ver DJ2014-02
as 13 Feb 2014

PowerPoint original
available on request

Intel Xeon Phi MIC Offload Programming Models

Kent Milfeld

July 2014

Key References

- Jeffers and Reinders, Intel Xeon Phi…
  - but some material is no longer current
- Intel Developer Zone
  - http://software.intel.com/en-us/mic-developer
  - http://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features
- Stampede User Guide and related TACC resources
  - Search User Guide for "Advanced Offload" and follow link

Other specific recommendations throughout this presentation

Overview

Basic Concepts

Three Offload Models

Issues and Recommendations

Source code available on Stampede:

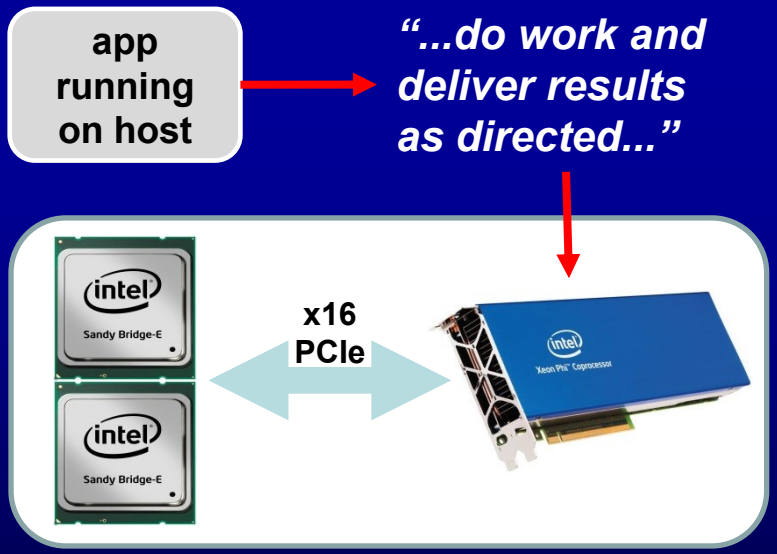
```
tar xvf ~train00/offload_demos.tar
```

Project code: **`TRAINING-HPC`** (TACC Portal) or **`TG-TRA140011`** (XSEDE Portal)

Offloading:
MIC as assistant processor

A program running on the host "offloads" work by directing the MIC to execute a specified block of code. The host also directs the exchange of data between host and MIC.

Ideally, the host stays active while the MIC coprocessor does its assigned work.



Offload Models

- Compiler Assisted Offload (CAO)
  - Explicit
    - Programmer explicitly directs data movement and code execution
  - Implicit
    - Programmer marks some data as “shared” in the virtual sense
    - Runtime automatically synchronizes values between host and MIC
- Automatic Offload (AO)
  - Computationally intensive calls to Intel Math Kernel Library (MKL)
  - MKL automatically manages details
  - More than offload: work division across host and MIC!

Explicit Model: Direct Control of Data Movement

- aka Copyin/Copyout, Non-Shared, COI*
- Available for C/C++ and Fortran
- Supports simple ("bitwise copyable") data structures (think 1d arrays of scalars)

*Coprocessor Offload Infrastructure

Explicit Offload

F90

```
program main

  use omp_lib

  integer :: nprocs


  nprocs = omp_get_num_procs()

  print*, "procs: ", nprocs

end program
```

```
ifort -openmp off00host.f90

icc -openmp off00host.c
```

C/C++

```
#include <stdio.h>
#include <omp.h>

int main( void )  {

    int totalProcs;


    totalProcs = omp_get_num_procs();

    printf( "procs: %d\n", totalProcs );
    return 0;

}
```

Simple Fortran and C codes that each return "procs: 16" on Sandy Bridge host…

TACC

Explicit Offload

F90

```
program main

  use omp_lib

  integer :: nprocs

  !dir$ offload target(mic)
  nprocs = omp_get_num_procs()

  print*, "procs: ", nprocs

end program
```

offload directive

runs on MIC

runs on host

```
ifort -openmp off01simple.f90
```

```
icc -openmp off01simple.c
```

