Migrating Offloading Software to Intel® Xeon Phi™ Processor

White Paper

February 2018

Document Number: 337129-001US

Contents

Figures

Revision History

Document Number	Revision Number	Description	Date
337129	001	• Initial Release	February 2018

§

1 Introduction

The Intel® Xeon Phi™ x100 coprocessor introduced the concept of many core architecture with more than 57 processor cores in one package, 4-way multithreading and 512-bit vector instructions (Intel® Initial Many Core Instructions – Intel® IMCI). It enabled many usage models and can still be found in numerous machines. At the moment of writing, the largest computing cluster with hosts equipped with Intel® Xeon Phi™ x100 coprocessors is ranked 2nd on the list of 500 fastest clusters in the world (TOP500 list from June 2017[1].

Many different programming models have been adopted for Intel® Xeon Phi™ x100 coprocessor and significant investments have been made into modernization of applications running on it.

There are three main programming models or types of applications that use Intel® Xeon Phi™ x100 coprocessor:

- Native applications running on Intel® Xeon Phi™ x100 coprocessors.
- Distributed applications
- which use Message Passing Interface (MPI) to communicate between processes (called MPI ranks) on Intel® Xeon Phi™ x100 coprocessors and platforms with Intel® Xeon® processors.
- Offload applications running on platforms with Intel® Xeon® processors and using compiler offloading features (Intel® Language Extensions for Offload or OpenMP* target directives) or communicating via Symmetric Communication Interface (SCIF) to execute code on Intel® Xeon Phi™ x100 coprocessors.

The introduction of Intel® Xeon Phi™ Processor brought another breakthrough. It not only extended the many core concept by adding more cores (64-72) but also increased computing power, improved 512-bit vector instructions and added new type of memory (MCDRAM). Finally, the next generation of devices can work independently as the main CPU of a bootable platform.

Thanks to the fact that Intel® Xeon Phi™ Processor is an independent device, the programming models known for Intel® Xeon® processors can be employed and porting highly parallel applications to Intel® Xeon Phi™ Processor is relatively straightforward. However, there are applications that have both strong serial and strong parallel parts and take advantage of heterogeneous nature of Intel® Xeon® hosts equipped with Intel® Xeon Phi™ x100 coprocessors.

This white paper proposes migration paths for such applications (referred to as heterogeneous applications in the document), both from hardware and software perspective.

§

1. https://www.top500.org/lists/2017/06/

2 Hardware Configurations

2.1 Introduction

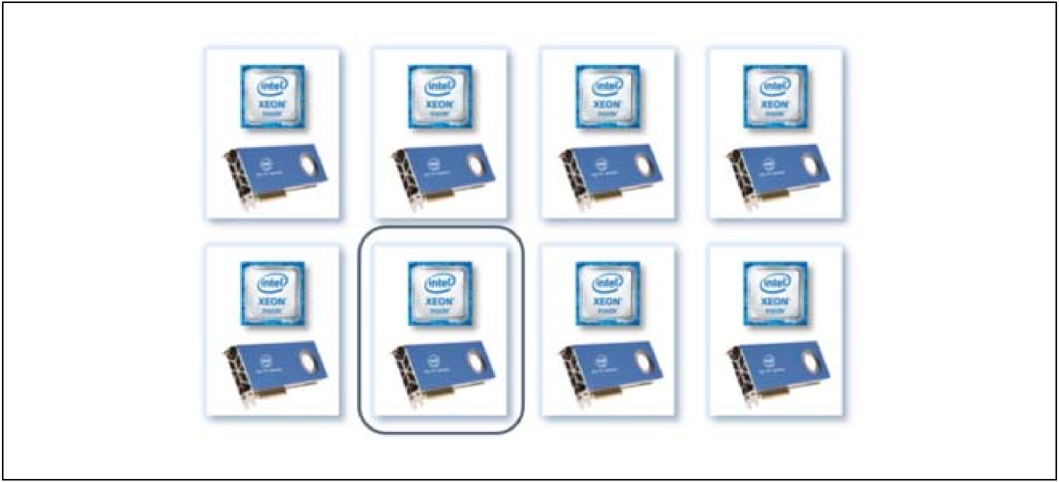
There are three Intel® hardware products that can not only replace Intel® Xeon Phi™ x100 coprocessors in computing platforms, but also offer better functionality and improved performance:

- Intel® Xeon® processors can power workstations or rack-mounted servers. Visit this link to learn more about Intel® Xeon® processors.
- Intel® Xeon Phi™ Processors are usually mounted in rack-mounted servers, but are also available in workstations (http://dap.xeonphi.com/). Follow this link to learn more about Intel® Xeon Phi™ x200 product family.
- Intel® Omni-Path Architecture (Intel® OPA) fast fabric adapters are usually separate PCI-e extension cards, but can also be integrated in Intel® Xeon Phi™ x200 processors. Intel® Omni-Path Architecture is designed to connect nodes in clusters, but also supports point to point communication, so just two nodes can be connected with fast fabric, for example a platform with Intel® Xeon® processor and a workstation with Intel® Xeon Phi™ x200 processor. It is also possible to install two Intel® OPA adapters in one machine to create a one-to-two configuration. Visit this link for more information.

2.2 Large and medium clusters

Traditionally Intel® Xeon Phi™ x100 coprocessors in large clusters were installed in servers with Intel® Xeon® processors, creating what can be called a hardware-defined heterogeneous topology. The following figure illustrates this approach. Usually, all nodes were connected with a fast fabric.

Figure 2-1. Fixed Assignment of Accelerators to Computing Nodes



If highly parallel applications with relatively small serial parts prevail in the anticipated usage model of a cluster, homogeneous cluster of Intel® Xeon Phi™ Processor based nodes should be the number one choice. If the planned usage model contains applications with both serial and parallel parts, the topology shown in Figure 2-1, "Fixed Assignment of Accelerators to Computing Nodes" on page 7 can be extended by adding Intel® Xeon Phi™ Processor based nodes and connecting them with a fast fabric to the original cluster. Intel® Xeon Phi™ x100 coprocessors could then be removed, as Intel® Xeon Phi™ Processors can take over their responsibilities.

Figure 2-2. Heterogeneous Computing Cluster

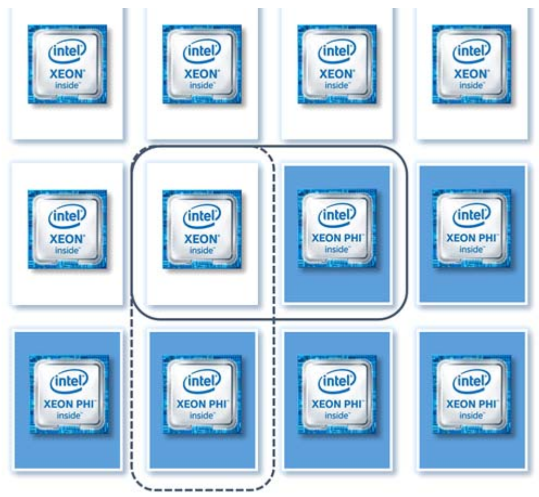
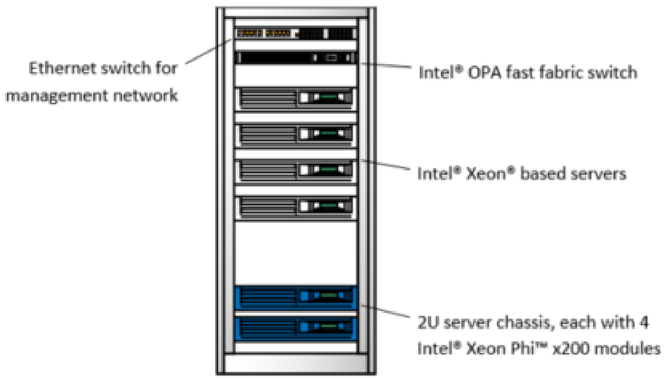


Figure 2-2 shows a heterogeneous cluster with systems based on Intel® Xeon® processors and Intel® Xeon Phi™ Processors. Heterogeneous computing nodes can be joined in different configurations based on requirements of particular applications. This flexibility is usually provided by a fast fabric network, such as Intel® OPA. Both types of servers can reside in one rack or cabinet (the following figure) or in separate racks.

