Noname manuscript No.
(will be inserted by the editor)

High-Level Approaches for Leveraging Deep-Memory Hierarchies on Modern Supercomputers

Antonio Gómez-Iglesias · Ritu Arora

Received: date / Accepted: date

Abstract There is a growing demand for supercomputers that can support memory-intensive applications to solve large-scale problems from various domains. Novel supercomputers with fast and complex memory subsystems are being provisioned to meet this demand. While complex and deep-memory hierarchies offer increased memory-bandwidth they can also introduce additional latency. Optimizing the memory usage of the applications is required to improve performance. However, this can be an effort-intensive and a time-consuming activity if done entirely manually. Hence, high-level approaches for supporting the memory-management and memory-optimization on modern supercomputers are needed. Such scalable approaches can contribute towards supporting the users at the open-science data centers - mostly domain scientists and students - in their code modernization efforts. In this paper, we present a memory management and optimization workflow based on high-level tools. While the workflow can be generalized for supercomputers with different architectures, we demonstrate its usage on the Stampede2 system at the Texas Advanced Computing Center that contains both Intel Knights Landing and Intel Xeon processors, and each Knights Landing node offers both DDR4 and MCDRAM.

1 Introduction

Performance of the memory system/sub-system often does not match the computational capability of the latest processing elements that are available in modern supercomputers. Even though the single core performance does not continue to grow as it did in the previous decades, the number of cores per processor have gone up with a decrease in the clock frequency of those cores. This comes from

A. Gómez-Iglesias
Intel Corporation
E-mail: antoniogi@gmail.com

R. Arora
Texas Advanced Computing Center, The University of Texas at Austin
E-mail: rauta@tacc.utexas.edu

the fact that dynamic power requirements for a transistor depends on the energy of a logic transition multiplied by the frequency of transitions [5].

The continuous growth in computing capabilities in existing and upcoming supercomputers has led to a memory bottleneck. Memory technologies (DDR to DDR4) have improved to offer more bandwidth. However, the growth rate for the memory has not been able to keep up with the growth in computing capabilities of the cores in the chip. This has meant that the memory bandwidth per core has decreased. While several High Performance Computing (HPC) applications benefit from the higher FLOPS that the modern compute nodes provide, there are several other memory-intensive applications (e.g., image processing applications [4] and Deep Neural Networks) that still suffer from the limited memory-bandwidth.

In order to address the memory-bandwidth issue, the manufactures are designing chips that have High Bandwidth Memory (HBM) [8] integrated ön package". The most recent self-bootable Intel Xeon Phi generation (Knights Landing - KNL) cards are packed with 16 GB of HBM that is named as MCDRAM (Multi-Channel DRAM - Intel's proprietary memory). The MCDRAM can be configured either as a third-level cache or as a NUMA (Non-Uniform Memory Access) node, and up to 384 GB of DDR4 memory. With the specification of HBM2 already defined, along with the fact that the HBM3 specifications ensuring higher memory-bandwidth than HBM2 have already been outlined by some manufacturers, it is clear that HBM/HBM2/HBM3 will continue to feature in the next generation supercomputers. In addition to HBM, some of the latest supercomputers also incorporate another level in the memory hierarchy in the form of Non-Volatile RAM (NVRAM). This further expands the possibilities of addressing the memory bandwidth needs of the users along with increasing the complexity of the system and, potentially making an application complex when trying to reach peak performance.

Based on the recent trends in large-scale HPC systems, and despite the fact that Intel has discontinued its KNL product-line, we consider that the KNL chip is a good representation of the complexity in the design of processors in current and future generation supercomputers. The KNL chip (or package) has: 1) many-core architecture where each core runs at a relatively low frequency and implements several simplifications over other Intel Xeon chips in order to meet power requirements [20]; 2) contained power consumption; and 3) different types of memories that satisfy different requirements (large capacity - low bandwidth; limited capacity - higher bandwidth). The inclusion of different types of addressable memory in a package increases its overall complexity and it is not a straightforward task to efficiently harness its capabilities. The memory-intensive applications running on such packages need to be adapted to achieve the best possible performance. Without these modifications, such applications will see an increased execution time and an overall reduction of the efficient utilization of the resources.

When running applications on systems with deep-memory hierarchies, it is important to analyze the memory usage characteristics of the applications, and determine the memory layers in which the applications' data structures should reside. This, however, can be a challenging task for many software developers. Moreover, different supercomputers target different communities: some systems are designed to meet the requirements of a handful of users and applications [3], while other machines are conceived with the aim of supporting large number of users, collaborations [22,15] and applications. In the latter, users of these systems are often either domain experts or graduate students who are using such complex

systems for the first time. Often times, these users do not have the skills or time to adapt their applications to take advantage of the latest features on the supercomputers. Therefore, high-level tools for memory-usage optimization on modern supercomputers are needed. Such tools can significantly reduce the amount of effort and time required for optimizing the applications. While these tools might not reach the performance that advanced and expert users would be able to attain, they should still be able to adapt the applications to obtain a significant percentage of the peak performance.

