Introduction to Intel Xeon Phi Coprocessors – Part II

Summer Supercomputing Institute
June 19th 2014

Carlos Rosales
carlos@tacc.utexas.edu
SYMMETRIC COMPUTING
Symmetric Computing

• Run MPI tasks on both MIC and host

• Also called “heterogeneous computing”

• Two executables are required:
  – CPU
  – MIC

• Currently only works with Intel MPI

• MVAPICH2 support coming
Definition of a Node

A “node” contains a Host component and a MIC component

- Host – refers to the Sandy Bridge component
- MIC – refers to one or two Intel Xeon Phi co-processor cards

<table>
<thead>
<tr>
<th>NODE</th>
<th>Host</th>
<th>MIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2x Intel 2.7 GHz E5-2680 16 cores</td>
<td>1-2 Intel Xeon PHI SE10P 61 cores / 244 HW threads</td>
</tr>
</tbody>
</table>
Environment variables for MIC

By default, environment variables are “inherited” by all MPI tasks

Since the MIC has a different architecture, several environment variables must be modified

- OMP_NUM_THREADS – # of threads on MIC
- LD_LIBRARY_PATH – must point to MIC libraries
- I_MPI_PIN_MODE – controls the placement of tasks
- KMP_AFFINITY – controls thread binding
Steps to create a symmetric run

1. Compile a host executable and a MIC executable
2. Determine the appropriate number of tasks and threads for both MIC and host
3. Create the batch script
4. Submit the batch script
   - `sbatch symmetric.slurm`
The hard way: symmetric run on 1 node

mpiexec.hydra

- `–n 16 –host localhost ./host.exe`
- `–env OMP_NUM_THREADS 30`
- `–env LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH`
- `–env I_MPI_PIN_MODE mpd`
- `–env KMP_AFFINITY balanced`

`–n 4 –host mic0 ./mic.exe`

16 tasks on host

4 tasks on mic0

Environment variables for MIC tasks
Steps to create a symmetric run

1. Compile a host executable and a MIC executable:
   – mpicc –openmp –o my_exe.cpu my_code.c
   – mpicc –openmp –mmic –o my_exe.mic my_code.c

2. Determine the appropriate number of tasks and threads for both MIC and host:
   – 16 tasks/host – 1 thread/MPI task
   – 4 tasks/MIC – 30 threads/MPI task
Steps to create a symmetric run

3. Create a batch script to distribute the job

```bash
#!/bin/bash
#------------------------------------------
# symmetric.slurm
# Generic symmetric script – MPI + OpenMP
#------------------------------------------
#SBATCH –J symmetric        #Job name
#SBATCH -o symmetric.%j.out #stdout; %j expands to jobid
#SBATCH –e symmetric.%j.err  #stderr; skip to combine
#SBATCH –p development      #queue
#SBATCH –N 2                #Number of nodes
#SBATCH –n 32               #Total number of MPI tasks
#SBATCH –t 00:30:00         #max time
#SBATCH –A TG-01234         #necessary if multiple projects

export MIC_PPN=4
export MIC_OMP_NUM_THREADS=30

ibrun.symm -m ./my_exe.mic -c ./my_exe.cpu
```
Symmetric launcher – ibrun.symm

Usage:

```
ibrun.symm -m ./<mic_executable> -c ./<cpu_executable>
```

- Analog of ibrun for symmetric execution
- # of MIC tasks and threads are controlled by env variables

```
MIC_PPN=<# of MPI tasks/MIC card>
MIC_OMP_NUM_THREADS=<# of OMP threads/MIC MPI task>
MIC_MY_NSLOTS=<Total # of MIC MPI tasks>
```
Symmetric launcher

- # of host tasks determined by batch script (same as regular ibrunch)
- ibrunch.symm does not support –o and –n flags
- Command line arguments may be passed within quotes

```
ibrunch.symm -m "./my_exe.mic args" -c "./my_exe.cpu args"
```
Symmetric launcher

