2017 IEEE International Parallel and Distributed Processing Symposium Workshops

A Memory Heterogeneity-Aware Runtime System for bandwidth-sensitive HPC applications

Kavitha Chandrasekar*, Xiang Ni*†, Laxmikant V. Kale*
*University of Illinois Urbana Champaign
†IBM T.J. Watson Center, NY
Email:{kchndrs2,xiangni2,kale}@illinois.edu

***Abstract*—Today's supercomputers are moving towards deployment of many-core processors like Intel Xeon Phi Knights Landing (KNL), to deliver high compute and memory capacity. Applications executing on such many-core platforms with improved vectorization require high memory bandwidth. To improve performance, architectures like Knights Landing include a high bandwidth and low capacity in-package high bandwidth memory (HBM) in addition to the high capacity but low bandwidth DDR4. Other architectures like Nvidia's Pascal GPU also expose similar stacked DRAM. In architectures with heterogeneity in memory types within a node, efficient allocation and data movement can result in improved performance and energy savings in future systems if all the data requests are served from the high bandwidth memory. In this paper, we propose a memory-heterogeneity aware runtime system which guides data prefetch and eviction such that data can be accessed at high bandwidth for applications whose entire working set does not fit within the high bandwidth memory and data needs to be moved among different memory types. We implement a data movement mechanism managed by the runtime system which allows applications to run efficiently on architectures with heterogeneous memory hierarchy, with trivial code changes. We show upto 2X improvement in execution time for Stencil3D and Matrix Multiplication which are important HPC kernels.**

I. Introduction

As we move towards Exascale era, the ratio of memory bandwidth to compute capacity of a node is expected to be low. In order to provide high bandwidth for such many-core platforms, stacked DRAMs can be used, like in Intel Xeon Phi, Knights Landing (KNL). Proposed future architectures like Traleika Glacier [1] for Exascale computing also envision a fast near memory and a slow far memory, namely Block Shared Memory and DRAM, respectively. Some architectures propose the use of Non Volatile Memory (NVM) as slow memory for scaling DRAM capacity, when the application's working set does not fit within DRAM. Stacked DRAM architectures exploit heterogeneity in memory types to sustain high memory bandwidth at large core counts per node. Architectures like KNL which are now being deployed in production (Stampede 2.0 and ALCF Theta) employ MCDRAM as the High Bandwidth memory (HBM), in addition to using DDR4 as slow memory. Slow memory can either mean high latency or low bandwidth or both. In KNL, DDR4 has about 4X lower bandwidth than MCDRAM with comparable latency for access. We show in Figure 1 the bandwidth difference for Stream benchmark [2] measured on an Intel Xeon Phi Knight's Landing. Due to much higher bandwidth of MCDRAM, it is

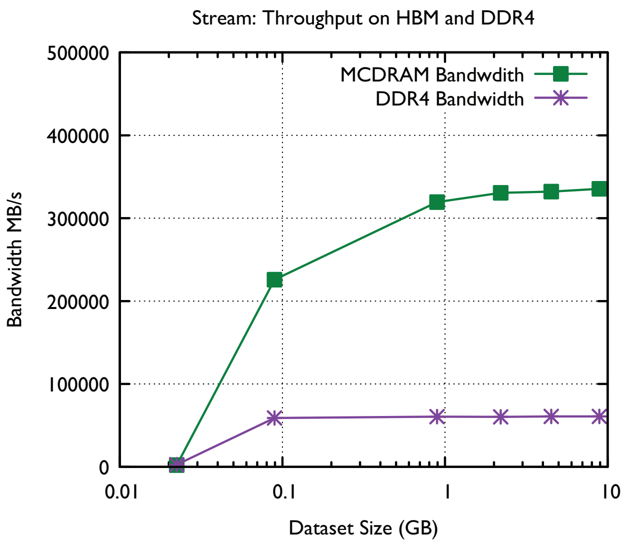


Fig. 1: Bandwidth comparison for stream

critical for bandwidth sensitive applications to make efficient use of HBM.

While architectures like KNL currently provide hardware caching mechanisms for fetching data into HBM, caching could result in increased latency from conflict misses or capacity misses and would not be practical for multiple tiers of memory hierarchy. Our focus is on handling memory heterogeneity in the absence of hardware-level caching mechanisms. Data movement costs are predicted to be the dominant energy costs in Exascale systems, shifting the onus for efficient data movement to the software stack.

Several HPC applications perform memory sensitive computation like Stencil3D, in which among 3D grid of objects, each object communicates with its 6 immediate neighbors. For dataset sizes worked on by 64 or 128 threads in many-core machines, the entire grid would not fit in small capacity HBM. As a result, any data that overflows from HBM and is allocated on slow memory can result in poor performance. In architectures like KNL, with upto 272 threads working simultaneously on the data, the application becomes bandwidth sensitive.