In this paper, we present a workflow based on high-levels tools for helping users in taking advantage of the deep-memory hierarchies without knowing the low-level microarchitectural details. The workflow is designed for usage on production-quality supercomputers and is already available on petascale systems used by thousands of users worldwide [21]. The tools can work in the user-space without the need of privileged access. This simplifies the implementation of the workflow on many production systems. The results presented in this paper demonstrate the value-addition done by our workflow during the process of memory-usage optimization. The rest of this paper is organized as follows: Section 2 introduces the related work; Section 3 details the overall workflow and the tools involved in its implementation. Section 4 introduces a set of benchmarks and real applications that were used for getting the results presented in this same section. Finally, Section 5 summarizes the main findings of the paper and introduces our future work.

2 Related Work

RTHMS [13] is a tool that analyzes parallel applications and provides a set of recommendations regarding data placement on systems with different types of memory. This work is similar to the work done by [11] in the sense that both define a metric for placing data structures on the HBM (MCDRAM). They both analyze whether a data structure fits in the available memory so that it can be fully allocated in it. RTHMS also considers some memory access patterns, which makes it a more advanced option. Such tools optimize the applications at compile time, and while they are useful, they can potentially mispredict the data structures that should be placed on the HBM since they lack critical information that only exists at run-time, like size of the dynamically allocated data structures.

Authors in [2] propose a runtime system that automatically moves data to and from MCDRAM on KNL when needed using the Charm++ runtime system. It, however, needs the users and programmers to annotate the code to describe which data structures might be bandwidth-sensitive. This is an effort that many users are either not willing to undertake or they cannot afford.

Both the aforementioned efforts focus on a particular part of the problem that we have described in Section 1 of the paper. Advanced users can use these approaches or they can use any of the available profilers to extract the information required for identifying the data structures that should be allocated on the HBM. Based on our experience, it can be stated that such advanced users rarely rely on new tools that simply act as an additional layer on top of the tools that they can already use. However, for not so advanced users, these tools still do not hide enough hardware complexity to make them useful.

Arora et al. developed a tool [1] to guide the users in compiling and running their applications efficiently on KNL nodes. The tool can also be used to interactively adapt codes developed in C/C++/Fortran to selectively allocate certain data structures used in the application on the MCDRAM. The authors mention extending the tool to support advance vectorization and provide advanced options for memory-usage optimization as part of their future work.

Rosales et al. [18] presented a tool that monitors the utilization of hardware resources on supercomputers by a given application with a very low overhead. This tool is in production on a number of supercomputers with thousands of users having access to it. It allows users to gather information about a vast number of resources available on a compute node. It also offers a summarized and a simple overview of the main issues that an application can be suffering from while running on a supercomputer.

3 Memory Management and Optimization Workflow

We present a user-friendly memory management and optimization workflow for applications running on modern supercomputers. The workflow can be adapted and adopted for future exascale systems as well. It consists of a set of tools for analyzing the application characteristics, especially those related to memory utilization. Users are not required to perform any manual re-engineering of their applications.

Our recommended workflow is diagrammatically shown in Figure 1 and the steps labelled in this Figure are explained in the following subsections. For simplicity and reproducibility, we are going to describe the implementation of the workflow on a supercomputer consisting of KNL nodes. We use a collection of external tools, libraries, hardware/performance counters, and the data in `sysfs`. All the steps in our workflow run in user-space, without the need for privileged permissions, to ensure portability and to simplify adoption. We expect that this workflow can be adapted for other chips with deep-memory hierarchies since those chips are very likely to have similar libraries and tools too.

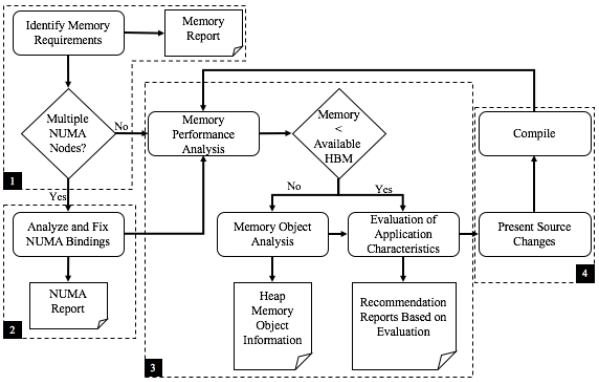

Fig. 1: Overall workflow. It consists of four different stages with reports after each step to ensure that users can make informed decisions regarding memory management and optimization.

3.1 Identify Memory Requirements

As shown in Figure 1, the first step of our workflow consists of identifying the memory requirements of the target application. This step applies to all the compute nodes used during run-time. The information that we collect per node includes:

- Memory used by the application: if the application consists of several other applications running on the same node, which is a typical case for many workflows or in Multiple Instruction, Multiple Data (MIMD) scenarios, we collect the overall memory footprint. We also consider that the actual memory used by an application includes both virtual memory and the ramdisk in `/dev/shm` since that also counts towards the main memory utilization.
- Free memory: capturing the memory used by an application can be difficult or the data might not be completely accurate under some scenarios. As a result, we also collect the information on the amount of memory available at all times during the application run-time. This value can provide a better insight in cases where the out-of-memory killer terminates the execution of the target application.
- NUMA utilization: similar to tracking main memory utilization, when NUMA is present, the utilization of each NUMA node is captured. This can be used to quickly detect NUMA affinity problems in NUMA-aware applications or NUMA problems in unoptimized applications.
- NUMA statistics: we collect the number of NUMA hits and misses and whether the hits or misses are local or remote. This information is important to later on analyze how NUMA-aware the code is and to decide if the code can be modified to improve the performance.