C/C++

```
#include <stdio.h>
#include <omp.h>

int main( void )  {

  int totalProcs;

  #pragma offload target(mic)
  totalProcs = omp_get_num_procs();

  printf( "procs: %d\n", totalProcs );
  return 0;

}
```

offload pragma

runs on MIC

runs on host

Add a one-line directive/pragma that offloads to the MIC the one line of executable code that occurs below it…

…codes now return "procs: 240"…

TACC

Explicit Offload

F90

```
program main

  use omp_lib

  integer :: nprocs

  !dir$ offload target(mic)
  nprocs = omp_get_num_procs()

  print*, "procs: ", nprocs

end program
```

don't use "-mmic"

```
ifort -openmp off01simple.f90
```

```
icc -openmp off01simple.c
```

C/C++

```
#include <stdio.h>
#include <omp.h>

int main( void )  {

  int totalProcs;

  #pragma offload target(mic)
  totalProcs = omp_get_num_procs();

  printf( "procs: %d\n", totalProcs );
  return 0;

}
```

Don't even need to change the compile line…

Explicit Offload

F90

```
program main

  use omp_lib

  integer :: nprocs

  !dir$ offload target(mic)
  nprocs = omp_get_num_procs()

  print*, "procs: ", nprocs

end program
```

off01simple

C/C++

```
#include <stdio.h>
#include <omp.h>

int main( void )  {

  int totalProcs;

  #pragma offload target(mic)
  totalProcs = omp_get_num_procs();

  printf( "procs: %d\n", totalProcs );
  return 0;

}
```

Not asynchronous (yet): the host pauses until MIC is finished.

Explicit Offload

F90

```
!dir$ offload begin target(mic)
   nprocs     = omp_get_num_procs()
   maxthreads = omp_get_max_threads()
!dir$ end offload
```

off02block

Can offload a block of code (generally safer than the one-line approach)…

C/C++

```
#pragma offload target(mic)
   {
     totalProcs = omp_get_num_procs();
     maxThreads = omp_get_max_threads();
   }
```

TACC

Explicit Offload

F90

```
program main

  integer, parameter :: N = 500000   ! constant
  real               :: a(N)         ! on stack

  !dir$ offload target(mic)
    !$omp parallel do
      do i=1,N
        a(i) = real(i)
      end do
    !$omp end parallel do
  ...
```

off03omp

C/C++

```
int main( void ) {

  double a[500000];
    // on the stack; literal here is important
  int i;

  #pragma offload target(mic)
    #pragma omp parallel for
      for ( i=0; i<500000; i++ )  {
        a[i] = (double)i;
      }
  ...
```

…or an OpenMP region defined by an omp directive…

Explicit Offload

```
integer function successor( m )
  ...

program main
  ...
  integer :: successor
  ...

  !dir$ offload target(mic)
    n = successor( m )
```

F90

off04proc

```
int  successor( int  m );
void increment( int* pm );

int main( void ) {

    int i;

    #pragma offload target(mic)
      {
        i = successor( 123 );
        increment( &i );
      }
```

C/C++

…or procedure(s) defined by the programmer

(though now there's another step)…

Explicit Offload

```
!dir$ attributes offload:mic :: successor
integer function successor( m )
  ...

program main
  ...
  integer :: successor
  !dir$ attributes offload:mic :: successor
  ...

  !dir$ offload target(mic)
    n = successor( m )
```

F90

off04proc

...mark prototypes to tell compiler to build executable code on both sides...

```
__declspec( target(mic) ) int  successor( int   m );
__declspec( target(mic) ) void increment( int* pm );

int main( void ) {

    int i;

    #pragma offload target(mic)
      {
        i = successor( 123 );
        increment( &i );
      }
```

C/C++

Explicit Offload

```
module mymodvars
  !dir$ attributes offload:mic :: mymoduleint
  integer                      :: mymoduleint
end module mymodvars

program main

  use mymodvars
  implicit  none

  integer        :: mylocalint = 123
  integer, save  :: mysaveint !no decoration required
```

