Clustering Modes in Knights Landing Processors: Developer's Guide

Andrey Vladimirov and Ryo Asai

Colfax International

May 11, 2016

Abstract

This publication is part of a developer guide focusing on the new features in 2nd generation Intel® Xeon Phi™ processors code-named Knights Landing (KNL). In this document, we discuss the clustering modes of the on-die mesh interconnect.

We start a discussion on what types of applications benefit from the clustering modes and why clustering modes help these applications. After that we cover the specifics of the available cluster modes: all-to-all, quadrant, hemisphere, SNC-4 and SNC-2. Finally, we discuss how to make the application NUMA-aware for use in SNC modes. In this context, we give recipes for nested OpenMP and hybrid MPI+OpenMP approaches combined with first-touch allocation policy, numactl tool and memkind library.

This publication and other white papers on KNL processors can be found on the Colfax Research Website: colfaxresearch.com/knl-guide

Table of Contents

1. Cache Organization in KNL

2nd generation Intel® Xeon Phi™ processors code-named Knights Landing (KNL) are specialized computing platforms capable of delivering better performance than general-purpose CPUs such as Intel® Xeon® products for some applications. Applications run best on KNL if they have high degree of parallelism and well-behaved communication with memory. Specifically,

- If memory traffic is negligible compared to the processing of arithmetic, the application is *compute-bound* and may run well on KNL due to its high arithmetic peak performance [1].
- If memory access has predictable, sequential pattern, the application is *bandwidth-bound* and may run well on KNL due to its high-bandwidth memory (HBM) [2].
- Finally, if an application is neither compute-bound, nor bandwidth-bound because it has significant irregular memory access pattern, it belongs to the class of *latency-bound* applications.

In 1st generation Intel® Xeon Phi™ processors code-named Knights Corner (KNC), latency-bound applications performed poorly compared to Intel Xeon processors of comparable power. However, in KNL, significant improvements in cache organization reduce the impact of latency-bound operations.

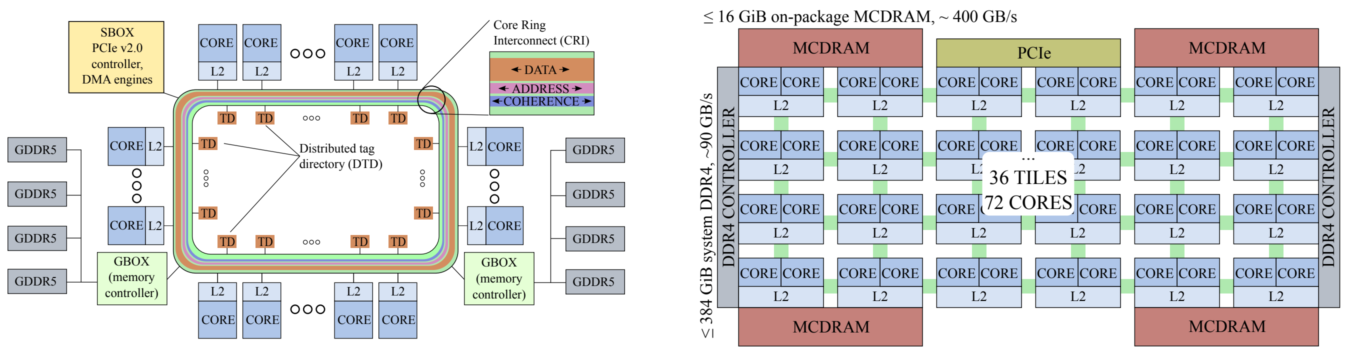
SBOX
PCIe v2.0 controller, DMA engines
CORE
L2
TD
Core Ring Interconnect (CRI)
DATA
ADDRESS
COHERENCE
Distributed tag directory (DTD)
GDDR5
GBOX (memory controller)
≤ 16 GiB on-package MCDRAM, ~ 400 GB/s
MCDRAM
PCIe
≤ 384 GiB system DDR4, ~90 GB/s
DDR4 CONTROLLER
36 TILES
72 CORES

