Tips for Running and Tuning Programs on Cray XC Systems with KNL

Heidi Poxon

Cobalt Batch System on Theta

man qsub

```
qsub -n <number of nodes> -t <wallclock time> --mode script -A <project(allocation)> \
     -q <queue> --env <var1=value1:var2=value2....> myScriptFile
```

EXAMPLE:

```
qsub -n 2 -t 15 --mode script -A Theta_ESP -q cache-quad --env OMP_NUM_THREADS=16 \
     ./runScript
```

Script Example

```
#!/bin/bash -evx
# Command line options will override the three settings below:
#COBALT -t 5
#COBALT -q cache-quad
#COBALT --project Theta_ESP
export n_nodes=$COBALT_JOBSIZE
export n_ranks_per_node=64
export n_mpi_ranks=$(($n_nodes * $n_ranks_per_node))
aprun --env CRAY_OMP_CHECK_AFFINITY=TRUE --env OMP_NUM_THREADS=4  \
   -n $n_mpi_ranks -N $n_ranks_per_node -cc depth -d 4 -j 4  ./x_hello_world_mpi_openmp
```

To simplify the aprun command, set environment variables within script

Theta ESP Hands-on Workshop Notes

- All Intel Tuning Guides:
  http://software.intel.com/en-us/articles/processor-specific-performance-analysis-papers

Theta ESP Hands-on Workshop Notes

- Theta hardware specifics:
  - 3240 nodes
  - Intel Xeon Phi 7230 (16GB MCDRAM, 1.30 GHz, 64 core)
    - http://ark.intel.com/products/94034/Intel-Xeon-Phi-Processor-7230-16GB-1_30-GHz-64-core
    - 192 GB DDR4 DRAM
    - 128 GB node-local SSD
    - 32 MB L2 cache

Compile / Link Using Cray Compiler

- **> module swap PrgEnv-intel PrgEnv-cray**
- **> module load craype-mic-knl**
- **> cc –o a.out my_c_code.c**
  - > CC –o a.out my_c++_code.cpp
  - > ftn –o a.out my_fortran_code.f90
- **Check affinity by setting environment variable:**
  - export CRAY_OMP_CHECK_AFFINITY=TRUE

Using CMake and GNU Autotools on Cray Systems

- See the Tips document:

http://docs.cray.com/books/S-2801-1608//S-2801-1608.pdf

Experimenting with MCDRAM

- **Try the following from least to most effort**

1. Run in 100% cache mode
   - Captures reuse, simplest, will just work
2. Run in quad, flat mode but don't allocate into MCDRAM
   - How sensitive is program to MCDRAM
3. Check if program fits in MCDRAM
   - Use numactl or memory high water mark info from perftools-lite
   - Best way to run if code fits
4. Use a combination of MCDRAM and DDR using numactl
   - `Also simple, but may not allocate "the right" data in MCDRAM`
5. Allocate bandwidth intensive arrays in MCDRAM
   - The most effort, need to identify which arrays to allocate

Allocating in MCDRAM or DDR in Flat Mode

- **Allocate out of MCDRAM**
  - `> aprun . . . numactl --membind=1 a.out`
- **Allocate out of DDR**
  - `> aprun . . . numactl --membind=0 a.out`
- **Allocate out of MCDRAM until exhausted**
  - `> aprun . . . numactl --preferred=1 a.out`

