1.1 Run “Hello World”

Now that the Intel® Xeon Phi™ coprocessors are up and running, execute a simple program several different ways.

1.1.1 “Hello World” Native Execution Using the Intel® C Compiler

This is a repeat of the previous example, but this time using Intel® C Compiler for compilation. Note that you will need to install the Intel® Compiler suite to build this example (and the follow-on example highlighting offload directives.)

Notice that the Intel® C compiler (icc) is used with an additional flag (*-mmic*) to indicate that the target architecture in this case is the Intel® Xeon Phi™ coprocessor.

```
[host]$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>
void
main()
{
    printf("Hello World \n");
}
[host]$ icc -mmic hello world.c -o hello world
```

Next, copy the code to the file system on the coprocessor using *scp*

```
[host]$ scp hello_world mic0:
hello world            100%   10KB  10.2KB/s   00:00
```

Invoke the application on the coprocessor.

```
[host]$ ssh mic0 /home/<USER>/hello_world
Hello World
```

1.1.2 “Hello World” via Compiler Based Offload Directives

This example demonstrates the use of offload directives to run code on the coprocessor.

```
[host]$ cat hello_offload.c
#include <stdio.h>
#include <stdlib.h>
void
main()
{
    #pragma offload target (mic:0)
    {
      printf("hello world from offloaded code running on the coprocessor \n");
    }
}
```

To build it, use the Intel® C compiler, as before with *-offload* flag.

```
[host]$ icc -offload hello_offload.c -o hello_offload
```

Finally, to run it, simply invoke the host side binary

```
[host]$ ./hello_offload
hello world from offloaded code running on the coprocessor
```

1.2 运行 INTEL MPI 程序

2. MPIRUN 运行方式

mpirun 是简化的启动 MPI 进程方式

单节点简略如：**mpirun –np 8 hostname**

```
$ mpirun -machinefile mpd.hosts -np 16 ./program
```

节点文件 mpd.hosts 格式如下：

```
node1:8
node2:8
```

也可以将 node1 和 node2 分别写 8 行。

1.3 编写 MIC 程序

MIC 程序运行有两种模式：offload 模式和 native 模式(详细可购买参考书《MIC 高性能计算编程指南》)。offload 模式类似于使用 GPU 作为计算设备的程序，即程序在 CPU 端启动并运行，中间遇到需要加速计算的部分则转移到 MIC 上运行，运行结束后再返回 CPU 端。native 模式则是指整个程序从启动到结束都在 MIC 上运行，与 CPU 端无关（用户直接登录 MIC 卡操作系统运行程序）。

服务器所有示例可见[root@node1 public]# ls /public/example/

1.3.1 串行程序

计算 PI 的串行程序如下：

文件名：pi-serial.c

```
#include <stdio.h>
#include <math.h>
int main(int argc,char *argv[])
{
    int     n, i;
    double pi, dx, x;
    n = 10000;
```

```
    dx = 1.0 / (double) n;
    pi = 0.0;
    for(i=0; i<n; i++)
    {
        x = dx * ((double)i + 0.5);
        pi += 4.0*dx/(1.0+x*x);
    }
    printf("pi is approximately %.16f\n", pi);
    return 0;
}
```

要利用 MIC 上众多核心的计算资源，首先需要对这个串行程序进行并行化，这里分别采用 openmp 和 mpi 两种方法进行并行。

1.3.2 OPENMP 并行程序

（1）使用 CPU 进行计算（加入蓝色粗体部分代码）：

文件名：pi-omp.c

```
#include <stdio.h>
#include <math.h>
int main(int argc,char *argv[])
{
    int       n, i;
    double pi, dx, x;
    n = 10000;
    dx = 1.0 / (double) n;
    pi = 0.0;
    #pragma omp parallel for reduction(+:pi)
    for(i=0; i<n; i++)
    {
        x = dx * ((double)i + 0.5);
        pi += 4.0*dx/(1.0+x*x);
    }
    printf("pi is approximately %.16f\n", pi);
    return 0;
}
```

使用 icc -openmp -o pi-omp pi-omp.c 进行编译，编译得到 pi-omp 的可执行文件。在 HOST 端运行即可利用单节点 CPU 多核资源进行 PI 的计算。

（2）在 MIC 上使用 native 模式进行计算。native 模式仍然使用以上 pi-omp.c 的代码，无需进行任何改动。但需要添加-mmic 参数进行编译，如：

```
icc -mmic -openmp -o pi-omp-native pi-omp.c
```

将编译得到的可执行程序 pi-omp-native 拷贝到 MIC 上的/tmp 目录下：

```
scp pi-omp-native root@mic0:/tmp
```

再将编译器安装目录下查找 libiomp5.so 文件，并将其拷贝到 MIC 卡上的/lib64 目录下，如：

```
cd /public/software/intel/composer_xe_2013/compiler/lib/mic
scp libiomp5.so root@mic0:/lib64
```

登录到 MIC 上，进入 tmp 目录，然后运行刚才拷贝过来的程序。

```
ssh root@mic0
cd /tmp
./pi-omp-native
```