For Stencil3D, we observe in Figure 2 that the performance

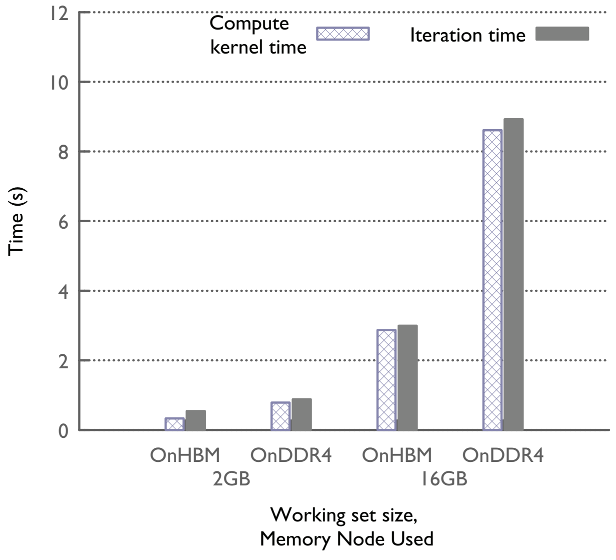


Fig. 2: Comparison of performance of Stencil3D on HBM and DDR4, when the dataset size fits in HBM. Compute kernel time refers to total time spent in bandwidth sensitive task.

on HBM is 3X higher than on DDR4, when the working set fits within HBM. This motivates our implementation within the runtime system for performing migrations across heterogeneous memory nodes, when the working set does not fit within HBM.

We present a runtime-aware prefetching mechanism within the CHARM++ [3] runtime system to prefetch and evict data to reduce the cost of low bandwidth access to far memory. We also demonstrate the overheads associated with data prefetch and eviction and scheduling. The contributions of this paper are:

- An implementation within the runtime system that tracks the data blocks used by tasks to perform memory-aware scheduling
- Incorporation of data movement inside the runtime system, i.e adding a system level abstraction that can be extended to other kinds of memory heterogeneity
- Performance evaluation of the proposed mechanism for bandwidth-sensitive HPC applications on Intel Xeon Phi KNL

The rest of this paper is organized as follows. Section II describes related work. In Section III we describe the CHARM++ programming model and runtime system, and KNL architecture. Section IV talks about design and implementation of scheduling and data movement strategies. Section V covers the evaluation of our strategies with commonly used HPC codes. Section VI is the conclusion.

II. Related Work

Khaldi *et al.* [4] study HBM aware allocation using compiler hints. This work uses compiler analysis to efficiently allocate bandwidth sensitive data in HBM. However, this study assumes that the bandwidth sensitive portion of data fits in HBM. Our work performs prefetch and eviction of data at the runtime level when the dataset does not fit in HBM.

Yount *et al.* [5] look at improving HBM usage in cache mode by performing tiling based mechanism to increase usage of cached data in HBM. This work focuses on cache mode in HBM. This work is not easily extendable to other applications, since it requires algorithmic changes to the application to reduce the working set size to enable it to fit within HBM.

In previous similar work [6] data reuse was explored in the context of simulation and analytics running on the same machine. It studies the bounded buffer problem in a producer-consumer scenario. It works at OS-level controls and does not explore lone applications with out-of-core memory requirements. Also, the experiments were performed on a functional simulator. We show our results using the latest Xeon Phi KNL.

Legion runtime system [7] allows data dependences between tasks to be indicated by programmers. The data dependence is used to generate parallelism and to move data across memory hierarchies. The evaluation of Legion runtime system is performed using GASNet memory to map data from remote nodes. Our focus is on evaluation of data movement strategies at node-level on architectures like KNL and future Exascale systems exploring within node memory hierarchy.

Sequoia [8] is another runtime system that performs task scheduling with the consideration of memory hierarchy. It is evaluated on Cell processor and distributed memory clusters, while our focus is on within node memory heterogeneity in recent and futuristic many-core systems.

In [9], similar mechanism of data prefetch and eviction has been used to use NVM as additional memory to relieve memory pressure when an application's dataset does not fit in DDR4. Our work uses similar infrastructure but is different since our work focuses on memory hierarchy where the slow memory is bandwidth restricted, whereas NVM is both bandwidth and latency restricted.

NVM has been used as persistent virtual memory [10] to perform out-of-core execution with OS based control. Mechanisms for limited buffer scheduling aim at migrating data between memory hierarchies by storing metadata or relying on application-supplied hints. The focus is on managing multiple applications running on a server. Also, prediction mechanisms for prefetching in general have to rely on regular application behavior.

