Preliminary Experience with OpenMP Memory Management Implementation

Adrien Roussel[123], Patrick Carribault[13], and Julien Jaeger[123]

[1] CEA, DAM, DIF, F-91297 Arpajon, France
{adrien.roussel, patrick.carribault, julien.jaeger}@cea.fr
[2] Exascale Computing Research Laboratory, Bruyères-le-châtel, France
[3] Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, 91680 Bruyères-le-Châtel, France

Abstract. Because of the evolution of compute units, memory heterogeneity is becoming popular in HPC systems. But dealing with such various memory levels often requires different approaches and interfaces. For this purpose, OpenMP 5.0 defines memory-management constructs to offer application developers the ability to tackle the issue of exploiting multiple memory spaces in a portable way. This paper proposes an overview of memory-management from applications to runtimes. Thus, we describe a convenient way to tune an application to include memory management constructs. We also detail a methodology to integrate them into an OpenMP runtime supporting multiple memory types (DDR, MCDRAM and NVDIMM). We implement our design into the MPC framework, while presenting some results on a realistic benchmark.

Keywords: OpenMP 5.0, Data Allocation, Memory Management

1 Introduction

For the past decades, the main trend has been to enhance the compute capabilities of processors through frequency increase, functional unit extension (*e.g.* SIMD) or core duplication. It leads to the current generation of supercomputer nodes design with several multi-core processors linked together. But this large spectrum of compute capabilities puts the stress on the memory part to keep feeding such functional units. For example, SIMD operations may require more memory bandwidth to enable issuing one instruction requiring a large number of register inputs. However, proposing a new memory type with a larger bandwidth, for the same overall cost, exposes a smaller storage. That is why various kinds of memory appeared in the HPC community. For example, Intel launched the Knights Landing many-core processor [26] which embeds a high bandwidth stacked memory named MCDRAM. The next generation of such an approach is called HBM and will be available in some processors like ARM-based Fujitsu A64FX [30]. This approach improves the throughput of bandwidth-hungry units, but there still is a need for a large storage with lower bandwidth but better performance than regular disks. This is why the notion of persistent memory appears in clusters like the flash memory NVDIMM technology [15].

While these new memory types may fulfill the requirements of many parallel applications (HBM/MCDRAM for compute-intensive parts, NVDIMM for I/O dedicated portions), allocating data in these target memory spaces in a portable and easy way is tedious. Thus, memory management is becoming one major concern for HPC application developers. Since version 5.0, OpenMP tackles this issue by introducing *memory management* extensions [22]. It is now possible to control data allocation and placement on specific memory spaces through OpenMP constructs. For this purpose, OpenMP defines multiple memory spaces: application developers have then to specify some parameters (traits) for a specific allocator (e.g., data alignment, pool size,. . .).

This paper proposes a first experience of implementing these OpenMP memory management constructs. It makes the following contributions:

- Preliminary implementation of OpenMP memory management constructs targeting DRAM, MCDRAM and NVDIMM into the MPC framework [7] [4] (a thread based MPI implementation with a OpenMP 3.0 runtime system)
- Port of a C++ mini-application with effort to support STL objects.
- Experiments on portability with various target architectures exposing different memory types.

This paper is organized as follows: Section 2 exposes an overview of the OpenMP specification for memory management. Section 3 presents related work in this area. Section 4 details our approach to implement these OpenMP constructs into the MPC framework and enable their support inside an application. Finally, this papers illustrates experimental results in Section 5 before concluding in Section 6.

```
// Initialization
omp_memspace_handle_t mcdram = omp_high_bw_mem_space;
omp_alloctrait_t mcdram_traits[1] = {omp_atk_alignment, 64};
omp_allocator_handle_t mcdram_alloc;
mcdram_alloc = omp_init_allocator(mcdram, 1, mcdram_traits);

// Allocation
void* Allocate(size_t size, omp_allocator_handle_t allocator) {
  return (void*)omp_alloc(size, allocator);
}

// Deallocation
void Release(void **ptr, omp_allocator_handle_t allocator) {
   if (*ptr != NULL) {
      omp_free(*ptr, allocator) ;
      *ptr = NULL ;
   }
}
```