F90

off05global

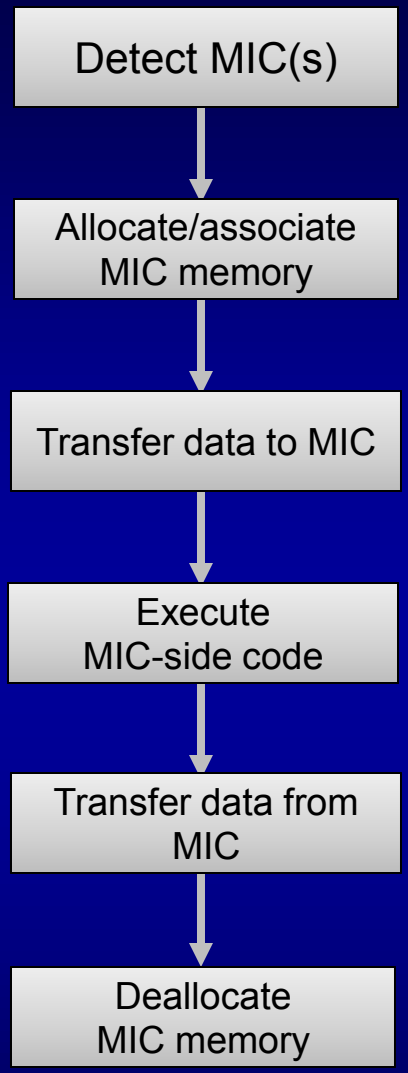
…and mark all global and static identifiers...

```
__declspec( target(mic) ) int myGlobalInt;

int main( void ) {

  int myLocalInt = 123;
  __declspec( target(mic) ) static int myStaticInt;
```

C/C++

Controlling the Offload

Additional decorations (clauses, attributes, specifiers, keywords) give the programmer a high degree of control over all steps in the process.



Explicit Offload

```
!dir$ offload target(mic)
  !$omp parallel do
    do i=1,N
      a(i) = real(i)
    end do
  !$omp end parallel do
```

F90

off03omp

…"target(mic)"
means
"find a MIC,
any ol' MIC"…

```
#pragma offload target(mic)
  #pragma omp parallel for
    for ( i=0; i<500000; i++ )  {
      a[i] = (double)i;
    }
```

C/C++

Explicit Offload

```
!dir$ offload target(mic:0)
  !$omp parallel do
    do i=1,N
      a(i) = real(i)
    end do
  !$omp end parallel do
```

F90

off03omp

…"target(mic:0)"
or "target(mic:i)"
means
"find a specific MIC"…

```
#pragma offload target(mic:0)
  #pragma omp parallel for
    for ( i=0; i<500000; i++ )  {
      a[i] = (double)i;
    }
```

C/C++

Explicit Offload

```
 integer, parameter :: N = 100000                ! constant
 real               :: a(N), b(N), c(N), d(N)    ! on stack
  ...
!dir$ offload target(mic)          &
 in( a ), out( c, d ), inout( b )
    !$omp parallel do
      do i=1,N
        c(i) =  a(i) + b(i)
        d(i) =  a(i) - b(i)
        b(i) = -b(i)
      end do
    !$omp end parallel do
```

F90

off06stack

…control data transfer between host and MIC...

```
double a[100000], b[100000], c[100000], d[100000];
  // on the stack; literal is necessary for now

...

#pragma offload target(mic)          \
  in( a ), out( c, d ), inout( b )
    #pragma omp parallel for
      for ( i=0; i<100000; i++ )  {
        c[i] =  a[i] + b[i];
        d[i] =  a[i] - b[i];
        b[i] = -b[i];
      }
```

C/C++

Explicit Offload

```
real, allocatable  :: a(:), b(:)
integer, parameter :: N = 5000000

allocate(  a(N), b(N)  )
...

!
!dir$ offload target(mic)                              &
 in(  a : alloc_if(.true.) free_if(.true.)  ), &
 out( b : alloc_if(.true.) free_if(.false.) )
  !$omp parallel do
    do i=1,N
      b(i) = 2.0 * a(i)
    end do
  !$omp end parallel do
```

(see source file for alignment directives)