Figure 2-3. Cluster Rack with Network Switches

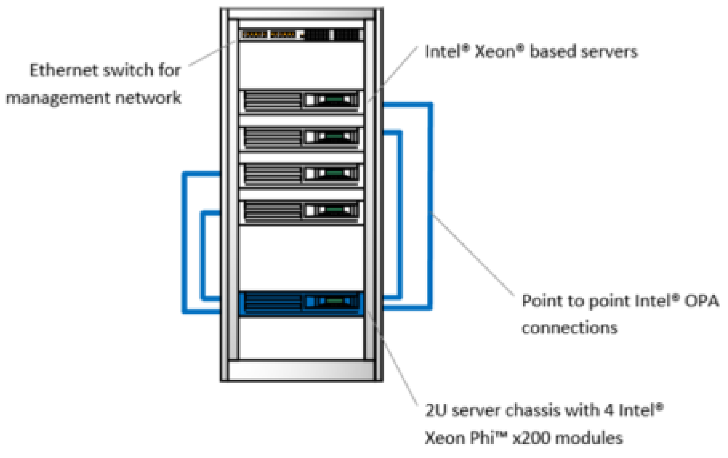


Ethernet switch for management network
Intel® OPA fast fabric switch
Intel® Xeon® based servers
2U server chassis, each with 4 Intel® Xeon Phi™ x200 modules

2.3 Small clusters and workstations

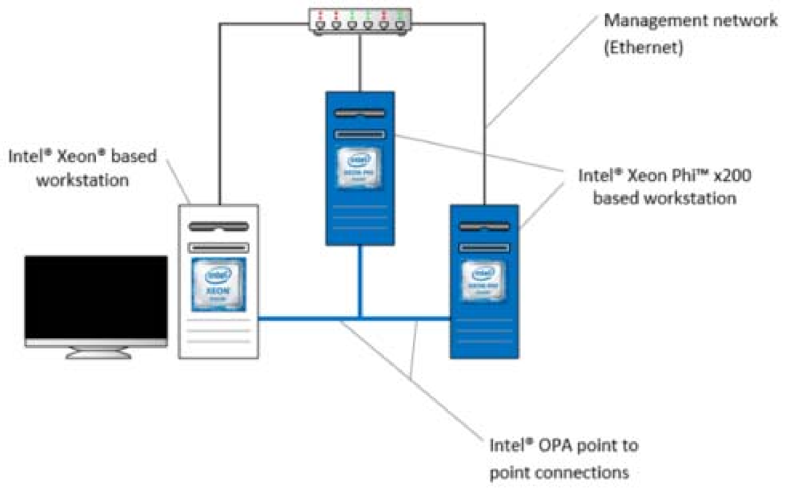
Creating a fast fabric network for small clusters and in workstation environments can significantly increase the cost of the installation. The main contribution to that is the cost of fast fabric network switches. However, it is possible to create small setups using Intel® OPA adapters in point to point configuration. The following figure shows an example configuration of a server rack with four Intel® Xeon® processor based servers connected using point to point connections to four servers with Intel® Xeon Phi™ Processors, which were installed in a single 2U server chassis.

Figure 2-4. Example Server Rack Using Point to Point Fabric Connections



Ethernet switch for management network
Intel® Xeon® based servers
Point to point Intel® OPA connections
2U server chassis with 4 Intel® Xeon Phi™ x200 modules

Figure 2-5. Example of a Workstation Based Setup (1:2)



Management network (Ethernet)
Intel® Xeon® based workstation
Intel® Xeon Phi™ x200 based workstation
Intel® OPA point to point connections

Both rack-mounted servers and workstations with Intel® Xeon Phi™ Processors are available (see http://dap.xeonphi.com/). It is therefore possible to build workstation setups connected with Intel® OPA fast fabric and, additionally, with regular Ethernet connections. This topology can utilize one to one connection or two Intel® Xeon Phi™ Processor based workstations serving one system with Intel® Xeon® processor (see Figure 2-5). All software solutions that can be used for migration will work on such setup.