III. Background

A. Charm++ Programming Model and Runtime System

We implement memory-heterogeneity aware execution using the CHARM++ runtime system. CHARM++ requires for work to be over-decomposed in work units called *chares*. Over-decomposition implies that there are more work units/*chares* than number of processors. Over-decomposition with migratability allows for load balancing of chares. We use overdecomposition of *chares* to schedule tasks that fit in the low capacity high bandwidth memory. Over-decomposition allows us to divide work units into pieces and schedule at a

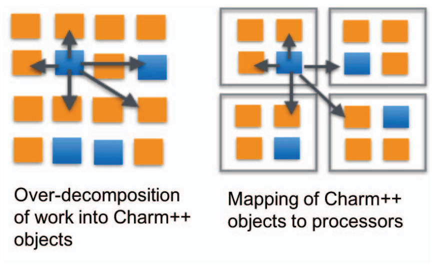

Over-decomposition of work into Charm++ objects
Mapping of Charm++ objects to processors

Fig. 3: Over-decomposition with CHARM++

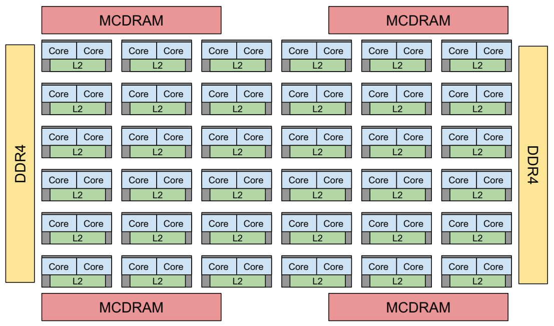


Fig. 4: KNL Architecture - Overview

time, only a set of work units that fit in HBM. Hence, even for very large input size, with the help of overdecomposition we can reduce the working set size of an application in HBM. The runtime system can use the knowledge of data block dependences for tasks to prefetch and evict data to mitigate performance penalty from accessing DDR4. It can avoid latency for fetching data by means of prefetching data. Naive methodologies that do not take memory heterogeneity into account would suffer from reduced performance as a result of increased accesses to low bandwdith memory.

The CHARM++ programming system allows us to provide simple hooks like attributes to indicate which methods are bandwidth sensitive. Each *chare* executes multiple entry methods or tasks per iteration in an iterative application. Entry methods allow a *chare*'s work to be broken down into finer grained tasks, to allow for overlap of communication and computation. While a *chare*'s entry method waits for its input data to arrive, the entry methods of other chares on the same Processing Entity (PE) whose input data is present, can be executed.

In Charm++, the converse layer performs delivery of messages for objects for execution. When a message arrives for an object, the converse scheduler delivers the message and in turn the object executes the corresponding *entry method* for the message. Objects do not migrate at anytime, they migrate only when load balancing explicitly moves them to a different PE. We look to intercept messages from the converse scheduler in order to perform prefetching for certain entry methods before having the objects execute the entry method. This is described later in IV-B.

B. KNL Architecture

We describe the architecture of Intel Xeon Phi Knight's Landing in detail. KNL has 68 cores with 4-way SMT, thereby providing 272 virtual cores if SMT is enabled. There are 34 L2 tiles in all, such that each tile is shared by 2 physical cores. We show a depiction of KNL in Figure 4. There are two types of memory: MCDRAM which is the High Bandwidth Memory with a capacity of 16GB and DDR4 which is the low bandwidth memory with a capacity of 96GB, which can be increased upto 384GB. MCDRAM has over 4X higher bandwidth than DRAM.

We now describe the different *memory modes* that can be configured in KNL.

1) Cache Mode: In cache mode, MCDRAM is configured to serve as a direct-mapped cache for data in DDR4. Any data that is accessed, that misses in MCDRAM, is fetched from DDR4 and cached in MCDRAM. Hence, misses in MCDRAM suffer from additional latency fetches from DDR4.
2) Flat Mode: In flat mode, MCDRAM and DDR4 are treated as separate memory nodes. Data can allocated in either memory nodes. This mode requires the application to perform memory node-aware allocation.
3) Hybrid Mode: Hybrid mode allows part of the MCDRAM to be configured in flat mode and part of the MCDRAM to be configured in cache mode with the DDR4. This avoids latency from misses for data in the flat mode portion of MCDRAM while also allowing memory node-agnostic allocation for applications with the partial cache mode.

Following *cluster modes* are possible in KNL.

1) All-to-All: Memory addresses are uniformly distributed across all tag directories on all tiles.
2) Quadrant: Memory addresses corresponding to the Memory controller in each quadrant are distributed to tag directories in tiles in that quadrant.

In our study we use KNL configured in Flat All-to-All mode. Flat mode allows programmer-control over allocation of data and is representative of architectures with memory hierarchy and memory heterogeneity. All-to-All cluster mode has the most impact on memory bandwidth, hence we use this mode since our emphasis is on heterogeneity in memory bandwidth.

IV. DESIGN AND IMPLEMENTATION

A. Data dependence API

In order to indicate the bandwidth-sensitive data in a CHARM++ application, we require the programmer to annotate the data and tasks with simple attributes.