- If the executables require redirection or complicated command lines, a simple shell script may be used:

<table>
<thead>
<tr>
<th>run_mic.sh</th>
<th>run_cpu.sh</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#!/bin/sh</code></td>
<td><code>#!/bin/bash</code></td>
</tr>
<tr>
<td><code>a.out.mic &lt;args&gt; &lt; inputfile</code></td>
<td><code>a.out.host &lt;args&gt; &lt; inputfile</code></td>
</tr>
</tbody>
</table>

`ibrun.symm -m ./run_mic.sh -c run_cpu.sh`

Note: The bash, csh, and tcsh shells are not available on MIC. So, the MIC script must begin with “`#!/bin/sh`”
Symmetric Launcher Example

... 

#SBATCH -N 4 -n 32
export OMP_NUM_THREADS=2
export MIC_OMP_NUM_THREADS=60
export MIC_PPN=2

The MPI tasks will be allocated in consecutive order by node (CPU tasks first, then MIC tasks). For example, the task allocation described by the above script snippet will be:

<table>
<thead>
<tr>
<th>NODE</th>
<th>Host Tasks</th>
<th>MIC Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE 1</td>
<td>8 (0-7)</td>
<td>2 (8-9)</td>
</tr>
<tr>
<td>NODE 2</td>
<td>8 (10-17)</td>
<td>2 (18-19)</td>
</tr>
<tr>
<td>NODE 3</td>
<td>8 (20-27)</td>
<td>2 (28-29)</td>
</tr>
<tr>
<td>NODE 4</td>
<td>8 (30-37)</td>
<td>2 (38-39)</td>
</tr>
</tbody>
</table>
Task Binding

When using IMPI, process binding may be controlled with the following environment variable:

- I_MPI_PIN_MODE=<pinmode>

<table>
<thead>
<tr>
<th>pinmode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpd</td>
<td>mdp daemon pins MPI processes at startup (Best performance for MIC)</td>
</tr>
<tr>
<td>pm</td>
<td>Hydra launcher pins MPI processes at startup (Doesn’t appear to work on MIC)</td>
</tr>
<tr>
<td>lib</td>
<td>MPI library pins processes BUT this does not guarantee colocation of CPU and memory (Default)</td>
</tr>
</tbody>
</table>

I_MPI_PIN_MODE=mpd (default for ibrun.symm)
Task Binding

You can also lay out tasks across the local cores

- Explicitly: `I_MPI_PIN_PROCESSOR_LIST=<proclist>`
  - `export I_MPI_PIN_PROCESSOR_LIST=1-7,9-15`
- Grouped: `I_MPI_PIN_PROCESSOR_LIST=<map>`

<table>
<thead>
<tr>
<th>bunch</th>
<th>The processes are mapped as closely as possible on the socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>scatter</td>
<td>The processes are mapped as remotely as possible to avoid sharing common resources: caches, cores</td>
</tr>
<tr>
<td>spread</td>
<td>The processes are mapped consecutively with the possibility to not share common resources</td>
</tr>
</tbody>
</table>
Task Binding

Be careful when using MIC and host

- MIC – 244 H/W threads and 1 socket
- Host – 16 cores and 2 sockets

To set `I_MPI_PROCESSOR_LIST` for MIC simply use the MIC prefix, e.g.

```
export MIC_I_MPI_PROCESSOR_LIST=1,61,121,181
```
Thread Placement

Thread placement may be controlled with the following environment variable

- **KMP_AFFINITY=*/<type>*

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compact</td>
<td>pack threads close to each other</td>
</tr>
<tr>
<td>scatter</td>
<td>Round-Robin threads to cores</td>
</tr>
<tr>
<td>balanced</td>
<td>keep OMP thread ids consecutive (MIC only)</td>
</tr>
<tr>
<td>explicit</td>
<td>use the proclist modifier to pin threads</td>
</tr>
<tr>
<td>none</td>
<td>does not pin threads</td>
</tr>
</tbody>
</table>

For **compact** mode, threads are packed close to each other:

```
0 1 2 3
4 5 6 7
```

For **scatter** mode, threads are distributed to cores in a Round-Robin manner:

```
0 1 2 3
4 5 6 7
```

For **balanced** mode, OMP thread ids are kept consecutive (MIC only):

```
0 1 2 3
4 5 6 7
```
**Balance**

- How to balance the code?