§

3 Software Migration

3.1 Introduction

Porting different application types described in Chapter 1, “Introduction” can be quite straightforward if the right tools are utilized in the process.

Migration of any application running the Intel® Xeon Phi™ x100 coprocessor consist of two major steps:

1. Porting the application to run on the Intel® Xeon Phi™ Processor based system
2. Tuning of the application to utilize available resources in the best possible way.

3.2 Software tools

3.2.1 Intel® MPI Library

Intel® MPI is an implementation of the Message Passing Interface, a highly optimized communication runtime standardized by the MPI Forum. It consist of hundreds of functions, but the simplest program can be built using just a few. The most basic communication is provided by *MPI_Send* and *MPI_Recv* pairs, used to exchange messages between processes (or ranks in the MPI nomenclature) that can run on different machines. MPI implementations are often highly optimized for a particular network type or even a particular network topology. The following example code demonstrates a simple program using MPI:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int my_rank;
    int ranks_no;


    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &ranks_no);

    if (my_rank == 0)
    {
        // Orchestrator
        int other_rank;
       for (other_rank = 1; other_rank < ranks_no; other_rank++)
        {
                // Receive rank numbers from other ranks
                int other_rank_received = -1;
                MPI_Recv(&other_rank_received, 1, MPI_INTEGER,
                         other_rank, 1, MPI_COMM_WORLD, &status);
               printf("Rank %d reported!\n", other_rank_received);

            }
        }
        else
        {
            // Report own rank to the orchestrator
            MPI_Send(&my_rank, 1, MPI_INTEGER, 0, 1, MPI_COMM_WORLD);
        }

        MPI_Finalize();

        return 0;
    }
```

Compile and run the example using the Intel® MPI Compiler:

```
$ mpiicc –o basic_mpi basic_mpi.c
$ mpirun –np 10 ./basic_mpi
```

The *np* parameter instructs the MPI runtime to start 10 ranks on a local machine.

To learn more about Intel® MPI Library visit https://software.intel.com/en-us/intel-mpi-library.

3.2.2 Offloading over Fabric

The Intel® C/C++ and Fortran Compilers support offloading directives in the source code. This feature allows the application developer to specify which parts of the program will be offloaded to the Intel® Xeon Phi™ processor-based coprocessors or nodes. The Intel® offloading runtime supports Intel® Language Extensions for Offload (Intel® LEO) and OpenMP* target directives.

An example code using OpenMP* target directives to execute code on a coprocessor can look like this:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(void)
{
    printf("Example: Offload using OpenMP target directives\n");
    printf("Hello from the host!\n");
    #pragma omp target
    {
        if (omp_is_initial_device() != 0)
        {
            printf("Offload executed on host.\n");
            exit(-1);
        }
```

```
            printf("Hello from the target!\n");
        }
        return 0;
    }
```

To compile this program and run it on a host equipped with Intel® Xeon Phi™ x100 coprocessor execute:

```
$ icc -qopenmp –o omp_basic omp_basic.c
$ ./omp_basic
```

With the introduction of the Intel® Xeon Phi™ Processor, the offloading programming model is implemented as Offload over Fabric (OOF) and enables offloading to compute nodes connected within a high-speed network, such as Intel® OPA. Communication with the networking layer is realized by the Open Fabric Interface API (OFI). OOF allows easy porting of applications using offloading programming models. To compile the same code example for Offload over Fabric and run it between a host with Intel® Xeon® processor named *host* and a host with Intel® Xeon Phi™ Processor named *target* execute:

```
$ icc –qopenmp -qoffload-arch=mic-avx512 –o omp_basic
omp_basic.c
$ OFFLOAD_NODES=target ./omp_basic
```

To learn more about Offload over Fabric visit https://software.intel.com/en-us/articles/how-to-use-offload-over-fabric-with-knights-landing-intel-xeon-phi-processor.