Listing 1.1: Functions for OpenMP Memory Management

[4] Available at https://mpc.hpcframework.com/

2 Memory Management in OpenMP 5.0

OpenMP 5.0 introduces constructs and API routines to manage portable data allocation in various memory banks. Thus it defines a set of memory spaces (`omp_memspace_handle_t`) and parameters (*traits*) that can affect the way data are allocated in this target memory (`omp_alloctrait_t`). Even though each implementation can propose its own spaces and traits, OpenMP 5.0 defines default spaces (*default*, *large capacity*, *constant*, *high bandwidth* and *low latency*), and allocator traits (such as *alignment*, *pool size* and *fallback*). Thus, the user has to create an allocator handle (`omp_allocator_handle_t`) by specifying a target memory space and a set of traits. This operation is performed by calling the initialization function named `omp_init_allocator`. Listing 1.1 highlights this process at the top part.

After this setup, the application can allocate data with this new allocator. The runtime will thus be responsible for allocating data into the target memory bank with the specified trait values. There are two ways to manage data allocations with OpenMP: functions and directives. The first method is to call the functions `omp_alloc` and `omp_free`. Both take an allocator handle as input (which should be initialized first). Listing 1.1 shows this approach.

```
float A[N], B[N];
#pragma omp allocate(A) allocator(omp_large_cap_mem_alloc)

#pragma omp parallel
{
  #pragma omp task allocate(omp_const_mem_alloc: B)
  {
    /* ... */
  }
}
```

Listing 1.2: Directives for OpenMP Memory Management

The second method relies on the `allocate` directive and clause (see Listing 1.2). The directive takes a list of data variables to allocate through the handle specified in the `allocator` clause. This clause can be used in several constructs such as `task`, `taskloop` and `target`. Users have to specify in this clause the handle to use to allocate data, and the list of data variables. The handle allocator can be defined by the user or a predefined allocator. There is one allocator given by the OpenMP standard per memory space.

3 Related Work

This section details different approaches to deal with multiple memory levels inside an HPC compute node: dedicated allocations, portable allocations and OpenMP implementations.

Dedicated Memory Management. The first approach deals with dedicated interfaces to allocate data inside a specific target memory type. Even if some memory kinds can be configured as a cache level to enable automatic hardware-driven management (*e.g.* MCDRAM in Intel KNL [18, 21, 9]), fine-grain data allocation can lead to better performance. Thus, some research papers set up this MCDRAM as *flat mode* meaning that a specific action is required to put data into this target memory. This action may have a coarse-grain scope (*e.g.* relying on the *numactl* library [8] or forcing the global memory-placement policy [19]), or a more fine-grain approach (*e.g.* using memory allocators like *memkind* [6] to deal with placement on a per-allocation basis [5]). Similar fine-grain initiatives also exist for other types of memory *e.g.*, persistent [16, 2, 25].

Portable Memory Management. Runtime systems are already able to deal with memory management for performance portability concerns [12, 1, 4]. This list is not exhaustive as multiple approaches exist in this domain, especially focusing on heterogeneous systems. Some other initiatives also deal with high bandwidth memory management like MCDRAM for KNL processors. These forms of memory management have been explored within domain specific languages in [24].

High level programming interfaces are widespread in the HPC community. Previous works such as [10, 11] have to deal with memory allocation in a portable way, but only for the GPGPU concerns now. These interfaces enable abstract memory allocations through wrapper functions or objects.

OpenMP Memory Management. While some initiatives have been already proposed for memory management in a portable way, there is no standard way to do it. OpenMP now provides a way to standardize memory management for a wide system spectrum. Based on directives and functions, software developers are able to address memory allocations in an easy way: they do not need to deal with the actual low-level allocation method into a specific memory. As far as we know, LLVM [20] has the most up-to-date OpenMP runtime implementation for the support of memory management constructs. Even if this runtime system is well advanced, the front-end part of supported compilers (Clang and Intel) does not support the full specification. Until now, it supports allocation in standard and high bandwidth memory banks only with few traits (*e.g.* `pool_size`, `fallback` and `alignment`). For the high bandwidth memory support, LLVM forwards memory allocations to `hbw_malloc` from the *memkind* library.

Paper position: While our work is similar to the LLVM approach (design and implementation of memory-management constructs inside an existing OpenMP runtime), the objective of this paper is to give a preview our implementation of multiple memory levels (DDR, MCDRAM/HBM and NVDIMM) its portability aspects. Moreover, we have experimented how to integrate those OpenMP functions in a portable solution with a C++ application.

