Intel Xeon Phi

A guide to Intel Xeon Phi usage

Intel Xeon Phi accelerator can be programmed in several modes. The default mode on the cluster is offload mode, but all modes described in this document are supported.

Intel Utilities for Xeon Phi

To get access to a compute node with Intel Xeon Phi accelerator, use the PBS interactive session

```
$ qsub -I -q qprod -l select=1:ncpus=24:accelerator=True:naccelerators=2:accelerator_model=phi
```

To set up the environment module "intel" has to be loaded, without specifying the version, default version is loaded (at time of writing this, it's 2015b)

```
$ module load intel
```

Information about the hardware can be obtained by running the micinfo program on the host.

```
$ /usr/bin/micinfo
```

The output of the "micinfo" utility executed on one of the cluster node is as follows. (note: to get PCIe related details the command has to be run with root privileges)

```
MicInfo Utility Log
Created Mon Aug 17 13:55:59 2015

    System Info
        HOST OS         : Linux
        OS Version      : 2.6.32-504.16.2.el6.x86_64
        Driver Version      : 3.4.1-1
        MPSS Version        : 3.4.1
        Host Physical Memory    : 131930 MB

Device No: 0, Device Name: mic0

    Version
        Flash Version        : 2.1.02.0390
        SMC Firmware Version     : 1.16.5078
        SMC Boot Loader Version  : 1.8.4326
        uOS Version          : 2.6.38.8+mpss3.4.1
        Device Serial Number     : ADKC44601414
```

```
    Board
        Vendor ID        : 0x8086
        Device ID        : 0x225c
        Subsystem ID         : 0x7d95
        Coprocessor Stepping ID  : 2
        PCIe Width       : x16
        PCIe Speed       : 5 GT/s
        PCIe Max payload size    : 256 bytes
        PCIe Max read req size   : 512 bytes
        Coprocessor Model    : 0x01
        Coprocessor Model Ext    : 0x00
        Coprocessor Type     : 0x00
        Coprocessor Family   : 0x0b
        Coprocessor Family Ext   : 0x00
        Coprocessor Stepping     : C0
        Board SKU        : C0PRQ-7120 P/A/X/D
        ECC Mode         : Enabled
        SMC HW Revision      : Product 300W Passive CS

    Cores
        Total No of Active Cores : 61
        Voltage          : 1007000 uV
        Frequency        : 1238095 kHz

    Thermal
        Fan Speed Control    : N/A
        Fan RPM          : N/A
        Fan PWM          : N/A
        Die Temp         : 60 C

    GDDR
        GDDR Vendor      : Samsung
        GDDR Version         : 0x6
        GDDR Density         : 4096 Mb
        GDDR Size        : 15872 MB
        GDDR Technology      : GDDR5
        GDDR Speed       : 5.500000 GT/s
        GDDR Frequency       : 2750000 kHz
        GDDR Voltage         : 1501000 uV

Device No: 1, Device Name: mic1

    Version
        Flash Version        : 2.1.02.0390
        SMC Firmware Version     : 1.16.5078
        SMC Boot Loader Version  : 1.8.4326
```

```
    uOS Version          : 2.6.38.8+mpss3.4.1
    Device Serial Number     : ADKC44500454

Board
    Vendor ID        : 0x8086
    Device ID        : 0x225c
    Subsystem ID         : 0x7d95
    Coprocessor Stepping ID  : 2
    PCIe Width       : x16
    PCIe Speed       : 5 GT/s
    PCIe Max payload size    : 256 bytes
    PCIe Max read req size   : 512 bytes
    Coprocessor Model    : 0x01
    Coprocessor Model Ext    : 0x00
    Coprocessor Type     : 0x00
    Coprocessor Family   : 0x0b
    Coprocessor Family Ext   : 0x00
    Coprocessor Stepping     : C0
    Board SKU        : C0PRQ-7120 P/A/X/D
    ECC Mode         : Enabled
    SMC HW Revision      : Product 300W Passive CS

Cores
    Total No of Active Cores : 61
    Voltage          : 998000 uV
    Frequency        : 1238095 kHz

Thermal
    Fan Speed Control    : N/A
    Fan RPM          : N/A
    Fan PWM          : N/A
    Die Temp         : 59 C

GDDR
    GDDR Vendor      : Samsung
    GDDR Version         : 0x6
    GDDR Density         : 4096 Mb
    GDDR Size        : 15872 MB
    GDDR Technology      : GDDR5
    GDDR Speed       : 5.500000 GT/s
    GDDR Frequency       : 2750000 kHz
    GDDR Voltage         : 1501000 uV
```