First, the bandwidth-sensitive task or entry method of *chare* needs to be annotated with *prefetch* for the runtime to ensure that the entry method's data is prefetched into HBM before execution. Secondly, the task or entry method's data dependence needs to be marked, so that the runtime system can ensure that the data is prefetched before execution. Following shows an example from an excerpt of CHARM++ *.ci* file:

```
module Compute{
...
entry [prefetch] void compute_kernel() [
    ↪ readwrite : A, writeonly : B]
...
};
```

In the above example, `compute_kernel` method requires prefetching of its data dependences namely A and B of types readwrite and writeonly respectively. In the C++ file, we need to declare the type as *CkIOHandle* which allows the runtime system to store and query metadata about the data block.

```
class Compute : public CBase_Compute {
...
public:
CkIOHandle<double> A;
CkIOHandle<double> B;
...
};
```

We reuse the API from [9] for specification of data dependence and for storing metadata on the dependence data blocks.

B. Scheduling work units and data movement

We make changes to the Converse scheduler in order to perform prefetching at node level. Before a chare's entry method is about to be executed by delivery of its input message, we intercept the call and check whether the entry method needs prefetching of data. If so, instead of delivering the message we queue the message and the corresponding object in a queue. We use two queues types: *wait* queues and *run* queues. There is one run queue per PE, though we plan to use a node-level run queue in the future. The wait queue contains tasks that need data to be prefetched and the run queue contains tasks that are ready to be scheduled by the Converse scheduler. Tasks are picked up in FIFO order from the run queue and scheduled.

When an object of type prefetch arrives, the message for the object is not delivered, instead the object calls preprocessing method which is generated specifically for every prefetch method. Preprocessing and post-processing methods corresponding to *[prefetch]* type entry method is generated as part of charmxi tool's autogeneration of header files. This allows methods to be executed within the runtime before and after execution of the entry method of interest [9]. In this method, the object along with its input dependences, i.e the input data that were annotated as specified in IV-A and input message are encapsulated as an OOCTask. Once the object's prefetch annotated entry method finishes executing, there is a similar post-processing method which allows for data eviction as needed. The prefetch and eviction scheduling policies implemented are described below:

```
Algorithm: IO thread
while While space remains in HBM do
    pop first task in wait queue;
    bring in data for task;
    if all data for task in HBM then
        add task to run queue;
    else
        bring in remaining data;
    end
end
Data blocks not in use are evicted to DDR4;
```

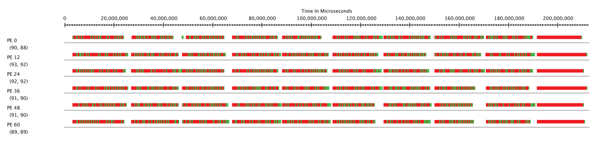
Algorithm 1: General Prefetch and Evict strategy

Multiple queues, Single IO thread In this strategy, we use multiple queues to add all pending tasks and a single IO thread to prefetch data and evict data. Multiple queues allow handle load imbalance that can arise if the IO thread served IO requests for chares mapped to the same PE. For example, with a single wait queue, it is possible that IO thread prefetches data for n tasks on PE0 instead of fetching data for n tasks on n PEs. This can result in load imbalance when there is no space left in HBM and tasks on some PEs have to wait for other tasks to finish execution to free up HBM space. This results in load imbalance across PEs. We avoid such load imbalance by having one queue per PE, so that the IO thread can serve same number of requests for each wait queue at a time, thereby serving all PEs equally. Every task/entry method, in its *preprocessing* method, checks whether all its data dependence blocks are in memory. The CkIOHandle specified in IV-A has two states: *INHBM* and *INDDR* that indicates whether the data block is in HBM or in DDR4. A task checks if it is ready to execute, i.e. if all the data dependences are in INHBM, if so, the task is immediately added to the *run queue*, ready to be executed. However, if the task is not ready, that is, not all data dependences are in state *INHBM*, the worker thread locks the corresponding PE's wait queue and adds the task to *wait queue[PEindex]*. The IO thread waits conditionally for a signal. When a worker adds a task to the wait queue, it sends a signal to wake up the IO thread to begin processing from the wait queue. The IO thread then wakes up, locks each wait queue (one per PE) one by one and pops the first candidate task in the queue. It then goes through the task's data dependences and for any dependence that is *INDDR*, brings it into HBM and changes its state to *INHBM*. It then verifies that all its dependences have been brought into HBM and adds the task to the *run queue* of the corresponding PE, ready to be scheduled by the Converse scheduler. The IO scheduler keeps track of the HBM memory in use out of the total 16GB by keeping track of each block size being brought into HBM. If there are no more tasks in the wait queue or if allocating a data block would exceed the remaining HBM capacity, then the IO thread goes to sleep/conditional wait. Whether a data block is in use, is tracked by a reference counter, incremented every time a task depending on the block is scheduled. When a task finishes execution, it evicts its data dependences to DDR4, if they are not currently in use by another task, by checking the data blocks' reference counts. If the IO thread is sleeping, the task wakes it up after the eviction, so that more data can be prefetched asynchronously.