<table>
<thead>
<tr>
<th></th>
<th>Sandy Bridge</th>
<th>Xeon Phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>32 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>Cores</td>
<td>16</td>
<td>61</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.7 GHz</td>
<td>1.1 GHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>51.2 GB/s(x2)</td>
<td>352 GB/s</td>
</tr>
<tr>
<td>Vector Length</td>
<td>4 DP words</td>
<td>8 DP words</td>
</tr>
</tbody>
</table>
Balance

Example: Memory balance
Balance memory use and performance by using a different # of tasks/threads on host and MIC

Host
16 tasks/1 thread/task
2GB/task

Xeon PHI
4 tasks/60 threads/task
2GB/task
Example: Performance balance
Balance performance by tuning the # of tasks and threads on host and MIC

Host
16 tasks/1 thread/task
2GB/task

Xeon PHI
4 tasks/30 threads/task
2GB/task
MPI with Offload Sections

ADVANTAGES
• Offload Sections may easily be added to MPI/OpenMP codes with directives
• Intel compiler will automatically detect and compile offloaded sections

CAVEATS
• However, there may be no MPI calls within offload sections
• Each host task will spawn an offload section
OFFLOAD COMPUTING
Offloading

- **Offloading Concepts**
  - Basics
  - Directive Syntax
  - Automatic Offloading (AO)
  - Compiler Assisted Offloading (CAO)
    - Directives (Code Blocks – Targets)
    - Preparation and Offload Process Steps (mechanism)
    - Data Transfers
    - Declaration for Functions andGlobals, Pointer Data
    - Persistent Data
    - Asynchronous Offloading
- **Offloading inside an OMP parallel region.**
Offloading Strategy

• Think threads
  – (Whether working on a MIC, GPU, ARM, etc.)

• Options:
  – Have the MIC do all of the work
    • May be viable for low-performance-CPU – MIC solution
  – Share the work - host and MIC
    • More reasonable for HPC system with MICs
    • Offload compute-intensive section. If it isn’t threaded, make it threaded
    • Optimize data transfers
    • Split calculation & use asynchronous mechanisms

• Great time to venture into many-core architectures
How Does Offloading Work?

• Send block of code to be executed on coprocessor (MIC).
  – Must have a binary of the code (code block or function).
  – Compiler makes the binary and stores it in the executable (a.out).

• During execution on the CPU, the “runtime” is contacted to begin executing the MIC binary at an offload point.
  – When the coprocessor is finished, the CPU resumes executing the CPU part of the code.
Offload Models

• Non-Shared memory
  – Host and MIC have separate memory sub systems—think distributed memory and bit-wise data copy between platforms.

• Virtual-Shared Memory
  – C/C++; complex data structures (pointer based structures, classes, etc.) can be shared; coherency overhead.

Offloading works best when
  – Compute complexity is $O(N^{i+1})$ and data complexity is $O(N^i)$
  – Code is not IO intensive
  – Offload can be done asynchronously
Automatic Offload

• Offloads some MKL routines automatically
  – No coding change
  – No recompiling
• Makes sense with BLAS-3 type routines
  – Minimal Data $O(n^2)$, Maximal Compute $O(n^3)$
• Supported Routines (more to come)

<table>
<thead>
<tr>
<th>Type</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-3 BLAS</td>
<td>xGEMM, xTRSM, STRMM</td>
</tr>
<tr>
<td>LAPACK 3 amigos</td>
<td>LU, QR, Cholesky</td>
</tr>
<tr>
<td>Eigen Solver</td>
<td></td>
</tr>
</tbody>
</table>
Enabling Automatic Offload

