Multi-/Many-Core Programming with Intel® Xeon Phi™ Coprocessors

Fundamentals of Parallel Programming using Intel´s Many Integrated Core (MIC) Architecture

Intel Xeon Phi coprocessor Workshop
Lab Workbook

may 2015

Heterogeneous Computing Working Group - Center for Scientific Computing

São Paulo State University (UNESP) - São Paulo, Brazil

Organized by: Rogério Iope

Engineering Team: Allan Szu, Beraldo Leal, Eduardo Bach, Gabriel Winckler, Márcio Costa, Raphael Cóbe, Silvio Luiz Stanzani

Overview

The Intel Xeon Phi Coprocessor, the first product of Intel's Many Integrated Core (MIC) Architecture, is a new accelerator technology developed by Intel to enable performance gains for highly parallel computing workloads. It possesses several interesting and appealing features, including the ability to use familiar programming models such as OpenMP and MPI. This hands-on training session is a comprehensive, practical introduction to the Xeon Phi architecture and programming models, aiming to demonstrate the processing power of the Intel Xeon Phi product family.

Participants will have access to a heterogeneous computing system equipped with Intel Xeon processors and Intel Xeon Phi coprocessors, as well as Intel software development tools. The computing system, a state-of-the-art server with two Intel Xeon processors (16 cores, 2 threads/core) and three Intel Xeon Phi coprocessors (171 cores, 4 threads/core), is hosted at the Center for Scientific Computing of the São Paulo State University (UNESP), in Brazil. The step-by-step hands-on activities have been planned to provide easy to follow instructions in order to allow the participants to have a real - though very introductory - experience on using a powerful 716-thread manycore system.

Learning Goals

Attendants of these hands-on labs will start issuing simple command-line tools to get basic information about the Intel Xeon Phi coprocessors, then will learn how to monitor what resources are being used and access their operating systems by establishing ssh sessions with them. Trainees will thus verify that the Intel Xeon Phi coprocessor is an IP-addressable PCIe device - managed by an independent environment provided by the MIC Platform Software Stack (MPSS) - that runs the Linux operating system.

Following the introductory part, participants will learn how to compile and run simple C/C++ applications directly into the coprocessors, and then compile and run example codes based on shared-memory parallelism with OpenMP and Cilk Plus and distributed-memory parallelism with MPI. They will also work on MPI application examples that should be executed simultaneously on the Xeon processors and the Xeon Phi coprocessors, explore the use of Intel Math Kernel Library, and develop insights on tuning parallel applications.

Useful References

- Intel Xeon Phi Coprocessor High-Performance Programming, by Jim Jeffers and James Reinders (Elsevier, 2013)
  http://www.lotsofcores.com/

- Intel Xeon Phi Coprocessor Architecture and Tools, by Rezaur Rahman (Apress, 2013)
  http://www.apress.com/9781430259268

- Parallel Programming and Optimization with Intel Xeon Phi Coprocessors (Colfax, 2013)
  http://www.colfax-intl.com/nd/xeonphi/book.aspx

- An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors
  http://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors.pdf

- Intel Xeon Phi Coprocessor Developer´s Quick Start Guide
  http://software.intel.com/sites/default/files/article/335818/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf

- Intel Xeon Phi Coprocessor: System Software Developers Guide
  http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html

- Intel C++ Compiler XE 13.1 User and Reference Guide (Linux OS)
  http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/

- Tutorial on OpenMP - Lawrence Livermore National Laboratory
  https://computing.llnl.gov/tutorials/openMP/

- Tutorial on MPI - Lawrence Livermore National Laboratory
  https://computing.llnl.gov/tutorials/mpi/

Remote access to the testing platform

This document assumes that the testing platform has been setup and is ready to use. We will be using a state-of-the-art server - loaned by Intel - with two Intel Xeon processors (16 cores, 2 threads/core) and three Intel Xeon Phi coprocessors (171 cores, 4 threads/core), as well as several Intel software development tools. To simplify nomenclature, we will refer to the testing platform as "the host" (or "the host system"), and the Xeon Phi coprocessors installed in the host system as "mic0", "mic1", and "mic2".

Participants should work alone or in pairs on a workstation - preferably running Linux or Mac - with Internet access. All the exercises are command-line based and should be executed on the host system by means of a secure shell (SSH) connection. Ideally the participant workstation should be able to run X11.

