Intel® MPI on MIC - Lab Instructions

Lab 0 - Prerequisites

Assumptions:

- MPSS Alpha2 is installed. A major difference to the previous version (which is exploited in the labs) is the existence of user accounts on the card, and the availability of `ssh/scp`.
- The compiler is installed in directory
  `ICCROOT=/opt/intel/composer_xe_2013.0.075`
- Intel MPI is installed in directory
  `MPIROOT=/opt/intel/impi/4.1.0.018`
- Intel Trace Analyzer and Collector (ITAC) is installed in directory
  `ITACROOT=/opt/intel/itac/8.1.0.016`

The path of the Intel MPI reference manual is `$MPIROOT/doc/Reference_Manual.pdf`, use it for details about the environment variables. However, in Beta it does not yet include specific MIC details, see `$MPIROOT/doc/Release_Notes_Addendum_for_MIC_Architecture.txt` instead (but: IP addresses and ssh access changed with the latest MIC drivers).

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

Enter the 0_Prerequisites directory where you will find the SOLUTION.txt files including the commands below for your convenience.

After a reboot of the KNC card several libraries have to be uploaded. For the ease of use the target directory is `/lib64/`, consequently no further `LD_LIBRARY_PATH` setting is required.

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 using the following script, plus the libraries used in the debugger lab:

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

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

Setup and check your user environment for the compiler, Intel MPI, and ITAC by executing the corresponding source files:

```
# source $ICCROOT/bin/compilervars.sh intel64

# source $MPIROOT/intel64/bin/mpivars.sh

# source $ITACROOT/intel64/bin/itacvars.sh impi4

# which icc mpiicc traceanalyzer
```

Check the ssh access which is required for MPI, it should work without a password:

```
# ssh mic0 'hostname; echo "If this works the prerequisites are done,
continue with the next lab!"'
```

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.

Specify the working directory on the MIC card, usually the home directory of the user account:

```
# export MICHOME=/home/$USER/
```

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 form the XEON one. It could be any suffix!

Now the MIC binary has to be uploaded somewhere onto the card. It is also possible to explicitly rename the target binary:

```
# scp test.MIC mic0:$MICHOME
```

An alternative to scp is to mount an NFS file system onto MIC. Copy the MIC binary into the NFS area and use it with the path visible on the MIC card.

Set the communication fabrics:

```
# export I_MPI_FABRICS=shm:tcp
```

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

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

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

```
# mpiexec -wdir $MICHOME -host mic0 -n 2 ./test.MIC
```

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:

```
# mpiexec -host `hostname` -n 2 ./test : -wdir $MICHOME -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 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 mpiexec (without "="):

```
# mpiexec -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:

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

# mpiexec -prepend-rank -wdir $MICHOME -host mic0 -n 2 ./test.MIC

# mpiexec -prepend-rank -host `hostname` -n 2 ./test : -wdir $MICHOME
-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.

Ignore the error output "ERROR - load_iblibrary", this is just a message from Intel MPI when analysing the available interconnects between the host and the MIC card. You can avoid it by setting explicitly:

```
# export I_MPI_FABRICS=shm:tcp
```

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:

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

# mpiexec -prepend-rank -env I_MPI_PIN_DOMAIN auto -wdir $MICHOME -
host mic0 -n 2 ./test.MIC
```

```
# mpiexec -prepend-rank -env I_MPI_PIN_DOMAIN 4 -host `hostname` -n 2
./test : -env I_MPI_PIN_DOMAIN 12 -wdir $MICHOME -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

# scp test-openmp.MIC mic0:$MICHOME
```

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

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

# mpiexec -prepend-rank -wdir $MICHOME -host mic0 -n 2 ./test-
openmp.MIC

# mpiexec -prepend-rank -host `hostname` -n 2 ./test-openmp : -wdir
$MICHOME -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:

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

```
# mpiexec -prepend-rank -env KMP_AFFINITY verbose -env OMP_NUM_THREADS
4 -env I_MPI_PIN_DOMAIN omp -wdir $MICHOME -host mic0 -n 2 ./test-
openmp.MIC 2>&1 | sort
```

```
# mpiexec -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 -
wdir $MICHOME -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.

