	Subject	Purpose	directory
1	OMP	Serial Compile and Run (CPU, MIC, Offload)	omp_c & omp_f90
2	MPI	MPI CPU+MIC Execution	mpi
3	Stencil	OMP Concurrent CPU+MIC Execution	stencil
4	Performance	Measurement of Single Prec. FLOPS/sec	flops
5	Vectors	Vector/novector and alignment	vector & align
6	Threads	OMP team creation and barrier overheads	thread_overhead

README omp_c

What you will do: (See next section for INSTRUCTIONS.)

   Rather than putting the compile commands in makefiles,
   we have you type out the commands so that you
   can see how simple it is to compile for all cases.

1. Open two windows on the login node
   (You can edit/compile on login/compute nodes; no compiling on MIC)
   srun onto a compute node in one window (use for compiling)
   ssh over to the MIC in the other window (for MIC native execution)

2. Look over the Code for the cases:

 a.)
  hello.c   i.) Reduction, run on host
  hello.c   ii.) Reduction, run natively on mic
  hello_off.c   iii.) Reduction, run on host and offload to mic

 b.)
  hello_omp.c   i.) OMP Reduction, run on host
  hello_omp.c   ii.) OMP Reduction, run natively on mic
  hello_omp_off.c iii.) OMP Reduction, run on host and offload to mic

3. Compile and Run Cases:

INSTRUCTION DETAILS

1.)
  Login to Stampede in two terminal windows.

  From window1
  ssh <my_login_name>@stampede.tacc.utexas.edu

  srun -p development -t 60 -n16 --pty /bin/bash -l

      <Your prompt is the interactive node name (e.g. c559-001)>
      <compile and work here>

  From window2
  ssh <my_login_name>@stampede.tacc.utexas.edu
              <once you are logged in ssh to the compute node>
  ssh c559-yyy
              <once you are on the compute node ssh to the mic>
  ssh mic0

2.)

a.)

The “hello” toy code does a simple reduction:

```
icc       hello.c      -o a.out.cpu
icc -mmic hello.c      -o a.out.mic
icc       hello_off.c  -o a.out.off
```

On the host (window1) execute:

```
./a.out.cpu

./a.out.off

./a.out.mic  (this will run on the MIC!)
```

Or, in the MIC window2 execute the MIC binary:

```
 ./a.out.mic
```

b.)

The toy code does a simple OMP reduction:

```
icc       -openmp hello_omp.c      -o a.out.omp_cpu
icc -mmic -openmp hello_omp.c      -o a.out.omp_mic
icc       -openmp hello_omp_off.c  -o a.out.omp_off
```

On the host (window1) execute:

```
export OMP_NUM_THREADS=16
./a.out.omp_cpu

export MIC_PREFIX=MIC
export MIC_OMP_NUM_THREADS=240
./a.out.omp_off
```

On the MIC (window2) execute:

(No need for MIC_ prefix on MIC when executing natively!)

```
export OMP_NUM_THREADS=244
./a.out.omp_mic
```

While you are on the MIC, kick the tires on BusyBox.

cat /proc/cpuinfo, etc.

Change the number of threads on the host and the mic.

README omp_f90

What you will do: (See next section for INSTRUCTIONS.)

   Rather than putting the compile commands in makefiles,
   we have you type out the commands so that you
   can see how simple it is to compile for all cases.

1. Open two windows on the login node
   (You can edit/compile on login/compute nodes; no compiling on MIC)
   srun onto a compute node in one window (use for compiling)
   ssh over to the MIC in the other window (for MIC native execution)

2. Look over the Code for the cases:

 a.)
  hello.F90   i.) Reduction, run on host
  hello.F90   ii.) Reduction, run natively on mic
  hello_off.F90   iii.) Reduction, run on host and offload to mic

 b.)
  hello_omp.F90   i.) OMP Reduction, run on host
  hello_omp.F90   ii.) OMP Reduction, run natively on mic
  hello_omp_off.F90 iii.) OMP Reduction, run on host & offload to mic

3. Compile and Run Cases:

INSTRUCTION DETAILS

1.)
  Login to Stampede in two terminal windows.

