Intel® Xeon Phi™ Coprocessor

Lab Instructions

CUG Lugano May 5th 2014

Intel MPI

Table of Contents

Lab 0 - Prerequisites

Assumptions:

- MPSS is installed: User accounts exist on the card, `ssh/scp` is available, as well as NFS.
- The compiler is installed in directory
  `ICCROOT=/global/opt/intel/composer_xe_2013_sp1.2.144`
- Intel MPI is installed in directory
  `I_MPI_ROOT=/global/opt/intel/impi/4.1.3.049`
- Intel Trace Analyzer and Collector (ITAC) is installed in directory
  `VT_ROOT=/global/opt/intel/itac/8.1.2.033/`

The path of the Intel MPI reference manual is `$I_MPI_ROOT/doc/Reference_Manual.pdf`, use it for details about the environment variables. You get access to ITAC by loading the ITAC module:

```
# module load itac
```

The trace analyzer reference guide is available at `$VT_ROOT/doc/ITA_Reference_Guide.pdf` (for the collector libraries at `$VT_ROOT/doc/ITC_Reference_Guide.pdf`), in Beta without MIC details.

For all runs with MPI ranks on Xeon Phi please set:

export I_MPI_MIC=1

Notice for CUG: The text below might not apply to this event. Environment on Beacon should be OK! Please go directly to section Lab 1 Basics.

Each lab directory contains a SOLUTION.txt file which includes the commands described in this manual (or slightly modified ones) for your convenience. Cut and paste the commands from the output of `` `cat SOLUTION.txt` `` for execution.

Enter the 0_Prerequisites directory where you will find the first SOLUTION.txt file. Please execute the included commands to get familiar with them for the first time. When revisiting the labs later it is sufficient to just source the file (`source 0_Prerequisites/SOLUTION.txt`) for the prerequisites setup.

First check your user environment for the compiler, Intel MPI, and ITAC:

```
# which icc mpiicc traceanalyzer
```

Add the paths in $MIC_LD_LIBRARY_PATH and $VT_ROOT/mic/slib to the LD_LIBRARY_PATH. Otherwise the OpenMP lib or the ITAC collector lib will not be found in the hybrid or ITAC labs, respectively.

```
# export LD_LIBRARY_PATH=
$LD_LIBRARY_PATH:$MIC_LD_LIBRARY_PATH:$VT_ROOT/mic/slib
```

The Intel MPI run command starts processes only on the KNC card when the environment variable I_MPI_MIC is set. It is required for all labs:

```
# export I_MPI_MIC=1
```

The generic short host name for the KNC card is `hostname -s`-mic0, and can be found in /etc/hosts. Throughout this document the short version mic0 is used, in the SOLUTION.txt the variable $MIC0 is defined with the actual value.

The general assumption for this lab is that the host directory /opt/intel is mounted with NFS onto the identical path /opt/intel on the KNC card. Check this, and implicitly ssh access which is required for MPI, it should work without a password:

```
# ssh mic0 'hostname; mount|grep /opt/intel; echo "If this works and
/opt/intel is mounted on `hostname` the prerequisites are done,
continue with the next lab!"'
```

The second NFS related assumption is that the working directory on the host is mounted with NFS onto the identical path on the KNC card. If this is not the case you have to upload the KNC executable with scp explicitly onto the card, e.g. into the user's home directory. And in the mpirun arguments for running MPI processes on the card the remote working directory has to be specified. These or analogue commands have to be added to all labs below:

```
# scp test.MIC mic0:/home/$USER
# mpirun … -wdir /home/$USER –host mic0 …
```

If /opt/intel is not available with NFS, several MPI libraries have to be uploaded (repeat this after each reboot of the KNC card). For the ease of use the target directory is `/lib64/`, consequently no further `LD_LIBRARY_PATH` setting is required. However, you need sudo access rights to upload to that directory. Otherwise you have to copy into a user directory, and set a related `LD_LIBRARY_PATH`.

Upload explicitly the OpenMP and the CilkPlus library from the compiler path:

```
# sudo scp $ICCROOT/compiler/lib/mic/libiomp5.so     mic0:/lib64/

# sudo scp $ICCROOT/compiler/lib/mic/libcilkrts.so.5 mic0:/lib64/
```

Upload the default Intel MPI binaries and libraries onto KNC, plus the libraries used in the debugger lab:

```
# sudo scp $I_MPI_ROOT/mic/lib/libmpi_dbg.so.4.1
mic0:/lib64/libmpi_dbg.so.4

# sudo scp $I_MPI_ROOT/mic/lib/libmpi_dbg_mt.so.4.1
mic0:/lib64/libmpi_dbg_mt.so.4
```

Upload the ITAC library for MIC and source the ITAC environment:

```
# sudo scp $VT_ROOT/mic/slib/libVT.so mic0:/lib64/
```

Lab 1 - Basics

PURE MPI

Each Intel MPI distribution includes a test directory which contains a simple MPI program coded in C, C++, or Fortran.