Figure 1: Distributed Tag Directory on Intel Xeon Phi Processors. Left: Knights Corner (1st generation), right: Knights Landing (2nd generation)

In KNL (see Figure 1, right), each of its $\leq$ 72 cores has an L1 cache, pairs of cores are organized into tiles with a slice of the L2 cache symmetrically shared between the two cores, and the L2 caches are connected to each other with a mesh. All caches are kept coherent by the mesh with the MESIF protocol (this is an acronym for Modified/Exclusive/Shared/Invalid/Forward states of cache lines). In the mesh, each vertical and horizontal link is a bidirectional ring. In contrast, KNC (Figure 1, left) had only one bidirectional ring connecting up to 61 cores.

To maintain cache coherency, KNL has a distributed tag directory (DTD), organized as a set of per-tile tag directories (TDs), which identify the state and the location on the chip of any cache line. For any memory address, the hardware can identify with a hash function the TD responsible for that address.

These improvements in cache organization in KNL come with increased complexity of the chip hardware. To manage this complexity and set the optimal mode of operation for any given computational application, the programmer has access to cache clustering modes. Their purpose and utilization is discussed in this paper.

2. Clustering Modes

When an application requests data from memory address, the processing tile (let's call it tile A) will first query the local cache to see if the requested memory address is present there. If it is, the calculation will proceed with minimal latency for data access. Otherwise, tile A will query the DTD for the cache line (i.e., a 64-byte block of memory) containing that data. This means that a message will be sent from tile A to a TD on another tile (call it tile B). If according to the TD, this cache line is present in some other tile's L2 cache (call it tile C), another message will be sent from tile B to tile C, and finally, tile C will send the data to tile A. If the requested memory address is not cached, tile B will forward the request to the memory controller responsible for this address (call it controller D). This may be on-package MCDRAM-based memory controller or on-platform DDR4-based memory controller.

Naturally, it is in the developer's interests to maintain locality of these messages to achieve the lowest latency and greatest bandwidth of communication with caches. KNL supports all-to-all, quadrant/hemisphere and sub-NUMA cluster SNC-4/SNC-2 modes of cache operation, and their properties and utilization are discussed below.

2.1. All-to-All

With the all-to-all clustering mode, memory addresses are uniformly distributed across all TDs on the chip. This mode can have "unfortunate cases" where the points A, B, C and D are far apart, and the latency of cache hits and cache misses is long. Figure 2 demonstrates this mode.

The all-to-all mode should not be used for day-to-day operation of KNL. It is supported for troubleshooting and for situations where other clustering modes cannot operate; for example, a missing memory module or faulty MCDRAM would require using the all-to-all mode.

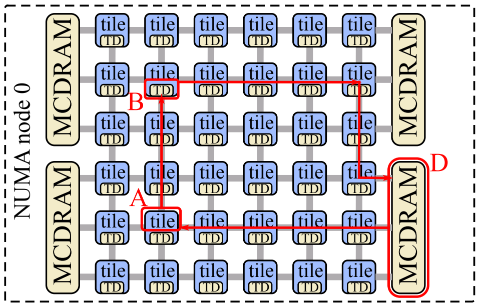

Figure 2: Cache miss in the all-to-all mode.

2.2. Quadrant/Hemisphere

In the quadrant clustering mode, the tiles are divided into four parts called quadrants, which are spatially local to four groups of memory controllers. Memory addresses served by a memory controller in a quadrant are guaranteed to be mapped only to TDs contained in that quadrant. Hemisphere mode functions the same way, except that the die is divided into two hemispheres instead of four quadrants. In the quadrant and hemisphere modes, the latency of L2 cache misses is reduced compared to the all-to-all mode because the worst-case path is shorter.

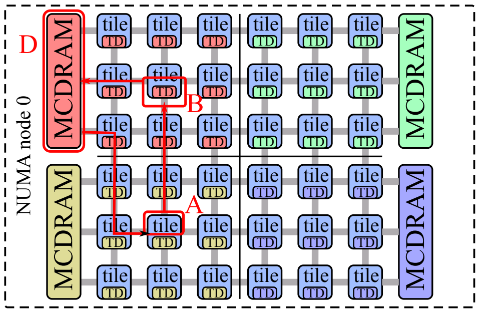

Figure 3: Cache miss in the quadrant clustering mode.