4 Application- and Runtime-Level OpenMP Memory Management

This section presents the main contribution of this paper: the support of multiple memory types inside an existing OpenMP implementation (4.1) and the port of a C++ mini-app (Section 4.2).

4.1 Runtime System Design for Memory Management Integration

MPC [23] is a thread-based MPI implemention which proposes an OpenMP [7] runtime system. It is compliant to OpenMP 3.1 and partially supports version 4.5. As MPC integrates its own NUMA aware allocator, we have decided to design and implement the memory management constructs inside this framework. MPC is compatible with GNU and Intel compilers for OpenMP lowering and thread-based specific features. But these compilers have currently a limited support of the `allocate` directive and clause. Thus, our work focuses on providing initialization functions and allocation/deallocation calls (`omp_alloc` and `omp_free`) for multiple memory types: DDR, high bandwidth MCDRAM and large-capacity NVDIMM. For this purpose, it is necessary to enhance the existing implementation with an advanced hardware topology detection and an approach for initializing and allocating data. This section details those steps.

Automatic memory banks discovery. OpenMP offers a set of predefined memory spaces with a way to fallback if the application tries to allocate in non-existing type. Thus, the first step is to discover available memory banks on the target machine. For this purpose, a hardware detection tool has to be integrated into the OpenMP runtime system to list available memory spaces on a system at execution time. Most of the runtime systems already rely on the *hwloc* library for hardware topology discovery and thread binding. Recent work [13] adds the support of heterogeneous memory types such as *high bandwidth memory* (*e.g.* MCDRAM from Intel's KNL processor) or *large capacity* memory bank like *NVDIMM* technology. On such machines, MCDRAM memory space is exposed as a no-core NUMA node with a special attribute named `MCDRAM`. Checking the presence of MCDRAM in a system is thus possible by browsing all the NUMA nodes and searching for the one which has this defined *hwloc* attribute. *Large capacity* memory spaces such as `NVDIMM` memory are viewed by *hwloc* as OS devices, and tagged with a special attribute. So, it is possible to detect such memory banks by listing all the OS devices and searching for *large capacity* memory banks.

The basic block of our design relies on the hardware detection module based on *hwloc* in the MPC framework. It is called at runtime initialization and for every runtime entry point that is not in a parallel region to detect and save available hardware components.

Memory Management Initialization. The allocation process is separated into two parts: the initialization of allocators and the data allocation (as seen in Listing 1.1).

First of all, the `omp_init_allocator` function initializes some user-defined traits in a `omp_alloctrait_t` structure, and link them into a memory space in a `omp_memspace_handle_t` structure. From a runtime point of view, it is necessary to save a dynamic collection of allocators. By default, in our implementation, this structure contains pre-defined allocators only. Its size can then be enlarged to allow users to create new allocators with some specific traits. This structure maps an allocator handle (*i.e.* `omp_alloc_handle_t`) to an allocator structure (*i.e.* `omp_alloc_t`). This collection is only accessible from each thread in a read-only access mode. Indeed, we do not ensure thread safety for this structure yet. Thus, all the allocators have to be initialized outside a parallel region. We are currently working on thread safety to enable the creation of allocators inside parallel regions. In this way, concurrent threads may benefit from allocators inside parallel regions within the `omp_alloc` function.

Various traits are proposed in the specification, such as memory alignment as illustrated in Listing 1.1. Traits have a default value, but the initialization process may set them user-defined values. The MPC framework currently supports the `alignment` and `fallback` traits. Future work is needed to support more traits in the framework.