这样就可以实现计算 PI 程序的 native 运行了。加了-mmic 参数编译的程序只能在 MIC 上运行，无法在 CPU 上运行。

（3）使用 offload 模式利用 MIC 进行计算，代码如下：

文件名：pi-omp-offload.c

```
#include <stdio.h>
#include <math.h>
int main(int argc,char *argv[])
{
    int       n, i;
    double pi, dx, x;
    n = 10000;
    dx = 1.0 / (double) n;
    pi = 0.0;
    #pragma offload target(mic)
    #pragma omp parallel for reduction(+:pi)
    for(i=0; i<n; i++)
    {
        x = dx * ((double)i + 0.5);
        pi += 4.0*dx/(1.0+x*x);
    }
    printf("pi is approximately %.16f\n", pi);
```

```
    return 0;
}
```

以上代码在 pi-omp.c 的基础上添加了一行编译制导语句（红色粗体部分）：

```
#pragma offload target(mic)
```

该编译制导语句提示编译器编译时将仅随其后的代码块同时编译为在 MIC 上执行的代码和在 CPU 执行的代码。使用 `icc -openmp -o pi-omp-offload pi-omp-offload.c` 对以上程序进行编译，得到的可执行文件为 pi-omp-offload，在 HOST 端直接运行该程序即可使用 MIC 进行 PI 的计算了。

1.3.3 MPI 并行程序

使用 OPENMP 实现的并行程序仅仅能在一个节点的计算资源，而且目前仅用到了一块 MIC 卡。下面使用 MPI 来实现使用多块卡进行 PI 的计算。

（1）使用 CPU 进行计算

文件名：pi-mpi.c

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main(int argc,char *argv[])
{
    int     i, n, ntmp, ifr, ito;
    int     myid, nprocs;
    double mypi, pi, dx, x;
    double startwtime, endwtime;
    int     namelen;
    char    processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    printf("Process %d of %d is on %s\n", myid, nprocs, processor_name);
    n = 10000;
    if (myid == 0)      startwtime = MPI_Wtime();
    n = (int)(n/nprocs) * nprocs;
    dx = 1.0 / (double)n;
```

```
    mypi = 0.0;
    ntmp = n / nprocs;
    ifr = myid * ntmp;
    ito = (myid + 1) * ntmp；
    for(i=ifr; i<ito; i++)
    {
        x = dx * ((double)i + 0.5);
        mypi += 4.0*dx/(1.0+x*x);
    }
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
    {
        endwtime = MPI_Wtime();
        printf("pi is approximately %.16f\n", pi);
        printf("wall clock time = %f seconds\n", endwtime-startwtime);
    }
    MPI_Finalize();
    return 0;
}
```

使用 mpiicc 进行编译：

```
mpiicc -o pi-mpi pi-mpi.c
```

得到可执行文件 pi-mpi，在 HOST 端运行 mpirun -np 12 ./pi-mpi 即可利用 12 个 CPU 核的资源进行计算。

（2）在 MIC 上使用 native 模式进行计算。native 模式仍然使用以上 pi-mpi.c 的代码，无需进行任何改动。但需要添加-mmic 参数进行编译，如：

```
mpiicc -mmic -o pi-mpi-native pi-mpi.c
```

将编译得到的可执行程序 pi-mpi-native 拷贝到 MIC 上的/tmp 目录下：

```
scp pi-mpi-native root@mic0:/tmp
```

然后登录到 MIC 上（ssh root@mic0），执行：

```
mpirun -np 240 ./pi-mpi-native
```

其中 240 为启动的进程数，这里的 MIC 有 60 个核心，每核心支持 4 个线程，即 240。

（3）使用 offload 模式利用 MIC 进行计算，代码如下：

文件名：pi-mpi-offload.c

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main(int argc,char *argv[])
{
    int       i, n, ntmp, ifr, ito;
    int       myid, nprocs;
    double mypi, pi, dx, x;
    double startwtime, endwtime;
    int       namelen;
    char      processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    printf("Process %d of %d is on %s\n", myid, nprocs, processor_name);
    n = 10000;
    if (myid == 0)    startwtime = MPI_Wtime();
    n = (int)(n/nprocs) * nprocs;
    dx = 1.0 / (double)n;
    mypi = 0.0;
    ntmp = n / nprocs;
    ifr = myid * ntmp;
    ito = (myid + 1) * ntmp;
    #pragma offload target(mic:myid)
    #pragma omp parallel for reduction(+:mypi)
    for(i=ifr; i<ito; i++)
    {
        x = dx * ((double)i + 0.5);
        mypi += 4.0*dx/(1.0+x*x);
    }
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
    {
        endwtime = MPI_Wtime();
        printf("pi is approximately %.16f\n", pi);
        printf("wall clock time = %f seconds\n", endwtime-startwtime);
    }
    MPI_Finalize();
    return 0;
```

```
}
```

以上代码添加了两行制导语句（红色黑体部分）：

```
    #pragma offload target(mic:myid)
    #pragma omp parallel for reduction(+:mypi)
```

其中第一行制导语句指示编译器将接下来的代码段同时编译为 CPU 和 MIC 上的可执行代码，第二行制导语句指示接下来的代码段使用 OPENMP 进行并行。

使用如下命令进行编译：

```
mpiicc -openmp -o pi-mpi-offload pi-mpi-offload.c
```

在 HOST 端，运行 mpirun -np 2 ./pi-mpi-offload 即可启动两个进程，每个进程可使用一块 MIC 卡进行计算。若存在 N 块 MIC 卡，则至少需要启动 N 个进程才能利用所有 MIC 卡的资源。