Use the syntax below to log in to the host system.

```
$ ssh –X phi01.ncc.unesp.br –l traineeN    (N is a number assigned to each participant)
```

Note: Before starting any lab exercises execute the commands below (this must be done each time a new terminal window is opened):

```
[phi01]$ export PATH=$PATH:/opt/intel/mic/bin
[phi01]$ source /opt/intel/composerxe/bin/compilervars.sh intel64
[phi01]$ source /opt/intel/impi/4.1.0/bin64/mpivars.sh
```

Also make sure that the Xeon Phi coprocessors are up and running (status = 'online'):

```
[phi01]$ micctrl --status    (show the status of all MIC cards installed on the host system)
```

Please refer to the teaching assistant(s) if you have any question.

Hands-on : "Intel MPI programming models"

2.1 Goals

This set of exercises will show you practical aspects of heterogeneous execution of parallel application in distributed memory with MPI, widely recognized as the de facto standard for parallel programming.

2.2 Overview of MPI

The Message Passing Interface (MPI) has been the de facto standard for parallel programming for nearly two decades. It defines a set of library routines that can be called from C and Fortran programs, and includes a communication protocol that allows multiple processes, which do not share common memory, to perform parallel calculations, communicating with each other by passing messages through a communication channel. MPI messages are arrays of predefined and user-defined data types. The purpose of MPI messages range from task scheduling to exchanging large amounts of data necessary to perform calculations. MPI guarantees that the order of sent messages is preserved on the receiver side.

In the old era of single-core compute nodes, the dominant MPI usage model in clusters of computers was to run one MPI process per physical machine. Nowadays, with the advent of multi-core, multi-socket, and now heterogeneous many-core systems, the range of usage models of MPI has grown. It is possible, for example, to run one MPI process per compute node, exploiting parallelism in each machine by means of a shared-memory parallel framework, such as OpenMP or Intel Cilk Plus. Alternatively, one single-threaded MPI process can run on each physical or logical core of each machine in a cluster of servers. In this case, MPI processes running on the same compute node still do not share memory address space. However, message passing between these processes is more efficient, because fast virtual fabrics can be used for communication. Another possible option is to run multiple multi-threaded MPI processes per compute node. In this case, each process exploits parallelism in shared memory, and MPI communication between processes adds distributed-memory parallelism. This hybrid approach may yield optimum performance for with a high frequency or large volume of communication.

MPI programs typically employ a single-program, multiple-data (SPMD) approach. Multiple instances, or MPI ranks, of the same program run concurrently on different data. Each rank computes a different part of a larger problem and uses MPI calls to communicate data between ranks. From the programmer's perspective, ranks may run either on the same node or in different nodes; the communication path may be different but that is transparent to the MPI program.

As we have seen in Session 1, Intel Xeon Phi coprocessors run its own operating system, independent of the host processor, and have their own memory domain. In other words, a Xeon Phi coprocessor behaves like an independent compute node. From this perspective, MPI is a natural fit. In principle, any existing MPI program can be run on a mixture of hosts and coprocessors without source code modification.

In heterogeneous clusters with Intel Xeon processors and Xeon Phi coprocessors, MPI programmers have a choice of running MPI processes on hosts and coprocessors natively, or running MPI processes only on hosts and performing offload to coprocessors.

2.3 Hands-on Activities

2.3.1 As we have seen in Session 1, before compiling and running MPI applications, the appropriate environment variables should be set by calling a shell script included in Intel MPI - it is important to set up the environment variables in such a way that the compilers generate code appropriate to the host system architecture. In the host system console, run the script below which will export the proper environment variables for the compilers, the Intel MKL (Math Kernel Library) and Intel TBB (Threading Building Blocks):

```
[phi01]$ source /opt/intel/composerxe/bin/compilervars.sh intel64
```

Remember that MPI applications must be compiled with special wrapper applications – ‘mpiicc’ for C and ‘mpiicpc’ for C++. In order to launch the resulting executable as a parallel MPI application, it should be run using the ‘mpirun’ script.

2.3.2 Intel MPI distribution includes a test directory, which contains a simple MPI program coded in C, C++, or Fortran. In directory ‘/home/traineeN/source-files’ you will find a copy of the source file ‘test.c’ from the Intel MPI distribution. Let us start working on this code as a quick remind on how to run an MPI program on the Intel Xeon Phi coprocessor. Compile the source file with the Intel compiler for the host with the usual Intel MPI wrapper:

```
[phi01]$ mpiicc -o test test.c
```

Now compile the source file for Intel Xeon Phi coprocessor using the "-mmic" compiler flag. Because of this flag the Intel MPI script will provide the Intel MPI libraries for Intel Xeon Phi coprocessor to the linker (add the verbose flag "-v" to see it):

```
[phi01]$ mpiicc -mmic -o test.mic test.c
[phi01]$ scp test.mic mic0:~/
```

The ".mic" suffix is added to distinguish the coprocessor binary from the host one (it could be any suffix). As a starter run the Xeon binary with 4 MPI processes alone:

```
[phi01]$ mpirun -n 4 ./test
```

Copy binary file 'test.mic' to the coprocessors and run the Intel MPI test program on one of the three Xeon Phi cards in coprocessor-only mode:

```
[phi01]$ mpirun -host mic0 –n 4 ./test.mic
```

An alternative would be to login onto the coprocessor and run the test from there in a straightforward manner. Try it if you like.