  From window1
  ssh <my_login_name>@stampede.tacc.utexas.edu

  srun -t 60 --pty /bin/bash -l

      <Your prompt is the interactive node name (e.g. c559-001)>
      <compile and work here>

  From window2
  ssh <my_login_name>@stampede.tacc.utexas.edu
              <once you are logged in ssh to the compute node>
  ssh c559-yyy
              <once you are on the compute node ssh to the mic>
  ssh mic0

2.)

a.)

The toy code does a simple reduction:

```
ifort       hello.F90      -o a.out.cpu
ifort -mmic hello.F90      -o a.out.mic
ifort       hello_off.F90  -o a.out.off
```

On the host (window1) execute:

```
./a.out.cpu

./a.out.off

./a.out.mic    (this will run on the MIC!)
```

Or, in the MIC window2 execute the MIC binary:

```
./a.out.mic
```

b.)

The toy code does a simple OMP reduction:

```
ifort       -openmp hello_omp.F90      -o a.out.omp_cpu
ifort -mmic -openmp hello_omp.F90      -o a.out.omp_mic
ifort       -openmp hello_omp_off.F90  -o a.out.omp_off
```

On the host execute:

```
export OMP_NUM_THREADS=16
./a.out.omp_cpu

export MIC_PREFIX=MIC
export MIC_OMP_NUM_THREADS=240
./a.out.omp_off
```

On the MIC execute:
(No need for MIC_ prefix on MIC when executing natively!)

```
export OMP_NUM_THREADS=244
./a.out.omp_mic
```

While you are on the MIC, kick the tires on BusyBox.
cat /proc/cpuinfo, etc.
Change the number of threads on the host and the mic.

README mpi

In this lab you will execute a simple MPI reduction
on a single Stampede node:

Natively on the E5 (Sandy Bridge) CPUs
Natively on the Phi(MIC) coprocessor
Across the CPU and the MIC.

We do not include a makefile so
that you can exercise the commands
yourself. The code is not optimized
nor compiled for performance-- it is
meant to be instructional.

*****!!!!! DO THIS WORK ON A COMPUTE NODE !!!!! *******

1.) DO THIS WORK ON A COMPUTE NODE (make sure you have a c559-yyy
prompt.

If you are not on a node:
srun -p development -t 60 -n16 --pty /bin/bash -l

You may want to open another window on the compute node with ssh:

ssh c559-yyy (in another window)

2.) cd to the mpi directory:

cd mic/mpi

3.) Source a sourceme file that sets up all of the env. vars.

source sourceme

4.) From the single source, compile an E5 and Phi
version of the program:

```
mpif90 -mmic mpi_reduction.f90 -o a.out.mic
mpif90 (does not work correctly right now)
or
mpicc  -mmic mpi_reduction.c   -o a.out.mic


mpiifort     mpi_reduction.f90 -o a.out.cpu
mpif90       (does not work correctly right now)
or
mpicc        mpi_reduction.c   -o a.out.cpu
```

5.) Run the mpi code with 4 tasks on the CPUs:

```
mpiexec.hydra -n 4 -host localhost ./a.out.cpu
```

6.) Run the mpi code with 8 tasks on the MIC:

```
mpiexec.hydra  -n 8 -host mic0 ./a.out.mic
```

```
IF YOU HAVE TIME, try it on the MIC:
ssh over to the MIC (mic0) and run 2 tasks there.
ssh mic0
cd mic/mpi
source sourceme_mic   #very important: sets MIC path
mpiexec.hydra -n 2 ./a.out.mic
```

7.) Now launch an execution on the CPUs and the MIC
(E5 and the PHi) with the command:

```
mpiexec.hydra -n 4 -host localhost ./a.out.cpu : \
              -n 8 -host mic0      ./a.out.mic
```

OR try the "almost ready for prime time" ibrun.sym command.
First, set the environment variables you will need for the MIC.
Use MIC_MY_NSLOTS to specify the number of MIC tasks.
Options –c and –m are used to specify the CPU and MIC executable.

```
#export MIC_OMP_NUM_THREADS=240 #Only if you have omp code

    export MIC_MY_NSLOTS=4
   ~train00/ibrun.symm -m ./a.out.mic -c  ./a.out.cpu
```

Executes 16 tasks on CPUS (from the srun -n option), 4 on the MIC.