- Compile as usual, use new –mkl
  - Works with serial, OpenMP and MPI codes.
- Enable with MKL_MIC_ENABLE variable

```
login1$ ifort -mkl -xhost -O2 app_has_MKLDgemm.f90
login1$ icc -mkl -xhost -O2 app_has_MKLDgemm.c
...

c559-001$ export OMP_NUM_THREADS=16

c559-001$ export MKL_MIC_ENABLE=1

c599-001$ ./a.out
```

See MKL_MIC_WORKDIVISION environment variable to set (force) a relative work load.
Offload Directives

• Directives can be inserted before code blocks and functions to run the code on the Xeon Phi Coprocessor (the “MIC”).

• **No recoding required.** (Optimization may require some changes.)

• **Directives are simple**, but more “details” (specifiers) can be used for optimal performance.

• **Data must be moved to the MIC.** For large amounts of data:
  – Amortize with large amounts of work.
  – Keep data resident (“persistent”).
  – Move data asynchronously.
• Insert Offload Directive:

• Compile with Intel Compiler:

• How to turn off offloading:

```c
int main()
{
    float a[10]; int i;
    #pragma offload target(mic)
    {
        for(i=0;i<10;i++)
            a[i]=(float) i;
    }
    #pragma offload target(mic)
    foo(a);
    printf(" %f \n",a[10]);
}
```

```c
program main
real :: a(10)

!dir$ offload begin target(mic)
do i=1,10
    a(i)=i; end do
!dir$ end offload
!dir$ offload target(mic)
call foo(a)
print*, a(10)
end program
```

```c
#pragma offload target(mic) C/C++
!dir$ offload target(mic) F
```

```c
icc prog.c ifort prog.f90
```

use --no-offload option
OpenMP regions can be offloaded directly.

OpenMP parallel regions can exist in offloaded code blocks or functions.
Compile & Run

- Compile on login node (as shown), or on compute node interactively (see idev in lab exercise).
  ```
  login2$  icc --openmp --xhost --O3 omp_prog.c
  login2$  ifort --openmp --xhost --O3 omp_prog.f90
  login2$  idev ...
  ```
- Run on compute node (or in batch script).
  ```
  c559-001$  export MIC_PREFIX=MIC
  c559-001$  export OMP_NUM_THREADS=16
  c559-001$  export MIC_OMP_NUM_THREADS=240
  c559-001$  ./a.out
  ```
- Use KMP_AFFINITY when thread count < 4*core count.
  Tells runtime to find MIC_ prefixed variables, strip off MIC_ and use them on MIC.

"C559-001$" is the shell prompt for a compute node (host+mic) after executing `idev`
Compiler Assisted Offload

• Compiler looks for `offload` directive everywhere:
  – Before blocks, functions (subroutines), statements
  – For global variables and function declarations
  – As stand-alone directives for data transfer and waits

• Target( `mic : dev_id` )

```plaintext
target(mic)       : Execute on runtime selected MIC, on cpu if error or not available
target(mic:-1)    : Execute on runtime selected MIC, fail otherwise
target(mic:0-n)   : Execute on dev_id=mod(#, no. of coprocs), fail otherwise
```

With more than 1 MIC, use `dev_id` with: `offload`, `offload_transfer`, `offload_wait`
Compiler Assisted Offload: Example

```c
int main()
{
...
#pragma offload target(mic:0)
{
  #pragma omp parallel for
  for (i=0; i<N;i+)
  {
    a[i]=sin(b[i])+cos(c[i]);
  }
}
...
}
```

```fortran
program main
...
!dir$ offload begin target(mic:0)
!$omp parallel do
do i = 1,N
  a(i)=sin(b(i))+cos(c(i))
end do
!dir$ end offload
...
end program
```

Data (a, b, and c) within lexical scope are moved implicitly.