At this step of the workflow, if there is only one NUMA domain, step 2 is skipped since that stage of the workflow focuses on fixing the NUMA bindings.

3.2 Analyze NUMA Utilization and Fix NUMA Bindings

In the previous step, we collected information about NUMA utilization as well as a set of NUMA counters. This information is analyzed and presented to the users so that they decide whether to continue with the workflow or to stop. This step of the workflow does not introduce code modifications, but rather helps users with an easy approach to improve the NUMA bindings of their applications. As previously stated, we are using the Intel KNL processor to demonstrate our recommended workflow. This processor supports different *cluster modes* [20], that are basically boot-time configurations, to expose the available MCDRAM as different NUMA nodes. In some of the cluster modes (`all-to-all`, `hemisphere` and `quadrant`), the MCDRAM is configured as a single NUMA node. However, there are other modes (`sub-NUMA, or SNC 2 or 4`) where the MCDRAM is configured as 2 or 4 different NUMA nodes.

If the MCDRAM is configured in `sub-NUMA 2 or 4` mode, the operating system will see the available MCDRAM as 2 or 4 NUMA nodes (and also, it will see 2 or 4 NUMA nodes for the DDR4 memory, resulting in 4 or 8 total NUMA nodes). Even though there is no physical distinction between the hardware based on how it is configured in software, as it is still the same chip, the mechanism for data

access changes. Accessing data that is not located on the memory controller of the local NUMA domain [20] imposes a latency penalty in the same way as accessing remote memory on a multi-socket system.

The workflow generates a set of plots that represent the NUMA utilization during the execution of the target application. The information on NUMA utilization is collected using the `numastat` tool. These plots include the memory available and used on each NUMA node (which considers all NUMA nodes independent of their level in the hierarchy), the local NUMA hits and misses, as well as the remote hits and misses. The visualization of the results gives a very quick idea of where problems reside at run-time.

This step of the workflow is required for those applications whose memory footprint makes them suitable for running entirely within a subset of the NUMA nodes. In systems with deep-memory hierarchies, different NUMA nodes will present different characteristics (i.e., some nodes will have HBM). There are different options for using the different levels in the hierarchy. If the memory requirements of an application can be met by the HBM, then all the data structures needed by the application can be created there. Typically, `numactl` is used to achieve this.

In cases where there is a single NUMA node for each type of memory in the hierarchy, using the `numactl` tool with the appropriate flags is sufficient to specify which NUMA node must be used first. The problem arises when there are multiple NUMA nodes for each type of memory in the hierarchy. In this case, setting the correct policy with `numactl` is more complicated if the target application has been parallelized with MPI. Consider the scenario where, in one node, the application is executed as `mpirun ./target_app` and the memory hierarchy of the node consists of DDR4 and HBM. There are two sockets in the node, and each socket has its own local DDR4 and HBM memory. In this case, there are two NUMA nodes for DDR4 and another two for the HBM. Also in this case, suppose that the two DDR4 NUMA nodes are nodes 0 and 1, while the HBM nodes are 2 and 3. By default, the application would run out of DDR4. However, if the user introduces `numactl` to set the memory bindings to use HBM, the application will use one of the two HBM nodes until full and then use the other HBM node. This can happen by using the following command: `mpirun numactl --membind=2,3 ./target_app`. However, there will still be NUMA misses. We introduce a tool that detects and fixes the CPU affinity of each individual MPI task at runtime. Based on that CPU affinity, it instructs `numactl` to set the memory policy to use the local NUMA nodes by default. Also, we use the `preferred` option for `numactl`, so that the application would use first the *local* HBM NUMA node and then, when all the HBM is completely used, it would allocate memory on the local DDR4 node. This tool can be configured to select which one of the local nodes to use. Our approach is independent of the MPI library used to compile/run the application and does not need any additional configurations. Thus, the command looks like `mpirun fix_affinity ./target_app`.

At this point, if the memory footprint of a given application is smaller than the available HBM, the workflow ends. However, if the memory access pattern of the given application is latency-sensitive instead of bandwidth-sensitive or the memory footprint is larger than the capacity provided by the HBM, there are additional steps in the workflow.

3.3 Identify Bandwidth-Critical Data Structures

As shown in Fig. 1, the third step while analyzing applications for improving performance is to identify the bandwidth-critical data that can reside on the HBM (or MCDRAM) instead of the DDR4. One of the tools in our workflow, named **ICAT**[1], can be used to analyze the application characteristics for identifying this data. ICAT runs the target application and gathers the information on microarchitectural events - such as instructions per cycle, L1 cache loads, L1 cache stores, L1 cache load misses, L1 cache store misses, Last Level Cache (LLC) loads, LLC cache stores, LLC cache load misses, and LLC cache store misses. It also collects the information on the fraction of cycles for which the processor was stalled on the different levels of cache and the DRAM. If needed, the tool computes the total sizes of the memory objects in an application, and the memory access patterns (such as strided access) by internally using hardware performance counters, software profiles, and built-in heuristics. These heuristics are related to: 1) size of the data structures, 2) the number of accesses, 3) whether or not the data structures are memory bound, 4) whether or not the data structures are DRAM bound, and 5) the amount of memory needed by the entire application. If the application fits into L2 cache then flat-mode with default settings is recommended and the memory allocation is done on DDR4. Otherwise, ICAT checks if the application fits in MCDRAM and `numactl` is available - if `numactl` is available, then flat-mode with all allocation to MCDRAM is recommended but if `numactl` is not available, then cache-mode is recommended with a warning of performance-penalty. Using the aforementioned information and metrics, ICAT does the analysis for decision-making purposes. On the basis of its analysis, it guides the users in prioritizing the allocation of memory objects on the HBM[1].