3.3 Porting application to the Intel® Xeon Phi™ Processor

3.3.1 Native and MPI (distributed) applications

Porting native and MPI applications to Intel® Xeon Phi™ Processor based platforms can be very straightforward. In many cases only a simple recompilation for new hardware is required to have a running application:

```
$ icc –xMIC-AVX512 …
$ mpiicc –xMIC-AVX512 …
```

This strategy can sometimes fail if the application is using explicit vectorization (intrinsics). Intel® Xeon Phi™ x100 coprocessor uses Intel® Initial Many Core Instructions (Intel® IMCI) instruction set that is not fully compatible with AVX-512. Code can be ported to the new instruction set or modernized to use Intel® Compiler auto-vectorization features or one of the standard approaches to vectorization, e.g. OpenMP* SIMD directives. The last approach is very highly recommended – it will allow for better code portability in the future.

3.3.2 Offloading applications

Offloading using compiler directives

Offloading applications can require a little bit more work, mainly connected to installation and configuration of the Offload over Fabric runtime software. The runtime (for both Intel® Xeon® processor and Intel® Xeon Phi™ Processor based nodes) can be found at Intel® Xeon Phi™ Processor Software page. The website also contains detailed documentation on required configuration of nodes and offloading runtime.

When the Offloading over Fabric software is installed and tested, the offloading application can be recompiled for offloading to Intel® Xeon Phi™ Processor based host. For applications using OpenMP* target directives the *-qoffload-arch=mic-avx512* switch should be added to the compiler options:

```
$ icc –qopenmp -qoffload-arch=mic-avx512 …
```

To compile applications using Intel® Language Extensions for Offload (Intel® LEO) the *-qoffload* switch should be replaced with *-qoffload-arch=mic-avx512* compiler option:

```
$ icc -qoffload-arch=mic-avx512 …
```

There are two situations when code change may be required:

1. Explicit vectorization is used in the offloaded code (i.e. intrinsics)
2. *__MIC__* preprocessor definition is used in the offloaded code

The first case was addressed in Section 3.3.1, "Native and MPI (distributed) applications."

__MIC__ preprocessor macro allows to compile a code segment only if the code is compiled for Intel® Xeon Phi™ x100 coprocessor. For example:

```
#pragma omp target
{
#ifdef __MIC__
   // This code is executed on
   // Intel® Xeon Phi™ x100 coprocessor (target)
#else
      // This code is executed on Intel® Xeon® processor (host)
   #endif
}
```

This construct is possible because each offload region is implicitly compiled two times – once for Intel® Xeon® processor and once for Intel® Xeon Phi™ x100 coprocessor. It allows to execute offload regions on the host processor if no coprocessor is present in the system. For Intel® Xeon Phi™ x200 processor the *__MIC__* macro has been replaced with the *__TARGET_ARCH_MIC* macro and the code of the application may have to be changed to reflect that. The above example, after modification for running on Intel® Xeon Phi™ Processor, should look like this:

```
#pragma omp target
{
#ifdef __TARGET_ARCH_MIC
   // This code is executed on
   // Intel® Xeon Phi™ Processor (target)
#else
   // This code is executed on Intel® Xeon® processor (host)
   #endif
}
```

It is worth noting that the OpenMP* target directives can be automatically executed on the host processor if no coprocessor is available. It is also possible to compile code using Intel® Language Extensions for Offload so that the presence of a coprocessor in the system is not required and the code will be executed entirely on the host:

```
$ icc -qoffload=optional …
```

This method has a penalty of executing offload semantics without doing actual offloading, but applications with small amount of serial work can perform reasonably well on Intel® Xeon Phi™ Processor based systems. Those applications can be later modernized so as not to use the offloading model at all.

Offloading using SCIF API

There is a subset of offloading applications that use Symmetric Communication Interface (SCIF) to communicate with Intel® Xeon Phi™ x100 coprocessor and execute offloaded code in client/server model. Such applications can be easily migrated to other APIs working in highly optimized network environments, such as MPI. Table below shows example mapping of SCIF functions to MPI functions.