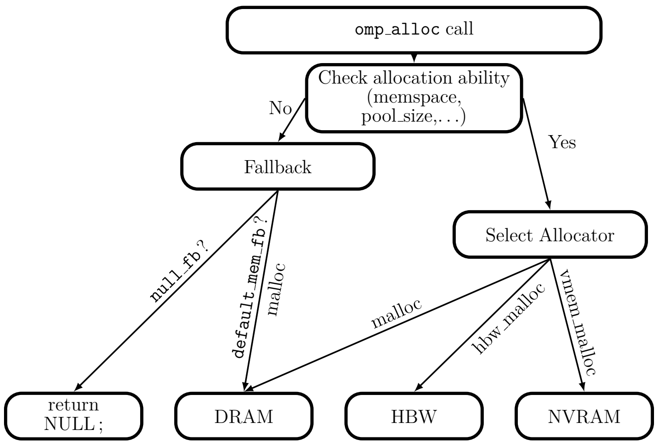

Fig. 1: `omp_alloc` call procedure

Data Allocation Process The final step is to implement the data-allocation mechanisms. Figure 1 sketches this process. For this purpose, an allocator can be retrieved from the collection based on the handle structure. Depending on the trait values, the data allocation process is different. For example, we test the value of the fallback trait if the runtime detects that the requested memory space is unavailable. Trait checking is quite similar no matter what the selected memory space is. Once the traits values are set, we need to link a memspace to an allocation function. Depending on the desired memory space, this function differs. For example, `malloc` from *glibc* is the function used to allocate data in DRAM memory space (denoted as `omp_default_mem_space`) while `hbw_malloc` from the *memkind* library enables data allocation in the high bandwidth memory space (denoted as `omp_high_bw_mem_space`). To support all the set of the specification defined memory spaces, runtime developers may have to integrate various allocator library inside the OpenMP runtime. For example, data allocation in high bandwidth or in large capacity memory spaces are currently well supported within the *memkind* library. [5] The MPC framework comes with its own NUMA-aware allocator [28] based on kernel page reuse. For more convenience, and to avoid multiple library integrations, we link our allocator to the OpenMP runtime. Data allocations in DRAM (refered to as default memory space) and in high bandwidth memory space are thus operated by our allocator. As MCDRAM is currently detected as a no-core NUMA node, we redirect all the dynamic allocations queries to this NUMA node. We also support data allocation in large capacity devices like `NVDIMM`. For this purpose, we have integrated the `nvmem` [6] library inside the MPC framework.

The deallocation process is quite similar to the allocation one. When `omp_free` is called, we check the memory space specified in the allocator structure and then call the appropriate data free function.

```
template <typename T>
class omp_allocator
{
public:
   ...
    omp_allocator_handle_t& allocator;

    omp_allocator(omp_allocator_handle_t& alloc) : allocator(alloc) {}

    pointer allocate(size_type n, const T* hint = 0) {
        return (T*)omp_alloc(sizeof(T) * n, allocator);
    }

    void deallocate(pointer p, size_type n) {
        omp_free(p, allocator);
    }
};
   ...
  std::vector<Real_t, omp_allocator<Real_t> > m_x(N, 0., omp_allocator<Real_t>(allocator);
```

Listing 1.3: Custom Allocator for STL Objects

[5] See https://memkind.github.io/memkind for more information
[6] Available at https://pmem.io

4.2 Enabling Portable Application Memory Management

After implementing partial memory management support into the runtime, it is necessary to modify the target application as presented in Listing 1.1. This section illustrates the case of LULESH [17], an hydrodynamics application from the CORAL benchmark suite. This example provides some valuable experience on how to port an existing C++ application to use OpenMP memory management functions. While porting C codes leads to addtional codes as shown in Listing 1.1, many C++ applications exploit STL objects [27] (*e.g.* vector, stack, list, ...). Such objects manage allocators through a template parameter `Allocator`. We propose a custom allocator object (see Listing 1.3) that integrates the OpenMP `omp_allocator_handle_t` structure features. Thus, all the methods (e.g., constructor/destructor, `resize` or `insert`) use OpenMP allocation constructs. This example also illustrates that the new allocator has to be passed as a template parameter of the STL object and as input of the constructor (to indicate the right memory spaces to the allocator).

5 Experimental Results

This section illustrates our implementation inside the MPC framework on one benchmkark allocating data in various memory banks based on the OpenMP 5.0 memory-management functions. For this purpose, we modified the LULESH benchmark (as previously explained in Section 4.2) by inserting calls to `omp_alloc` functions to allocate data in various memory spaces.

Experimental Environment. On the hardware side, the target platforms cover the different memory kinds that our implementation supports. Thus we rely on 4 different systems. The first one is a compute node containing a 68-core Intel Knights Landing processor [26], 16GB of MCDRAM and 96GB of regular DRAM memory. This configuration will be used to evaluate the allocation inside a high bandwidth memory. To test the large storage memory, we use a compute node equipped with two NVDIMM technology storages (with a capacity of 1.5TB each). This persistent-memory node is composed of two 24-core Intel Cascade Lake processors (Xeon Platinum 8260L), each clocked at 2.40 GHz. Finally, two systems are available to ensure portability by exposing only regular DDR: one AMD ROME node (two 32-core AMD EPYC 7502 processors at 2.50 GHz with 128GB of DDR) and one Intel Skylake node (two 24-core Intel Xeon Platinium 8168 Skylake processor at 2.70 GHz with 96GB of DDR). On the software side, all benchmark versions were compiled with `-O3` and linked to the MPC framework (configured with standard options) for the OpenMP runtime.