As mentioned in Section 3.2, the KNL nodes can be configured in different *memory* and *cluster* modes, and the performance of an application can be impacted by the choice of these modes [17]. While there are some default configurations of these modes recommended by Intel, one would need to understand the memory usage characteristics of a given application to determine the most suitable modes to use for running it. There can be some trial-and-error involved in this process, and tools like Vtune [16] could be required to understand the memory usage pattern of a given application. Understanding the output of Vtune and making the decision on the appropriate modes to use can be difficult for many supercomputer users. Such users can take advantage of the decision-support tool that is incorporated in our workflow. Our tool already analyzes the aforementioned microarchitectural characteristics of a given serial or parallel application to advise on bandwidth-critical data structures and it can also recommend the best KNL configuration modes for running a given application. It prepares the recommendation reports for the user and also advises on the appropriate usage of the `numactl` options.

3.4 Iterative Code Adaptation for Optimizing Memory Use

The applications running on KNL processors that have the MCDRAM configured as an L3 cache are not required to undergo any change. However, if the MCDRAM

[1] https://colfaxresearch.com/knl-mcdram/

is configured in flat-mode, as an extension of the DDR4 memory address space, the users can choose to allocate memory for specific bandwidth-critical data structures on the MCDRAM using the `hbwmalloc` interface, and thereby, gain some performance [7]. ICAT can determine if there are bandwidth-critical data structures that should be allocated to the MCDRAM. It can not only inform the user but can also re-engineer the code to use the `hbwmalloc` interface [7] for dynamic memory allocation in C, C++, and Fortran applications. The main changes required for updating an application to use the `hbwmalloc` interface include: 1) adding an include statement for a header file; 2) allocating memory dynamically for the bandwidth-critical data structures by using the `hbw_malloc` call that is available through the `hbwmalloc` interface instead of calling the `malloc` function; and 3) updating calls to the function `free` with the calls to the `hbw_free` function. However, additional lines of code are required if it is desired to make the application portable across systems that do not contain MCDRAM. ICAT can re-engineer a given application for optimally using the MCDRAM and adds the necessary checks in the code so that the application does not fail when it is run on systems that do not contain MCDRAM or do not have the `memkind` library [7].

ICAT has a very light-weight parser, code analyzer, and code translator written in C++ and bash for identifying certain grammar rules and patterns in the applications written using the following base languages: C, C++, and Fortran. ICAT can handle ambiguities in the C++ language on the basis of the context. ICAT works with 100% accuracy for code adaptation. Code modification at few places may not be hard to do manually. However, manually 1) writing portable code that is easy to maintain and run on systems that may or may not have MCDRAM, and 2) identifying potential candidates for allocation on MCDRAM can be difficult for several domain scientists and students - the target audience of our tools and workflow. These are the situations in which ICAT is the most valuable.

It should be noted that ICAT is not fully automatic. While it is capable of 1) identifying the data structures that are appropriate candidates for allocation on MCDRAM, and 2) making appropriate code changes for allocating those data structures on the MCDRAM, it relies on user-guidance to shortlist the data structures that should be allocated on MCDRAM. It produces a report on all the data structures that are good candidates for allocation on MCDRAM. It then iteratively prompts the user to agree or disagree to allocate those data structures on MCDRAM. Only when the user agrees to the suggestions made by the tool, the tool makes the changes to the code.

4 Experimental Set Up & Results

We implemented and tested our workflow on the KNL nodes in the Stampede2 supercomputer [21]. The KNL nodes offer a memory-bandwidth of up to 479 GB/s when using MCDRAM directly, while this value decreases to 85 GB/s in the case of DDR4 [19]. We measured the run-time of our benchmarks using different configuration modes of the KNL nodes, and present an analysis of the results. While hard to quantify, we also demonstrate that our workflow raises the level of abstraction of using systems with deep-memory hierarchies.

4.1 Selected Benchmarks

We have selected a set of benchmarks that are representative of common workloads on supercomputers at several open-science data centers. We have focused on applications that are easily available and that also allow replication of our experiments with the arguments and configurations that we detail in this section. All codes were compiled with the `-O3` and `-xMIC-AVX512` flags:

1. MiniMD [6] is a parallel molecular dynamics simulation code and is part of the Mantevo project [12]. It computes physical properties like energy and pressure for a simulated space containing atoms. We focus on the MPI+OpenMP version of the code. We used Intel MPI 17.0.3 and Intel Compiler 17.0.4. We chose a problem size of 64^3, atomic density of 0.8442, 200 timesteps and performed reneighboring after every 10 steps. The experiments were carried out on a single Intel KNL node, with 8 MPI processes and 8 threads per process. The OpenMP affinity was set to `spread`. Finally, we set `OMP_PLACES` to `threads`, so that each OpenMP thread can be assigned to individual hardware threads on the target machine.
2. MiniFE [6] is another mini application that is also part of the Mantevo benchmark suite. It mimics the implementation of finite element generation, assembly, and solution for unstructured grid problems. It works over a 3-D box, with the dimensions of the box being passed as arguments. The implementation supports both MPI and OpenMP. We selected the optimized hybrid version for KNL, both with and without `memkind` support [7]. For our runs, we used a 3-D box of size 256^3. The code was executed using 16 MPI tasks and 4 OpenMP threads on a KNL node. Similar to MiniMD, we set `OMP_PLACES` to `threads` and used a `spread` affinity for the threads.
3. SPPARKS [14] is a Monte Carlo application that can be run using any of the following algorithms: Kinetic Monte Carlo (KMC), rejection KMC (rKMC), and Metropolis Monte Carlo (MMC). There are multiple subcategories of SPPARKS applications and the supported computational models. For our experiments, we chose to run an Ising model. The region for the problem is a block of sizes [0,500] and [0,500]. We simulated the problem for 100 seconds, setting a random sweep. Since the code is a pure MPI code, we ran it with 1 MPI task per physical core of the KNL node (total of 68 tasks).
4. LULESH (Livermore Unstructured Lagrange Explicit Shock Hydrodynamics [10]) is another mini-app. It is representative of 3D Lagrangian hydrodynamics on an unstructured mesh. It is a very popular application when used as a benchmark due to its characteristics and the opportunities that it brings in terms of generating load-imbalances in processors [9]. For our tests, we focused on the pure MPI version of the code. We had to use 64 tasks on a KNL node. We set the number of elements in the mesh to 24 (using the `-s` argument).

4.2 Workflow Implementation

Next, we ran the different steps in our workflow. This section details the results that we achieved for those applications.

4.2.1 Identify Memory Requirements

We first ran all the benchmarks on a KNL node configured in "Flat SNC-4" mode. This mode exposes the available memory (DDR4 and MCDRAM) as a total of 8 NUMA nodes: the first four nodes [0-3] are DDR4, while the last four [4-7] belong to MCDRAM. By default, the memory is allocated on DDR4 if no modification in the code or in the NUMA affinity is introduced. The results can be seen in Fig. 2. This figure shows the amount of memory allocated on each NUMA node throughout the execution time of each application. For example, for MiniMD it can be seen how approximately 2.5 GB are allocated on Node 0, 1.3 GB on Node 1, 1.7 GB on Node 2, 1.2 GB on Node 3, and no data is allocated on the other 4 NUMA nodes (MCDRAM). Since we are running applications with enough MPI tasks to have at least one task per NUMA node, all four DDR4 NUMA nodes are used. The MPI library sets the placement of the task and then the operating system allocates the memory on the closest NUMA node. Overall, the memory utilization remains constant until the end of the execution when it is deallocated. Similar behaviour can be observed for all the applications presented in the paper.

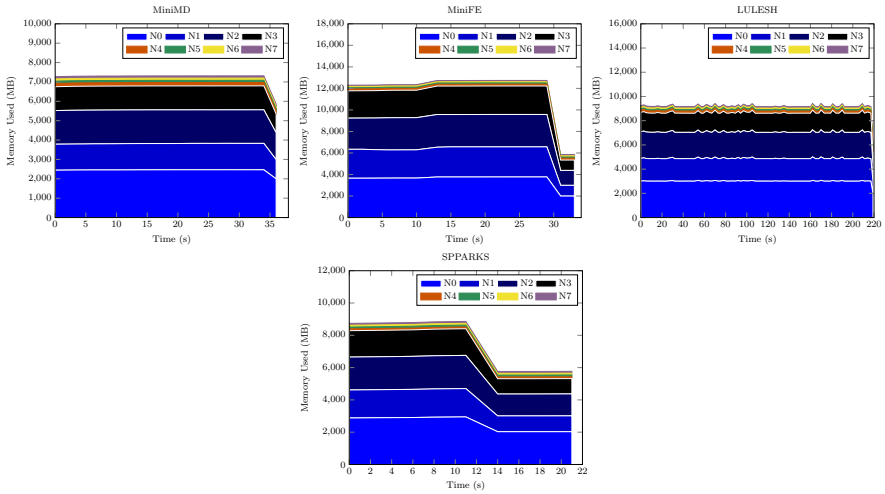

Fig. 2: NUMA utilization for all applications running in "Flat SNC4" mode with default settings

4.2.2 Analyze NUMA Utilization and Fix NUMA Bindings

In the previous experiment all the test cases ran using DDR4 only. We also observed that for all the test cases, the memory footprint was smaller than 16 GB, which makes these test cases as ideal candidates for running directly from MCDRAM. Next, we will show the results of running the test cases using MCDRAM only. For applications with multiple MPI tasks per node, it is necessary for users to introduce additional steps to fix the affinity of the processes involved in the computation so that the MCDRAM NUMA node that is closest to them is used.

For these tests, we have used a KNL node configured in `Flat SNC4` mode, which exposes a total of 8 NUMA nodes.

Our workflow captures the CPU usage when each task is executed, and instructs `numactl` to set the appropriate memory policy for the task. Initially, all data is stored in the local MCDRAM node. Once it is full, the remaining data is allocated in the local DDR4 node. Indeed, this approach works optimally when all the data fits within the MCDRAM. However, it might put key data structures on the DDR4 if the memory footprint is large enough. Our workflow covers this scenario by analyzing the placement of individual data structures.