Figure 3 shows an example of an L2 cache miss in the quadrant mode. Because the memory controller and the tag directory are in the same area, the memory request will never need to go across quadrants.

The division into quadrants is hidden from the operating system: there are no "breaks" in the address space and the memory appears to be one contiguous block from the user's perspective. This is the recommended model to use for applications that treat KNL as a symmetric multi-processor (SMP).

2.3. SNC-4/SNC-2

The sub-NUMA cluster modes SNC-4 and SNC-2 also partition the chip into four quadrants or two hemispeheres, and, in addition, expose these quadrants (hemispheres) as NUMA nodes. In this mode, NUMA-aware software can pin software threads to the same quadrant (hemisphere) that contains the TD and accesses NUMA-local memory.

Figure 4 shows an example of an L2 cache miss in the SNC-4 mode. Because tiles A, B and C and memory controller D are physically close to each other, this mode has the lowest latency provided that communication stays within a NUMA domain. At the same time, if in SNC-4 or SNC-2 mode cache traffic crosses NUMA boundaries, this path is more expensive than in the quadrant mode. KNL in SNC-4 (-2) mode is similar to a 4-way (2-way) Intel Xeon processor.

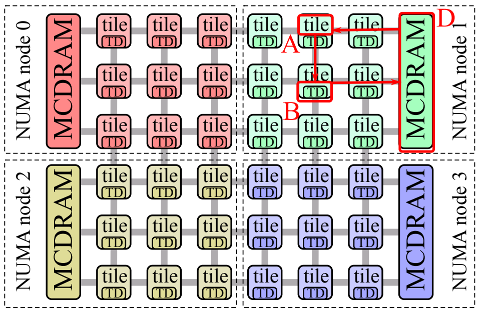

Figure 4: Cache miss in the SNC-4 mode.

Sub-NUMA clustering is the recommended mode of operation for NUMA-aware applications, i.e., applications that pin processing threads and their memory to the respective NUMA nodes. We discuss programming considerations for this mode in Section 3.

2.4. Setting the Clustering Mode

Clustering mode is a boot-time decision. The setting of the mode is done in the BIOS. There is no way to modify the clustering mode on KNL without restarting the system. The location of this setting in the BIOS interface depends on the vendor. For example, in one of our machines, it is located under Advanced → Chipset → North Bridge → QPI Configuration → Cluster Mode.

To check what mode the system is loaded in, there is a modified version of the `hwloc-dump-hwdata` for KNL processors that can output the mode. For example, the listing below reports "Cluster mode: SNC-4".

```
user@knl% sudo hwloc-dump-hwdata
Dumping KNL SMBIOS Memory-Side Cache information:
...
  Getting MCDRAM KNL info. Count=8 struct size=12
  MCDRAM controller 0
  Size = 2048 MB
  MCDRAM controller 1
  Size = 2048 MB
...
  Total MCDRAM 16384 MB
  Cluster mode: SNC-4
  Memory Mode: Flat
  Flat Mode: No MCDRAM cache available, nothing to dump.
```

3. Programming with Sub-NUMA Clusters

SNC-2 and SNC-4 clustering expose locally-communicating sub-domains of the chip as NUMA nodes. However, to take advantage of this architecture, the developer must make the application NUMA-aware by taking measures discussed in this section.

