

LAB



# Using the Xeon Phi Coprocessor

*Kent Milfeld*

`milfeld@tacc.utexas.edu`

Stampede Training  
January 11, 2013

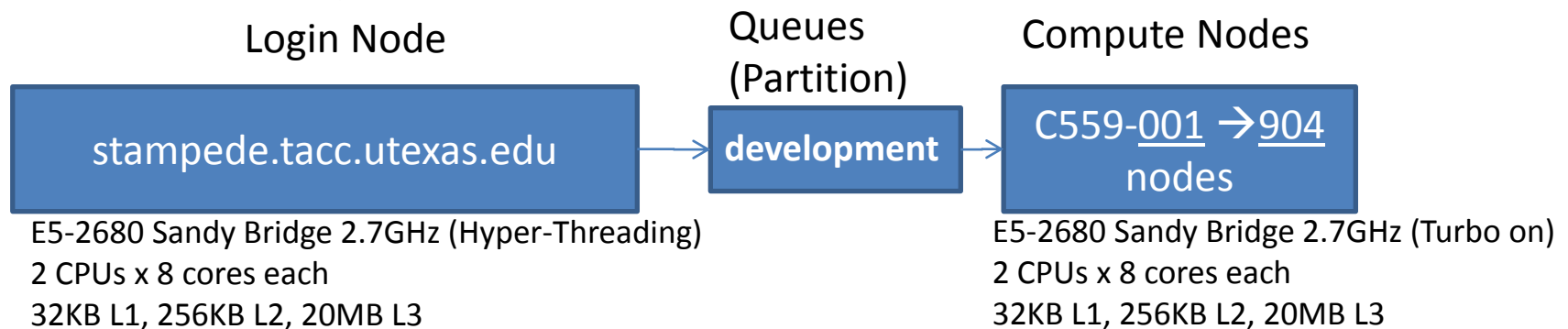
# Stampede System Login & Setup

- Login to Stampede:  
`ssh <my_login>@stampede.tacc.utexas.edu`
- Untar lab files to your directory:  
`tar xvf ~train00/mic_lab.tar`
- cd to the appropriate directory and **read the instructions here** and/or in the **README** file:

	Subject	Purpose	directory
1	Dev. Env. OMP	Interactive Development, Compile, Run CPU, MIC and Offload	omp_c or omp_f90
2	MPI	CPU, MIC and Symmetric (CPU+MIC) Execution	mpi
3	Stencils	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

# SLURM: Batch Utility

- slurm is the batch system. Details at:  
<https://computing.llnl.gov/linux/slurm/documentation.html>
- `sinfo` -- show current partitions (queues)
- `squeue/showq` -- show your queued jobs
- `sbatch myjob` -- submit batch job
- `scancel <jid>` -- delete job with id =<jid>
- `srun <args>` -- **run interactively on a node!!!**



# SLURM: srun

- You can obtain interactive access to a compute node through SLURM with the **srun** command.
- **srun** submits an “interactive” job to the queue, and gives you an **interactive prompt** when you get a compute node.
- Do the labs in an interactive session. (Feel free to try a batch job.)

EXAMPLE:

```
login2$ srun -t 60 -p development -n 16 --pty /bin/bash -l
--> Verifying availability of home dir (/home1/00770/milfeld)...OK
--> Verifying access to desired queue (devel)...OK
c559-802$
```

1hr, development queue, 16 cores.  
If asked, put an account (-A ...) here.

You are now on a compute node with a Intel Xeon Phi coprocessor.  
If you exit the window, the session is terminated.

# SLURM: sinfo, jobscript, sbatch

## login2\$ sinfo

```

PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
...
development up    2-00:00:00    1  down* c559-412
development up    2-00:00:00   39  idle  c559-[001-004,101-104, ...]

```

If you submit a job, you will get one of these nodes.

## login2\$ cat job

```

#!/bin/bash
#SBATCH -t=5           ← Time (minutes)
#SBATCH -p development ← Partition (queue)
#SBATCH -N 1          ← Number of nodes
module load mkl       ← If you use MKL, load module
export OMP_NUM_THREADS=16
./a.out

```

## login2\$ sbatch job

← Output: slurm-<jobid>.out

# SLURM: sbatch, squeue

EXAMPLE:

```
login2$ sbatch job
```

```
Submitted batch job 2058
```

```
login2$ squeue
```

JOBID	PARTITION	NAME	USER	ST	TIME	NODES	NODELIST (REASON)
2058	sc12	job	milfeld	R	0:09	1	c3-401

```
login2$ squeue
```