- C/C++ use `{...}` (curly braces) to mark a block
- Fortran use `begin` and `!dir$ end offload` to mark block
The Offload Preparation

- Code is instrumented with directives.

- Compiler creates a CPU binary and a MIC binary for offloaded code block.

- Loader places both binaries in a single file. (→ a.out)

- During CPU execution of the application an encountered offload code block is executed on a coprocessor (through runtime), subject to the constraints of the target specifier…
The basic operations of an offload rely on interaction with the runtime to:

- Detect a target phi coprocessor
- Allocate memory space on the coprocessor
- Transfer data from the host to the coprocessor
- Execute offload binary on coprocessor
- Transfer data from the coprocessor back to the host
- Deallocate space on coprocessor

Binaries are moved on first offload
Offload Directive: Syntax

C/C++

Fortran

```
#pragma offload specifier [ [,] specifier ]
!dir$ offload specifier [ [,] specifier ]
```

**specifier:**

- **target** (targ-name [:dev_id])
- **if** (if-specifier) or **mandatory**
- **signal** (tag) **wait** (tag)
- **data_specifier**(...)

Intel calls this “offload-parameter”.

“clauses”
Offload Directive: DataSpecifier

data_specifier:

\texttt{in} (identifier \{\texttt{[\texttt{[},]}identifier\texttt{\ldots]}\} [: modifier \{\texttt{[\texttt{[},]}modifier\texttt{\ldots]}\})

\texttt{out} ("")

\texttt{inout} ("")

\texttt{nocopy} ("")

For explicit data transfers

\begin{itemize}
  \item \texttt{variables}
  \item \texttt{arrays}
  \item \texttt{\ldots}
  \item \texttt{length()}
  \item \texttt{alloc_if()}
  \item \texttt{free_if()}
  \item \texttt{align}
  \item \texttt{storage handlers}
\end{itemize}
Offload Directive: Summary

C/C++ starts with: #pragma ...
Fortran starts with: !dir$ ...

offload*
offload_attribute
offload_transfer
offload_wait

Stand Alone directives (no offload code)

Specifies MIC vars & functions
data Host ↔ MIC
Wait for async. offload

* Fortran uses offload begin ... end offload, C/C++ uses {...}

__attribute__ and __declspec “decorations” can be used in lieu of offload_attribute in C/C++.
Use !dir$ attributes list in Fortran.
Data Transfers

• If you know the intent of data usage, minimize unnecessary transfers with in/out/inout data specifiers.

```
#pragma offload target(mic [:dev_id]) data_specifier(identifier_list) //syntax

#pragma offload target(mic) in( b, c ) // Only copy b and c into MIC
#pragma offload target(mic) out(a ) // Only return a
#pragma offload target(mic) inout(d ) // Default, copy into and out of
```
int main(){

    #pragma offload target(mic) \ 
    in(b,c) out(a)

    {
        #pragma omp parallel for
        for (i=0; i<N;i+){
            a[i]=sin(b[i])+cos(c[i]);
        }
    }

    ...
}

program main

... 
!dir$ offload begin target(mic) & 
    in(b,c) out(a)

!$omp parallel do
    do i = 1,N
        a(i)=sin(b(i))+cos(c(i))
    end do
!dir$ end offload

... 
end program
Offloading Functions and Globals: Attributes

- “Decorate” all functions **used** in offloads with a target “attribute”.
- Likewise with globals

```c
__attribute__((target(mic))) <followed by function/global declaration> C/C++
__declspec(target(mic)) <followed by function/global declaration>
!dir$ attributes offload:mic :: <function/subroutine name or variables> F90
```
Offloading Functions and Globals: Example

C/C++

```c
__declspec(target(mic))
int global = 0;

__declspec(target(mic))
int foo()
{
    return ++global;
}

main()
{
    int i;
    #pragma offload target(mic) inout(global)
    { i = foo(); }

    printf("global: i=%d:%d both=1\n", global, i);
}
```