3.1. Querying NUMA Information

To see the NUMA configuration of KNL in sub-NUMA cluster modes, you can use the command `numactl` with the argument `-H` (see Listing 1).

```
user@knl% numactl -H
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  ... rest of node 0 cpus ...
node 0 size: 24452 MB
node 0 free: 22976 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 ... rest of node 1 cpus ...
node 1 size: 24576 MB
node 1 free: 23698 MB
node 2 cpus: 48 49 50 51 52 53 54 55 56 57 58 59 60 ... rest of node 2 cpus ...
node 2 size: 24576 MB
node 2 free: 23843 MB
node 3 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 ... rest of node 3 cpus ...
node 3 size: 24576 MB
node 3 free: 23693 MB
node 4 cpus:
node 4 size: 4096 MB
node 4 free: 3982 MB
node 5 cpus:
node 5 size: 4096 MB
node 5 free: 3982 MB
node 6 cpus:
node 6 size: 4096 MB
node 6 free: 3982 MB
node 7 cpus:
node 7 size: 4096 MB
node 7 free: 3982 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  21  21  21  31  41  41  41
  1:  21  10  21  21  41  31  41  41
  2:  21  21  10  21  41  41  41  31
  3:  21  21  21  10  41  41  31  41
  4:  31  41  41  41  10  41  41  41
  5:  41  31  41  41  41  10  41  41
  6:  41  41  41  31  41  41  10  41
  7:  41  41  31  41  41  41  41  10
```

Listing 1: Output from `numactl -H`. This particular system had 64 cores.

In our example, we have the SNC-4 mode set up, so cores appear grouped into four nodes with exactly a quarter of the on-platform memory in each node. In addition, our system uses the on-package high-bandwidth memory (HBM) in the flat mode (see [2]), which adds four more NUMA nodes with a quarter of the HBM in each node.

We can understand the relationship between these nodes by querying the distances between them shown at the bottom of Listing 1. To the nodes containing codes (0, 1, 2 and 3), the nearest memory is in the the same node (distance 10), after that is the memory in other on-platform nodes (distance 21) and after that is the memory in nearby HBM nodes (distance 31); memory in other HBM nodes is the farthest (distance 41). The distance to the HBM is greater than the distance to the on-platform memory, however, the bandwidth of HBM is greater than the bandwidth of on-platform memory – this is by design. Figure 5 illustrates the NUMA structure in this example.

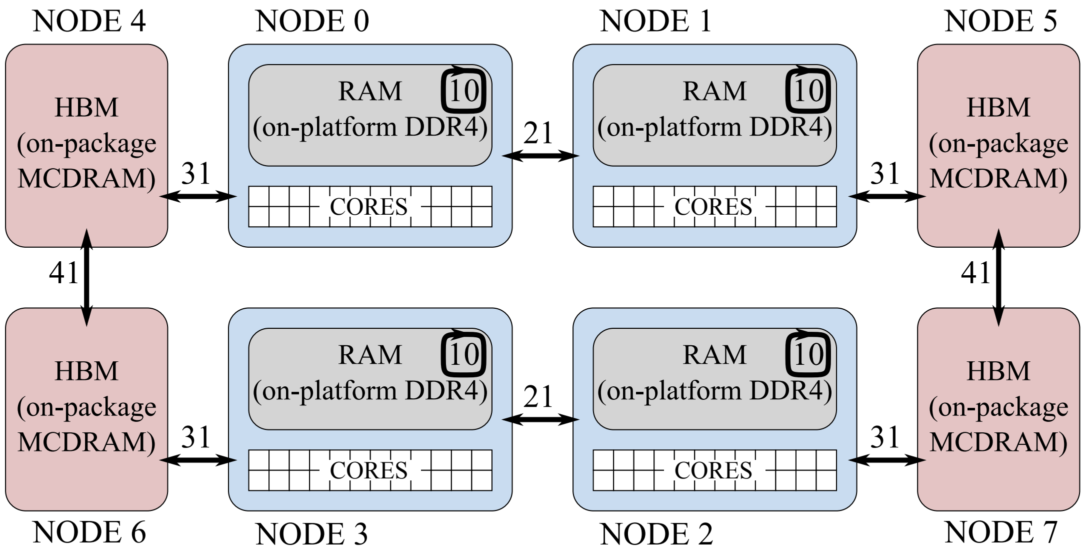

Figure 5: NUMA configuration in SNC-4 cache mode with HBM in flat mode. Labels on arrows indicate NUMA distances reported by `numactl`.