Experiment with changing the number of tasks. NOTE: if the
reduction values are not identical, it is because the partial sums
are used and roundoff will affect the results.

README stencil

General:

Rather than putting the compile commands in makefiles,
we have you type out the commands so that you
can see how simple it is to compile for all cases.

Please review the code sten.F90. It runs the same routine on the host and the mic concurrently with OMP.

Note: The same code is used for both architectures.
(Different optimizations can be applied for each.)
(But certainly more advanced code will
use #ifdef's with __MIC__.)

"signal" uses a variable as a handle for an event.

No optimization (compiler and code) are performed
here-- it is just a simple stencil for illustrating
the concurrency mechanism.

1.) sten.F90 is a simple code that shows how to compute
on the MIC and host concurrently.

The code shows several features of offloading:

a.) offloading a routine
b.) persistent data (data stays on the MIC between offloads)
c.) asynchronous offloading (for host-mic concurrent computing)

The "doit" script shows how to set up the execution environment.

2.) You don't even need a makefile for this:
(Script doit runs a.out with OPENMP env. vars for host & MIC.)

```
ifort -openmp sten.F90
./doit
```

the output will report the time of the concurrent execution.
Change the value of "L" (between 1 and 4,000) to change the
distribution of work for the CPU and MIC. See code.

3.) To run the code solely on the CPU (host), execute:
(Script doit host runs a.out with OPENMP env. vars for the E5.)

```
ifort -openmp -no-offload sten.F90
./doit host
```

4.) When compiling you can have the compiler report on offload statements and MIC vectorization with the following option:

```
ifort -openmp  -offload-copts:"-vec-report3 -O2"  \
               -opt-report-phase:offload sten.F90
```

5.) If you want to see what is going on in the offload regions during execution, set the H_TRACE to a level of verbosity {1-5}. E.g.

```
export H_TRACE=2
./doit
```

6.) Having fun:

Try adjusting the value of L in the code to change the amount of work on the host and the MIC.

Try a native execution:
Look in doit and check out the “mic” option for executing natively on the MIC. How would you compile the sten.F90 code? (Hint: can you combine the –no-offload and –mmic?) Compile the code for native execution and run the script with the mic option.

README flops

What you will do and learn: (See next section for INSTRUCTIONS.)

   Rather than putting the compile commands in makefiles,
   we have you type out the commands so that you
   can see how simple it is to compile for all cases.

1. Using interactive development environment for this MIC experiment.

   Open two windows on the login node.
   Use one window to edit and compile on a login node.
   Use the other window to create an interactive session on a compute
   node with the SLURM srun command; then ssh to the MIC for executing
   code. (You don't need to use batch for development!)

2. Look over the flops codes (C and/or Fortran versions):

```
         serial_nopt.c  *.f90 (not optimized)
          serial_opt.c  *.f90 (optimized)
       threaded_opt.c  *.f90 (Openmp parallelized)
```

   You will run the compiled executables directly on the MIC
   (i.e., you will run it natively on the MIC).

   Measure the floating point performance for a loop containing:

```
       fa[k] = a * fa[k] + fb[k]
```

   * The floating point operations use data from the L2 cache.
   * Since the reals are 4-byte, one can obtain 32floats/Cycle,
   * twice the number for 8-byte reals!

3. Compile and Run Cases, and report the performance:

INSTRUCTION DETAILS

1.) Login to Stampede in two terminal windows and cd into the flops directory:

FROM WINDOW1:

```
      ssh <my_login_name>@stampede.tacc.utexas.edu
      ...
      cd mic/flops
```

FROM WINDOW2:

```
ssh <my_login_name>@stampede.tacc.utexas.edu
cd mic/flops
```

Once you have logged in, execute srun to get a compute node interactively:

```
srun -p development -t 60 -n16 --pty /bin/bash -l

<Your prompt is the interactive node name (e.g. c559-001)>

ssh mic0            ( you will get this prompt:  " ~ $" )
```