JOBID	PARTITION	NAME	USER	ST	TIME	NODES	NODELIST (REASON)
-------	-----------	------	------	----	------	-------	-------------------

```
login2$ ls
```

```
overhead  slurm-2058.out  job      info
Makefile  overhead.c      scan_compact
```

---

# SLURM: showq

EXAMPLE:

```
login2$ showq
```

```
ACTIVE JOBS-----
JOBID  JOBNAME  USERNAME  STATE    CORES  REMAINING  STARTTIME

6058   bash     subramon  Running   512    11:08     11/9 05:27
6060   bash     akshay    Running   16     11:42     11/9 06:02
6061   bash     milfeld   Running   16     3:54      11/9 06:14

3 Active jobs:    544 of  92160 Processors Active ( 0.59%)

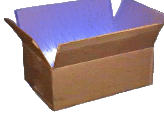
WAITING JOBS-----
JOBID  JOBNAME  USERNAME  STATE  PROC  WCLIMIT  QUEUE TIME
6057   vec     milfeld   Waiting  16  0:05:00  11/9 05:16

ERRORED JOBS-----
JOBID  JOBNAME  USERNAME  STATE    PROC  WCLIMIT  QUEUE TIME

5946   mpirun  dmalhotr  Completing  32  0:35:39  11/8 17:43

Total Jobs: 3 Active Jobs: 3 Idle Jobs: 0 Blocked Jobs:0
```

# Compute Node

- You can compile and run executables directly on compute nodes.
- “ssh mic0” on your compute node to access the Phi coprocessor.
- The  $\mu$ OS on the Phi is BusyBox  [www.busybox.net](http://www.busybox.net).
- It supports a subset of your favorite Linux commands.
- To execute native code directly on MIC compiled with `-mmic`.

EXAMPLE:

You cannot compile on the MIC. ←

```
login2$ ifort -mmic myprog.f90 -o a.out.mic
login2$ srun -t 60 -p development -n 16 --pty /bin/bash -l
...
C559-001$ ssh mic0
... added 'mic0,192.168.1.1' (RSA) to the list of known hosts.
TACC Stampede System - MIC Co-Processor
[Access enabled for user milfeld]
~ $ cd $HOME/whatever
~ $ ./a.out.mic
Hello World from MIC.
~ $
```

\$HOME mounted on Phi coprocessors. ←



# OMP Access/Compile/Run Experiment

What you will learn:

How to access the Stampede system.

How to access a Stampede node for interactive use. Stampede nodes have 2 E5 (Sandy Bridge) CPUs and 1 Xeon Phi (MIC).

How to compile the same code for running natively on the CPU or natively on the MIC, and offloading a the computational block of code.

You can compile on the login node or the compute node.

Compile & run:

- \* CPU-only program
- \* MIC-only (same a CPU) program
- \* Offload program

About the experiment

[See the README file in the directory for directions.](#)

\* Use `-mmic` to compile code for native execution on the MIC.  
\* ssh into the MIC, cd to the directory, and execute it like you would on any Linux box.

\* You can even execute it on the CPU and it will run on the MIC!

\* For offloading, include the directives (done for you),

\* compile as usual (no special compiler flags needed), then

\* run the executable on the CPU.

OMP is completely supported on the MIC, same directives work on MIC and the CPU.

# MPI: Experiment

## General:

In “symmetric” computing mode MPI tasks can be executed on the host and the MIC.

For symmetric computing two separate executables are made, one for the host and another for the MIC (also called heterogeneous computing).

The host and mic executables are started by an `mpiexec.hydra` launcher with arguments that specify the number of cores on each platform (E5s and MICs).

[See the README file in the directory for directions.](#)

## About the experiment

Do your work and launching on a compute node.

Use the “-mmic” compiler option to compile for the MIC. Advised: suffix MIC executables with a prefix. E.g.

```
mpif90 -mmic prog.f90 -o a.out.mic
mpif90      prog.f90 -o a.out
```

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

Try the new `ibrun.sym` command:

```
export MIC_MY_NSLOTS=8
ibrun.sym -c a.out -m a.out.mic
```

In develop node you can even ssh to mic0 (the MIC) and launch native code there:  
`mpiexec.hydra -n 8 ./a.out.mic`

# OMP Concurrent Execution on MIC + CPU

What you will learn:

The “sten” code shows how to employ offloading and continue execution on the host.

The same stencil update code (a function) is used for offloading and CPU execution.

“Persistent data” is used on the MIC to avoid copying data back to the host between different offloads.

Asynchronous offloading allows simultaneous execution of the CPU portion and MIC portion of work.

About the experiment

See the [Readme file in the directory for directions](#).