These distances indicate that in the SNC-4 mode, applications running completely in the on-platform memory (DDR4) need to allocate memory in the local NUMA nodes. If applications need to use the HBM, then for locality, threads on node 0 should allocate on node 4, threads on node 1 – on node 5, threads on node 2 – on node 7 and threads on node 3 – on node 6 (distance of 31).

For developers who wish to explicitly control NUMA allocation, the above information should provide sufficient guidance. However, in many cases, it is possible to take advantage of NUMA locality with generic methods that do not use explicit node numbers. These methods are discussed below.

3.2. Pinning Threads to Sub-NUMA Clusters

3.2.1. Nested OpenMP

OpenMP is an open-standard framework for multi-threading supported by Intel C, C++ and Fortran compilers as well as by the GNU Compiler Collection (GCC) and a number of other compilers. Nested parallelism is supported since OpenMP 2.5, and it can be used to effectively take advantage of the sub-NUMA cluster mode.

To create nested parallel regions in OpenMP, code and environment variables shown in Listing 2 should be used. In this example we create 4 teams with 64 threads each.

At the time of writing this paper, the best known method for setting this is to use a combination of `OMP_PLACES` and `OMP_PROC_BIND` environment variables. Assuming that the user needs 4 teams with 64 threads and compact placement of threads within a team, the variables shown in Listing 3 can be used.

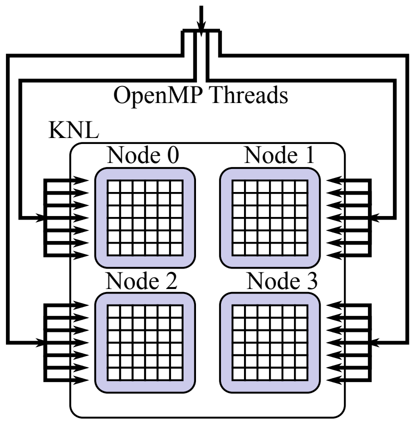

Figure 6: Using nested OpenMP for Quadrant mode.

```
#pragma omp parallel
{ /* 4 teams to be mapped across sub-NUMA clusters... */
#pragma omp parallel
  { /* 64 threads to work within each sub-NUMA clusters ... */ }
}
```

```
user@knl% export OMP_NESTED=1
user@knl% export OMP_NUM_THREADS=4,64
```

Listing 2: Example of nested OpenMP parallelism with 4 teams of 64 threads.

```
user@knl% export OMP_PLACES=0,1,2,3,4,5,6,7,8, ...other numa0cpus...,16,17,18,\
> 19,20,21,22,23,...other numa1cpus...,48,49,50,51,52,53,...other numa2cpus...,\
> 32,33,34,35,36,37,38,...other numa3cpus...,239
user@knl% export OMP_PROC_BIND=spread,close
```

Listing 3: Setting up thread affinity.

The environment variable `OMP_PLACES` defines places, i.e., groups of OS procs that threads can be bound to. As the value of this variable we listed 64 OS procs belonging to NUMA node 0 followed by 64 OS procs belonging to NUMA node 1, etc. With `OMP_PROC_BIND=spread,close`, the master threads of each team are going to be distributed uniformly across these places due to the "spread" argument, i.e., master thread for team 0 will land on OS proc 0, for team 1 on 16, etc. Within each team, threads will be placed close to each other, i.e., team 0 will populate NUMA node 0, team 1 – node 1, etc.

`OMP_PLACES` can be automatically generated from the output of `numactl`:

```
root@knl% export OMP_PLACES=`numactl -H | grep cpus | \
>        awk '(NF>3) {for (i = 4; i <= NF; i++) printf "%d,", $i}' | sed 's/.$//'`
```

3.2.2. Hybrid MPI+OpenMP

Message Passing Interface (MPI), industry-standard framework for communication in distributed-memory applications, and OpenMP can be combined to take advantage of the SNC-4 and SNC-2 modes. IN this setup, we will run four (for SNC-4) or two (for SNC-2) MPI processes with as many threads in each as there are logical processors in each sub-NUMA cluster (see Figure 7). Listing 4 illustrates the usage of OpenMP inside of an MPI process.

```
int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
#pragma omp parallel
  {
    // ...
  }
  MPI_Finalize();
}
```

Listing 4: Hybrid MPI+OpenMP programming.

