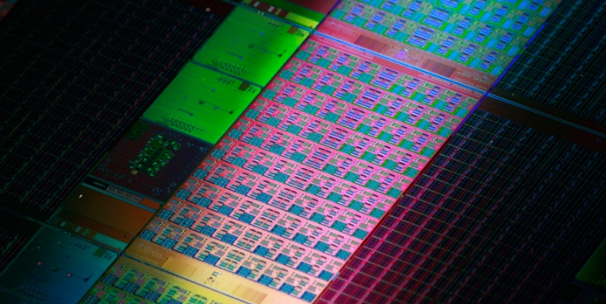


LAB

Preparing for Stampede: Programming Heterogeneous Many-Core Supercomputers

Dan Stanzione, Lars Koesterke, Bill Barth, Kent Milfeld
dan/lars/bbarth/milfeld@tacc.utexas.edu

XSEDE 12
July 16, 2012

Discovery System Login & Setup

- Login to our discovery system (train2## is your assigned login):

  ssh train2xx@login2.discovery.tacc.utexas.edu

- Untar lab files to your directory:

  tar xvf ~milfeld/xsede_12.tar

- Cd to the appropriate directory and read the instructions here and/or in the **info** file:

0.) How to use SLURM Batch system (slurm)

OpenMP Experiments

1.) Overhead in OpenMP (omp_overhead)
2.) Setting Affinity in code (affinity)
3.) QR factorization (caqr)

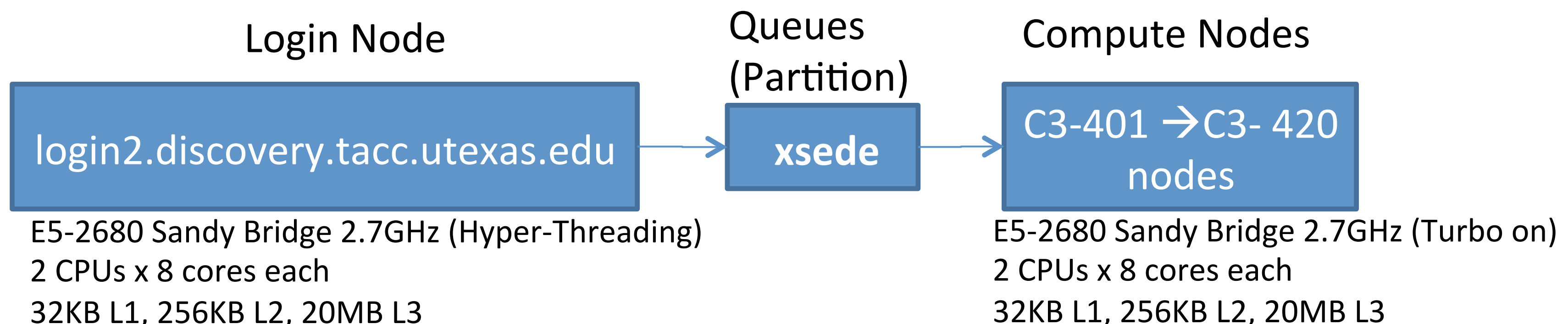
Vectorization Experiments

1.) Vectorization (vector)
2.) Alignment (align)
3.) TBD ()

(…)=directory

SLURM: Batch Utility

- slurm is the batch system. Details at:
  https://computing.llnl.gov/linux/slurm/documentation.html
- sinfo -- show current partitions (queues)
- squeue -- show your queued jobs
- sbatch myjob -- submit batch job
- scancel <jid> -- delete job with id =<jid>



SLURM: sinfo, jobscript, sbatch

login2$ sinfo

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
  …
xsede        up 2-00:00:00      1  down* c3-412
xsede        up 2-00:00:00     23   idle c3-[401-411,413-424]
```

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

login2$ cat job

```
#!/bin/bash
#SBATCH --time=5
#SBATCH -p xsede
#SBATCH -w "c3-401"

module load mkl
export OMP_NUM_THREADS=16
./a.out
```

(The line `#SBATCH -w "c3-401"` is struck through on the slide.)

- Time (minutes)
- Partition (queue)
- Specific HOST – **you won't need to use this.**
- If you use MKL, load module

login2$ sbatch job

Output: slurm-<jobid>.out

SLURM: sbatch, squeue, srun

EXAMPLE:

```
login2$ sbatch job
Submitted batch job 2058

login2$ squeue
JOBID PARTITION     NAME     USER  ST        TIME  NODES NODELIST(REASON)
 2058     xsede      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
```

You can obtain interactive access to a compute node through slurm with the srun command. Srun submits an "interactive" job to the queue, and give you an interactive prompt when you get a node. Example:

```
login2$ srun -p xsede --pty /bin/bash -l
c3-401$
```

Advise us if you want you use this.

SLURM: Experiment

- Go to the slurm directory
- Look over the job script and submit it with sbatch
- Monitor the progress of your job with squeue
- Kill your job with scancel if it hasn't finished.
- List details about all queue with the sinfo command

Note that the output is returned in a file with the jobid in its name.
How many nodes were in the alloc state when you ran sinfo? ______
Read the output file.

What is the PWD directory for batch execution? __________________
What option would you use to request a specific node? ___________
How many CPUS/CORES are in a node? _____ cores ______cpus
What size is the L3 cache? ________MB
What is the nominal chip frequency? ______GHz

Vector: Experiment

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 SNB 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 info 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 + pow(y,2) + sin(PI*z) integrand?

_________ speedup

Align: Experiment

Align

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 +/- 80 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

Align

About the experiment

You will change the leading dimension of a 3-D array used in a Fortran subroutine, so that the 1st 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 info file.

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 just average over what each thread sees as the time (this is good enough for semi-quantitative work).

The code uses the rtsc 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.

The experiment will execute and collect the times for 1 through 16 threads, sleeping 1 second to let the system become quiescent.

OMP-overhead: Experiment

About the experiment

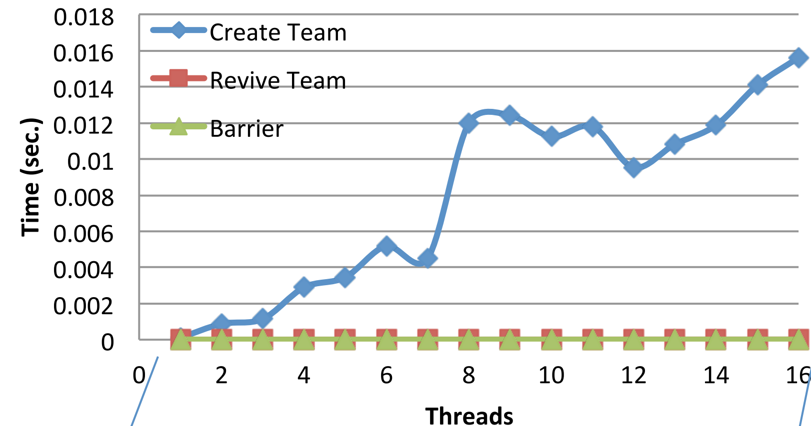
You will measure the costs for forking, reviving, and synchronizing (barrier) for 1 through 16 threads. Use a compute node for this (submit the job script).

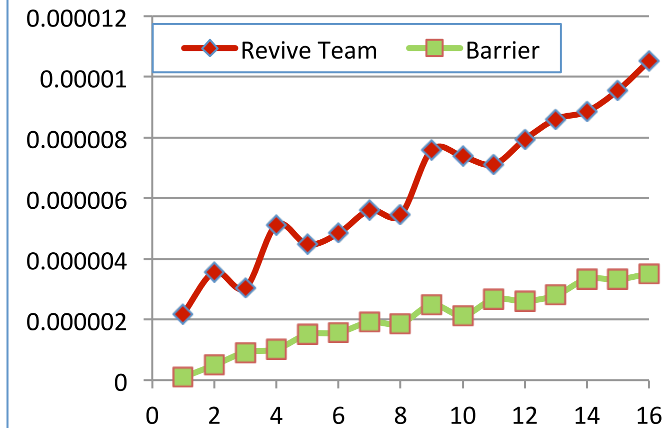
Follow the instructions in the info file.

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

Thread Creation: __ to __ milliseconds
Thread Revival : __ to __microseconds
Thread Barrier : __ to __ microseconds

Note, first time you use a parallel regions, thereis 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.





Overhead

Affinity: Experiment

General:

For some applications, developers want full control over which cores their threads are placed. If there are two sockets on a node, each with 8 cores, a developer may want to increase the number of threads on a single socket from 1-8 (no threads on the other socket). This can be done by mapping the thread id to the logical cpu (core id). Threads assignments to cores are done in sched_setaffinity (). Intel numbers the cores on a two-socket system in a round robin fashion. Core ids on socket 0 are 0,2,4,6,8,10,12,14, and on socket 1 they are 1,3,5,7,9,11,13,15.

In the affinity code, a modulo expression is used to give the numbers
0,2,4,6,8,10,12,14,1,3,5,7,9,11,13,15
for the thread ids (linear sequence):
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

General:

Hence, for a thread team of n, the thread ids 0,1,2,...n-1 can easily be mapped to 0,2,4,...13,15, so as to fill up one socket and then the other as the number of threads are increased..

QR: Experiment

- QR Factorization (Communication Avoiding)
  http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-131.pdf

- **Objective:** Evaluate the scalability of the caqr code David Hudak ported to the KNF. (Thank you David!)

- Go to the caqr omp directory: cd caqr
- Load up MKL in your environment: module load mkl
- Create the caqr executable: make
- Run a single execution with the job script: sbatch job
- Modify job to run from 1-16 threads with MKP_AFFINITY set to scatter and compact.
- Which one wins, scatter or compact?