F90

off07heap

…manage MIC memory and its association with dynamically allocated memory on the host...

```
int N = 5000000;
double *a, *b;

a = ( double* ) memalign( 64, N*sizeof(double) );
b = ( double* ) memalign( 64, N*sizeof(double) );
...
#pragma offload target(mic)                          \
 in(  a :                alloc_if(1) free_if(1) ), \
 out( b :                alloc_if(1) free_if(0) )
  #pragma omp parallel for
    for ( i=0; i<N; i++ )  {
      b[i] = 2.0 * a[i];
    }
```

C/C++

Explicit Offload

```
real, allocatable  :: a(:), b(:)
integer, parameter :: N = 5000000

allocate(  a(N), b(N)  )
...

! Fortran allocatable arrays don't need length attribute...
!dir$ offload target(mic)                              &
 in(  a : alloc_if(.true.) free_if(.true.)  ), &
 out( b : alloc_if(.true.) free_if(.false.) )
  !$omp parallel do
    do i=1,N
      b(i) = 2.0 * a(i)
    end do
  !$omp end parallel do
```

(see source file for alignment directives)

F90

off07heap

…Dynamically allocated arrays in C/C++ require an additional "length" attribute...

```
int N = 5000000;
double *a, *b;

a = ( double* ) memalign( 64, N*sizeof(double) );
b = ( double* ) memalign( 64, N*sizeof(double) );
...
#pragma offload target(mic)                          \
 in(  a : length(N) alloc_if(1) free_if(1) ), \
 out( b : length(N) alloc_if(1) free_if(0) )
  #pragma omp parallel for
    for ( i=0; i<N; i++ )  {
      b[i] = 2.0 * a[i];
    }
```

C/C++

Explicit Offload

```
integer  :: n = 123

!dir$ offload begin target(mic:0)  signal( n )
  call incrementslowly( n )
!dir$ end offload

...

print *, "  n: ", n
```

for now, any initialized 4 or 8 byte integer

F90

off08asynch

…Asynchronous offload with "signal": work on host continues while offload proceeds...

```
int n = 123;

#pragma offload target(mic:0)  signal( &n )
  incrementSlowly( &n );

...

printf( "\n\tn = %d \n", n );
```

pointer to any initialized variable

C/C++

Explicit Offload

```
integer  :: n = 123

!dir$ offload begin target(mic:0)  signal( n )
  call incrementslowly( n )
!dir$ end offload

...
!dir$ offload_wait target(mic:0)  wait( n )



print *, "  n: ", n
```

F90

off09transfer

…offload_wait pauses the host but initiates no new work on MIC...

```
int n = 123;

#pragma offload target(mic:0)  signal( &n )
  incrementSlowly( &n );

...
#pragma offload_wait target(mic:0)  wait( &n )
```

device number required

```
printf( "\n\tn = %d \n", n );
```

C/C++

Explicit Offload

```
integer  :: n = 123

!dir$ offload begin target(mic:0)  signal( n )
  call incrementslowly( n )
!dir$ end offload

...
!dir$ offload begin target(mic:0) wait( n )
  print *, "  procs: ", omp_get_num_procs()
  call flush(0)
!dir$ end offload

print *, "  n: ", n
```

F90

off09transfer

```
int n = 123;

#pragma offload target(mic:0)  signal( &n )
  incrementSlowly( &n );

...
#pragma offload target(mic:0)  wait( &n )
  {
  printf( "\n\tprocs: %d\n", omp_get_num_procs() );
  fflush(0);
  }

printf( "\n\tn = %d \n", n );
```

C/C++

…classical offload (as opposed to offload_wait) will offload the next line/block of code...

...both constructs need a wait() clause with tag

Explicit Offload

```
!dir$ offload_transfer target(mic:0)                    &
  in(     a : alloc_if(.true.) free_if(.false.) ), &
  nocopy( b : alloc_if(.true.) free_if(.false.) )  &
  signal( tag1 )
```

these examples are asynchronous

F90

off09transfer

…offload_transfer is a data-only offload (no executable code sent to MIC)...