Enter directory 1_Basics where you will find a copy of the source file test.c from the Intel MPI distribution. You will use this code and two variants for first runs with Intel MPI on MIC.

Compile and link the source file with the Intel compiler for the XEON host with the usual Intel MPI script:

```
# mpiicc -o test test.c
```

And compile and link the source file for MIC using the "-mmic" compiler flag. Because of the flag the Intel MPI script will provide the Intel MPI libraries for MIC to the linker (add the verbose flag "-v" to see it):

```
# mpiicc -mmic -o test.MIC test.c
```

The ".MIC" suffix is added by the user to distinguish the binary from the XEON one. It could be any suffix!

As a starter run the XEON binary with 2 MPI processes alone:

```
# mpirun -n 2 ./test
```

Now run your first Intel MPI program on MIC in coprocessor mode:

```
# micmpiexec -host mic0 -n 2 ./test.MIC
```

micmpiexec is a NICS wrapper script.
An alternative would be to login onto the card and run the test from there in a straightforward manner. Try it if you like (and ask me in case of issues).

Pulling it together you can run the test code on XEON and MIC as one MPI program in symmetric mode. Each argument set (command line sections separated by ":") is defined independently, therefore 4 MPI processes are chosen on the MIC card as an example:

```
# micmpiexec -host `hostname` -n 2 ./test : -host mic0 -n 4 ./test.MIC
```

Please notice: In the symmetric mode you have to provide the "-host" flag also for the MPI processes running on the XEON host!

As an alternative to the previous command you can declare a machinefile with hosts and number of processes per host defined. Use this machinefile together with the environment flag I_MPI_MIC_POSTFIX which value is added to the executable by default on the MIC card:

```
# echo `hostname`:2 > machinefile
# echo mic0:4      >> machinefile

# export I_MPI_MIC_POSTFIX=.MIC
# micmpiexec –machinefile machinefile ./test
```

As preparation for the next lab on hybrid programs, the mapping/pinning of Intel MPI processes will be investigated in the following. Set the environment variable I_MPI_DEBUG equal or larger than 4 to see the mapping information, either by exporting it:

```
# export I_MPI_DEBUG=4
```

Or by adding it as a global environment flag ("-genv") onto the command line close to micmpiexec (without "="):

```
# micmpiexec -genv I_MPI_DEBUG 4 ...
```

For pure MPI programs (non-hybrid) 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 the rule applies "1 MPI process for 1 domain".

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

```
# mpirun -prepend-rank -n 2 ./test

# micmpiexec -prepend-rank -host mic0 -n 2 ./test.MIC

# micmpiexec -prepend-rank -host `hostname` -n 2 ./test : -host mic0 -
n 4 ./test.MIC
```

Sorting of the output can be beneficial for the mapping analysis although this changes the order of the output, add "2>&1 | sort" 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. By exporting I_MPI_PIN_DOMAIN in the shell or using the global "-genv" flag the variable is identically exported to

the host and the MIC card. Typically this is not beneficial and an architecture adapted setting with "`-env`" is recommended:

```
# mpirun -prepend-rank -env I_MPI_PIN_DOMAIN auto -n 2 ./test

# micmpiexec -prepend-rank -env I_MPI_PIN_DOMAIN auto -host mic0 -n 2
./test.MIC

# micmpiexec -prepend-rank -env I_MPI_PIN_DOMAIN 4 -host `hostname` -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).

HYBRID

Now we want to run our first hybrid MPI/OpenMP program on MIC. A simple printout from the OpenMP threads was added to the Intel MPI test code:

```
# diff test.c test-openmp.c
```

Compile and link with the "-openmp" compiler flag. And upload to the MIC card as before:

```
# mpiicc -openmp -o test-openmp test-openmp.c

# mpiicc -openmp -mmic -o test-openmp.MIC test-openmp.c
```

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:

```
# unset I_MPI_DEBUG   # to reduce the output for now

# mpirun -prepend-rank -n 2 ./test-openmp

# micmpiexec -prepend-rank -host mic0 -n 2 ./test-openmp.MIC

# micmpiexec -prepend-rank -host `hostname` -n 2 ./test-openmp : -host
mic0 -n 4 ./test-openmp.MIC
```

Lot of output! The default for the OpenMP library is to assume as many OpenMP threads as there are logical processors. For the next tests, explicit OMP_NUM_THREADS values (different on host and MIC) 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 with 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:

```
# micmpiexec -prepend-rank -env KMP_AFFINITY verbose -env
OMP_NUM_THREADS 4 -env I_MPI_PIN_DOMAIN auto -n 2 ./test-openmp 2>&1 |
sort

# micmpiexec -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

# micmpiexec -prepend-rank -env KMP_AFFINITY verbose -env
OMP_NUM_THREADS 4 -env I_MPI_PIN_DOMAIN 4 -host `hostname` -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 usual beneficial to avoid splitting of cores on MIC 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" (MIC specific) to modify the default OpenMP affinity.

```
# micmpiexec -prepend-rank -env KMP_AFFINITY
verbose,granularity=thread,scatter -env OMP_NUM_THREADS 4 -env
I_MPI_PIN_DOMAIN auto -n 2 ./test-openmp

# micmpiexec -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

# micmpiexec -prepend-rank -env KMP_AFFINITY
verbose,granularity=thread,compact -env OMP_NUM_THREADS 4 -env
I_MPI_PIN_DOMAIN 4 -host `hostname` -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 MIC.

OFFLOAD

Now we want to run the Intel MPI test program with some offload code on MIC. The simple printout from the OpenMP thread is now offloaded to MIC:

```
# diff test.c test-offload.c
```

Compile and link for the XEON host with the "-openmp" compiler flag as before. The latest compiler automatically recognizes the offload pragma and creates the binary for it. If required offloading would be switched off with the "-no-offload" flag:

```
# mpiicc -openmp -o test-offload test-offload.c
```

Execute the binary on the host:

```
# OFFLOAD_REPORT=2 mpirun -prepend-rank -env OMP_NUM_THREADS 4 -n 2
./test-offload
```

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

```
# mpirun -prepend-rank -env KMP_AFFINITY verbose -env OMP_NUM_THREADS
4 -n 2 ./test-offload 2>&1 | grep bound | sort
```

All OpenMP threads will be mapped onto identical MIC threads! The variable I_MPI_PIN_DOMAIN cannot be used because the domain splitting will be calculated due to the number of logical processors on the XEON host!

One solution is to specify explicit proclists per MPI process:

```
# OFFLOAD_REPORT=2 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:

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

An alternative solution utilizes the environment variable KMP_PLACE_THREADS for thread placement. The principle syntax is
KMP_PLACE_THREADS =<#cores>Cx<#threads_par_core>T,<core_offset>O
but there are several syntax variants described in the compiler reference (e.g. without C,T,O). The environment variables avoids the calculation of specific HW thread numbers, only the core offset has to be calculated. Without OMP_NUM_THREADS set, it specifies also the number of OpenMP threads:

```
# mpirun -prepend-rank -genv KMP_AFFINITY verbose -env
KMP_PLACE_THREADS=4Cx1T,0O -n 1 ./test-offload : -env
KMP_PLACE_THREADS=4Cx1T,4O -n 1 ./test-offload 2>&1 | grep bound |
sort
```

Please notice that inside a thread group the default mapping is still scatter:

```
# mpirun -prepend-rank -genv KMP_AFFINITY verbose -env
KMP_PLACE_THREADS=2Cx2T,0O -n 1 ./test-offload : -env
KMP_PLACE_THREADS=2Cx2T,2O -n 1 ./test-offload 2>&1 | grep bound |
sort
```

This can be overwritten by specifying KMP_AFFINITY compact or balanced in addition.

The third solution for the thread mapping uses a wrapper script which calculates the offset for KMP_PLACE_THREADS based on the MPI process rank which is provided in the process environment variable PMI_RANK. One example of such a wrapper script is provided as file runoff.sh, it takes the <#cores>Cx<#threads_par_core>T definition as parameter:

```
# cat runoff.sh

# mpirun -prepend-rank -genv KMP_AFFINITY verbose -n 2 ./runoff.sh
3Cx2T 2>&1 | grep bound | sort

# mpirun -prepend-rank -genv KMP_AFFINITY verbose,compact -n 2
./runoff.sh 3Cx2T. 2>&1 | grep bound | sort

# mpirun -prepend-rank -genv KMP_AFFINITY verbose,compact -genv
OMP_NUM_THREADS 5 -n 2 ./runoff.sh 3Cx2T 2>&1 | grep bound | sort
```

Lab 2 - Hybrid MPI/OpenMP

Enter the directory 2_MPI_OpenMP.