5.1 Coarse-Grain OpenMP Memory Allocation

This section describes the experiments conducted on the modified LULESH benchmark targeting a single memory space. Thus, it aims at testing the ability of our implementation to allocate data in a specified memory space following the OpenMP 5 standard.

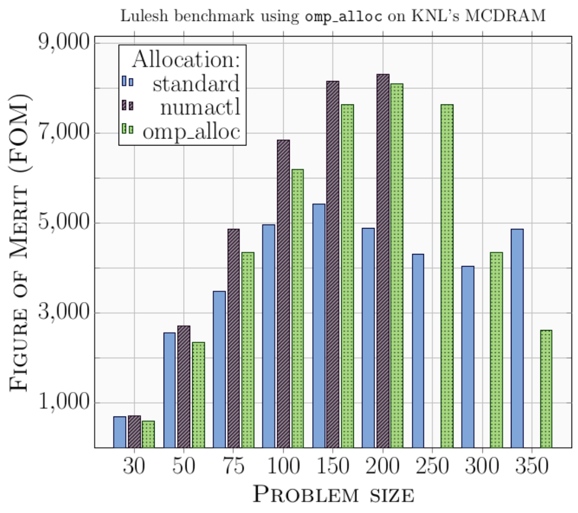

Fig. 2: Coarse-Grain Data Allocation Management in MCDRAM

High-Bandwidth Memory. The first evaluation concerns the MCDRAM memory on Intel KNL node with 64 threads. Figure 2 shows the *Figure of Merit* (FOM - number of elements solved per microsecond) according to the mesh size on different versions of LULESH (with a fixed total number of iterations: 100). The first bar (*standard*) represents the regular run (everything is allocated in DDR) while the second bar (*numactl*) is controled by the numactl command that places all data into the MCDRAM. The third version (*omp_alloc*) is modified with OpenMP memory management constructs. All of the three executions were compiled and run within the MPC OpenMP implementation. These results show only a 5% difference in performance between the execution with `numactl` and the application modified to use `omp_alloc`, for problem sizes from 30 to 200. There are no results for problem sizes greater than 200 for the *numactl* version because all the data does not fit in MCDRAM and the application stops. From 200 to 350, however, the *omp_alloc* execution can execute even though the allocation does not fit into MCDRAM, and the performance diminishes. This is due to the cache memory mode, as data does not fit in MCDRAM, the application performance is lead the DRAM bandwidth because some data are allocated in it. Performance of the original application without the use of `numactl` are lower than the two previous ones. In conclusion, we are able to allocate data in the MCDRAM high bandwidth memory bank with the OpenMP memory management functions. The performance difference between `omp_alloc` and `numactl` charts is due to the fact that `numactl` is much more agressive and allocates all the data in MCDRAM. In our modified implementation, we only moved data dynamically allocated such as arrays and vectors, so not all the data are moved to MCDRAM.

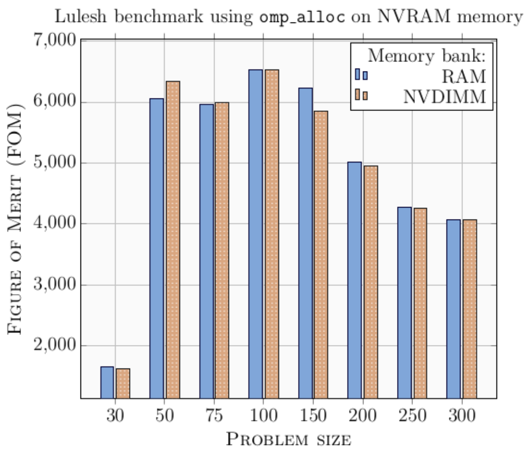

Fig. 3: Coarse-Grain Data Allocation Management in NVDIMM

Large-Capacity Memory Space. Figure 3 presents the FOM running LULESH on a dual-socket 24-core Skylake node (i.e., 48 OpenMP threads) equipped with a *NVDIMM* device. While the two versions rely on OpenMP to allocate data, the first one (*RAM*) allocates all data in regular DDR (default allocator) while the second one (*NVDIMM*) changes the OpenMP allocator to target the *large capacity* space. With minor modifications, this graphic shows the ability to perform data allocation in large capacity memory devices like *NVDIMM* technology. As explained in [14], the NVDIMM memory can be configured in two modes. Our results are similar for both selected memory spaces. We can conclude that the node is configured in a 2LM mode (*i.e.* similar to KNL cache memory mode): all the data allocation are thus directed to NVDIMM memory. Since no error message are emitted from the vmem library, we are assured all allocated memory is in NVRAM. We do not have any error message when using *vmem* library, which means that we are able to allocate memory in NVRAM.