Offload Mode

To compile a code for Intel Xeon Phi a MPSS stack has to be installed on the machine where compilation is executed. Currently the MPSS stack is only installed on compute nodes equipped with accelerators.

```
$ qsub -I -q qprod -l select=1:ncpus=24:accelerator=True:naccelerators=2:accelerator_model=phi
$ module load intel
```

For debugging purposes it is also recommended to set environment variable "OFFLOAD_REPORT". Value can be set from 0 to 3, where higher number means more debugging information.

```
export OFFLOAD_REPORT=3
```

A very basic example of code that employs offload programming technique is shown in the next listing. Please note that this code is sequential and utilizes only single core of the accelerator.

```
$ vim source-offload.cpp

#include <iostream>

int main(int argc, char* argv[])
{
    const int niter = 100000;
    double result = 0;

 #pragma offload target(mic)
    for (int i = 0; i < niter; ++i) {
        const double t = (i + 0.5) / niter;
        result += 4.0 / (t * t + 1.0);
    }
    result /= niter;
    std::cout << "Pi ~ " << result << 'n';
}
```

To compile a code using Intel compiler run

```
$ icc source-offload.cpp -o bin-offload
```

To execute the code, run the following command on the host

```
./bin-offload
```

Parallelization in Offload Mode Using OpenMP

One way of paralelization a code for Xeon Phi is using OpenMP directives. The following example shows code for parallel vector addition.

```
$ vim ./vect-add

#include <stdio.h>

typedef int T;

#define SIZE 1000

#pragma offload_attribute(push, target(mic))
T in1[SIZE];
T in2[SIZE];
T res[SIZE];
#pragma offload_attribute(pop)

// MIC function to add two vectors
__attribute__((target(mic))) add_mic(T *a, T *b, T *c, int size) {
  int i = 0;
  #pragma omp parallel for
    for (i = 0; i < size; i++)
      c[i] = a[i] + b[i];
}

// CPU function to add two vectors
void add_cpu (T *a, T *b, T *c, int size) {
  int i;
  for (i = 0; i < size; i++)
    c[i] = a[i] + b[i];
}

// CPU function to generate a vector of random numbers
void random_T (T *a, int size) {
  int i;
  for (i = 0; i < size; i++)
    a[i] = rand() % 10000; // random number between 0 and 9999
}

// CPU function to compare two vectors
int compare(T *a, T *b, T size ){
  int pass = 0;
  int i;
  for (i = 0; i < size; i++){
    if (a[i] != b[i]) {
      printf("Value mismatch at location %d, values %d and %dn",i, a[i], b[i]);
      pass = 1;
    }
  }
```

```
  if (pass == 0) printf ("Test passedn"); else printf ("Test Failedn");
  return pass;
}

int main()
{
  int i;
  random_T(in1, SIZE);
  random_T(in2, SIZE);

  #pragma offload target(mic) in(in1,in2)  inout(res)
  {

    // Parallel loop from main function
    #pragma omp parallel for
    for (i=0; i<SIZE; i++)
      res[i] = in1[i] + in2[i];

    // or parallel loop is called inside the function
    add_mic(in1, in2, res, SIZE);

  }

  //Check the results with CPU implementation
  T res_cpu[SIZE];
  add_cpu(in1, in2, res_cpu, SIZE);
  compare(res, res_cpu, SIZE);

}
```

During the compilation Intel compiler shows which loops have been vectorized in both host and accelerator. This can be enabled with compiler option "-vec-report2". To compile and execute the code run

```
$ icc vect-add.c -openmp_report2 -vec-report2 -o vect-add

$ ./vect-add
```

Some interesting compiler flags useful not only for code debugging are:

Debugging openmp_report[0|1|2] - controls the compiler based vectorization diagnostic level vec-report[0|1|2] - controls the OpenMP parallelizer diagnostic level

Performance ooptimization xhost - FOR HOST ONLY - to generate AVX (Advanced Vector Extensions) instructions.

Automatic Offload using Intel MKL Library

Intel MKL includes an Automatic Offload (AO) feature that enables computationally intensive MKL functions called in user code to benefit from attached Intel Xeon Phi coprocessors automatically and transparently.

Behavioural of automatic offload mode is controlled by functions called within the program or by environmental variables. Complete list of controls is listed here.