SCIF functionality	SCIF functions	MPI functions
Connection management	*scif_connect, scif_listen, scif_accept*	*Not necessary – handled by MPI runtime*
Sending/receiving messages	*scif_send/scif_recv*	*MPI_Send/MPI_Recv*
Remote Direct Memory Access (RDMA)	*scif_writeto/scif_readfrom scif_vwriteto/scif_vreadfrom*	*MPI_Put/MPI_Get*
Memory registration for RDMA	*scif_register/scif_unregister*	*MPI_Win_create/ MPI_Win_free*
Data transfer (RDMA) synchronization	*scif_fence_wait*	*MPI_Win_fence*

This is an example mapping that may not lead to optimal code, so more careful investigation of the application algorithms and MPI features should be performed before pursuing this migration path.

To achieve full performance the application should be compiled into two binaries – one optimized for Intel® Xeon® processor and one optimized for Intel® Xeon Phi™ Processor. The two binaries should be both started using a single *mpirun* command, which establishes communication between ranks running on different nodes. For example the command line below starts one rank (from the binary file named *app_mpi)* on a machine with hostname *hostname* and another rank (from the binary file optimized for Intel® Xeon Phi™ Processor named *app_mpi_phi*) on a machine called *targetname*:

```
$ mpirun -np 1 -hosts <hostname> ./app_mpi : \
   -np 1 -hosts <targetname> ./app_mpi_phi
```

3.4 Tuning applications for Intel® Xeon Phi™ Processor

3.4.1 Increased number of cores

Applications written with portability in mind, i.e. not hardcoded for the number of cores available on Intel® Xeon Phi™ x100 coprocessor, should scale to more cores. However, some new bottlenecks can emerge, so careful testing of the scalability of the application should be performed. Use tools such as Intel® VTune Amplifier XE to diagnose any potential issues.

3.4.2 Intel® Advanced Vector Extensions (Intel® AVX)-512

The SIMD width of Intel® Xeon Phi™ Processor vector instruction is the same as the width of Intel® Xeon Phi™ x100 coprocessor instructions, but the Instruction Set Architecture (ISA) is not fully compatible. One of the most important changes is the fact that Intel® Xeon Phi™ Processor (unlike Intel® Xeon Phi™ x100 coprocessor) provides additional support to SSE and Intel AVX-2 instruction sets. In portable programs most of the issues connected to vectorization should be handled by the compiler and runtimes, but the utilization of the vector processing units (VPUs) and vectorization quality should be monitored using tools such as Intel® Vector Advisor and Intel® Compiler vectorization reports. Those tools can advise the user on how to refactor their code to release its full potential when run on Intel® Xeon Phi™ Processor.

3.4.3 Cluster modes

The Intel® Xeon Phi™ Processor can run in several cluster (cluster on a die) modes, but for most workloads Quadrant mode is recommended. Some codes can achieve better performance when processor is run in SNC-4 cluster mode. In this mode the Intel® Xeon Phi™ Processor is virtually divided into four (in Cache memory mode) or eight (in Flat memory mode) Non-uniform Memory Access (NUMA) nodes. This affects how the operating system kernel works and may have both negative and positive effects on the performance of the application. It is recommended to test both SNC-4 and Quadrant modes to find the most suitable mode for particular application.

3.4.4 High bandwidth memory

Intel® Xeon Phi™ Processor is equipped with 16GB of high bandwidth memory (MCDRAM). This memory can be used automatically to extend existing cache hierarchy (when the processor runs in Cache memory mode) or can extend the available physical address space when the processor runs in Flat mode. It should be noted, that MCDRAM's latency can be slightly higher than the latency of DDR4 memory[1]. To saturate the bandwidth, it is recommended to access the MCDRAM from many threads running on Intel® Xeon Phi™ Processor.

Non-uniform Memory Access (NUMA) mechanism is used in Flat and Hybrid memory modes to expose the MCDRAM memory to the operating system. Learn more

about this mechanism.