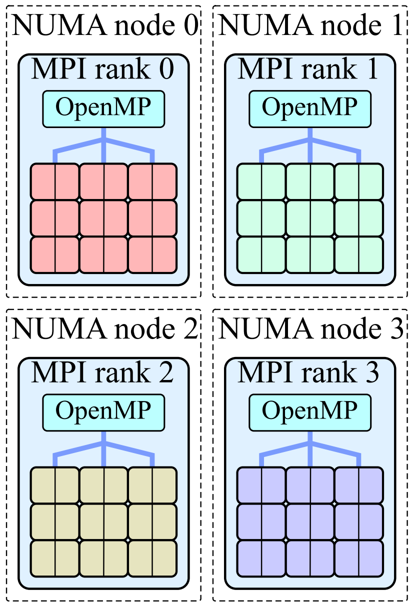

Figure 7: MPI+OpenMP for the SNC-4 mode on KNL.

NUMA-aware MPI implementations can pin processes to NUMA nodes of the KNL processor when the user specifies the corresponding number of processes per node. For instance, with Intel MPI we can confirm this behavior as shown Listing 5. There we set the environment variable `I_MPI_DEBUG=5` to produce the diagnostic output with pin CPU information.

```
user@knl% export I_MPI_DEBUG=5
user@knl% mpirun -host knl -np 4 ./my-hybrid-app
...
[0] MPI startup(): Rank   Pid     Node name   Pin cpu
[0] MPI startup(): 0      12326   knl         {0,1,2,3,4,5,6,7,8,9,10,11,12,13, ...
[0] MPI startup(): 1      12327   knl         {16,17,18,19,20,21,22,23,24,25,26, ...
[0] MPI startup(): 2      12328   knl         {48,49,50,51,52,53,54,55,56,57,58, ...
[0] MPI startup(): 3      12329   knl         {32,33,34,35,36,37,38,39,40,41,42, ...
...
```

Listing 5: Output of `mpirun` with `I_MPI_DEBUG=5` set.

Note that it is possible to use hybrid MPI+OpenMP approach in the quadrant mode as well. Similarly, in the SNC mode, it is possible to run a single MPI process per KNL processor with enough threads to scale across cores. In some applications, the balance between OpenMP and MPI (i.e., the number of threads per MPI process) is a performance tuning parameter (see, e.g., [3]). Therefore, the approach shown above (four MPI processes in SNC-4) should be treated not as a universal recipe, but as a starting point for application performance tuning.

3.3. Binding Memory to NUMA Nodes

With software threads pinned to the respective sub-NUMA clusters, the next task is to bind the memory objects used by these threads to the local (or near) NUMA nodes. With the methods shown in Sections 3.2.1 and 3.2.2, it is possible to do it in a natural way using the first-touch allocation policy, which is the default NUMA allocation policy in Linux. According to this policy, when a thread running on a NUMA node first touches a newly allocated object (i.e., writes into an array after calling `malloc`), the touched memory page is placed on the thread's NUMA node.

3.3.1. On-Platform Memory

If our intent is to allocate data in on-platform memory (the DDR4-based RAM), we can simply call `malloc()` followed by data initialization, and this will place the touched arrays into the NUMA on which the allocating thread is running. Listing 6 illustrates this approach.

```
#pragma omp parallel
{
  // Master thread of every outer
  // OpenMP region will initialize
  // its own array
  float *A = new float[N];

  // First touch places the array
  // on the local on-platform node
#pragma omp parallel for
  for (int i = 0; i < N; i++)
    A[i] = 0.0f;
}
```

```
int main(int argc, char **argv) {
{
  MPI_Init(argc, argv);
  // Master thread of each MPI proc.
  // will initialize its own array
  float *A = new float[N];

  // First touch places the array
  // on the local on-platform node
#pragma omp parallel for
  for (int i = 0; i < N; i++)
    A[i] = 0.0f;
}
```

Listing 6: Allocation of NUMA-local memory using first touch policy.