```
export KMP_AFFINITY=scatter
for i in `seq 1 16`; do
export OMP_NUM_THREADS=$i
echo "team: $i threads"
./caqr 2048 2048 10000 0
done
```

QR

QR: Experiment -- KMP_AFFINITY

- Get some basic information about threads using KMP_AFFINITY verbose. Submit a job with:

```
export KMP_AFFINITY="vebose,none"
export OMP_NUM_THREADS=2
./caqr 2048 2048 10000 0
```

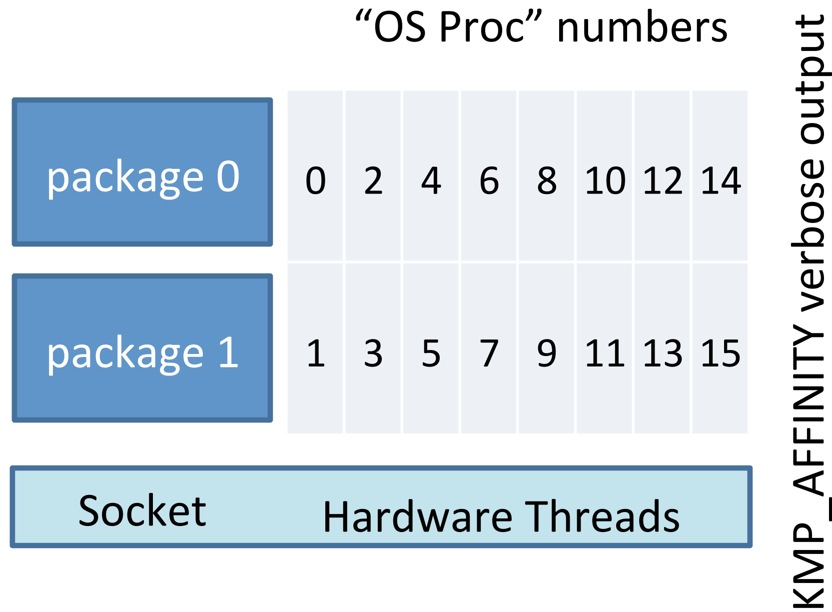
OUTPUT:

OMP: Info #156: KMP_AFFINITY: 80 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY:
2 packages x 8 cores/pkg x 1 threads/core (16 total cores)
…
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set **{0,1,2,3,4, … 14,15}**
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set **{0,1,2,3,4, … 14,15}**
…

- Include "verbose,none" in KMP_AFFINITY to get
- basic information and
- default mask.

Means thread can execute on any of the 16 cores.

QR: Experiment
Socket (package) -- OS Proc map



QR: Experiment

- Now run a job with KMP_AFFINITY=verbose,compact and KMP_AFFINITY=verbose,scatter for 8 threads.
- Determine how the threads are distributed on the sockets for scatter and compact affinity, using the table on the previous page.

- For the more adventurous: Change the scheduling in QR_matrix.cp to runtime, e.g. include the clause "schedule(runtime)" in the #pragma omp parallel for loops. and experiment with different scheduling, by setting the OMP_SCHEDULE environment variable. (e.g. export OMP_SCHEDULE="dynamic,#", where # is a chunk size number.)