...use it to move data and manage memory (alloc and free)...

```
#pragma offload_transfer target(mic:0)                  \
  in(     a : length(N) alloc_if(1) free_if(0) ), \
  nocopy( b : length(N) alloc_if(1) free_if(0) )  \
  signal( &tag1 )
```

these examples are asynchronous

C/C++

TACC

Summary

functions, global space

transfer

offload

space alloc
data motion
async

wait

attributes/declspec

offload_transfer

alloc_if free_if
in, out, inout, length
signal, wait

offload_wait

Data Transfers
Asynchronous Execution
Persistence

TACC

Detecting/Monitoring Offload

- `export OFFLOAD_REPORT=2  # or 1, 3`
- Compile time info: `-opt-report-phase=offload`
- `__MIC__` macro defined on device
  - can be used for conditional compilation
  - use only within offloaded procedure
  - use capitalized “F90” suffix to pre-process during compilation
- `ssh mic0` (not mic:0) and run top
  - offload processes owned by “micuser”

Other Key Environment Variables

OMP_NUM_THREADS

- default is 1; that's probably not what you want!

MIC_OMP_NUM_THREADS

- default behavior is 244 (var undefined); you definitely don't want that

MIC_STACKSIZE

- default is only 12MB

MIC_KMP_AFFINITY and other performance-related settings

Offload: making it worthwhile

- Enough computation to justify data movement
- High degree of parallelism
  - threading, vectorization
- Work division: keep host and MIC active
  - asynchronous directives
  - offloads “from OpenMP regions”
- Intelligent data management and alignment
  - persistent data on MIC when possible

http://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features

Automatic Offload (AO)

- Feature of Intel Math Kernel Library (MKL)
  - growing list of computationally intensive functions
  - xGEMM and variants; also LU, QR, Cholesky
  - kicks in at appropriate size thresholds
    - (e.g. SGEMM: (M,N,K) = (2048, 2048, 256)
  - http://software.intel.com/en-us/articles/intel-mkl-automatic-offload-enabled-functions-for-intel-xeon-phi-coprocessors
- Essentially no programmer action required
  - does more than offload: work division across host and MIC
  - http://software.intel.com/en-us/articles/performance-tips-of-using-intel-mkl-on-intel-xeon-phi-coprocessor

Automatic Offload

```
...

M =  8000
P =  9000
N = 10000

...

CALL DGEMM( 'N','N',M,N,P,ALPHA,A,M,B,P,BETA,C,M )
```

Fortran

ao_intel

…call one of the supported MKL functions for sufficiently large matrices...

```
#include "mkl.h"

...
m = 8000;
p = 9000;
n = 10000;

...

cblas_dgemm(
  CblasRowMajor, CblasNoTrans, CblasNoTrans,
  m, n, p, alpha, A, p, B, n, beta, C, n );
```

C/C++

TACC

Automatic Offload

Must set 1 env. var for this to work. See next page..

```
                        ifort -openmp -mkl main.f
...

M =  8000
P =  9000
N = 10000

...

CALL DGEMM( 'N','N',M,N,P,ALPHA,A,M,B,P,BETA,C,M )
```

Fortran

ao_intel

```
#include "mkl.h"
                              icc -openmp -mkl main.c
...
m = 8000;
p = 9000;
n = 10000;

...

cblas_dgemm(
  CblasRowMajor, CblasNoTrans, CblasNoTrans,
  m, n, p, alpha, A, p, B, n, beta, C, n );
```

C/C++

…use Intel compiler and link to MKL...

...ldd should show libmkl_intel_thread...

Automatic Offload

- Set at least three environment variables before launching your code:

```
export MKL_MIC_ENABLE=1
export OMP_NUM_THREADS=16
export MIC_OMP_NUM_THREADS=240
```

- Other environment variables provide additional fine-grained control over host-MIC work division et al.

http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_userguide_lnx/GUID-3DC4FC7D-A1E4-423D-9C0C-06AB265FFA86.htm

MKL Offload: Other Opportunities

- Apps that call MKL “under the hood” can exploit AO
  - Need to build with Intel and link to threaded MKL
    - In other words, use `-mkl` or `-mkl=parallel`; do not use `-mkl=sequential`
  - Matlab on Stampede:

    ```
    export BLAS_VERSION=$TACC_MKL_LIB/libmkl_rt.so
    ```

  - AO for R temporarily available with "module load R_mkl"
    - New AO-enabled parallel R coming soon
  - AO for Python: coming soon to Stampede
- Can also explicitly offload MKL functions

Implicit Offload: Virtual Shared Memory

- aka Shared Memory, MYO*
- Programmer marks data as shared between host and MIC; runtime manages synchronization
- Supports “arbitrarily complex” data structures, including objects and their methods
- Available only for C/C++

*”Mine-Yours-Ours”

Implicit Offload

_Cilk_shared marks global data as usable and synchronized between host and MIC. Runtime handles the details.