5.2 Fine-Grain OpenMP Memory Allocation

An previous analysis [3] of the LULESH benchmark has already determined the relevant data to be placed in high bandwidth memory bank. The purpose of this work was to detect which functions are bandwidth bound. Application performance can thus be improved if data operated in these functions are placed in memory banks with a higher bandwidth. This analysis demonstrated that some functions are sensitive to bandwidth, such as *EvalEOSForElems*, *AllocateGradients* and *CalcForceNodes*. We thus placed all the data relative to these function in `omp_high_bw_mem_space` (*i.e.* MCDRAM here) while the other data are placed in `omp_default_mem_space` (*i.e.* DRAM memory).

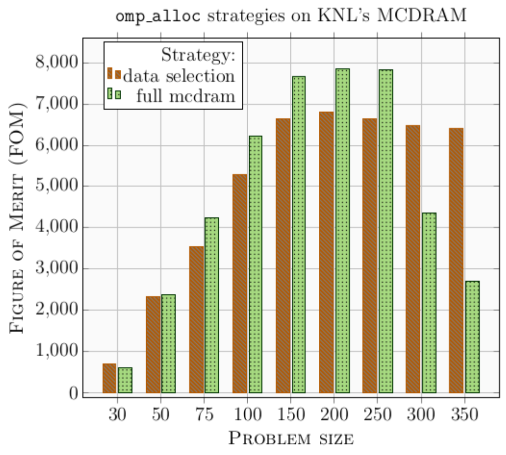

Fig. 4: Fine-Grain Data Allocation Management in MCDRAM

The objective of this experiment is to illustrate the benefit of fine-grain management for data allocation: i.e., choose in which memory bank to allocate through the `omp_alloc` function. Figure 4 presents the results of this approach (*data selection*) compared to allocating everything inside the MCDRAM (*full mcdram*). For this experiment, we change the number of iterations performed to 250 from the previous experiment while still varying the problem size. We can observe that allocating all data in MCDRAM is generally about 10% better between problem sizes of 75 and 250. Performance is about 50% less that the data selection method beyond problem sizes of 250. Below a problem size of 75, then are about the same. Performance decreases beyond and are worse than standard allocation model. These results show that selecting MCDRAM may suffer a bit in performance at small scale, it may be significantly important at large scale, where many HPC applications run. For all the tested configurations, we observe no performance decrease. However, performance seems to be bound to 6500.

In conclusion, as previously stated in several papers like [5], we show that allocating all data in MCDRAM memory might not be the most relevant choice. Indeed, when data do not fit inside this high bandwidth memory bank, application performance is deteriorated, even more than without the use of MCDRAM. From this experiment, we aim to warn developers about data allocation. OpenMP offers ways to allocate easily, in a portable way, data in various memory banks. However, the strategy to select which data to move from one memory bank to another is still in charge of the developers. Currently, no runtime mechanism to automatially move data from memory banks exists.

5.3 Portabilty Across Hardware Platforms

The design of OpenMP memory-management constructs enables portability of applications whatever the available memory types on the target hardware. With the help of the *fallback* trait, data can be allocated in a default memory space (`default_mem_fb`) if the specified one does not exist. However, an application can also terminate if the fallback property is set to `abort_fb`. We propose here an experiment that highlights the portability of our OpenMP memory-management implementation. For this purpose, we ran the LULESH benchmark with the fine-grain data allocation strategy as sketched in the previous section (selected functions allocated in the high bandwidth memory space while the other ones target a default allocator). We executed it on several platforms, without any code modification. The selected machines are the ones composed of AMD EPYC, Intel Skylake and Intel KNL processors previously described in 5. All the runs were performed with 48 threads, and we fix the size problem to 350 for 100 iteration. We set the fallback trait to `default_mem_fb` to forward data allocation to DRAM memory space if high bandwidth memory is not found. Results are gathered in Table 1.