Execute the following commands to build the Poisson binaries for the XEON host and the MIC card. The compilation will use the "`-openmp`" flag to create the hybrid version of the code. The OpenMP threads are used in file compute.c:

```
# make clean; make

# make clean; make MIC
```

Execute the Poisson application on the XEON host, the MIC card and in symmetric mode on both. The code accepts the following flags:

"-n x" change size of grid. The default is x=1000.

"-iter x" change number of max iterations. The default is x= 4000.

"-prows x" change number of processor rows. The default is computed.

```
# micmpiexec -env OMP_NUM_THREADS 16 -n 1 ./poisson -n 9600 -iter 50

# micmpiexec -env OMP_NUM_THREADS 60 -host mic0 -n 1 ./poisson.MIC -n
9600 -iter 50

# micmpiexec -env OMP_NUM_THREADS 16 -host `hostname` -n 1 ./poisson -
n 9600 -iter 50 : -env OMP_NUM_THREADS 60 -host mic0 -n 1
./poisson.MIC -n 9600 -iter 50
```

Vary the number of MPI processes and OpenMP threads (most likely different on the XEON host and MIC) to optimize the performance. Use the knowledge about MPI process mapping and OpenMP thread affinity from the basic lab.

Lab 4 - Intel Trace Analyzer and Collector

Enter the directory 4_ITAC.

Execute the following commands to build the Poisson binaries for the XEON host and the MIC card with the "-tcollect" flag which will instrument the code:

```
# make clean; make

# make clean; make MIC
```

Execute the Poisson application on the XEON host, the MIC card and in symmetric mode on both (see commands below). Each run creates an ITAC trace file in the stf format which can be analyzed with the traceanalyzer.

Start in the traceanalyzer with your preferred approach, for example: Select “Charts->Event Timeline”, select “Charts->Message Profile”, zoom into the Event Timeline (klick into it, keep pressed, move to the right, and release the mouse) and see menu Navigate to get back. Right klick the “Group MPI->Ungroup MPI”.

The trace analyzer reference guide is available at `$VT_ROOT/doc/ITA_Reference_Guide.pdf` (for the collector libraries at `$VT_ROOT/doc/ITC_Reference_Guide.pdf`), in Beta without MIC details.

```
# export VT_LOGFILE_FORMAT=stfsingle
# micmpiexec -env OMP_NUM_THREADS 1 -n 12 ./poisson -n 3500 -iter 10
# traceanalyzer poisson.single.stf

# micmpiexec -env OMP_NUM_THREADS 1 -host mic0 -n 12 ./poisson.MIC -n
3500 -iter 10
# traceanalyzer poisson.MIC.single.stf

# micmpiexec -env OMP_NUM_THREADS 1 -host `hostname` -n 12 ./poisson -
n 3500 -iter 10 : -env OMP_NUM_THREADS 1 -host mic0 -n 12
./poisson.MIC -n 3500 -iter 10
# traceanalyzer poisson.single.stf
```

For the symmetric run with identical number of MPI processes on host and MIC you will likely see load imbalance due to the difference characteristics of the architectures (clock rate, …).

Alternatively to the instrumentation of the code (which introduces calls to the ITAC library into the binaries) re-use the binaries without changes from directory 2_MPI_OpenMP (enter the 2_MPI_OpenMP directory, do not forget to upload the MIC binary again), but execute all runs with the "-trace" flag. Load the *.stf trace file into the traceanalyzer and start the analysis:

```
# export VT_LOGFILE_FORMAT=stfsingle
# micmpiexec -trace -env OMP_NUM_THREADS 1 -n 12 ./poisson -n 3500 -
iter 10
```

```
# traceanalyzer poisson.single.stf

# micmpiexec -trace -env OMP_NUM_THREADS 1 -host mic0 -n 12
./poisson.MIC -n 3500 -iter 10
# traceanalyzer poisson.MIC.single.stf

# micmpiexec -trace -env OMP_NUM_THREADS 1 -host `hostname` -n 12
./poisson -n 3500 -iter 10 : -env OMP_NUM_THREADS 1 -host mic0 -n 12
./poisson.MIC -n 3500 -iter 10
# traceanalyzer poisson.single.stf
```