C/C++ only

```
int           _Cilk_shared mySharedInt;
COrderedPair  _Cilk_shared mySharedP1;
```

Implicit Offload

_Cilk_shared also marks functions as suitable for offload. Signatures in prototypes and definitions determine how shared and unshared functions operate on shared data.

C/C++ only

```
int          _Cilk_shared mySharedInt;
COrderedPair _Cilk_shared mySharedP1;

int  _Cilk_shared incrementByReturn( _Cilk_shared int  n );
void _Cilk_shared incrementByRef(    _Cilk_shared int& n );
void _Cilk_shared modObjBySharedPtr(
  COrderedPair _Cilk_shared *ptrToShared );
```

Implicit Offload

_Cilk_offload executes a shared function on MIC (does not operate on a block of code)

C/C++ only

```
int          _Cilk_shared mySharedInt;
COrderedPair _Cilk_shared mySharedP1;

int  _Cilk_shared incrementByReturn( _Cilk_shared int  n );
void _Cilk_shared incrementByRef(    _Cilk_shared int& n );
void _Cilk_shared modObjBySharedPtr(
  COrderedPair _Cilk_shared *ptrToShared );

...
mySharedInt
  = _Cilk_offload incrementByReturn( mySharedInt );

_Cilk_offload modObjBySharedPtr( &mySharedP1 );
```

Implicit Offload

C/C++ only

```
int           _Cilk_shared mySharedInt;
COrderedPair  _Cilk_shared mySharedP1;

int  _Cilk_shared incrementByReturn( int n );
void _Cilk_shared incrementByRef( _Cilk_shared int& n );
void _Cilk_shared modObjBySharedPtr(
  COrderedPair _Cilk_shared *ptrToShared );

...
mySharedInt
  = _Cilk_offload incrementByReturn( mySharedInt );

_Cilk_offload modObjBySharedPtr( &mySharedP1 );
```

But the devil's in the details...

Implicit Offload: Issues

- Shared data must be global
- Shared vs unshared datatypes
  - need for casting and overloading (equality, copy constructors)
- Special memory managers
  - “placement new” to share STL classes
- Infrastructure less stable and mature
  - Intel sample code available, but other resources are sparse
  - we all have a lot to learn about this
- By its nature a little slower than explicit offload

Offload: Issues and Gotchas

- Fast moving target
  - Functionality/syntax varies across compiler versions
  - Documentation often lags behind ground truth
- First offload takes longer
  - Consider an untimed warupMIC offload
- Memory limits
  - ~6.7GB available for heap; 12MB default stack
- Serious File I/O essentially impossible from offload region
  - console output ok; flush buffer
- Optional offload in transition
  - -no-offload compiler flag works on Stampede

Building Static Libraries

- Offload routines are compiled to make MIC & x86_64 objects.
- Use the Intel xiar library with -qoffload-build to build 1 library.

```
c557-800$ icpc -c mylib.cpp
c557-800$ ls
          mylibMIC.o  mylib.o ...     # host & MIC libs created.

c557-800$ xiar  cru -qoffload-build mylib.a mylib.o
          xiar: executing 'ar'
          xiar: executing 'ar'

c557-800$ ls
          mylibMIC.o  mylib.o  mylib.a ... # host & MIC libs in 1 file
c557-800$ icpc driver.cpp mylib.a
```

Building Static Libraries

- How to tell difference between x86-64 and MIC objects:

```
$ file mylib.o
      mylib.o:    ELF 64-bit LSB relocatable, x86-64, ver 1 (GNU/Linux)
$ file mylibMIC.o
      mylibMIC.o: ELF 64-bit LSB relocatable,         ver 1 (GNU/Linux)
```