The Automatic Offload may be enabled by either an MKL function call within the code:

```
mkl_mic_enable();
```

or by setting environment variable

```
$ export MKL_MIC_ENABLE=1
```

To get more information about automatic offload please refer to "Using Intel® MKL Automatic Offload on Intel ® Xeon Phi™ Coprocessors" white paper or Intel MKL documentation.

Automatic offload example #1

Following example show how to automatically offload an SGEMM (single precision - g dir="auto">eneral matrix multiply) function to MIC coprocessor.

At first get an interactive PBS session on a node with MIC accelerator and load "intel" module that automatically loads "mkl" module as well.

```
$ qsub -I -q qprod -l select=1:ncpus=24:accelerator=True:naccelerators=2:accelerator_model=phi
$ module load intel
```

The code can be copied to a file and compiled without any necessary modification.

```
$ vim sgemm-ao-short.c
```

‘ #include #include #include #include

include "mkl.h"

int main(int argc, char **argv) { float *A,* B, *C; /* Matrices */

MKL_INT N = 2560; /* Matrix dimensions */ *MKL_INT LD = N; /* Leading dimension */ int matrix_bytes; /* Matrix size in bytes */ int matrix_elements; /* Matrix size in elements */*

float alpha = 1.0, beta = 1.0; /* Scaling factors / *char transa = 'N', transb = 'N'; /* Transposition options */

int i, j; /* Counters */

matrix_elements = N * N; matrix_bytes = sizeof(float) * matrix_elements;

/* Allocate the matrices */ A = malloc(matrix_bytes); B = malloc(matrix_bytes); C = malloc(matrix_bytes);

/* Initialize the matrices */ for (i = 0; i < matrix_elements; i++) { A[i] = 1.0; B[i] = 2.0; C[i] = 0.0; }

printf("Computing SGEMM on the hostn"); sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, &beta, C, &N);

printf("Enabling Automatic Offloadn"); /* Alternatively, set environment variable MKL_MIC_ENABLE=1 / *mkl_mic_enable(); int ndevices = mkl_mic_get_device_count(); /* Number of MIC devices */ printf("Automatic Offload enabled: %d MIC devices presentn", ndevices);

printf("Computing SGEMM with automatic workdivisionn"); sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, &beta, C, &N);

/* Free the matrix memory */ free(A); free(B); free(C);

printf("Donen");

return 0; } '

Please note: This example is simplified version of an example from MKL. The expanded version can be found here: $MKL_EXAMPLES/mic_ao/blasc/source/sgemm.c**

To compile a code using Intel compiler use:

```
$ icc -mkl sgemm-ao-short.c -o sgemm
```

For debugging purposes enable the offload report to see more information about automatic offloading.

```
$ export OFFLOAD_REPORT=2
```

The output of a code should look similar to following listing, where lines starting with [MKL] are generated by offload reporting:

```
[user@r31u03n799 ~]$ ./sgemm
Computing SGEMM on the host
Enabling Automatic Offload
Automatic Offload enabled: 2 MIC devices present
Computing SGEMM with automatic workdivision
[MKL] [MIC --] [AO Function]    SGEMM
[MKL] [MIC --] [AO SGEMM Workdivision]    0.44 0.28 0.28
```

```
[MKL] [MIC 00] [AO SGEMM CPU Time]    0.252427 seconds
[MKL] [MIC 00] [AO SGEMM MIC Time]    0.091001 seconds
[MKL] [MIC 00] [AO SGEMM CPU->MIC Data]    34078720 bytes
[MKL] [MIC 00] [AO SGEMM MIC->CPU Data]    7864320 bytes
[MKL] [MIC 01] [AO SGEMM CPU Time]    0.252427 seconds
[MKL] [MIC 01] [AO SGEMM MIC Time]    0.094758 seconds
[MKL] [MIC 01] [AO SGEMM CPU->MIC Data]    34078720 bytes
[MKL] [MIC 01] [AO SGEMM MIC->CPU Data]    7864320 bytes
Done
```

Behavioral of automatic offload mode is controlled by functions called within the program or by environmental variables. Complete list of controls is listed here.

To get more information about automatic offload please refer to "Using Intel® MKL Automatic Offload on Intel ® Xeon Phi™ Coprocessors" white paper or Intel MKL documentation.

Automatic offload example #2

In this example, we will demonstrate automatic offload control via an environment vatiable MKL_MIC_ENABLE. The function DGEMM will be offloaded.