In the case of nested OpenMP, the master thread of every outer parallel region runs in its own sub-NUMA cluster (if the application is executed with environment variables shown in Section 3.2.1). First touch from thread team number 0 places the array `A` for that thread on NUMA node 0 (see Figure 5), first touch from team 1 places its `A` on node 1, etc.

It is possible to achieve NUMA locality with just one level of threading (i.e., without nested parallelism) as shown in Figure 8.

```
int main() {
  float *A = new float[N];
#pragma omp parallel for
  for (int i=0; i<N; i++)
    A[i] = 0.0f;
}
```

Figure 8: Parallel first touch for NUMA locality.

In this case, memory pages for parts of `A` first touched by threads in NUMA node 0 will be placed on node 0, parts touched by threads in node 1 will be placed on node 1, etc. To benefit from NUMA locality with one level of nesting, the parallel pattern with which `A` is used must be the same as the pattern with which it is allocated. Note that removing `#pragma omp parallel for` from initialization will generally degrade performance because the entire array will be placed on one NUMA node.

3.3.2. Entire Application in HBM

If the entire computational application fits within the amount of memory available in HBM (≤16 GiB in KNL), and if the application can benefit from HBM (see [2]), it is possible to place the application in HBM using the tool `numactl`. Examples of doing that in the SNC-4 mode is shown in Listing 7.

```
user@knl% # Memory of entire application
user@knl% # goes to NUMA nodes 4,5,6,7
user@knl% numactl -m 4,5,6,7 ./myApp
```

```
user@knl% #Same for an MPI applicatn.
user@knl% mpirun -np 4 -host knl \
>           numactl -m 4,5,6,7 ./myApp
```

Listing 7: Binding the entire application to HBM NUMA nodes in SNC-4. Left: nested OpenMP, right: MPI+OpenMP.

Even though we specify all four HBM NUMA nodes as targets for memory allocation, each of the four teams of OpenMP threads or each of the four MPI processes will allocate to its NUMA-local memory as long as the threads themselves are bound to NUMA nodes as discussed in Sections 3.2.1 and 3.2.2. That is, the team of threads (or MPI process) running on NUMA node 0 will allocate all of its data on NUMA node 4, team running on node 1 will allocate data to NUMA node 5, etc. This is because of the first-touch policy and because the local HBM nodes are closer than remote nodes to the respective core-containing nodes (see node distances in Section 3.1).

3.3.3. Selective Allocation in HBM

As described in [2], if the application needs more than 16 GiB of memory, or if not all objects in the application are bandwidth-critical, it is possible to store the bulk of the data in large DDR4-based memory, but selectively place certain objects or buffers in MCDRAM-based HBM. The memkind library can be used for that [4].

In the sub-NUMA cluster modes, allocating to NUMA-local HBM nodes with memkind works out-of-box as long as the OpenMP teams or MPI processes that allocate and touch memory are themselves bound to NUMA nodes as discussed in Sections 3.2.1 and 3.2.2. Listing 8 illustrates the recipe.

```c
#pragma omp parallel
{
  // Thread 0 -> NUMA node 4
  // Thread 1 -> NUMA node 5
  // etc.
  float *A = (float *)
    hbw_malloc(sizeof(float)*N);

  // First touch places the array
  // on the local HBM node
#pragma omp parallel for
  for (int i = 0; i < N; i++)
    A[i] = 0.0f;
}
```

```c
int main(int argc, char **argv) {
{
  // Rank 0 -> NUMA node 4
  // Rank 1 -> NUMA node 5
  // etc.
  float *A = (float *)
    hbw_malloc(sizeof(float)*N);

  // First touch places the array
  // on the local HBM node
#pragma omp parallel for
  for (int i = 0; i < N; i++)
    A[i] = 0.0f;
}
```

Listing 8: Allocation of NUMA-local memory with the memkind library.