F90

```fortran
module mydat
!
  !dir$ attributes offload:mic :: global
  integer :: global = 0
end module mydat
!
  !dir$ attributes offload:mic :: foo
  integer function foo
  use mydat
  global = global + 1
  foo = global
end function foo
program main
  use mydat
  integer i
  integer,external :: foo
!
  !dir$ attributes offload:mic :: foo
!
  !dir$ offload target(mic:0) inout(global)
  i = foo()
  print *, "global: i=", global, i,"(both=1)"
end program main
```
Offload Attributes Example

- Offload attributes can be applied to an **entire file** through a compiler option:

```
icpc | icc | ifort  -c -offload-attribute-target=mic  my_fun.cpp | c | f90
icpc | icc | ifort  my_fun.o                      my_app.cpp | c | f90
```

- **C/C++** has file scoping, **FORTRAN** does not:

```c
#pragma offload_attribute(push, target(mic))
void fun1(int i) {i=i+1;}
void fun2(int j) {j=j+2;}
#pragma offload_attribute(pop)
```

```fortran
module my_globs
!dir$ options /offload_attribute_target=mic
real, allocatable :: in1(:), in2(:), out1(:), out2(:)
!dir$ end options
end module
```
Offloading Pointer Data

- C pointer to contiguous data requires \texttt{length modifier} (default copy is 1 element).
- Not required for Fortran allocated arrays.

```c
... 

a=(double *) malloc(N *sizeof(double));
b=(double *) malloc(N *sizeof(double));
c=(double *) malloc(N *sizeof(double));
d=(double *) malloc(M *sizeof(double));
e=(double *) malloc(N*2*sizeof(double));

#pragma offload target(mic:0) in( a,b,c : length( N ) ) // pointers a, b & c, length N
#pragma offload target(mic:0) out( d : length( M ) ) // pointer d has length M
#pragma offload target(mic) inout( e : length(2*N) ) // pointer e has length of N\times2
```

Alignment might be important
Persistent Data

• **Default implicit and explicit** behavior:
  – allocate space for all data **before offload**
  – deallocate (free) **on offload completion**.

\[
\text{alloc\_if( } \text{logic\_expression } \text{ )} \quad - \quad \text{if true allocate space at begin}
\]
\[
\text{free\_if( } \text{logic\_expression } \text{ )} \quad - \quad \text{if true free space at end}
\]

• The stand-alone **offload\_transfer** directive allows data management (data specifiers) without a code block.
Persistent Data

- Fortran and C/C++ syntaxes are identical, except:
  - Sentinels are different: #pragma versus !dir$
  - Truth variables: Fortran: logical .true./.false. C/C++ int 1/0

```c
#pragma offload data_specifier( identifier(s): alloc_if(TorF) free_if(TorF) )
```

```c
#pragma offload ... in( a : alloc_if(1) free_if(0) ) //allocate space, don’t free at end
{...}
#pragma offload ... inout( a : alloc_if(0) free_if(0) ) //don’t allocate, don’t free at end
{...}
#pragma offload ... out( a : alloc_if(0) free_if(1) ) //don’t allocate, free at end
{...}
```

```c
#pragma offload_transfer... in( a : alloc_if(1) free_if(0) ) //allocate space, don’t free at end
...`
```

```c
#pragma offload_transfer... out( a : alloc_if(0) free_if(1) ) //don’t allocate, free space at end
```

... == target(mic)
## Alloc/Free Truth Table

<table>
<thead>
<tr>
<th>Allocation Operation</th>
<th>Deallocation (Free) Operation</th>
<th>Operations Performed (Use Case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc_if(true)</td>
<td>free_if(true)</td>
<td>This is the default when no storage operations are specified. Allocate space at beginning, free at end.</td>
</tr>
<tr>
<td>alloc_if(true)</td>
<td>free_if(false)</td>
<td>Allocate space, don’t free (make space available on device, and retain for future use).</td>
</tr>
<tr>
<td>alloc_if(false)</td>
<td>free_if(true)</td>
<td>don’t allocate, but free (reuse device storage, but will not need later)</td>
</tr>
<tr>
<td>alloc_if(false)</td>
<td>free_if(false)</td>
<td>don’t allocate, don’t free (reuse device storage, and leave for future use)</td>
</tr>
</tbody>
</table>
Asynchronous Offloading