At first get an interactive PBS session on a node with MIC accelerator.

```
$ qsub -I -q qprod -l select=1:ncpus=24:accelerator=True:naccelerators=2:accelerator_model=phi
```

Once in, we enable the offload and run the Octave software. In octave, we generate two large random matrices and let them multiply together.

```
$ export MKL_MIC_ENABLE=1
$ export OFFLOAD_REPORT=2
$ module load Octave/3.8.2-intel-2015b

$ octave -q
octave:1> A=rand(10000);
octave:2> B=rand(10000);
octave:3> C=A*B;
[MKL] [MIC --] [AO Function]    DGEMM
[MKL] [MIC --] [AO DGEMM Workdivision]    0.14 0.43 0.43
[MKL] [MIC 00] [AO DGEMM CPU Time]    3.814714 seconds
[MKL] [MIC 00] [AO DGEMM MIC Time]    2.781595 seconds
[MKL] [MIC 00] [AO DGEMM CPU->MIC Data]    1145600000 bytes
[MKL] [MIC 00] [AO DGEMM MIC->CPU Data]    1382400000 bytes
[MKL] [MIC 01] [AO DGEMM CPU Time]    3.814714 seconds
[MKL] [MIC 01] [AO DGEMM MIC Time]    2.843016 seconds
[MKL] [MIC 01] [AO DGEMM CPU->MIC Data]    1145600000 bytes
[MKL] [MIC 01] [AO DGEMM MIC->CPU Data]    1382400000 bytes
```

```
octave:4> exit
```

On the example above we observe, that the DGEMM function workload was split over CPU, MIC 0 and MIC 1, in the ratio 0.14 0.43 0.43. The matrix multiplication was done on the CPU, accelerated by two Xeon Phi accelerators.

Native Mode

In the native mode a program is executed directly on Intel Xeon Phi without involvement of the host machine. Similarly to offload mode, the code is compiled on the host computer with Intel compilers.

To compile a code user has to be connected to a compute with MIC and load Intel compilers module. To get an interactive session on a compute node with an Intel Xeon Phi and load the module use following commands:

```
$ qsub -I -q qprod -l select=1:ncpus=24:accelerator=True:naccelerators=2:accelerator_model=phi

$ module load intel
```

Please note that particular version of the Intel module is specified. This information is used later to specify the correct library paths.

To produce a binary compatible with Intel Xeon Phi architecture user has to specify "-mmic" compiler flag. Two compilation examples are shown below. The first example shows how to compile OpenMP parallel code "vect-add.c" for host only:

```
$ icc -xhost -no-offload -fopenmp vect-add.c -o vect-add-host
```

To run this code on host, use:

```
$ ./vect-add-host
```

The second example shows how to compile the same code for Intel Xeon Phi:

```
$ icc -mmic -fopenmp vect-add.c -o vect-add-mic
```

Execution of the Program in Native Mode on Intel Xeon Phi

The user access to the Intel Xeon Phi is through the SSH. Since user home directories are mounted using NFS on the accelerator, users do not have to copy binary files or libraries between the host and accelerator.

Get the PATH of MIC enabled libraries for currently used Intel Compiler (here was icc/2015.3.187-GNU-5.1.0-2.25 used) :

```
$ echo $MIC_LD_LIBRARY_PATH
/apps/all/icc/2015.3.187-GNU-5.1.0-2.25/composer_xe_2015.3.187/compiler/lib/mic
```

To connect to the accelerator run:

```
$ ssh mic0
```

If the code is sequential, it can be executed directly:

```
mic0 $ ~/path_to_binary/vect-add-seq-mic
```

If the code is parallelized using OpenMP a set of additional libraries is required for execution. To locate these libraries new path has to be added to the LD_LIBRARY_PATH environment variable prior to the execution:

```
mic0 $ export LD_LIBRARY_PATH=/apps/all/icc/2015.3.187-GNU-5.1.0-2.25/composer_xe_2015.3.187/
```

Please note that the path exported in the previous example contains path to a specific compiler (here the version is 2015.3.187-GNU-5.1.0-2.25). This version number has to match with the version number of the Intel compiler module that was used to compile the code on the host computer.

For your information the list of libraries and their location required for execution of an OpenMP parallel code on Intel Xeon Phi is:

/apps/all/icc/2015.3.187-GNU-5.1.0-2.25/composer_xe_2015.3.187/compiler/lib/mic

libiomp5.so libimf.so libsvml.so libirng.so libintlc.so.5

Finally, to run the compiled code use:

```
$ ~/path_to_binary/vect-add-mic
```

OpenCL

OpenCL (Open Computing Language) is an open standard for general-purpose parallel programming for diverse mix of multi-core CPUs, GPU coprocessors, and other parallel processors. OpenCL provides a flexible execution model and uniform programming environment for software developers to write portable code for systems running on both the CPU and graphics processors or accelerators like the Intel® Xeon Phi.