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

```
# mpiexec -prepend-rank -env KMP_AFFINITY
verbose,granularity=thread,compact -env OMP_NUM_THREADS 4 -env
I_MPI_PIN_DOMAIN omp -wdir $MICHOME -host mic0 -n 2 ./test-openmp.MIC
2>&1 | sort
```

```
# mpiexec -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 -wdir $MICHOME -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:

```
# H_TRACE=1 mpiexec -prepend-rank -env KMP_AFFINITY
granularity=thread,scatter -env OMP_NUM_THREADS 4 -n 2 ./test-offload
```

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

```
# mpiexec -prepend-rank -env KMP_AFFINITY
verbose,granularity=thread,scatter -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!

The solution is to specify explicit proclists per MPI process:

```
# H_TRACE=1 mpiexec -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:

```
# mpiexec -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
```

Lab 2 - Hybrid MPI/OpenMP

Enter the directory 2_MPI_OpenMP.

Specify the working directory on the MIC card, usually the home directory of the user account:

```
# export MICHOME=/home/$USER/
```

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; make upload
```

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.

```
# mpiexec -env OMP_NUM_THREADS 12 -n 1 ./poisson -n 3500 -iter 10
```

```
# mpiexec -env OMP_NUM_THREADS 12 -wdir $MICHOME -host mic0 -n 1
./poisson.MIC -n 3500 -iter 10
```

```
# mpiexec -env OMP_NUM_THREADS 12 -host `hostname` -n 1 ./poisson -n
3500 -iter 10 : -env OMP_NUM_THREADS 12 -wdir $MICHOME -host mic0 -n 1
./poisson.MIC -n 3500 -iter 10
```

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 3 - Hybrid MPI/CilkPlus

Enter the directory 3_MPI_CilkPlus.

Specify the working directory on the MIC card, usually the home directory of the user account:

```
# export MICHOME=/home/$USER/
```

Execute the following commands to build the Poisson binaries for the XEON host and the MIC card. The Cilk Plus statements in the compute.c file establish the hybrid version of the code:

```
# make clean; make
```

```
# make clean; make MIC; make upload
```

Please notice that this lab is a non-optimized functionality test, execution of run commands will take more time to complete than in the MPI/OpenMP lab!

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.

```
# mpiexec -env CILK_NWORKERS 12 -n 1 ./poisson -n 1000 -iter 10

# mpiexec -env CILK_NWORKERS 12 -wdir $MICHOME -host mic0 -n 1
./poisson.MIC -n 3500 -iter 10

# mpiexec -env CILK_NWORKERS 12 -host `hostname` -n 1 ./poisson -n
3500 -iter 10 : -env CILK_NWORKERS 12 -wdir $MICHOME -host mic0 -n 1
./poisson.MIC -n 3500 -iter 10
```

Vary the number of MPI processes and Cilk Plus workers (most likely different on the XEON host and MIC).

Lab 4 - Intel Trace Analyzer and Collector

Enter the directory 4_ITAC.

Specify the working directory on the MIC card, usually the home directory of the user account:

```
# export MICHOME=/home/$USER/
```

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; make upload
```

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 `$ITACROOT/doc/ITA_Reference_Guide.pdf` (for the collector libraries at `$ITACROOT/doc/ITC_Reference_Guide.pdf`), in Beta without MIC details.

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

# mpiexec -env OMP_NUM_THREADS 1 -wdir $MICHOME -host mic0 -n 12
./poisson.MIC -n 3500 -iter 10
# scp mic0:$MICHOME/poisson.MIC.single.stf ./
# traceanalyzer poisson.MIC.single.stf

# mpiexec -env OMP_NUM_THREADS 1 -host `hostname` -n 12 ./poisson -n
3500 -iter 10 : -env OMP_NUM_THREADS 1 -wdir $MICHOME -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
# mpiexec -trace -env OMP_NUM_THREADS 1 -n 12 ./poisson -n 3500 -iter
10
# traceanalyzer poisson.single.stf

# mpiexec -trace -env OMP_NUM_THREADS 1 -wdir $MICHOME -host mic0 -n
12 ./poisson.MIC -n 3500 -iter 10
# scp mic0:$MICHOME/poisson.MIC.single.stf ./
# traceanalyzer poisson.MIC.single.stf

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

Lab 5 - Debugging

Execute and check all setups of the “Lab 0 –Prerequisites” section for the compiler and Intel MPI, in particular the upload of the (debug) libraries, plus the setup of the user environment:

```
# source $ICCROOT/bin/compilervars.sh intel64
```

```
# source $MPIROOT/intel64/bin/mpivars.sh
```

```
# which icc mpiicc
```

A. Debugging on the Host

Compile the MPI program (test.c) for the host with debug flag `“-g”`

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

Start the host Intel debugger `idb`

```
# mpiexec -n 2 idb ./test
```

This command will start two debugger sessions for two ranks on the host called knc0, and two debuggers attach to these ranks.

In each host debugger, click File-> Open Source File and navigate to the source code that you are interested in. Insert break point and hit F5 (run).

Note: if you just need to run the Intel debugger with the rank you want to monitor.

For example, the following command starts three ranks on the host, and only the first rank has debugger:

```
# mpiexec -n 1 idb ./test : -n 2 ./test
```

B. Debugging on host and MIC simultaneously

In one window (called the first window), follow the instructions from the previous section to build a debug executable for the host.

In another window (called the second window), the MIC executable will be build and finally debugged.

Execute and check again the setup of the user environment for the compiler and Intel MPI of the “Lab 0 –Prerequisites” section:

```
# source $ICCROOT/bin/compilervars.sh intel64
```

```
# source $MPIROOT/intel64/bin/mpivars.sh
# which icc mpiicc
```

Specify the working directory on the MIC card, usually the home directory of the user account:

```
# export MICHOME=/home/$USER/
```

Compile the MPI program for MIC with debug flag -g (there are some warnings which can be ignored):

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

Upload the executable and debug library to the MIC card:

```
# scp test.MIC mic0:$MICHOME
```

Upload a missing library, required in the debugger:

```
# sudo scp /opt/intel/mic/myo/lib/libmyodbl-service.so mic0:/lib64/
```

[Note: Depending on your application you may have to upload additional compiler libraries, for example:

```
# sudo scp $ICCROOT /lib/mic/libintlc.so.5 mic0:/lib64/
# sudo scp $ICCROOT /lib/mic/libsvml.so    mic0:/lib64/
# sudo scp $ICCROOT /lib/mic/libimf.so     mic0:/lib64/
]
```

In the first window, run the program with the host Intel debugger idb, after setting the communication fabrics:

```
# export I_MPI_FABRICS=shm:tcp

# mpiexec -host `hostname` -n 1 idb ./test : -wdir $MICHOME -host mic0
-n 1 ./test.MIC
```

This starts one rank on the host with the debugger and one rank on MIC.

In the host debugger, click File-> Open Source File and navigate to the source code that you are interested. Insert break point (e.g. on the `MPI_Barrier (line 37)` and hit F5 (run).

In the second window, start the command line debugger `idbc_mic` for the MIC target and attach it to the MPI process running on MIC. Retrieve the process id (pid) of the MPI process before by using a one times call of `top` on MIC:

```
# ssh mic0 top -n 1 | grep test.MIC
```

(Note that starting with KNC idb_mic is not supported, so we use idbc_mic instead in this lab.)

```
# idbc_mic -tco -rconnect=tcpip:mic0:2000
```

In the target debugger, attach to the pid of the MPI process running on the MIC:

```
# (idb) attach <pid> target-bin-path-on-host
```

e.g.,

```
# (idb) attach 1927 ./test.MIC
# (idb) list
# (idb) b 37
# (idb) cont
```

In Debugger Commands, type `l (list).`