Native/MPI applications

1. https://sites.utexas.edu/jdm4372/2016/12/06/memory-latency-on-the-intel-xeon-phi-x200-knights-landing-processor/

There are two approaches to using MCDRAM memory in Flat mode by native (and MPI) applications:

1. Using NUMA control features of the Linux* operating system.
2. Using explicit API calls: system or a library, such as
3. *memkind* library.

The first method can be used for workloads that require relatively small amounts of memory (up to 16GB) and therefore can fit entirely within MCDRAM. In Quadrant cluster mode, the MCDRAM memory is assigned to NUMA node 1 and the application can be bound to this node:

```
$ numactl –m 1 ./application
```

This command instructs the operating system to use strict *bind* policy: all allocations will be performed by the operating system from NUMA node 1 and therefore from MCDRAM memory. It is also possible to use the *preferred* policy:

```
$ numactl –p 1 ./application
```

This policy instructs the operating system to allocate from the NUMA node 1 (and thus MCDRAM) first and, once this resource is exhausted, from regular DDR memory. The performance of an application using preferred policy can be less predictable.
The second method (i.e. using the *memkind* library) allows the user to place only selected allocations in MCDRAM memory and can often achieve better results. Visit the *memkind* website and learn more about the library.
The OpenMP* committee is currently working on new features in the OpenMP* standard (see OpenMP* TR5 document for details) that will expose new kinds of memory to the users in an easy and portable way.

Offload applications

Offload regions running on Intel® Xeon Phi™ Processor based nodes use MCDRAM mode by default if it is available. The runtime can be configured to use different memory polices. The user can define the first kind of memory to be used (MCDRAM or DDR) and fallback mechanism. Fallback defines the action taken when the first kind of memory is exhausted: allocate from the other kind of memory or simply let the operating system abort the application.

It is also possible to use the *memkind* library to selectively place allocations made by the offloaded regions in the MCDRAM memory and use OpenMP* target pointers to register those allocations in the OpenMP* runtime:

```
#pragma omp target map(from: target_data)
   is_device_ptr(target_data)
{
    target_data = (double *)hbw_malloc(SIZE);
}

omp_target_memcpy(target_data, host_data, SIZE, 0, 0, 0,
omp_get_initial_device());


#pragma omp target is_device_ptr(target_data) map(from: result)
{
    result = compute(target_data);
}

#pragma omp target is_device_ptr(target_data)
{
    hbw_free(target_data);
}
```

This code can be compiled with the following command line:

```
$ icc <...> -lmemkind -qoffload-option,mic,compiler,\
'-lmemkind‘ <...>
```

See Offload over Fabric documentation for details about configuring the offloading runtime.