Pulling it together: you can run the test code on both the host processor and on one of the Xeon Phi coprocessor as one MPI program in symmetric mode by defining each argument set independently (with command line sections separated by a colon):

```
[phi01]$ mpirun -host localhost -n 4 ./test : -host mic0 -n 8 ~/test.mic
```

Notice that in the symmetric mode you must provide the '-host' flag for the MPI processes running on the Xeon host!

2.3.3 As a preparation for the next exercises on hybrid programming, the mapping/pinning of Intel MPI processes will be investigated step by step. Set the environment variable I_MPI_DEBUG equal or larger than 4 to see the mapping information:

```
[phi01]$ export I_MPI_DEBUG=4
```

For pure (non-hybrid) MPI programs the environment variable I_MPI_PIN_PROCESSOR_LIST controls the mapping/pinning. For hybrid codes the variable I_MPI_PIN_DOMAIN takes precedence. It splits the (logical) processors into non-overlapping domains for which this rule applies: "one MPI process for one domain".

Repeat the Intel MPI test from before with I_MPI_DEBUG set. Because of the amount of output use the flag "-prepend-rank", which puts the MPI rank number in front of each output line:

```
[phi01]$ mpirun -prepend-rank -n 4 ./test
[phi01]$ mpirun -prepend-rank -host mic0 -n 4 ~/test.mic
[phi01]$ mpirun -prepend-rank -host localhost -n 4 ./test : -host mic0 -n 8 ~/test.mic
```

Sorting of the output can be beneficial for the mapping analysis, although this changes the order of the output. Try adding "2>&1 | sort" to sort the output if you like.

Now set the variable I_MPI_PIN_DOMAIN with the '-env' flag. Possible values are 'auto', 'omp' (which relies on the OMP_NUM_THREADS variable), or a fixed number of logical cores. We have learned before (session 1, exercise 2.2.4) that by exporting I_MPI_PIN_DOMAIN in the host's command shell, the variable is identically exported to the host and to the Xeon Phi coprocessors. Typically this is not beneficial and an architecture adapted setting using '-env' is recommended:

```
[phi01]$ mpirun -prepend-rank -env I_MPI_PIN_DOMAIN auto -n 4 ./test
[phi01]$ mpirun -prepend-rank -env I_MPI_PIN_DOMAIN auto -host mic0 -n 4
./test.mic
[phi01]$ mpirun -prepend-rank -env I_MPI_PIN_DOMAIN 4 -host localhost -n 2 ./test :
-env I_MPI_PIN_DOMAIN 12 -host mic0 -n 4 ~/test.mic
```

Experiment with pure Intel MPI mapping by setting I_MPI_PIN_PROCESSOR_LIST if you like. (See the Intel MPI reference manual for details).

2.3.4 Now we are going to run a hybrid MPI/OpenMP program on the Intel Xeon Phi coprocessor. Have a look at the source code 'test_openmp.c', in which a simple printout from the OpenMP threads was added to the previous Intel MPI test code. You can compare the difference between the two files by means of the diff utility:

```
[phi01]$ diff test.c test_openmp.c
```

Compile with the "-openmp" compiler flag and upload to the Intel Xeon Phi coprocessor as usual:

```
[phi01]$ mpiicc -openmp test_openmp.c -o test_openmp
[phi01]$ mpiicc -openmp -mmic test_openmp.c -o test_openmp.mic
[phi01]$ scp test_openmp.mic mic0:~/
```

Because of the '-openmp' flag, Intel MPI will link the code with the thread-safe version of the Intel MPI library (libmpi_mt.so) by default. Run the Intel MPI tests from before:

```
[phi01]$ unset I_MPI_DEBUG      (to reduce the output for now)
[phi01]$ mpirun -prepend-rank -n 2 ./test_openmp
[phi01]$ mpirun -prepend-rank -host mic0 -n 2 ~/test_openmp.mic
[phi01]$ mpirun -prepend-rank -host localhost -n 2 ./test_openmp : -host mic0 -n 4
~/test_openmp.mic
```

The execution generates a lot of output! The default for the OpenMP library is to assume as many OpenMP threads as there are logical processors. For the next steps, explicit OMP_NUM_THREADS values (different on host and Intel Xeon Phi coprocessor) will be set.