2.) In WINDOW 1 (compute node CPU), make MIC binaries from the codes

a.)
Look over the compute loop in the code, it is executed many times to acquire an average value with a simple timer. Compile the code for the MIC using the -mmic option. (The default optimization is -O2.)

```
icc      -mmic serial_noopt.c    -o a.out_noopt.mic
icc -O3  -mmic serial_opt.c      -o a.out_opt.mic
icc -O3  -mmic threaded_opt.c    -o a.out_omp.mic -openmp
  or
ifort     -mmic serial_noopt.f90 -o a.out_noopt.mic
ifort -O3 -mmic serial_opt.f90   -o a.out_opt.mic
ifort -O3 -mmic threaded_opt.f90 -o a.out_omp.mic -openmp
```

In WINDOW 2 (on MIC):

On the MIC execute:

```
./a.out_noop.mic    ****This will take ~35 seconds ******
```

What if the Peak Performance of a MIC core for 4-byte data?
hint: Processor Speed (GHz) x FLOPs/Clock_period x cores,
1.1GHz, 16 FLOPs/CP add and 16 FLOPs/CP multiplies

_____TFLOPS (Peak)

What is the performance for the code?

_____TFLOPS (Single-core, non-optimized flops code.)

b.)
Review the serial_opt and threaded_opt code (either C or F90).
Note the additional optimization directives (and openmp directives).
Run the executables:

In WINDOW 2 -- on the MI:

Execute the optimized serial version of the code natively:

```
./a.out_opt.mic
```

On the MIC execute the OpenMP version natively:

```
export OMP_NUM_THREADS=122
 export KMP_AFFINITY=scatter
./a.out_omp.mic
```

What is the performance for this verions?

_____TFLOPS (Serial Version -- a.out_opt.mic)
_____TFLOPS (OpenMP Version -- a.out_omp.mic)

Try the calculations with 61 and 183 threads. Try using -O2.

In the OMP code an outer loop is added, with an offset
to the arrays, to provide more work on additional data. This
outer loop is parallelized with openmp. Also, data alignment
of the data (in the C code), and loop unrolling and loop-count
directives allow the compiler to significantly optimize the
performance of the loop by a factor greater than 10.
Data alignment often only contributes enhancements up to 15%.
For the C code, it allowed the compiler to aggregate the whole
inner loop into a set of instructions. Likewise, for the Fortran
code the Loop Count & unroll directives provided the same information,
in a different form that allows the compiler to create the
same compact set of instructions. See the instructor for more
details on how to see this with the assembly print options (-S).

Because each iteration of the computation loop is independent,
openmp directive are used to execute the code in parallel.

README vector

General:

The vector code calculates a simple Riemann sum on a unit cube. It consists of a triply-nested loop. The grid is meshed equally in each direction, so that a single array can be used to hold incremental values and remain in the L1 caches.

When porting to a Sandy Bridge (SNB) and MIC it is always good to determine how well a code vectorizes, to get an estimate of what to expect when running on a system with larger vector units. Remember, the Sandy Bridgge can perform 8 FLOPS/CP (4 Mults + 4 ADDS), while the MIC can do double that!

The function "f" (in f.h) uses by default the default integrand: x*x + y*y + z*z

1.) Create the executable with the make command:

```
make
```

Note the vectorization.

2.) Run the code:

```
./vector
```

3.) You will get the total number of cycles, called clock periods, CPs, required to execute your code.

```
record the CP for the vector code
```

The output will look like this
(clock periods): 18864693
To determine the time, divide the clock periods (CPs) by the speed of the core: 2.7x10**9 CPs/Sec on our Sandy Bridges.

4.) Remake the vector executable with vectorization turned off:

```
make clean
make VEC_OPT=-no-vec
./vector
record the CPs for the no vector code.
```

5.) Determine the speedup (CPs non-vec/CPs vec).

6.) What is the speedup? _____

7.) Change the integrand (in the f.h file),
and perform the same vec/no-vec calculations
in 1-6 above.

8.) What is the speedup when sin and pow
are used in the integrand? _____

If you don't see any difference in the
vectorized and non-vectorized version of
an applications, then you should expect
any gain in a system (E5/MIC) with has
larger vector functional units. It would
also be a good time to consider re-thinking
an algorithmic strategy for using vector
units in the application.