OMP is completely supported on the MIC, same directives work on MIC and the CPU. Compile with “-openmp” for CPU and MIC.

Look over the code and review the class notes. The stencil\_it function is called twice each iteration. Once as an offload with M-L amount of work and then as a CPU execution for L amount of work. The offload is asynchronous, and a wait occurs at the end of the loop.

The stencil matrix is allocated once before the loop, and persists on the MIC throughout all offloads of the loop.

# Performance: Experiment

## General:

The performance of a loop containing:

$$fa(k) = a * fa(k) + fb(k)$$

for 4-byte reals is evaluated with: no optimization, optimization, and OpenMP parallelization.

Vector units on a MIC are 64 Bytes wide.  
Add and Multiply units operate concurrently.

Vector Units hold:

16 Single Precision (SP): 4-byte elements.

8 Double Precision (DP) 8-byte elements.

Floating point operations per clock period(CP):

SP: 16add + 16mult per CP

DP: 8add + 8mult per CP

## About the experiment

You can use this exercise to time your own loops, our routines that are hot spots in your code.

The C optimized code shows how to force array alignment with an `__attribute__`. Dynamically allocated arrays can be aligned with `__mm__malloc()`, `memalign()` or `posix_memalign()`.

The Fortran code requires an unroll and loop count directive for optimal performance. Allocated arrays are aligned.

What is the performance?

\_\_\_\_\_ (GFLOPS) Serial Non Optimized

\_\_\_\_\_ (GLFOPS) Serial Optimized

\_\_\_\_\_ (TFLOPS) Parallel Optimized

# Vector: Experiment

## 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 (E5) 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 E5 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:  $x*x + y*y + z*z$

## About the experiment

You will measure the number of clock period (CP) it takes to execute a 3-D Riemann sum that is vectorized. You will then determine the time (in CPs) to do the calculation without vectorization.

Follow the instructions in the README file.

What is the vectorization speed-up for the  $x*x + y*y + z*z$  integrand?

\_\_\_\_\_ speedup

What is the vectorization speed-up for the  $x + \text{pow}(y,2) + \text{sin}( \text{PI}*z )$  integrand?

\_\_\_\_\_ speedup

# Align: Experiment

## General:

For subroutines/functions compilers have limited information, and cannot make adjustments for misaligned 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 Fortran subroutine.

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 needed for Westmere, and 32 bytes for Sandy Bridge.

## General:

The stencil update-arrays in sub5.F90 have dimensions of 15x16x16. By padding the first dimension to 16, better 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 +/- 25 CPs. Large timing variations are not due to the time, but to 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.

# Align: Experiment

## About the experiment

You will change the leading dimension of a 3-D array used in a Fortran subroutine, so that the 1<sup>st</sup> element of the leading dimension is always aligned by 32-bytes, no matter what indices are in the other dimensions.

Follow the instructions in the README file in the work directory.

What is the performance improvement when the array is aligned?

\_\_\_\_\_ %

Can the `-ipo` option allow the compiler to work around the apparent alignment problem?

\_\_\_\_\_ %

# OMP-overhead: Experiment

## General:

Often developers have no idea of the overhead/costs for creating a parallel region (for the first time), reforming subsequent parallel regions, and synchronizing on a barrier.

Review the overhead.c code. It has timers around two separate parallel regions. The first region forks a set of threads (creation), and the other reuses (revives) the threads. A subsequent parallel region measures the cost of a barrier. For timing a barrier, we use the barrier cost for thread0 (this is good enough for semi-quantitative work).

The code uses the rdtsc hardware clock counter, good to around 25 Clock Periods (CPs), to obtain an accurate time for the OpenMP operations. Run the experiment several times so that you can see the variability.

## About the experiment

Follow the instructions in the README file in the work directory.

You will measure the costs for forking, reviving, and synchronizing (barrier) : 1-16 on the cpu and 1-240 on the MIC. (Use srun.)

After removing the outliers determine the range for the types of OpenMP operations:

CPU: 1-16 threads:

Thread Creation: \_\_ to \_\_ milliseconds

Thread Revival : \_\_ to \_\_ microseconds

Thread Barrier : \_\_ to \_\_ microseconds

MIC: 1-240 threads:

Thread Creation: \_\_ to \_\_ milliseconds

Thread Revival : \_\_ to \_\_ microseconds

Thread Barrier : \_\_ to \_\_ microseconds

Note, first time you use a parallel regions, there is a large cost to "fork" the threads. Note, a revival is only about 1/2 the cost of a barrier for a large number of threads.