On Anselm OpenCL is installed only on compute nodes with MIC accelerator, therefore OpenCL code can be compiled only on these nodes.

```
module load opencl-sdk opencl-rt
```

Always load "opencl-sdk" (providing devel files like headers) and "opencl-rt" (providing dynamic library libOpenCL.so) modules to compile and link OpenCL code. Load "opencl-rt" for running your compiled code.

There are two basic examples of OpenCL code in the following directory:

```
/apps/intel/opencl-examples/
```

First example "CapsBasic" detects OpenCL compatible hardware, here CPU and MIC, and prints basic information about the capabilities of it.

```
/apps/intel/opencl-examples/CapsBasic/capsbasic
```

To compile and run the example copy it to your home directory, get a PBS interactive session on of the nodes with MIC and run make for compilation. Make files are very basic and shows how the OpenCL code can be compiled on Anselm.

```
$ cp /apps/intel/opencl-examples/CapsBasic/* .
$ qsub -I -q qprod -l select=1:ncpus=24:accelerator=True:naccelerators=2:accelerator_model=phi
$ make
```

The compilation command for this example is:

```
$ g++ capsbasic.cpp -lOpenCL -o capsbasic -I/apps/intel/opencl/include/
```

After executing the complied binary file, following output should be displayed.

```
./capsbasic

Number of available platforms: 1
Platform names:
    [0] Intel(R) OpenCL [Selected]
Number of devices available for each type:
    CL_DEVICE_TYPE_CPU: 1
    CL_DEVICE_TYPE_GPU: 0
    CL_DEVICE_TYPE_ACCELERATOR: 1

** Detailed information for each device ***

CL_DEVICE_TYPE_CPU[0]
    CL_DEVICE_NAME:        Intel(R) Xeon(R) CPU E5-2470 0 @ 2.30GHz
    CL_DEVICE_AVAILABLE: 1

...

CL_DEVICE_TYPE_ACCELERATOR[0]
    CL_DEVICE_NAME: Intel(R) Many Integrated Core Acceleration Card
    CL_DEVICE_AVAILABLE: 1

...
```

More information about this example can be found on Intel website: http://software.intel.com/en-us/vcsource/samples/caps-basic/

The second example that can be found in "/apps/intel/opencl-examples" >directory is General Matrix Multiply. You can follow the the same procedure to download the example to your directory and compile it.

```
$ cp -r /apps/intel/opencl-examples/* .
$ qsub -I -q qprod -l select=1:ncpus=24:accelerator=True:naccelerators=2:accelerator_model=phi
$ cd GEMM
$ make
```

The compilation command for this example is:

```
$ g++ cmdoptions.cpp gemm.cpp ../common/basic.cpp ../common/cmdparser.cpp ../common/oclobject.
```

To see the performance of Intel Xeon Phi performing the DGEMM run the example as follows:

```
./gemm -d 1
Platforms (1):
 [0] Intel(R) OpenCL [Selected]
Devices (2):
 [0] Intel(R) Xeon(R) CPU E5-2470 0 @ 2.30GHz
 [1] Intel(R) Many Integrated Core Acceleration Card [Selected]
Build program options: "-DT=float -DTILE_SIZE_M=1 -DTILE_GROUP_M=16 -DTILE_SIZE_N=128 -DTILE_G
Running gemm_nn kernel with matrix size: 3968x3968
Memory row stride to ensure necessary alignment: 15872 bytes
Size of memory region for one matrix: 62980096 bytes
Using alpha = 0.57599 and beta = 0.872412
...
Host time: 0.292953 sec.
Host perf: 426.635 GFLOPS
Host time: 0.293334 sec.
Host perf: 426.081 GFLOPS
...
```

Please note: GNU compiler is used to compile the OpenCL codes for Intel MIC. You do not need to load Intel compiler module.

MPI

Environment setup and compilation

To achieve best MPI performance always use following setup for Intel MPI on Xeon Phi accelerated nodes:

```
$ export I_MPI_FABRICS=shm:dapl
$ export I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u,ofa-v2-scif0,ofa-v2-mcm-1
```

This ensures, that MPI inside node will use SHMEM communication, between HOST and Phi the IB SCIF will be used and between different nodes or Phi's on diferent nodes a CCL-Direct proxy will be used.

