PPCES 2013: Intel Xeon Phi Programming

March 15, 2013
Dirk Schmidl, Tim Cramer
{schmidl,cramer}@rz.rwth-aachen.de

0. Preparations

- Login to one of our frontend nodes, e.g.:
  `$ ssh cluster.rz.rwth-aachen.de`
- Switch to one of the Intel Xeon Phi machines.
  `$ ssh linuxphi???`
  `(Note: Which machine is shown in the slide at the front.)`
- Change into the lab directory:
  `$ cd ex_phi/C (for C exercises) or ex_phi/F(for Fortran exercises)`
- switch to the latest intel compiler module to get offload support
  `$ module switch intel intel/13.1`

1. Hello World

Switch into the 01_hello directory. The files contain simple "Hello World!" programs. There is also a Makefile to build the program.

1.1. Native on the Host

Use the Makefile to build the program on the host ("`make host`") and execute it ("`./hello_host.exe`").

The program executes on the host system and prints the message "Hello from HOST!".

1.2. Native on the MIC

Edit the Makefile to compile the program for native execution on the Intel Xeon Phi. Add the necessary Compiler Flag to CFLAGS_MIC or FFLAGS_MIC in line 4 of the Makefile and compile the executable for the coprocessor ("`make phi`"). Login to the coprocessor and run the executable.

```
$ ssh -i ~/.ssh/mic/id_rsa linuxphi<???>-mic0
$ cd /home/<hpclab??>/ex_phi/01_hello
$ ./hello_phi.exe
```

The program prints “Hello from Phi!”

Return to the host machine.
$ exit

1.3. Intel Language Extensions for Offload

Edit the file `hello_offload.c` (line 27) or hello_offload.F90 (line17) to offload the function `print_hello` from the host to the coprocessor.

Hint: You also need to add the attribute to the function declaration as presented in the slides.

Compile the code and run the executable.

```
$ make offload
$ ./hello_offload.exe
```

The function `print_hello` is called two times, one time from the host and one time from the Intel Xeon Phi. The output in both cases is “Hello from HOST or Phi.” Change the function print_hello (line 6 or line 5) to print “Hello from Phi.”, if it is executed on the coprocessor and “Hello from HOST” otherwise.

Hint: The function is build two times by the compiler, one version for the host and one version for the coprocessor. You can use the __MIC__ define to include code that should be only executed on the coprocessor.

1.4. MPI

You can also start MPI programs on the Intel Xeon Phi coprocessor and on the host system. Since both systems are not fully binary compatible, you need to build two executables.

```
$ module unload openmpi
$ module load BETA intelmpi/4.1mic
$ make mpi
```

The Makefile builds 2 executables, `hello_mpi.exe` for the host and `hello_mpi.exe.mic` for the Phi. To start the MPI program, you need to edit the file `hosts` and specify the machines you want to start the job on. Open the file hosts and replace the question marks with the machine name on your login sheet.

Execute the program with one process running on the host and one process running on the coprocessor.

```
$ $MPIEXEC -n 2 -machine hosts hello_mpi.exe
```

Note: The executable hello_mpi.exe.mic is automatically used on the coprocessor. In our environment you always have to add the postfix .mic for MPI executables that shall be run on the coprocessor.

2. Jacobi

Switch into the 02_jacobi directory. There you can find the Jacobi program parallelized with OpenMP from Wednesday. There is a Makefile to build and run the executable (“make build” and “make run”).

2.1. Offloading

The compute intensive part of the kernel (jacobi.c, line 88, jacobi.f90 line 159) has already been parallelized with OpenMP. Use Intel’s Language Extensions for Offload to offload the parallel region to the coprocessor.

Hint: You need to transfer the arrays u, f and uold to the coprocessor, e.g. with the inout clause.

2.2. Analyze the data transfer

Set the environment variable OFFLOAD_REPORT to 2 and run the program again.

```
$ export OFFLOAD_REPORT=2
$ make run
```

The parallel region is executed and offloaded once per iteration (5 times for the input file). You can see 5 blocks in the output showing the execution time and the amount of transferred data, one block every time an offload occurs.
Analyze the data to find out if the data is transferred every time or if the data is reused from previous iterations.

Unset the environment variable before you go on with the exercises.

```
$ unset OFFLOAD_REPORT
```

2.3. Minimize the data transfer

As you have seen the data is transferred to the device and back every time. Since it is not changed on the host you can optimize the code by copying the data outside of the while-loop to the device and back after the loop is over.

Hint: Use `offload_transfer` clauses to do the data transfer and use a copy of length zero at the offload pragma. Specify `free_if(0)` and `alloc_if(0)` where needed.

2.4. For experts only: Asynchronous execution

While the coprocessor is executing the code, the processor is just waiting. You need to split the work in two parts to utilize the host and the coprocessor simultaneously.
The following steps are needed:

1. Duplicate the parallel region and let every instance run over half of the iteration space.
2. Offload one of the regions to the device. Use asynchronous offloading to allow execution on host and coprocessor at the same time.
3. Transfer data before and after the offload region that needs to be exchanged.