In the following test the default OpenMP affinity is checked. Please notice that the range of logical processors is always defined by the splitting the threads based on the I_MPI_PIN_DOMAIN variable. This time we also use I_MPI_PIN_DOMAIN=omp, see how it depends on the OMP_NUM_THREADS setting:

```
[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY verbose -env
OMP_NUM_THREADS 4 -env I_MPI_PIN_DOMAIN auto -n 2 ./test-openmp 2>&1 | sort
[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY verbose -env
OMP_NUM_THREADS 4 -env I_MPI_PIN_DOMAIN omp -host mic0 -n 2 ~/test-
openmp.MIC 2>&1 | sort
[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY verbose -env
OMP_NUM_THREADS 4 -env I_MPI_PIN_DOMAIN 4 -host localhost -n 2 ./test-openmp :
-env KMP_AFFINITY verbose -env OMP_NUM_THREADS 6 -env I_MPI_PIN_DOMAIN 12 -
```

```
host mic0 -n 4 ~/test-openmp.MIC 2>&1 | sort
```

Remember that it is usually beneficial to avoid splitting of logical cores on Intel Xeon Phi coprocessor between MPI processes; either the number of MPI processes should be chosen so that I_MPI_PIN_DOMAIN=auto creates domains which cover complete cores or the environment variable should be a multiply of 4.

Use "scatter", "compact", or "balanced" (Intel Xeon Phi coprocessor specific) to modify the default OpenMP affinity.

```
[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY
verbose,granularity=thread,scatter -env OMP_NUM_THREADS 4 -env
I_MPI_PIN_DOMAIN auto -n 2 ./test_openmp

[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY
verbose,granularity=thread,compact -env OMP_NUM_THREADS 4 -env
I_MPI_PIN_DOMAIN omp -host mic0 -n 2 ~/test_openmp.mic 2>&1 | sort

[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY
verbose,granularity=thread,compact -env OMP_NUM_THREADS 4 -env
I_MPI_PIN_DOMAIN 4 -host localhost -n 2 ./test_openmp : -env KMP_AFFINITY
verbose,granularity=thread,balanced -env OMP_NUM_THREADS 6 -env
I_MPI_PIN_DOMAIN 12 -host mic0 -n 4 ~/test_openmp.mic 2>&1 | sort
```

Notice that, as well as other options, the OpenMP affinity can be set differently per Intel MPI argument set, i.e. different on the host and on the Intel Xeon Phi coprocessor.

2.3.5 Now we want to run the Intel MPI test program with some offload code on the Xeon Phi coprocessor. Have a look at source file ‘test_offload.c’, in which the simple printout from the OpenMP thread is now offloaded to the coprocessor. Compare the difference between the two files using the diff utility:

```
[phi01]$ diff test.c test_offload.c
```

Compile for the Xeon host with the ‘-openmp’ compiler flag as before. The Intel compiler will automatically recognize the offload pragma and create the binary for it.

```
[phi01]$ mpiicc -openmp test_offload.c -o test_offload
```

(Note that, if necessary, offloading could be switched off with the ‘-no-offload’ flag).

Execute the binary on the host:

```
[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY granularity=thread,scatter -env
OMP_NUM_THREADS 4 -n 2 ./test_offload
```

Now repeat the execution, but 'grep' and 'sort' the output to focus on the essential information:

```
[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY granularity=thread,scatter -env
OMP_NUM_THREADS 4 -n 2 ./test_offload 2>&1 | grep bound | sort
```

All OpenMP threads are mapped onto identical Intel Xeon Phi coprocessor threads! The variable I_MPI_PIN_DOMAIN cannot be used because the domain splitting would be calculated according to the number of logical processors on the Xeon host!

The solution is to specify explicit proclists per MPI process:

```
[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY granularity=thread,proclist=[1-
16:4],explicit -env OMP_NUM_THREADS 4 - n 1 ./test_offload : -env KMP_AFFINITY
granularity=thread,proclist=[17-32:4],explicit -env OMP_NUM_THREADS 4 -n 1
./test_offload
```

Repeat the execution, but 'grep' and 'sort' the output to focus on the essential information:

```
[phi01]$ mpirun -prepend-rank -env KMP_AFFINITY granularity=thread,proclist=[1-
16:4],explicit -env OMP_NUM_THREADS 4 - n 1 ./test_offload : -env KMP_AFFINITY
granularity=thread,proclist=[17-32:4],explicit -env OMP_NUM_THREADS 4 -n 1
./test_offload 2>&1 | grep bound | sort
```