- Always use optimal performance and reporting options for your application !

```
$ icpc -c -xAVX -vec-report3 -openmp -O3 mylib.cpp
$ xiar  cru -qoffload-build   mylib.a    mylib.o
$ icpc    -xAVX -vec-report3 -openmp -O3 mylib.a driver.cpp
```

How to Write Your Own Blazingly Fast Library of Special Functions for Intel Xeon Phi Coprocessors, Vadim Karpusenko (Colfax International), Andrey Vladimirov (Stanford University), May 3, 2013; http://research.colfaxinternational.com/ See example and pdf file in lab.

Simultaneous Execution on Host and MIC -- SIMPLE

```
!dir$ offload begin target(mic:0) signal(isig)
!$omp parallel num_threads(60)
   call sometime(noff, n1)                    !T1
!$omp end parallel
!dir$ end offload                                  !T0

!$omp parallel num_threads(16)
      call sometime(n, n2)                    !T2
!$omp end parallel

!dir$ offload_wait target(mic:0) wait(isig)
```

MIC

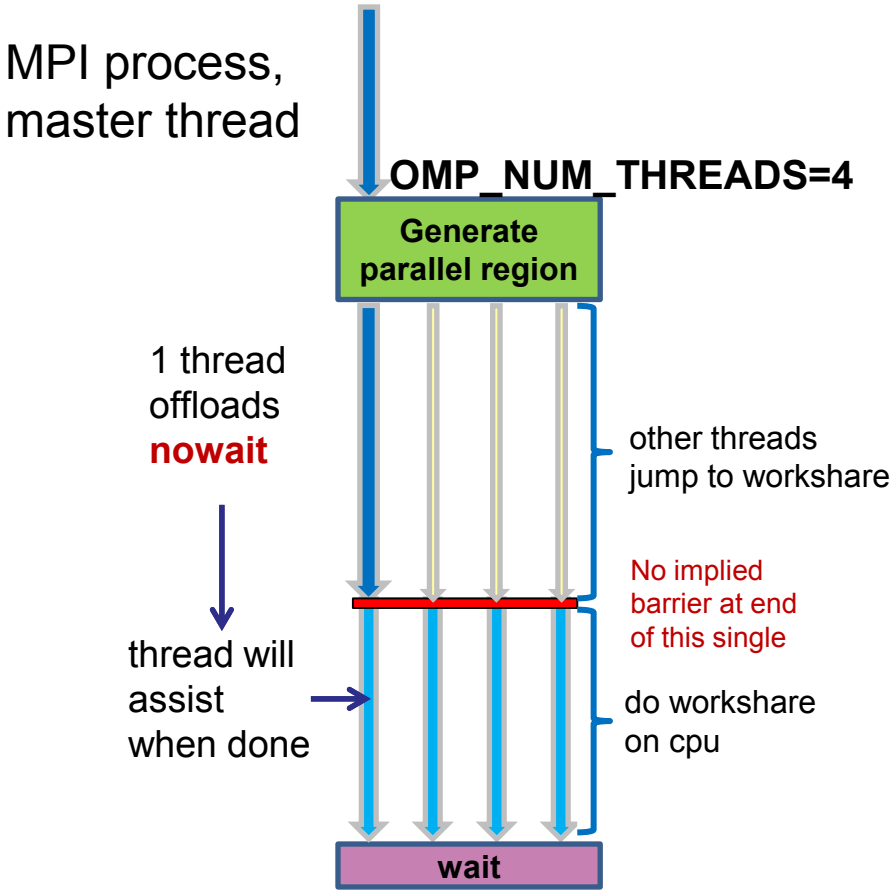
CPU

- Intel async allows offload "generator" thread to execute immediately in host's parallel region.