3.4. Did it Work?

As of today, there is no function in memkind to check whether a given memory address is stored in the local or a remote SNC node. However, for the purposes of prototyping and testing, you may want to know where your data is going in memory. To do that, you can use the tool `numastat` as shown in Listing 9.

```
root@knl% numastat
                     node0    node1     node2    node3     node4    node5     node6    node7
numa_hit           2143516   110059   2443892   110059     38390    18918     19281     5059
numa_miss                0        0         0        0         0        0         0        0
...
```

Listing 9: Querying NUMA status to see where in memory data is allocated.

By making a note of `numa_hit` (counter of successful allocations to local NUMA nodes) before the start of your application comparing it to the same metric after the appication has run, you can identify whether nodes 4 through 7 (HBM) were utilized. Any non-zero values for `numa_miss` would indicate cross-SNC traffic.

Additionally, by specifying a process ID, we can get per-node usage information for a specific process as shown in Listing 10.

```
user@knl% numastat -p 10601

Per-node process memory usage (in MBs) for PID 10601 (mpi_test)
                           Node 0          Node 1          Node 2
                  --------------- --------------- ---------------
Huge                         0.00            0.00            0.00
Heap                         0.00            0.00            0.00
Stack                        0.00            0.00            0.00
Private                      2.21            0.51            0.46
----------------  --------------- --------------- ---------------
Total                        2.21            0.51            0.46

...

                           Node 6          Node 7           Total
                  --------------- --------------- ---------------
Huge                         0.00            0.00            0.00
Heap                         0.13            0.00            0.13
Stack                       12.47            0.00           12.47
Private                   1517.91            0.00         1521.28
----------------  --------------- --------------- ---------------
Total                     1530.52            0.00         1533.88
```

Listing 10: Querying per-node NUMA utilization for a specific process.

In this example, we can see that the process allocates memory to NUMA node 6, which is an HBM node local to on-platform node 3.

4. Summary

Knights Landing processors are more forgiving to applications sensitive to cache traffic than their predecessors (KNC) due to more complex cache structure. Additional performance improvements in such applications may come from tuning their execution environment and parallel pattern for the clustering modes supported by KNL. For applications that treat the KNL chip as an SMP, the quadrant and hemisphere mode may be used. For NUMA-aware applications, sub-NUMA cluster modes (SNC-4 and SNC-2) may be used.

Making an application NUMA-aware for use with SNC modes requires two components:

1. Pinning groups of executing threads to the respective core-containing NUMA domains. This may be achieved with
   - Nested OpenMP with special environment variable settings or
   - Hybrid MPI with 4 (2) processes per KNL processor for SNC-4 (-2)
2. Taking advantage of the NUMA policy of local allocation on first touch, i.e., allocating and initializing arrays by threads running on the respective sub-NUMA clusters. That means:
   - To allocate the entire application in NUMA-local DDR4-based nodes, only the pinning of threads needs to be done, and allocated arrays must be initialized from the outer parallel regions or from the master thread of MPI processes.
   - To allocate the entire application in NUMA-local MCDRAM-based nodes, in addition to the above, `numactl -m ...` can be used.
   - To selectively allocate in NUMA-local MCDRAM-based nodes, memkind library allocators may be used in combination with the thread pinning and first-touch recipes.

See related paper [2] for more information on MCDRAM-based HBM in KNL.

References

[1] Guide to Automatic Vectorization with Intel AVX-512 Instructions in Knights Landing Processors. http://colfaxresearch.com/knl-avx-512/.

[2] MCDRAM as High-Bandwidth Memory (HBM) in Knights Landing Processors: Developer's Guide. http://colfaxresearch.com/knl-mcdram/.

[3] Andrey Vladimirov. Cluster-Level Tuning of a Shallow Water Equation Solver on the Intel MIC Architecture. http://research.colfaxinternational.com/post/2014/05/12/Shallow-Water.aspx.

[4] Open source memkind library on github. https://github.com/memkind/memkind.