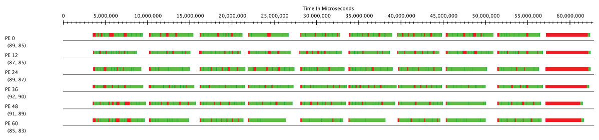
One of the issues with this scheduling strategy is that the IO thread preteches data for all tasks. Hence, before an iteration begins the IO thread fetches data for atleast 64 tasks at a time, if there are 64 worker threads. Hence some tasks have to wait longer for their data to be fetched. This causes some additional wait time as shown in 5a. We mitigate load imbalance by using one wait queue per PE. Another mechanism to mitigate load imbalance could be by using a node-level run queue.

Multiple queues, no IO thread We use parallel fetch and eviction in order to reduce overhead for prefetching data. This handles the load imbalance problem described earlier and also reduces overhead of all worker threads waiting for a single IO thread, especially for many-core systems like KNL. Each wait queue is a FIFO-based queue. When a task arrives on a PE, if there is sufficient allocation space in HBM, it fetches it's own data in the *preprocessing* step. If it is able to bring in all its dependences to HBM, then it schedules itself by adding itself to the corresponding PE's *run queue*. If there is no space in HBM, it adds itself to the PE's *wait queue*. When a task finishes executing, it calls its postprocessing step, where it evicts its own data dependences, as long as they are not in use by other tasks, by checking the reference count on the data blocks. After evicting its own data, it checks in the wait queue on its PE, to see if there are any tasks waiting to be scheduled on the PE. As a result of its own data eviction, it can now bring in data blocks for a waiting task and schedules the task onto the run queue. This mechanism allows for parallel fetch and parallel evict instead of a single IO thread performing all fetches and evicts. However this method has two drawbacks. The fetch and evicts are synchronous calls, as a result, they add overhead to the execution time. This time could worsen, if the data dependence count is very high, requiring a large number of blocks to be fetched and evicted for each task.

Multiple queues, multiple IO threads The next strategy we employ has the benefits of both of the previously described mechanisms. It has two benefits, namely, it uses multiple wait queues (one per PE), to allow for parallel processing of fetches and evictions along with ensuring load balance in the tasks that are scheduled and it uses multiple IO threads so as to allow the fetch and evict to be performed asynchronously, thereby allowing tasks to run as data is being prefetched for the next task. In this implementation, there is one IO thread per worker thread. When a task arrives at its preprocessing step, it simply adds itself to the corresponding PE's wait queue. The IO thread is then woken up by the worker thread. Each IO thread pops tasks from the wait queue of that PE and brings in data till the HBM is full. All IO threads are likely working in parallel, hence there is no starvation problem. The IO thread then adds tasks to the corresponding run queue and enters conditional



(a) Multiple Queues, Single IO thread



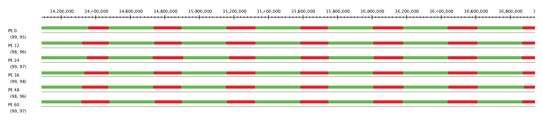
(b) Multiple Queues, Multiple IO threads (Asynchronous)

Fig. 5: Projections of Stencil3d comparing naive HBM allocation with Single and Multiple IO threads' asynchronous data prefetch

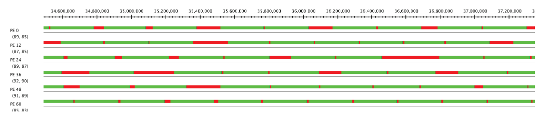
wait, since there are no more tasks in the wait queue or the HBM is full. Once a task finishes execution, in the post process step, it evicts its own data, evicting only those that have a reference count of 0. It then wakes up the IO thread for the PE, since it has evicted data, allowing any more additional tasks to have their data prefetched and be scheduled. We also plan on finding more optimal IO thread count such that one IO thread can be assigned to a subgroup of wait queues. Currently the IO threads are scheduled on the hyperthread cores corresponding to the worker threads, so as to not increase the usage of the number of physical cores being used.

No Prefetch/Evict - Baseline In our baseline mechanism, we do not perform any prefetch or eviction of data. *numactl* tool allows controlling the memory node on which the data for the process needs to be allocated. We assume sufficient data is available from the combination of both memory nodes (HBM and DDR4). We use `numa_alloc_onnode` described in Section IV-C to place data blocks in HBM and any remaining data blocks that do not fit within the 16GB HBM are placed in DDR4, which is memory node 0. The numactl policy *–preferred 1*, which indicates that it is preferable to allocate data to memory node 1 (HBM) is an alternate way of performing naive allocation. We use the former mechanism to maintain consistency across *libnuma library* allocation APIs used by the runtime system for prefetch and eviction. We allocate close to 15GB or more on HBM in Baseline case depending on data block sizes, ensuring that we do not oversubscribe the HBM memory.