Time (sec) measured on host:

	Total,	T0	T1	T2
Async off	2.11117		1.12804	0.98314
Async on	1.08984		0.00002	1.01347

Simultaneous Execution within host parallel region



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
   !DIR$ offload target(mic)
     call foo();
   !$omp end single nowait

   !$omp do schedule(dynamic)
     do i=1,N; ...
     end do
!$omp end parallel
```

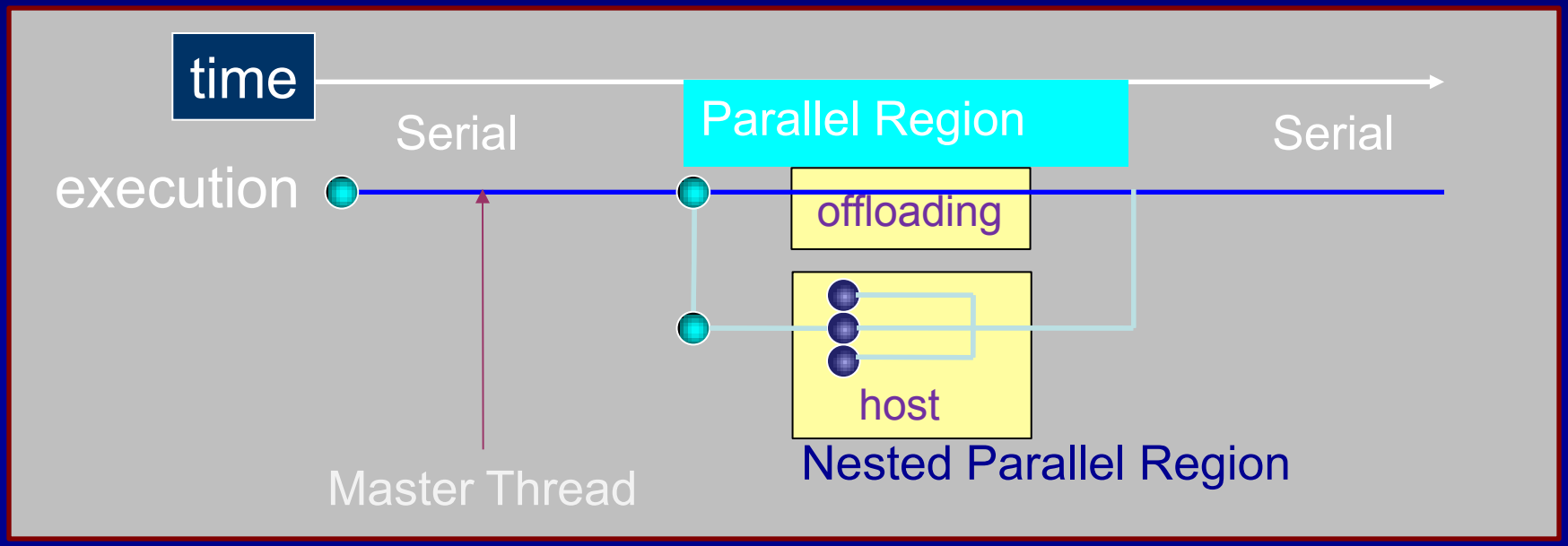
Simultaneous Execution within host parallel region w.o. Async

```
#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]);
}
```

MIC

CPU

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.

Offload in parallel region

```
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);}
      }

    }
  }
```

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!

.

Summary

- Offload may be for you if your app is...
  - computationally intensive
  - highly parallel (threading, vectorization)
- Best practices revolve around...
  - asynchronous operations
  - intelligent data movement (persistence)
- Three models currently supported
  - explicit: simple data structures
  - automatic: computationally-intensive MKL calls
  - implicit: complex data structures (objects and their methods)

Exercise Options (pick and choose)

- Option A: `tar xvf ~train00/offload_lab.tar`
  - Exercise 1: Simple Offload Examples
  - Exercise 2: Data Transfer Optimization
  - Exercise 3: Concurrent and Asynchronous Offloads
- Option B: `tar xvf ~train00/offload_demos.tar`
  - Explicit offload: exercises based on TACC examples from presentation
  - Automatic offload: exercises based on Intel examples from presentation

Project code: **`TRAINING-HPC`** (TACC Portal) or **`TG-TRA140011`** (XSEDE Portal)

Kent Milfeld
milfeld@tacc.utexas.edu
(512) 475-9458

For more information:
www.tacc.utexas.edu

TACC

License