Please note: Other FABRICS like tcp,ofa may be used (even combined with shm) but there's severe loss of performance (by order of magnitude). Usage of

single DAPL PROVIDER (e. g. I_MPI_DAPL_PROVIDER=ofa-v2-mlx4_0-1u) will cause failure of Host<->Phi and/or Phi<->Phi communication. Usage of the I_MPI_DAPL_PROVIDER_LIST on non-accelerated node will cause failure of any MPI communication, since those nodes don't have SCIF device and there's no CCL-Direct proxy runnig.

Again an MPI code for Intel Xeon Phi has to be compiled on a compute node with accelerator and MPSS software stack installed. To get to a compute node with accelerator use:

```
$ qsub -I -q qprod -l select=1:ncpus=24:accelerator=True:naccelerators=2:accelerator_model=phi
```

The only supported implementation of MPI standard for Intel Xeon Phi is Intel MPI. To setup a fully functional development environment a combination of Intel compiler and Intel MPI has to be used. On a host load following modules before compilation:

```
$ module load intel impi
```

To compile an MPI code for host use:

```
$ mpiicc -xhost -o mpi-test mpi-test.c
```

To compile the same code for Intel Xeon Phi architecture use:

```
$ mpiicc -mmic -o mpi-test-mic mpi-test.c
```

Or, if you are using Fortran :

```
$ mpiifort -mmic -o mpi-test-mic mpi-test.f90
```

An example of basic MPI version of "hello-world" example in C language, that can be executed on both host and Xeon Phi is (can be directly copy and pasted to a .c file)

‘ #include #include

int main (argc, argv) int argc; char *argv[]; { int rank, size;

int len; char node[MPI_MAX_PROCESSOR_NAME];

MPI_Init (&argc, &argv); /* starts MPI / *MPI_Comm_rank (MPI_COMM_WORLD, &rank);* / get current process id / *MPI_Comm_size (MPI_COMM_WORLD, &size);* / get number of processes */

MPI_Get_processor_name(node,&len);

printf("Hello world from process %d of %d on host %s n", rank, size, node); MPI_Finalize(); return 0; } ‘

MPI programming models

Intel MPI for the Xeon Phi coprocessors offers different MPI programming models:

Host-only model** - all MPI ranks reside on the host. The coprocessors can be used by using offload pragmas. (Using MPI calls inside offloaded code is not supported.)**

Coprocessor-only model** - all MPI ranks reside only on the coprocessors.

Symmetric model** - the MPI ranks reside on both the host and the coprocessor. Most general MPI case.

Host-only model

In this case all environment variables are set by modules, so to execute the compiled MPI program on a single node, use:

```
$ mpirun -np 4 ./mpi-test
```

The output should be similar to:

```
Hello world from process 1 of 4 on host r38u31n1000
Hello world from process 3 of 4 on host r38u31n1000
Hello world from process 2 of 4 on host r38u31n1000
Hello world from process 0 of 4 on host r38u31n1000
```

Coprocessor-only model

There are two ways how to execute an MPI code on a single coprocessor: 1.) lunch the program using "**mpirun**" from the coprocessor; or 2.) lunch the task using "**mpiexec.hydra**" from a host.

Execution on coprocessor**

Similarly to execution of OpenMP programs in native mode, since the environmental module are not supported on MIC, user has to setup paths to Intel MPI libraries and binaries manually. One time setup can be done by creating a "**.profile**" file in user's home directory. This file sets up the environment on the MIC automatically once user access to the accelerator through the SSH.

At first get the LD_LIBRARY_PATH for currenty used Intel Compiler and Intel MPI:

```
$ echo $MIC_LD_LIBRARY_PATH
/apps/all/imkl/11.2.3.187-iimpi-7.3.5-GNU-5.1.0-2.25/mkl/lib/mic:/apps/all/imkl/11.2.3.187-i
```

Use it in your ~/.profile:

```
$ vim ~/.profile

PS1='[u@h W]$ '
export PATH=/usr/bin:/usr/sbin:/bin:/sbin

#IMPI
export PATH=/apps/all/impi/5.0.3.048-iccifort-2015.3.187-GNU-5.1.0-2.25/mic/bin/:$PATH

#OpenMP (ICC, IFORT), IMKL and IMPI
export LD_LIBRARY_PATH=/apps/all/imkl/11.2.3.187-iimpi-7.3.5-GNU-5.1.0-2.25/mkl/lib/mic:/app
```

Please note: - this file sets up both environmental variable for both MPI and OpenMP libraries. - this file sets up the paths to a particular version of Intel MPI library and particular version of an Intel compiler. These versions have to match with loaded modules.