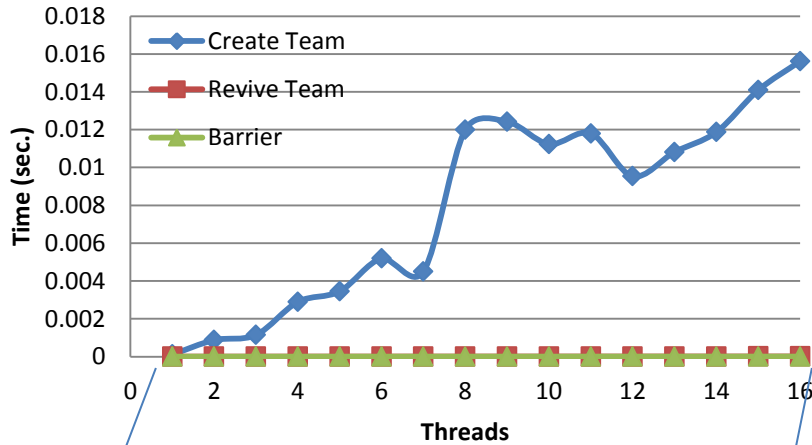


# OMP-overhead: Experiment

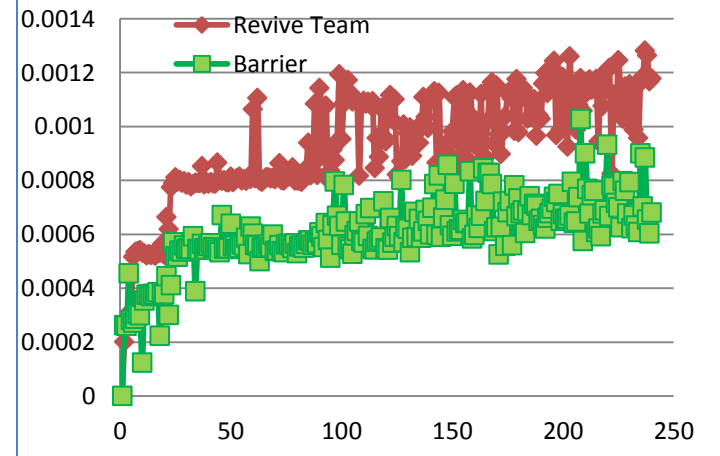
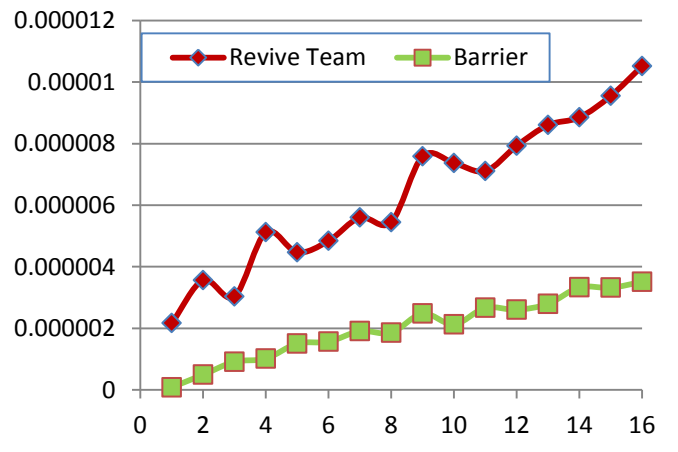
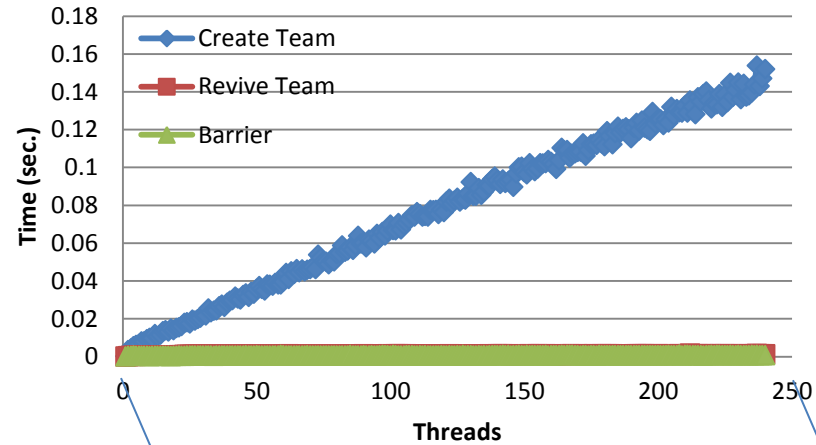
These are results from our similar discovery system, not stampede.

These are MIC results (no averaging).

## OpenMP Overhead -CPU



## OpenMP Overhead --MIC



6 Threads