, D. Kalamkar, R. Li, and D. Toussaint, "MILC staggered conjugate gradient performance on Intel KNL," p. 270, 12 2016.

[8] "Joint Laboratory for System Evaluation, JLSE cluster," http://www.jlse.anl.gov/.

[9] "MPICH: A high performance and widely portable implementation of the Message Passing Interface (MPI) standard," www.mpich.org.

[10] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem," in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1, May 2006, pp. 10 pp.–530.

[11] ——, "Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem," *Parallel Computing*, vol. 33, no. 9, pp. 634–644, 2007, selected Papers from EuroPVM/MPI 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819107000786

[12] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond, "memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies," 3 2015.

[13] B. Goglin, "Exposing the locality of heterogeneous memory architectures to HPC applications," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: ACM, 2016, pp. 30–39. [Online]. Available: http://doi.acm.org/10.1145/2989081.2989115

[14] J. D. McCalpin, "A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers," 1995.

[15] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, Feb. 2005. [Online]. Available: http://dx.doi.org/10.1177/1094342005051521

[16] S. Smith, J. Park, and G. Karypis, "Sparse tensor factorization on many-core processors with high-bandwidth memory," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 1058–1067.

[17] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra, and P. Beckman, "Exploring data migration for future deep-memory many-core systems," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept. 2016, pp. 289–297.