To access a MIC accelerator located on a node that user is currently connected to, use:

```
$ ssh mic0
```

or in case you need specify a MIC accelerator on a particular node, use:

```
$ ssh r38u31n1000-mic0
```

To run the MPI code in parallel on multiple core of the accelerator, use:

```
$ mpirun -np 4 ./mpi-test-mic
```

The output should be similar to:

```
Hello world from process 1 of 4 on host r38u31n1000-mic0
Hello world from process 2 of 4 on host r38u31n1000-mic0
Hello world from process 3 of 4 on host r38u31n1000-mic0
Hello world from process 0 of 4 on host r38u31n1000-mic0
```

**

Execution on host

If the MPI program is launched from host instead of the coprocessor, the environmental variables are not set using the ".profile" file. Therefore user has to specify library paths from the command line when calling "mpiexec".

First step is to tell mpiexec that the MPI should be executed on a local accelerator by setting up the environmental variable "I_MPI_MIC"

```
$ export I_MPI_MIC=1
```

Now the MPI program can be executed as:

```
$ mpirun -genv LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH -host mic0 -n 4 ~/mpi-test-mic
```

or using mpirun

```
$ mpirun -genv LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH -host mic0 -n 4 ~/mpi-test-mic
```

Please note: - the full path to the binary has to specified (here: "**>~/mpi-test-mic**") - the LD_LIBRARY_PATH has to match with Intel MPI module used to compile the MPI code

The output should be again similar to:

```
Hello world from process 1 of 4 on host r38u31n1000-mic0
Hello world from process 2 of 4 on host r38u31n1000-mic0
Hello world from process 3 of 4 on host r38u31n1000-mic0
Hello world from process 0 of 4 on host r38u31n1000-mic0
```

Please note that the "mpiexec.hydra" requires a file "**>pmi_proxy**" from Intel MPI library to be copied to the MIC filesystem. If the file is missing please contact the system administrators. A simple test to see if the file is present is to execute:

```
$ ssh mic0 ls /bin/pmi_proxy
/bin/pmi_proxy
```

**

Execution on host - MPI processes distributed over multiple accelerators on multiple nodes

To get access to multiple nodes with MIC accelerator, user has to use PBS to allocate the resources. To start interactive session, that allocates 2 compute nodes = 2 MIC accelerators run qsub command with following parameters:

```
$ qsub -I -q qprod -l select=2:ncpus=24:accelerator=True:naccelerators=2:accelerator_model=phi

$ module load intel impi
```

This command connects user through ssh to one of the nodes immediately. To see the other nodes that have been allocated use:

```
$ cat $PBS_NODEFILE
```

For example:

```
r38u31n1000.bullx
r38u32n1001.bullx
```

This output means that the PBS allocated nodes r38u31n1000 and r38u32n1001, which means that user has direct access to "**r38u31n1000-mic0**" and "**>r38u32n1001-mic0**" accelerators.

Please note: At this point user can connect to any of the allocated nodes or any of the allocated MIC accelerators using ssh: - to connect to the second node : ** $ ssh >r38u32n1001 **- to connect to the accelerator on the first node from the first node:** $ ssh r38u31n1000-mic0** or **$ ssh mic0** -**

to connect to the accelerator on the second node from the first node: **$ ssh r38u32n1001-mic0**

At this point we expect that correct modules are loaded and binary is compiled. For parallel execution the mpiexec.hydra is used. Again the first step is to tell mpiexec that the MPI can be executed on MIC accelerators by setting up the environmental variable "I_MPI_MIC", don't forget to have correct FABRIC and PROVIDER defined.

```
$ export I_MPI_MIC=1
$ export I_MPI_FABRICS=shm:dapl
$ export I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u,ofa-v2-scif0,ofa-v2-mcm-1
```

The launch the MPI program use:

```
$ mpirun -genv LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH
 -host r38u31n1000-mic0 -n 4 ~/mpi-test-mic
: -host r38u32n1001-mic0 -n 6 ~/mpi-test-mic
```

or using mpirun:

```
$ mpirun -genv LD_LIBRARY_PATH
 -host r38u31n1000-mic0 -n 4 ~/mpi-test-mic
: -host r38u32n1001-mic0 -n 6 ~/mpi-test-mic
```