Performance visualization We show differences in performance in the different strategies via projections, a performance visualization tool for CHARM++. Figure 5 compares the data prefetch and eviction performed with single IO thread with asynchronous data prefetches performed with multiple IO thread. The red portion shows wait time caused due to delays from scheduling tasks, data prefetch, eviction and locking of queues and data blocks. As can be seen, single IO thread has a lot more overhead (red) than multiple IO threads case.



(a) Zoomed In: Multiple Queues, No IO thread (Synchronous)



(b) Zoomed In from 5b: Multiple Queues, Multiple IO threads (Asynchronous)

Fig. 6: Projections of Stencil3d comparing synchronous and asynchronous data prefetch. Synchronous incurs fetch/evict overheads while asynchronous masks these overheads.

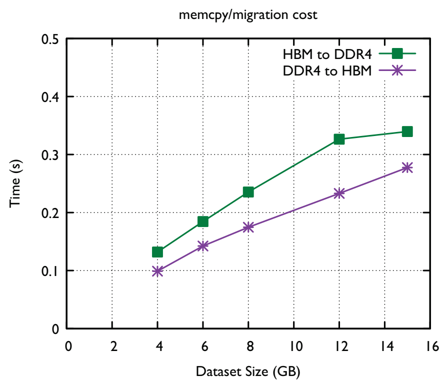


Fig. 7: memcpy cost for data migration

Figure 6 shows difference between synchronous and asynchronous data fetch. We observe that the preprocessing time before compute kernels which is of order of 20 ms is removed from asynchronous scheduling. There are still some delays that are caused by waiting for queue locks and data block locks.

C. Data movement Methodology

For moving data across memory hierarchies, we provide a hardware level abstraction that can be ported across different memory hierarchies.

We use two operations to allow data movement across HBM and DDR4: create space in destination memory and then move the data to the destination location. Here move itself is a two step process, consisting of copy to destination and then freeing the source. The creating of space in destination memory could be avoided if we maintain a memory pool in each memory type. We plan to perform this optimization in the future to further reduce the overhead of prefetch.

```
void *numa_alloc_onnode(size_t size, int
  ↪ node);
```

numa_alloc_onnode allows allocation of a data block on a memory node. HBM is exposed to the userspace as Memory node 1 and DDR4 is exposed as Memory node 0.

Once a same sized buffer on the destination memory is allocated, a data block is moved to the other memory node, by performing memcpy. memcpy can be used to move data between buffers within a memory node or between memory nodes, in userspace. It is described below:

```
void * memcpy ( void * destination, const
  ↪ void * source, size_t num );
```

Then we free source buffer by numa_free.

We use memcpy as the mechanism for moving data between the memory nodes for two reasons.

- Pointers to buffers are more relevant to the data types used by the CHARM++ runtime system compared to other recommended operation of migrate_pages which migrates memory pages. This would require conversion between bytes and 4K sized pages or other size depending on the OS setting. It might require padding to perform correct conversions.
- Previous work [11] paper evaluates migrate_pages and memcpy mechanisms and projects memcpy to be a more scalable mechanism for Xeon Phi KNL

D. When to Prefetch

By principle, the prefetch and eviction of data needs to be overlapped with the computation of tasks. Over-decomposition allows for such overlap between data prefetch and computation.

We present the cost associated with prefetching and evicting data to provide an insight into the associated overheads. In order to measure the cost of migrating data between HBM and DDR4, we try to stress the bandwidth by having 64 threads simultaneously perform prefetches. In addition, we consider a working set size such that high amount of data is moved between the memory nodes. The average costs associated, in seconds, with the main step performed as part of the data migration routine, memcpy is shown in Figure 7. We find memcpy costs for HBM to DDR4 to be slightly higher.

V. EXPERIMENTS

We evaluate Stencil3D and Matrix Multiplication , on Intel Xeon Phi Knights Landing. Stencil3D accesses large amounts of data in quickly executing loops which makes it bandwidth sensitive. Matrix multiplication with optimizations for Xeon Phi KNL and with vectorization becomes bandwidth sensitive as a result of several threads simultaneously accessing data from memory. Our experiments are conducted on a single machine to focus solely on within node memory heterogeneity. The KNL node used in the experiments has the configuration: Flat, All-to-All mode and was one of the nodes from Stampede 2.0. Each KNL node has 68 physical cores with 4-way SMT, hence providing a total of 272 cores. HBM has about 4X higher bandwidth than DDR4 as shown in Figure 1. In our study, all active application data is accessed from HBM. While some of DDR4 bandwidth is used for prefetching, the remaining DDR4 bandwidth could potentially be used to augment the bandwidth provided by high bandwidth memory. The capacity of HBM, MCDRAM is 16GB, whereas the capacity of DDR4 is 96 GB, 6 times that of HBM. We use only 64 out of the 68 cores for our experiments. An aspect we do not consider in our study is comparison with cache mode, which will be considered in the future. We do not use SMT since different applications could benefit differently from hyperthreading [12].