Fig. 3 shows the memory utilization on each NUMA node throughout the execution time for each test case. It can be seen that after fixing the affinity on the basis of the recommendation of the tools in our workflow, the applications effectively use the available MCDRAM nodes rather than the DDR4 nodes. It is also worth noticing that the execution time has significantly decreased for all the test cases as a result of the increased memory bandwidth provided by the MCDRAM. All the test cases show a behavior similar to what was previously described when using DDR4. However, the SPPARKS memory footprint changed significantly with the change in problem/model type.

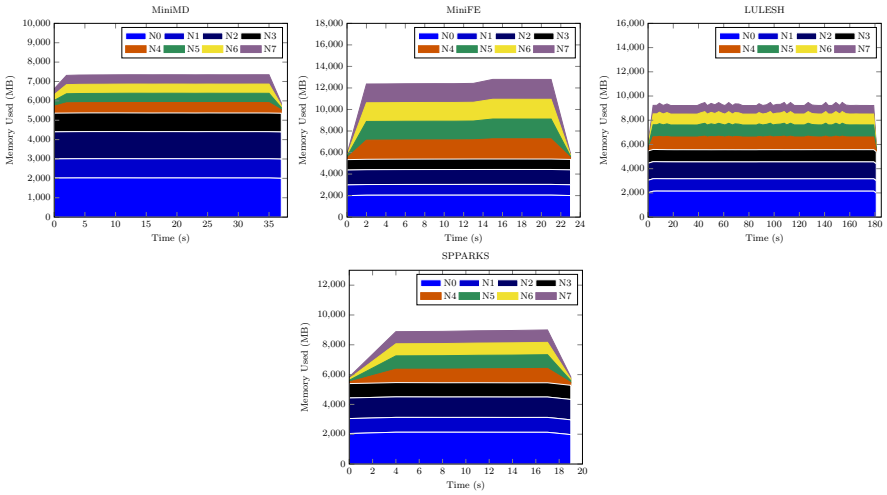

Fig. 3: NUMA utilization for all codes after fixing the affinity

Left plot in Fig. 4 shows the hits on the different NUMA nodes for LULESH after correctly setting the affinity on the basis of the recommendations of the tool. It can be seen how all hits are to MCDRAM NUMA nodes instead of DDR4 nodes. For comparison, the plot on the right shows the hits on the different MCDRAM nodes when only `numactl` is used globally for all tasks (naive approach, as in `mpirun numactl --membind=4,5,6,7 ./lulesh2.0`). In the second case, while all the NUMA hits still take place on MCDRAM nodes, they all hit the first MCDRAM node. Many of those hits are remote hits. This is equivalent to incurring

NUMA misses. We can see how the execution time is higher in the figure on the right as a result of these misses.

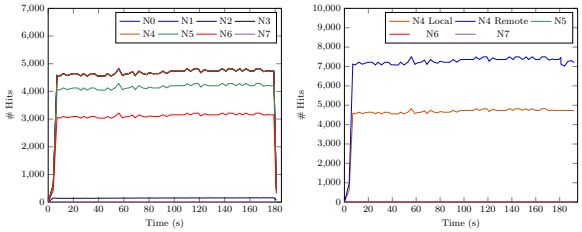

Fig. 4: NUMA utilization

4.3 Selecting Relevant MCDRAM Configurations

We ran the test cases on KNL nodes configured in different memory and cluster modes. In this section we present the results of running the test cases in 1) cache-quadrant mode (i.e., MCDRAM is configured for usage as an L3 cache, and the KNL tiles are logically divided in four parts which are spatially local to the four groups of memory controllers); 2) flat-quadrant mode (i.e., MCDRAM is configured for usage as addressable memory that complements DDR4, and the KNL tiles are logically divided in four parts which are spatially local to the four groups of memory controllers); and 3) flat-SNC4 mode (i.e., MCDRAM is configured for usage as addressable memory that complements DDR4, and the KNL tiles are divided into four separate NUMA nodes).

The tool that we use at this step (ICAT) follows an iterative process as depicted in Fig. 5. For the purpose of this paper, we focus on the options in the tool that are relevant for memory optimization. The user interacts with the tool after each step, agreeing on continuing or answering simple questions. At the end of the execution of the tool, a set of recommendations are presented. The users can implement these recommendations to improve the performance of the code. The recommendations include the best memory/cluster mode to be used as well as instructions for improving the memory bindings of the target code.

The results shown in Table 1 depict the best KNL configuration for running the different applications according to ICAT. This saves the user from spending time and effort in discovering those modes by trial-and-error, and without feeling burdened about the information on the low-level microarchitectural details of the processor and the application characteristics. Table 1 also shows how, for the case of SPPARKS (we include two different use cases, a large run and a short one), the results for the large run do not follow the recommendation. This is due to the nature of the problem being solved, with a set of random components that directly impact the overall execution time.

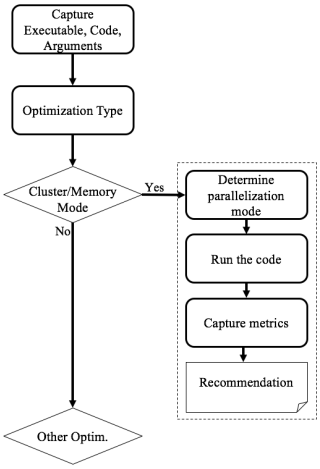

Fig. 5: Iterative process for generation of memory/cluster mode configuration

Table 1: Performance comparison of test cases run in different memory and cluster modes - Run-time in seconds used for comparison.