9.) Make a vector and novector executable for
the MIC, then ssh to mic0 and execute
them natively. Do this on a compute node.

```
make clean
make -f Makefile_mic VEC_OPT=-no-vec
mv vector novector

make clean
make -f Makefile_mic

ssh mic0
./vector
./novector
```

Note that the ratio between novector/vector
is only about as good as the E5. This is because
it is necessary to have 2 threads running on
a MIC core to have a new instruction issued
every clock period. Multi-threading is required
for optimal floating point performance on the
MIC.

README align

General:

For subroutines/functions compilers
have limited information, and cannot
make adjustments for mis-aligned data,
because the routines may be in other
programming units and/or cannot be
inlined. Because this is the usual
case, our experiment code is within
a subroutine in another file.

In a multi-dimensioned Fortran array if
the first dimension is not a multiple of
the alignment, the performance is
diminished.

e.g.for a real*8 a(15,16,16) array
15*8bytes is not a multiple of
16 bytes for Westmere and 32 bytes
for Sandy Bridge.

The stencil updates arrays (in align_sub.F90) of
dimensions of 15x16x16. By padding the first
dimension to 16, addition performance can be
obtained. In the experiment we make sure the
arrays are within the cache for the timings.

The timer captures time in machine
clock periods from the rdtsc instruction.
The way it is used here, it is accurate
within +/- 80 CPs. But timings may
have a wider range for conditions we
cannot control.

You will measure the performance with/without
alignment in the first dimension of a fortran
multi-dimensional array. You will also see
how inter-procedural optimization can be as
effective as padding.

Instructions:

1.) Build the code (execute "make"),
    and run the code (./align) with
    15 as the leading dimension and
    record the time.
2.) Change the dimension to 16,
    rebuild and record the time.
    By what percent has the performance increased?

3.) Try putting -ipo in the FFLAGS.
    rebuild and record the time.
    What has happened????

4.) Advanced users should try building a mic version
    (compile with –mmic) and execute it natively on
    the MIC.

README thread_overhead

General:

What you will do: (See next section for INSTRUCTIONS.)

   Rather than putting the compile commands in makefiles,
   we have you type out the commands so that you
   can see how simple it is to compile for all cases.

1. Open two windows on the login node
   (You can edit/compile on login/compute nodes; no compiling on MIC)
   srun onto a compute node in one window (use for compiling)
   ssh over to the MIC in the other window (for MIC native execution)

2. Look over the overhead.c code, and the scan scripts:

     overhead.c (Times new and 2nd parallel, and barrier regions)
     scan.cpu   (Runs 1-16 threads, and prints timings.)
     scan.mic   (Runs 1-244 threads, and prints timings.)
     Makefile   Makes mic and cpu executables (MAC=cpu or MAC=mic)

3. Make a CPU and MIC version of overhead.c.

4. Run scan.cpu on the compute node.
5. Run scan.mic on the MIC

INSTRUCTION DETAILS

1.) Login to Stampede in two terminal windows.

    FROM WINDOW1
    ssh <my_login_name>@stampede.tacc.utexas.edu

    srun -p development -t 60 -n16 --pty /bin/bash -l

    <Your prompt is the interactive node name (e.g. c559-001)>
    <compile and work here>

    FROM WINDOW2
    ssh <my_login_name>@stampede.tacc.utexas.edu
              <once you are logged in ssh to the compute node>
    ssh c599-yyy
              <once you are on the compute node ssh to the mic>
    ssh mic0

2.) Create overhead.cpu and overhead.mic

```
c559_001$ make clean; make MAC=cpu
c559_001$ make clean; make MAC=mic
```

3.) In WINDOW1 (on the CPU) execute:

```
login1$ ./scan.cpu
```

```
1-16 threads:
Thread Creation: ____ to ____ milliseconds
Thread Revival : ____ to ____ microseconds
Thread Barrier : ____ to ____ microseconds
```

4.) In WINDOW2 (on the MIC) execute:

```
./scan.mic    (ON MIC -- prompt is "~/mic/thread_overhead $ "

1-240 threads:
Thread Creation: ____ to ____ milliseconds
Thread Revival : ____ to ____ microseconds
Thread Barrier : ____ to ____ microseconds
```

Experiment with different KMP_AFFINITY settings.