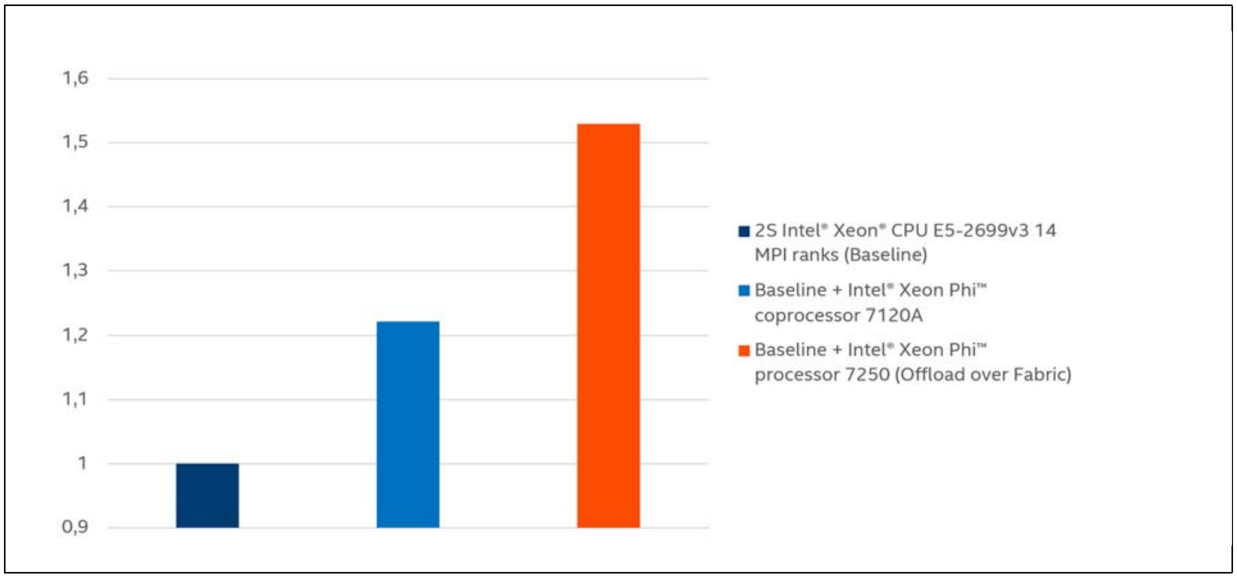
3.4.5 Load balancing of the application

Some offloading and heterogeneous MPI applications implemented automatic load balancing algorithm for dividing workloads between Intel® Xeon® processor and Intel® Xeon Phi™ x100 coprocessor. Those applications should also work well on Intel® Xeon Phi™ Processor based nodes. Programs without those mechanisms should be analyzed taking into account the fact that Intel® Xeon Phi™ Processor is a more powerful and conversely the work division that worked well for Intel® Xeon Phi™ x100 coprocessor will most likely be invalid when cooperating between Intel® Xeon® processors and Intel® Xeon Phi™ Processors based hosts.

3.4.6 Application examples

Techniques described in this section were successfully applied to existing applications and proved that the presented approach can improve the performance of existing applications. To confirm that we recompiled LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) for Offloading over Fabric and compared the results of running one of Liquid Cristal benchmarks to results achieved on a machine with two Intel® Xeon® processors and the same machine with Intel® Xeon Phi™ x100 coprocessor. It was a naive port, with no additional optimizations for Intel® Xeon Phi™ Processor. See Chapter 4, “Benchmarking and Benchmarks Details” for more information on benchmarking and configuration

Figure 3-1. LAMMPS* Ported to Offloading to Intel® Xeon Phi™ Processor



Another example of successful migration is porting the Parallel Tissue Modeling Framework (Timothy). Learn more about the results of this effort.

§

4 Benchmarking and Benchmarks Details

Software and workloads used in performance tests may have been optimized for performance on Intel® microprocessors only.

Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including performance of combinations of products. For more complete information visit www.intel.com/benchmarks.

Intel® measured results as of November 2016 using LAMMPS* benchmarks from the *USER-INTEL* package (*src/USER-INTEL/TEST*) with naive port to Offload over Fabric (compilation for offloading to Intel® Xeon Phi™ Processor).

OFFLOAD HOST AND BASELINE CONFIGURATION:

- Dual Socket Intel® Xeon® processor E5-2699 v3 (45 M Cache, 2.3 GHz, 18 Cores) with Intel® Hyper-Threading and Turbo Boost Technologies enabled
- 64 GB DDR4-2133 MHz memory
- Red Hat Enterprise Linux* 7.2 (Maipo)
- Intel® Omni-Path Host Fabric Interface Adapter 100 Series 1 Port PCIe* x16
- Intel® Server Board S2600WT2
- 500GB SATA drive ST500DM002 and 1TB ST1000NM0033 Disks.

14 MPI ranks and 2 threads on the Intel® Xeon® processor based host used in the benchmark, Intel® package, 1 offload target, automatic load balancing.

OFFLOAD TARGET CONFIGURATION:

- One node Intel® Xeon Phi™ Processor 7250 (16 GB, 1.4 GHz, 68 Cores) in Intel® Server System LADMP2312KXXX41
- 64GB DDR4, quad cluster mode
- MCDRAM flat memory mode
- Red Hat Enterprise Linux* 7.2 (Maipo)
- Intel® Omni-Path Fabric Interface
- 250MB SATA WD2502ABYS-0 System Disk.
- Intel® Compiler 17.0.0
- Intel® MPI Library 2017
- LAMMPS* code base: 30 Jul 16

§