System	AMD ROME	Intel Skylake	Intel KNL
FOM (z/s)	4834.99	4416.18	5397.94
Time (s)	890	970	790

Table 1: FOM results

The execution on KNL achieve better performance than the two other machines. Indeed, as KNL platform benefits from MCDRAM, some selected data allocation are directed to high bandwidth memory based on OpenMP constructs. However, the two others do not have the MCDRAM memory type, so all data allocations are forwarded to classical DRAM memory without any application cancellation. We conclude that we are able to ensure portability with OpenMP memory management constructs. The main advantage of this approach is to achieve this result without any significant modification to the application, and maintain portability

6 Conclusion & Future Work

This paper presents our experiences with the OpenMP memory management constructs at the application-level and the runtime-level. From the application side, developers should integrate data allocation calls with a standard like OpenMP, to provide portability. Through the LULESH benchmark, we have illustrated that these new constructs are easy to integrate. However, C++ STL objects users have to change the default allocator and implement a new one which encapsulates OpenMP function calls. We also have implemented these constructs into the OpenMP runtime of the MPC framework. While we do not support all the specification yet, we have implemented the major basic blocks into an OpenMP runtime system targeting various memory levels (DDR, MC-DRAM and NVDIMM). For this purpose, we detail our implementation from hardware detection to data allocation process.

Our results show that this implementation is feasible and that there are advantages that provide performance improvements for user applications. We illustrate that it is easy to make portable applications with slight modifications. Our implementation is able to allocate data in default, high bandwidth and large capacity memory spaces. Our experiments also show that data allocations should be performed with care: the best strategy is not always to allocate all data in the fastest memory.

As a future work, we plan to support all the features provided by the specification, especially remaining traits. We also plan to work on coupling the OpenMP memory management constructs with the `affinity` clause. Since the last version of the OpenMP, 5.0, the specification introduces a new `affinity` clause in `task` directives to give some hints at scheduling time in order to enhance data locality. This clause has been already implemented and evaluated by others [29]. Information from allocators will be needed from the runtime for the affinity clause. Indeed, this information can assist the task scheduler to make smarter decisions about affinity. In addition of that, it has a low memory footprint to keep this information and can significantly improve application performance. Thus we plan to evaluate this coupling to enhance the task scheduler.

Acknowledgments: This work was performed under the Exascale Computing Research collaboration, with the support of CEA, Intel and UVSQ.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011), https://hal.inria.fr/inria-00550877
2. Bhandari, K., Chakrabarti, D.R., Boehm, H.J.: Makalu: Fast recoverable allocation of non-volatile memory. ACM SIGPLAN Notices 51(10), 677–694 (2016)
3. Brunie, H., Jaeger, J., Carribault, P., Barthou, D.: Profile-Guided Scope-Based Data Allocation Method. In: MEMSYS 2018 - International Symposium on Memory Systems. Alexandria, United States (Oct 2018), https://hal.inria.fr/hal-01897917
4. Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguad, E., Labarta, J.: Productive programming of gpu clusters with ompss. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium. pp. 557–568 (2012)
5. Butcher, N., Olivier, S.L., Berry, J., Hammond, S.D., Kogge, P.M.: Optimizing for knl usage modes when data doesnt fit in mcdram. In: Proceedings of the 47th International Conference on Parallel Processing. ICPP 2018, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3225058.3225116