In this case four MPI processes are executed on accelerator r38u31n1000-mic and six processes are executed on accelerator r38u32n1001-mic0. The sample output (sorted after execution) is:

```
Hello world from process 0 of 10 on host r38u31n1000-mic0
Hello world from process 1 of 10 on host r38u31n1000-mic0
Hello world from process 2 of 10 on host r38u31n1000-mic0
Hello world from process 3 of 10 on host r38u31n1000-mic0
Hello world from process 4 of 10 on host r38u32n1001-mic0
Hello world from process 5 of 10 on host r38u32n1001-mic0
Hello world from process 6 of 10 on host r38u32n1001-mic0
Hello world from process 7 of 10 on host r38u32n1001-mic0
Hello world from process 8 of 10 on host r38u32n1001-mic0
Hello world from process 9 of 10 on host r38u32n1001-mic0
```

The same way MPI program can be executed on multiple hosts:

```
$ mpirun -genv LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH
 -host r38u31n1000 -n 4 ~/mpi-test
: -host r38u32n1001 -n 6 ~/mpi-test
```

Symmetric model

In a symmetric mode MPI programs are executed on both host computer(s) and MIC accelerator(s). Since MIC has a different architecture and requires

different binary file produced by the Intel compiler two different files has to be compiled before MPI program is executed.

In the previous section we have compiled two binary files, one for hosts "**mpi-test**" and one for MIC accelerators "**mpi-test-mic**". These two binaries can be executed at once using mpiexec.hydra:

```
$ mpirun
 -genv $MIC_LD_LIBRARY_PATH
 -host r38u32n1001 -n 2 ~/mpi-test
: -host r38u32n1001-mic0 -n 2 ~/mpi-test-mic
```

In this example the first two parameters (line 2 and 3) sets up required environment variables for execution. The third line specifies binary that is executed on host (here r38u32n1001) and the last line specifies the binary that is execute on the accelerator (here r38u32n1001-mic0).

The output of the program is:

```
Hello world from process 0 of 4 on host r38u32n1001
Hello world from process 1 of 4 on host r38u32n1001
Hello world from process 2 of 4 on host r38u32n1001-mic0
Hello world from process 3 of 4 on host r38u32n1001-mic0
```

The execution procedure can be simplified by using the mpirun command with the machine file a a parameter. Machine file contains list of all nodes and accelerators that should used to execute MPI processes.

An example of a machine file that uses 2 >hosts (r38u32n1001 and r38u33n1002) and 2 accelerators **(r38u32n1001-mic0** and r38u33n1002-mic0**) to run 2 MPI processes on each of them:

```
$ cat hosts_file_mix
r38u32n1001:2
r38u32n1001-mic0:2
r38u33n1002:2
r38u33n1002-mic0:2
```

In addition if a naming convention is set in a way that the name of the binary for host is **"bin_name"** and the name of the binary for the accelerator is **"bin_name-mic"** then by setting up the environment variable **I_MPI_MIC_POSTFIX** to **"-mic"** user do not have to specify the names of booth binaries. In this case mpirun needs just the name of the host binary file (i.e. "mpi-test") and uses the suffix to get a name of the binary for accelerator (i..e. "mpi-test-mic").

```
$ export I_MPI_MIC_POSTFIX=-mic
```

>To run the MPI code using mpirun and the machine file "hosts_file_mix" use:

```
$ mpirun
```

```
-genv LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH
-machinefile hosts_file_mix
~/mpi-test
```

A possible output of the MPI "hello-world" example executed on two hosts and two accelerators is:

```
Hello world from process 0 of 8 on host r38u31n1000
Hello world from process 1 of 8 on host r38u31n1000
Hello world from process 2 of 8 on host r38u31n1000-mic0
Hello world from process 3 of 8 on host r38u31n1000-mic0
Hello world from process 4 of 8 on host r38u32n1001
Hello world from process 5 of 8 on host r38u32n1001
Hello world from process 6 of 8 on host r38u32n1001-mic0
Hello world from process 7 of 8 on host r38u32n1001-mic0
```

Using the PBS automatically generated node-files

PBS also generates a set of node-files that can be used instead of manually creating a new one every time. Three node-files are genereated:

Host only node-file: - /lscratch/$PBS_JOBID/nodefile-cnMIConlynode-file : ă - /lscratch/{PBS_JOBID}/nodefile-mic Host and MIC node-file: - /lscratch/${PBS_JOBID}/nodefile-mix

Please note each host or accelerator is listed only per files. User has to specify how many jobs should be executed per node using "-n" parameter of the mpirun command.

Optimization

For more details about optimization techniques please read Intel document Optimization and Performance Tuning for Intel® Xeon Phi™ Coprocessors