- Default behavior: CPU process waits for offload to complete.
- **Signal and wait specifiers** allow CPU to continue executing after the offload code block, once the runtime is notified to perform the offload (i.e. offload becomes asynchronous).
- **offload_wait** is a stand-alone directive (no code block).
- A **device id** is mandatory with an asynchronous clause.
- **tag** is pointer or address in C/C++, and an 8-byte integer in Fortran.
- The tag must be set to a unique value >1 in Fortran.

```c
#pragma offload target(mic:dev_id) ... signal(tag)
#pragma offload target(mic:dev_id) ... wait(tag [,tag, ...])
#pragma offload_wait target(mic:dev_id) ... wait(tag [,tag, ...])
```

(Only one tag is allowed for signal. A wait can block on multiple signal tags.)
Asynchronous Offloading: Example

Offload events are identified by unique value of tag.
F90: `signal(var)`
C/C++: `signal(&var)`

Wait/signal can have only a single tag.

Directives can have wait and signal specifiers.

```c
#define N 10000
__attribute__((target(mic:0))) void work(int, int, int, int *);

int main(){
  int sig1=1, i, knt=1, *a, NSm, NEm, NSc, NEC;
  a=(int*)malloc(N*sizeof(int));
  do{
    NSm=0; NEm=N/2;
    #pragma offload target(mic:0) signal(&sig1) \ 
      inout(a:length(NEm-1))
    work(knt,NSm,NEm, a);
    NSc=N/2; NEC=N;
    work(knt,NSc,NEc, a);
    #pragma offload_wait target(mic:0) wait(&sig1)
    knt=knt+1;
  }while (knt < 10);
  // CPU & MIC work on different parts of a
}
```
**Offload Thread Placement**

Controlled through environment variable: `KMP_AFFINITY=<type>`

<table>
<thead>
<tr>
<th>Type</th>
<th>Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>compact</td>
<td>pack threads close to each other</td>
</tr>
<tr>
<td>scatter</td>
<td>Round-Robin threads to cores</td>
</tr>
<tr>
<td>balanced</td>
<td>keep OMP thread ids consecutive (MIC only)</td>
</tr>
<tr>
<td>explicit</td>
<td>use the proclist modifier to pin threads</td>
</tr>
<tr>
<td>none</td>
<td>does not pin threads</td>
</tr>
</tbody>
</table>

System with only 4 MIC CORES

- **compact**
  - 0 1 2 3
  - 4 5 6 7

- **scatter**
  - 0 4
  - 1 5
  - 2 6
  - 3 7

- **balanced**
  - 0 1
  - 2 3
  - 4 5
  - 6 7

Explicit, `proclist=[1,2,3,4,9,10,11,12]`

- 0 1 2 3
- 8 9 10 11

**CPU id (4/core)**

**software thread id**

8 OpenMP threads

Offload automatically avoids last core (HW threads 0,241,242,243), and with scatter/compact.

Be careful if you pin threads with `explicit`, offload communication/transfers occur on last core.
Offload From Within A Parallel Region

MPI process, master thread

OMP_NUM_THREADS=4

Generate parallel region

1 thread offloads nowait

thread will assist when done

other threads jump to workshare

No implied barrier at end of this single

do workshare on cpu

wait

C/C++

```
#pragma omp parallel
{
  #pragma omp single nowait
  #pragma offload target(mic)
  { foo(); }

  #pragma omp for schedule(dynamic)
  for(i=0; i<N; i++) {...}
}
```

