Quick Start Guide for Symmetric Mode Execution

Arturo Argueta, Roberto Camacho Barranco, Esthela Gallardo, and Pat Teller
UTEP Stampede Technology Insertion Project
Leonardo Fialho and Jim Browne
UT-Austin Stampede Technology Insertion Project

Xeon Phis can execute a program concurrently with node processors. For example, in the case of a Stampede node, 2 Sandy Bridge (SB) CPUs (Intel 8-core E5 processors) and 1 or 2 Xeon Phis can concurrently execute a parallel program. The program can be an MPI program or a hybrid program, e.g., one that employs both MPI and OpenMP. In the former case, MPI ranks are run on the cores of both the SB CPUs and Phis. In the latter case, the SB CPUs and Phis communicate via MPI, while OpenMP is employed on the SB CPUs and/or Phis to execute specified parts of the program on the cores of each.

The following four steps demonstrate how to set up the execution environment and execute a hybrid program that employs both MPI and OpenMP on 2 SB CPUs and 1 Phi of a Stampede node:

1. Set up the Intel MPI environment, i.e., `impi`, by executing the following command on the host CPU:

```
$ module swap mvapich2 impi
```

2. Assuming the program (example.c) is written in the C programming language, create object (binary) files for the SB CPUs and Phi by executing the following two commands on the host processor:

```
$ mpicc -openmp example.c -mmic -o example.mic
$ mpicc -openmp example.c       -o example.cpu
```

3. Create the following `impi` environment variables by executing the following commands on the host CPU:

```
$ export I_MPI_FABRICS=shm:dapl
$ export I_MPI_DAPL_PROVIDER="ofa-v2-mlx4_0-1u"
$ export I_MPI_MIC=1
$ export I_MPI_OFA_ADAPTER_NAME=mlx4_0
```

   For more details about `impi` environment variables see:

   - *http://software.intel.com/sites/products/documentation/hpc/ics/impi/41/lin/Reference_Manual/index.htm - Communication_Fabrics_Control.htm*
   - *http://software.intel.com/sites/products/documentation/hpc/ics/impi/41/lin/Reference_Manual/Environment_Variables_MIC.htm*

- *http://software.intel.com/sites/products/documentation/hpc/ics/impi/41/lin/Reference_Manual/OFA-capable_Network_Fabrics_Control.htm*

4. Execute the program on the 2 SB CPUs and Xeon Phi using a script similar to the one below:

```
mpiexec.hydra \
-n 16 -host localhost ./example.cpu : \
-env OMP_NUM_THREADS 30 \
-env LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH \
-env I_MPI_PIN_MODE mpd \
-n 8 -host mic0 ./example.mic
```

   Using this script, the program will be executed on 16 MPI nodes (one per core) of the 2 SB CPUs and 8 MPI nodes (8 cores) of the Phi that will use 30 OpenMP threads in case there is a OpenMP parallel region.

Sample Code

Rather than provide a sample code, we provide pointers to codes that have been executed in symmetric mode on nodes that include Xeon Phis. An MPI program, e.g., see *https://software.intel.com/en-us/articles/using-the-intel-mpi-library-on-intel-xeon-phi-coprocessor-systems#compiling*, can be run on the cores of both the SB CPUs and Phis of Stampede nodes. Alternatively, a hybrid code, e.g., see *https://software.intel.com/en-us/articles/wrf-conus25km-on-intel-xeon-phi-coprocessors-and-intel-xeon-processors-in-symmetric-mode*, which employs both MPI and OpenMP, can be executed on Stampede nodes.

Performance Gain

The execution time of a program or a section of code can be measured using `gettimeofday()`. Its usage is shown in the *Quick Start Guide for Offload Mode Execution* [URL]. Using `gettimeofday()`in the way indicated allows you to compare the execution times of programs executed in different execution modes.

Note: Even if it has been determined that Symmetric Mode may be the best option for the code at hand, load balancing to attain good performance scaling can still be challenging.

Data Transfer Costs

The cost of data transfers via MPI between the SB CPUs and Phis varies depending on the MPI library used, the directives employed, the values of the parameters selected, and, of course, the amount of data moved. Thus, we leave the quantification of data transfer costs to the user, but provide guidance on how to measure them.

Assuming that a program is executed on 1 SB CPU and 1 Phi of a Stampede node, the code below demonstrates how to obtain the number of bytes moved between the SB CPU and the Phi and the execution-time cost of the data transfers. If you are interested in the cost per byte transferred in the two directions (which is different), divide the latter by the former.

As can be seen by inspecting the code, the SB CPU and Phi each execute one `MPI_Send` to send the array `c` to the other. `MPI_Wtime()` is used to record the current time before each `MPI_Send` and after the corresponding `MPI_Recv`. Accordingly, the message passing time, i.e., the execution-time cost of a data transfer, in each direction is calculated by subtracting the former from the latter. In each case, the number of bytes transmitted is the size of the `int arraySize` variable, which is printed out by the program along with the execution-time costs of the data transfers in both directions.

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

int main(int argc, char *argv[]) {
    int rank, size;
    int i;
    int arraySize = 500000000;  // Max Value = 500000000
    int c[arraySize];
    int buffer[arraySize];

    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2) {
        printf("Please run with 2 processes.\n");
        fflush (stdout);
        MPI_Finalize();
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        double start = 0;
        double end = 0;

        for (i = 0; i < arraySize; i++)
            c[i] = i;

        // Start timing CPU-to-Phi communication.
        start = MPI_Wtime();
        MPI_Send(c, arraySize, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(buffer, arraySize, MPI_INT, 1, 1,
MPI_COMM_WORLD, &status);
        // End timing CPU-to-Phi communication.
        end = MPI_Wtime();
```

```
        double computationTime = end - start;

#ifdef __MIC__
        printf("Roundtrip from MIC: %f
seconds\n",computationTime);
#else
        printf("Roundtrip from CPU: %f seconds\n",
computationTime);
#endif
    }

    if (rank == 1) {
        double start = 0;
        double end = 0;

        // Start timing Phi-to-CPU communication.
        start = MPI_Wtime();
        MPI_Recv(buffer, arraySize, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);
        MPI_Send(buffer, arraySize, MPI_INT, 0, 1,
MPI_COMM_WORLD);
        // End timing Phi-to-CPU communication.
        end = MPI_Wtime();

        double computationTime = end - start;

        // Report Phi-to-SB communication cost.
#ifdef __MIC__
        printf("Roundtrip from MIC: %f
seconds\n",computationTime);
#else
        // Report SB-to-Phi communication cost.
        printf("Roundtrip from CPU: %f seconds\n",
computationTime);
#endif

        fflush (stdout);
    }

    MPI_Finalize();

    // Report size of data transmitted.
    int sizes = sizeof(c);
    printf("The size of the array is: %d \n", sizes);
    return 0;
}
```