Application	Recommendation	Cache-Quad	Flat-Quad	Flat-SNC4
MiniMD	Flat/Cache-Quadrant	34.688	40.930	36.125
MiniFE	Flat-SNC4	64.768	45.743	44.538
LULESH	Flat/Cache-Quadrant	315.083	328.459	514.193
SPPARKS small	Flat/Cache-Quadrant	18.878	17.501	23.022
SPPARKS large	Flat/Cache-Quadrant	2314.830	2260.723	**1812.511**

4.4 Code Adaptation

Code re-engineering may be necessary for improving the parallelization and vectorization of applications ported to systems with deep-memory hierarchies. Memory optimization of the application may also be required. As previously described, on the basis of an application's characteristics, one of our tools in the workflow, ICAT, can advise the user on modifying, compiling, and optimally running the application on the KNL processors. If a user desires, our tool can also automatically modify the application code for implementing any required changes for using the `hbwmalloc` interface, and thereby can help in optimally using the MCDRAM.

As previously stated, the memory footprint for all our test cases was less than 16 GB, therefore, all the data fitted into the MCDRAM. In such scenarios, as previously shown, usually the applications can be run optimally by allocating the required data on the MCDRAM. However, since the MCDRAM latency is higher than that of DDR4, applications with latency-sensitive data structures and memory access patterns would see a penalty when all the data structures are allocated on MCDRAM. For such latency-sensitive applications, a combination of MCDRAM and DDR4, where the latency sensitive data structures are allocated into DDR4, would offer the best performance. Moreover, there can be several applications that may benefit by prioritizing the assignment of large data structures to the MCDRAM - that is, the data structure/s that will be used for a longer duration over the lifetime of a program and are <= 16 GB in size can be assigned

higher priority for assignment in the MCDRAM. However, upon experimentation, we found that none of the applications selected as test cases for this paper fall into this category.

Nonetheless, for demonstrating the code re-engineering capabilities of the tool present in our workflow, we chose the *"openmp-opt-knl"* version of the MiniFE application despite knowing that our tool advised that no such re-engineering is required. Hence, we do not expect any significant improvement in the application performance by using the `hbwmalloc` interface.

The Mantevo benchmark suite already contains a version of the MiniFE code that is capable of allocating the memory of selected data structures on the MCDRAM using the `memkind` interface - the *"openmp-opt-knl-memkind"* version. We show that the performance of the code generated by our tool (that re-engineers the *"openmp-opt-knl"* version to use the `hbwmalloc` interface) is comparable to this version of the MiniFE code that uses the `memkind` interface. It should be noted that both the `memkind` interface and the `hbwmalloc` interface are supported by the same `memkind` library. The results of the comparison are presented in Table 2. Since the MiniFE application performed best on the KNL nodes configured in the Flat memory mode and SNC4 cluster mode, we used this mode for the results in Table 2. It is worth noticing that without requiring the users to introduce any changes in the code, our tool can help them in adapting their applications to optimally take advantage of the `hbwmalloc` interface without incurring any significant loss in performance.

Table 2: Run-time comparison of different versions of MiniFE.

Version	Time (s)
openmp-opt-knl	45.73
openmp-opt-knl-memkind	46.26
re-engineered-openmp-opt-knl	45.922

5 Conclusions

While hardware manufactures release innovative chip designs and memory hierarchies for powering science and discoveries, the users of those chips should not be burdened by the continuous need to manually update application code to take advantage of the innovative hardware features. Hence, high-productivity workflows and tools are needed. In this paper, we have presented one such high-productivity workflow that helps the users in efficiently utilizing the deep-memory hierarchies. Our workflow can be used on existing petascale supercomputers as demonstrated in this paper. The workflow, and eventually the tools that comprise the workflow, free the users from the burden of learning about low-level microarchitectural details of the deep-memory hierarchies during the process of porting their applications to supercomputers.

We have demonstrated the applicability of our approach to commonly used HPC applications as well as to the benchmarks that are representative of the workloads on the supercomputers at several open-science data centers. The results

show that we are able to achieve performance comparable to applications optimized by experts.

We have adopted a user-centric approach for implementing our workflow. Instead of offering complex tools with significant overheads and convoluted metrics, we offer a workflow that is easy to implement and that can be applied to a large number of applications. The workflow is suitable for running HPC applications developed using the most common programming languages and models, and involves tools that are available for public usage. We plan to do the usability analysis of our tools and the workflow in future. We also plan on expanding the capabilities of the tools in our workflow to support additional hardware elements like non-volatile memories.

Acknowledgment

We are very grateful to the National Science Foundation for grant #1642396, ICERT REU program (National Science Foundation grant #1359304), XSEDE (National Science Foundation grant #ACI-1053575), and Texas Advanced Computing Center (TACC) for providing resources required for this project. We are grateful to Tiffany Connors and Lars Koesterke for their contributions to the ICAT codebase. Stampede2 is generously funded by the National Science Foundation (NSF) through award ACI-1540931.