A. *Stencil3D*

In this section, we show speedup normalized to baseline performance described in Section IV-B, for Stencil3D on KNL. Stencil3D is a communication and memory intensive benchmark. It is a commonly used kernel in several applications like MIMD Lattice computation. It involves communication with its immediate neighbors in 2D or 3D space. For our evaluation purposes, we consider only Stencil3D. The total working set size for the grid that we consider is 32 GB. Our scheme supports total working set size upto the capacity of DDR4. The reduced working set size as a result of over-decomposition is varied between 2GB and 8 GB. We perform 20 iterations to mimic tiling patterns that increase computation to reduce the overhead incurred by data communication [13], a commonly used technique of performing computations after receiving the updated values from neighbors in the grid. The Stencil3D communication and computation pseudo-code is shown in Algorithm 2.

```
Algorithm: Run method on each stencil chare
while not converged do
    Perform iteration:
    while receive message from neighbors do
        collect all data;
    end
    update all grid elements with received data;
    send updated data to neighbors;
end
```

Algorithm 2: Stencil3D computation

Figure 8 shows the application iteration time speedup for different queuing strategies.

Single IO thread. We observe considerable slowdown in the application iteration time when performing Single IO thread fetches. This is because, in Stencil3d, the update of grid elements by each chare is done independently, i.e. each chare reads and writes to independent data blocks in each iteration. As a result, the IO thread needs to perform prefetch of blocks for each chare on each PE, hence increasing the wait time as observed in projections in Section IV-B.

Multiple queues, no IO thread. Compared to Single IO thread this performs better since each chare has to wait only for data for itself as the prefetch and eviction is done in parallel.

Multiple queues, Multiple IO threads. This shows best performance since the prefetch and eviction are done asynchronously. For applications that have low sharing of data blocks across tasks, multiple IO threads would work best.

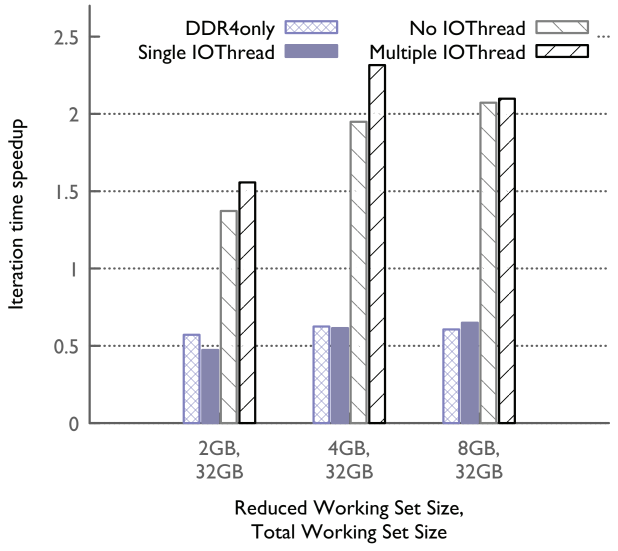


Fig. 8: Speedup from data movement in Stencil3D. Single IO thread is significantly slow since it fetches data for atleast one chare per PE, for all PEs, before scheduling the tasks.

B. *Matrix Multiplication*

Matrix multiplication divides the work units into a 2 dimensional array of chares. The data is divided such that the entire 2D grid of elements for input matrices A and B and output matrix C are distributed into blocks of *sub-rows* X *sub-columns* across the 2D array of chares. A and B input matrices are readonly blocks and hence can be shared across chares. The IO threads process the chares in a FIFO manner. For our experiments, the total working set size for the matrices is varied between 24 GB and 54 GB, while keeping the reduced working set size constant at 6GB. Increasing the total working set size allocates several blocks to DDR4, in Naive method. Since several input A and B blocks are reused across chares, as a result of overflow to DDR4, we see a significant slowdown in Naive method as we increase the total working set size.

In our Matrix multiplication implementation, we use MKL's `cblas_dgemm` calls which are highly tuned for performing matrix multiplication computations. In order to share the common input readonly blocks across tasks depending on them, we use a nodegroup in CHARM++ which allows caching of data at node-level. We find that the MKL library's `cblas_dgemm` call for KNL has been optimized internally to perform allocations on HBM. It is likely that the MKL library uses the memkind library [14] in order to perform allocation of datastructures created and used within the call. In order to make our measurements independent of such optimizations, we set `MEMKIND_HBW_NODES` to 0, so that such allocations performed internally go to memory node 0, i.e low bandwidth memory DDR4 in all our runs. This gives us explicit control over where the primarily used matrices (input matrices A and B, and output matrix C) are allocated. For DDR4only case, all matrices are allocated on DDR4. For Naive case, once the MCDRAM is full, remaining sub-blocks of matrices A, B and C are allocated to DDR4. As before, with fetch and eviction strategies, data is allocated on DDR4 and fetched into MCDRAM before being accessed by the dgemm call.