F90

```
!$omp parallel
 !$omp single
 !$omp do schedule(dynamic)
   call foo();
 !$omp end do
 !$omp end parallel
```
#include <omp.h>
#include <stdio.h>

int main() {
    const int N=100000000;
    int i, nt, N_mic, N_cpu;
    float *a;

    a = (float *) malloc(N*sizeof(float));
    for(i=0;i<N;i++) a[i]=-1.0; a[0]=1.0;

    N_mic = N/2; N_cpu = N/2;
    nt = 16; omp_set_num_threads(nt);

    #pragma omp parallel
    {
        #pragma omp single nowait
        {
            #pragma offload target(MIC:0) out(a:length(N_MIC))
            #pragma omp parallel for
            for(i=0;i<N_mic;i++) { a[i]=(float)i; }
        }

        #pragma omp for schedule(dynamic,N/nt)
        for(i=N_cpu;i<N;i++) { a[i]=(float)i; }
    }

    printf("a[0],a[N-1] %f %f\n",a[0],a[N-1]);
}
Nesting Parallel Regions

- OpenMP 3.0 supports nested parallelism, older implementations may ignore the nesting and serialize inner parallel regions.
- A nested parallel region can specify any number of threads to be used for the thread team, new id’s are assigned. Scheduling: static, etc.
omp_set_nested(1);
omp_set_max_active_levels(2);
omp_set_num_threads(2);

#pragma omp parallel
{
    printf("reporting in from %d\n", \
            omp_get_thread_num());

    #pragma omp sections
    {
        #pragma omp section
        {

            #pragma offload target(mic)
            bar(1);

        }

        #pragma omp section
        {

            #pragma omp parallel for num_threads(3)
            for(i=2;i<5;i++) {bar(i);}

        }
    }
}

Nesting Example

Sections allows 1 generating thread in each section.

Nested level re-defines a thread team with new thread ids. (Worksharing team is no longer dependent upon original parallel region team size.) Scheduling can be static!
## Compiler Options and Env Vars

### Compiler Options

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-no-offload</td>
<td>Ignore offload directives</td>
</tr>
<tr>
<td>-offload-attribute-target=mic</td>
<td>Flag every global data object and routine with the offload attribute</td>
</tr>
<tr>
<td>-opt-report-phase=offload</td>
<td>Optimization phase report for offload</td>
</tr>
<tr>
<td>-offload-option,mic,compiler,&quot;option list&quot;</td>
<td>Compiler options for MIC</td>
</tr>
<tr>
<td>-offload-option, ld,compiler,&quot;option list&quot;</td>
<td>Loader options for MIC</td>
</tr>
</tbody>
</table>

### Environment Variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIC_ENV_PREFIX</td>
<td>(usually =MIC) Controls variables passed to MIC.</td>
</tr>
<tr>
<td>OFFLOAD_REPORT</td>
<td>(=1</td>
</tr>
<tr>
<td>MIC_STACKSIZE</td>
<td>Specifies the stack size of the main thread for the offload. (default =12M)</td>
</tr>
<tr>
<td>MKL_MIC_ENABLE</td>
<td>(=1) Sets automatic offloading on.</td>
</tr>
<tr>
<td>MKL_MIC_WORKDIVISION</td>
<td>Sets fraction of automatic offload work for MIC/HOST.</td>
</tr>
<tr>
<td>MKL_HOST_WORKDIVISION</td>
<td></td>
</tr>
</tbody>
</table>

---

TACC
MIC Information


- **Programming and Compiling for Intel® Many Integrated Core Architecture**

- Intel Compiler Manuals: [C/C++] [Fortran](http://software.intel.com/mic-developer)
  (Key Features ➔ Intel ® MIC Architecture)

- Example code: /opt/apps/intel/13/composer_xe_2013.2.146/Samples


References

In these Compiler User Guides for offload details GO TO:
Key Features→Intel MIC Architecture→Programming for Intel MIC Architecture


Intel MIC Programming and Computing


Developer’s Guide


MKL