References

1. Arora, R., Koesterke, L.: Interactive code adaptation tool for modernizing applications for intel knights landing processors. In: Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact, PEARC17, pp. 28:1–28:8. ACM, New York, NY, USA (2017). DOI 10.1145/3093338.3093352. URL http://doi.acm.org/10.1145/3093338.3093352
2. Chandrasekar, K., Ni, X., Kale, L.V.: A memory heterogeneity-aware runtime system for bandwidth-sensitive HPC applications. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1293–1300 (2017). DOI 10.1109/IPDPSW.2017.168
3. Harrod, W.: A journey to exascale computing. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:, pp. 1702–1730. IEEE (2012)
4. Hartmann, C., Fey, D.: An extended analysis of memory hierarchies for efficient implementations of image processing applications. Journal of Real-Time Image Processing (2017). DOI 10.1007/s11554-017-0723-2. URL https://doi.org/10.1007/s11554-017-0723-2
5. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Fifth Edition: A Quantitative Approach, 5th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2011)
6. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving performance via mini-applications. Tech. Rep. SAND2009-5574, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550 (2009)
7. Intel: Memkind. http://memkind.github.io/memkind/ (2017). [Online; accessed 25-Jun-2018]
8. Jun, H., Cho, J., Lee, K., Son, H.Y., Kim, K., Jin, H., Kim, K.: HBM (High Bandwidth Memory) DRAM technology and architecture. In: 2017 IEEE International Memory Workshop (IMW), pp. 1–4 (2017). DOI 10.1109/IMW.2017.7939084
9. Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B.L., Cohen, J., Devito, Z., Haque, R., Laney, D., Luke, E., Wang, F., Richards, D., Schulz, M., Still, C.H.: Exploring traditional and emerging parallel programming models using a proxy application. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing,

IPDPS '13, pp. 919–932. IEEE Computer Society, Washington, DC, USA (2013). DOI 10.1109/IPDPS.2013.115. URL http://dx.doi.org/10.1109/IPDPS.2013.115
10. Karlin, I., Keasler, J., Neely, J.: Lulesh 2.0 updates and changes. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA (2013)
11. Khaldi, D., Chapman, B.: Towards automatic hbm allocation using llvm: A case study with knights landing. In: Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC, LLVM-HPC '16, pp. 12–20. IEEE Press, Piscataway, NJ, USA (2016). DOI 10.1109/LLVM-HPC.2016.7. URL https://doi.org/10.1109/LLVM-HPC.2016.7
12. Laboratory, S.N.: Mantevo Project Homepage. http://mantevo.org (2018). [Online; accessed 25-June-2018]
13. Peng, I.B., Gioiosa, R., Kestor, G., Cicotti, P., Laure, E., Markidis, S.: RTHMS: A tool for data placement on hybrid memory system. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, ISMM 2017, pp. 82–91. ACM, New York, NY, USA (2017). DOI 10.1145/3092255.3092273. URL http://doi.acm.org/10.1145/3092255.3092273
14. Plimpton, S., Battaile, C., Chandross, M., Holm, L., Thompson, A., Tikare, V., Wagner, G., Zhou, X., Garcia-Cardona, C., Slepoy, A.: Crossing the mesoscale no-man's land via parallel kinetic monte carlo (2009)
15. PRACE: Partnetship foR Advanced Computing in Europe. http://www.prace-ri.eu/ (2018). [Online; accessed 25-Jun-2018]
16. Reinders, J.: Vtune performance analyzer essentials. Intel Press (2005)
17. Rosales, C., Cazes, J., Milfeld, K., Gómez-Iglesias, A., Koesterke, L., Huang, L., Vienne, J.: A comparative study of application performance and scalability on the intel knights landing processor. In: M. Taufer, B. Mohr, J.M. Kunkel (eds.) High Performance Computing, pp. 307–318. Springer International Publishing, Cham (2016)
18. Rosales, C., Gómez-Iglesias, A., Predoehl, A.: REMORA: A resource monitoring tool for everyone. In: Proceedings of the Second International Workshop on HPC User Support Tools, HUST '15, pp. 3:1–3:8. ACM, New York, NY, USA (2015). DOI 10.1145/2834996.2834999. URL http://doi.acm.org/10.1145/2834996.2834999
19. Rosales, C., James, D., Gómez-Iglesias, A., Cazes, J., Huang, L., Liu, H., Liu, S., Barth, W.: KNL utilization guidelines. Tech. Rep. TR-16-03, Texas Advanced Computing Center, Austin, Texas (2013)
20. Sodani, A.: Knights Landing (KNL): 2nd generation Intel Xeon Phi processor. In: 2015 IEEE Hot Chips 27 Symposium (HCS), pp. 1–24 (2015). DOI 10.1109/HOTCHIPS.2015.7477467
21. Stanzione, D., Barth, B., Gaffney, N., Gaither, K., Hempel, C., Minyard, T., Mehringer, S., Wernert, E., Tufo, H., Panda, D., Teller, P.: Stampede 2: The evolution of an XSEDE supercomputer. In: Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact, PEARC17, pp. 15:1–15:8. ACM, New York, NY, USA (2017). DOI 10.1145/3093338.3093385. URL http://doi.acm.org/10.1145/3093338.3093385
22. Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G.D., Roskies, R., Scott, J.R., Wilkins-Diehr, N.: XSEDE: Accelerating scientific discovery. Computing in Science & Engineering **16**(5), 62–74 (2014). DOI 10.1109/MCSE.2014.80. URL doi.ieeecomputersociety.org/10.1109/MCSE.2014.80