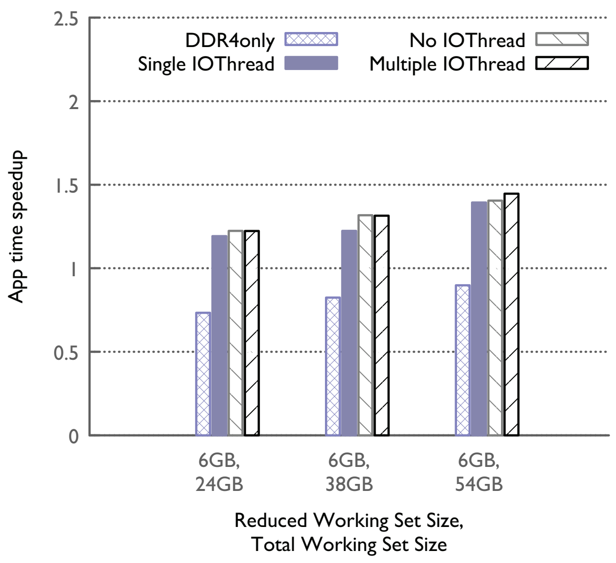


Fig. 9: Speedup from data movement for Matrix Multiplication. Single IO thread performs as well as Multiple IO threads, due to high data reuse of read-only data blocks.

Single IO thread. We observe in Figure 9 that single IO thread performs almost as well as other cases, primarily due to high data reuse in matrix multiplication. When a read-only block is being used by another chare, it is not evicted. As a result, when a data block is fetched into HBM, it is consequently reused before eviction to DDR4.

Multiple queues, no IO thread. Parallel fetch does not see much additional improvement compared to Single IO thread which benefits from data reuse.

Multiple queues, Multiple IO threads. Similar to parallel fetch, results are comparable to Single IO thread case. For applications with high data block sharing and reuse, Single IO thread would work well and incur low overhead.

VI. Conclusion

In this paper, we demonstrate the benefits of memory-heterogeneity aware execution within the runtime system and showe large speedups on the latest Intel Xeon Phi KNL in flat mode memory configuration. We demonstrate scalable mechanisms for prefetching and evicting data for many-core systems with memory heteroegeneity. Our implementation within the runtime system allows for such mechanism to be used across any set of HPC applications by making trivial annotations in the code.

Benefits were shown on a heterogeneous memory architecture where memory nodes differ in their bandwidth. Architectures with heterogeneity in both latency and bandwidth would benefit even more. We plan to extend this implementation to other heterogeneous memory architectures. We will also perform comparisons with cache mode in KNL in the future and in multi-node cluster settings.

References

[1] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu, “Runnemede: An architecture for ubiquitous high-performance computing,” in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 198–209. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2013.6522319

[2] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[3] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, “Parallel Programming with Migratable Objects: Charm++ in Practice,” ser. SC, 2014.

[4] D. Khaldi and B. Chapman, “Towards automatic hbm allocation using llvm: A case study with knights landing,” in *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 12–20. [Online]. Available: https://doi.org/10.1109/LLVM-HPC.2016.7

[5] C. Yount and A. Duran, “Effective use of large high-bandwidth memory caches in hpc stencil computation via temporal wave-front tiling,” in *Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, ser. PMBS ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 65–75. [Online]. Available: https://doi.org/10.1109/PMBS.2016.12

[6] K. Chandrasekar, B. Seshasayee, A. Gavrilovska, and K. Schwan, “Improving data reuse in co-located applications with progress-driven scheduling,” in *RESPA Workshop, co-located with SC*, vol. 15, 2015.

[7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[8] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, “Sequoia: Programming the memory hierarchy,” in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC ’06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188543

[9] X. Ni, “Mitigation of failures in high performance computing via runtime techniques,” Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2016.

[10] J. C. Mogul, E. Argollo, M. A. Shah, and P. Faraboschi, “Operating system support for nvm+ dram hybrid main memory.” in *HotOS*, 2009.

[11] 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.

[12] C. Rosales, J. Cazes, K. Milfeld, A. Gómez-Iglesias, L. Koesterke, L. Huang, and J. Vienne, *A Comparative Study of Application Performance and Scalability on the Intel Knights Landing Processor*. Cham: Springer International Publishing, 2016, pp. 307–318. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46079-6_22

[13] J. Ramanujam and P. Sadayappan, “Tiling multidimensional iteration spaces for nonshared memory machines,” in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’91. New York, NY, USA: ACM, 1991, pp. 111–120. [Online]. Available: http://doi.acm.org/10.1145/125826.125893

[14] C. Cantalupo, V. Venkatesan, J. R. Hammond, K. Czurylo, and S. Hammond, “User extensible heap manager for heterogeneous memory platforms and mixed memory policies,” *Architecture document*, 2015.