6. Cantalupo, C., Venkatesan, V., Hammond, J., Czurlyo, K., Hammond, S.D.: memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2015)
7. Carribault, P., Pérache, M., Jourdren, H.: Enabling low-overhead hybrid mpi/openmp parallelism with mpc. In: Sato, M., Hanawa, T., Mller, M., Chapman, B., de Supinski, B. (eds.) Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Proceedings of the 6th International Workshop on OpenMP (IWOMP 2010), Lecture Notes in Computer Science, vol. 6132, pp. 1–14. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-13217-9_1
8. Chandrasekar, K., Xiang Ni, 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 (May 2017)
9. Demeshko, I., Salinger, A.G., Spotz, W.F., Tezaur, I.K., Guba, O., Heroux, M.A.: Towards performance-portability of the albany finite element analysis code using the kokkos library of trilinos. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia (2016)
10. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: A domain specific language for building portable mesh-based pde solvers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC 11, Association for Computing Machinery, New York, NY, USA (2011), https://doi.org/10.1145/2063384.2063396
11. Edwards, H.C., Sunderland, D.: Kokkos array performance-portable manycore programming model. In: Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores. p. 110. PMAM 12, Association for Computing Machinery, New York, NY, USA (2012), https://doi.org/10.1145/2141702.2141703
12. Gautier, T., Ferreira Lima, J.V., Maillard, N., Raffin, B.: XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In: 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS). Boston, Massachusetts, United States (May 2013), https://hal.inria.fr/hal-00799904
13. Goglin, B.: Exposing the locality of heterogeneous memory architectures to hpc applications. In: Proceedings of the Second International Symposium on Memory Systems. p. 3039. MEMSYS 16, Association for Computing Machinery, New York, NY, USA (2016), https://doi.org/10.1145/2989081.2989115
14. Goglin, B., Rubio Proaño, A.: Opportunities for Partitioning Non-Volatile Memory DIMMs between Co-scheduled Jobs on HPC Nodes. In: Euro-Par 2019: Parallel Processing Workshops. Göttingen, Germany (Aug 2019), https://hal.inria.fr/hal-02173336
15. Huang, H.F., Jiang, T.: Design and implementation of flash based nvdimm. In: 2014 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA). pp. 1–6. IEEE (2014)
16. Iwabuchi, K., Lebanoff, L., Gokhale, M., Pearce, R.: Metall: A persistent memory allocator enabling graph processing. In: 2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3). pp. 39–44. IEEE (2019)
17. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Tech. Rep. LLNL-TR-641973 (August 2013)

18. Kayraklioglu, E., Chang, W., El-Ghazawi, T.: Comparative performance and optimization of chapel in modern manycore architectures. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 1105–1114 (May 2017)
19. Kirk, R.O., Mudalige, G.R., Reguly, I.Z., Wright, S.A., Martineau, M.J., Jarvis, S.A.: Achieving performance portability for a heat conduction solver mini-application on modern multi-core systems. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 834–841 (Sep 2017)
20. LLVM Foundation: LLVM Compiler Infrastucture, version 10.0.0 (2020), https://llvm.org/releases/download.html#10.0.0
21. Nagasaka, Y., Matsuoka, S., Azad, A., Buluç, A.: High-performance sparse matrix-matrix products on intel knl and multicore architectures. In: Proceedings of the 47th International Conference on Parallel Processing Companion. ICPP 18, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3229710.3229720
22. OpenMP Architecture Review Board: OpenMP application program interface version 5.0 (2018), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
23. Pérache, M., Jourdren, H., Namyst, R.: Mpc: A unified parallel runtime for clusters of numa machines. In: Proceedings of the 14th International Euro-Par Conference on Parallel Processing. p. 7888. Euro-Par '08, Springer-Verlag, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-85451-7_9
24. Reguly, I.Z., Mudalige, G.R., Giles, M.B.: Beyond 16gb: Out-of-core stencil computations. In: Proceedings of the Workshop on Memory Centric Programming for HPC. p. 2029. MCHPC17, Association for Computing Machinery, New York, NY, USA (2017), https://doi.org/10.1145/3145617.3145619
25. Schwalb, D., Berning, T., Faust, M., Dreseler, M., Plattner, H.: nvm malloc: Memory allocation for nvram. ADMS@ VLDB 15, 61–72 (2015)
26. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights landing: Second-generation intel xeon phi product. Ieee micro 36(2), 34–46 (2016)
27. Standard C++ Foundation: ISO International Standard ISO/IEC 14882:2017(E) Programming Language C++ (2017), https://isocpp.org/std/the-standard
28. Valat, S., Pérache, M., Jalby, W.: Introducing kernel-level page reuse for high performance computing. In: Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness. MSPC 13, Association for Computing Machinery, New York, NY, USA (2013), https://doi.org/10.1145/2492408.2492414
29. Virouleau, P., Roussel, A., Broquedis, F., Gautier, T., Rastello, F., Gratien, J.M.: Description, implementation and evaluation of an affinity clause for task directives. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) OpenMP: Memory, Devices, and Tasks. pp. 61–73. Springer International Publishing, Cham (2016)
30. Yoshida, T.: Fujitsu high performance cpu for the post-k computer. In: Hot Chips 30th Symposium (HCS) (August 2018)