aprun Example Using Intel

- ```
  $ KMP_AFFINITY=none \
    OMP_NUM_THREADS=2 \
    aprun –n 1020 –N 32 –d2 –j1 –cc depth a.out
  ```
- -n = --pes == number of MPI ranks
- -N = --pes-per-node == number of ranks per node
- -d = --cpus-per-pe == separation between ranks; depth
- -j = --CPUs == number of Hyper-Threads per physical core
- -cc = --cpu-binding == CPU affinity binding
  ```

aprun Examples Using GNU or CCE

- **1 node, 4 MPI ranks with 4, 16 or 64 OpenMP threads**
  - $ OMP_NUM_THREADS=4 aprun –n 4 –d 4 –j1
  - $ OMP_NUM_THREADS=16 aprun –n 4 –d 16 –j1
  - $ OMP_NUM_THREADS=64 aprun –n 4 –d 64 –j4
    - This example requires using 4 hyperthreads

More aprun Examples

- **1 node, 256 MPI ranks, no OpenMP threads, 4 ranks per core**
  - $ aprun –n 256 –j4
- **1 node, 128 MPI ranks, 2 hyperthreads threads per rank, 2 OpenMP threads per rank**
  - $ OMP_NUM_THREADS=2 aprun –n 128 –d 2 –j4
- **1 node, 16 MPI ranks, 4 hyperthreads per core, 16 OpenMP threads per rank**
  - $ OMP_NUM_THREADS=16 aprun –n 16 –d 16 –j4

Guidance on Hyper-Threading

- **There does not seem to be any reasonable predictor if Hyper-Threading (HT) will be better or not**
  - But when it is better, HT=2 often seems the best
- **While HT may help with hiding latency, it also can significantly increase pressure on all levels of cache**
- **Try it out and see what is best**

Using Core Specialization

- **Offloads some kernel and MPI work to unused Hyper-Thread(s)**
- **Good for large jobs and latency sensitive MPI collectives**
- **Highest numbered unused thread on node is chosen**
  - Usually the highest numbered HT on the highest numbered physical core
- **`Examples`**
  - `aprun -r 1 …`
  - `aprun –r N … # use several extra threads`
- **`Cannot oversubscribe, OS will catch`**
  - Illegal: `aprun -r1 -n 272 -N 272 -j 4 a.out`
  - Legal: `aprun -r1 -n 271 -N 271 -j 4 a.out`
  - Legal: `aprun -r8 -n 264 -N 264 -j 4 a.out`

ALPS Affinity vs KMP_AFFINITY

- Cray ALPS affinity conflicts with Intel® KMP_AFFINITY
  - aprun affinity and KMP affinity should not be used together
- ALPS default is `-cc=cpu` which binds each thread to a core
- Use `–cc none` to let KMP have control and use KMP_AFFINITY settings
  - Only works as expected if you have a single MPI rank per node
- Use `-d` and `–cc depth` to let ALPS have control and to take care of helper threads

General Guidance

- **Profile your code to minimize time investment and help focus optimizations**

- **Too few MPI ranks per node can result in poor performance**
  - Too few ranks will likely result in too few cores driving the network
  - The relatively slow cores will not be able to fully utilize the network bandwidth or number of outstanding references, leaving performance on the table

- **Be very careful with any single node performance studies**
  - Performance trends may lead you to believe that higher levels of OpenMP are desirable
  - Single node performance studies do not have any off node component.
  - This may also be the result of simply driving MPI time to zero, which will never happen in a real multi-node job

- **OpenMP and MPI scaling studies should be done on as many nodes as possible**
  - The tradeoffs should be considered at the level of parallelism that you expect to run at

General Guidance (2)

- **Try core specialization (`-r 1`) whenever possible**
- **Try using huge pages**
  - Especially helpful if you suspect TLB issues or if you are using MPI collectives like MPI_Alltoall()
- **Use MPI defaults**
  - Don't start by setting MPI-related environment variables
  - Defaults are often the best choice
- **If MPI communication is dominant, try rank reordering with CrayPat / grid_order**
- **Do not build with –g when running optimized code or measuring performance**

Vectorization is Important!

- **KNL's overall performance is highly correlated with good, clean vector code**

- **Without good to excellent vectorization, the slow scalar performance will act as a serious drag on overall performance**
  - May extend to "partial vectorization"
    - Loops that are vectorized, but have levels of indirection, non-unit-stride memory references, or conditionals make vectorization less than clean

Summary

- Application scaling studies should be done on as many nodes as possible
- Start testing using a moderate number of OpenMP threads and move up from there, but don't think you have to go > 32 threads
- Vectorization is critical to sustaining good performance
- One must consider how to utilize MCDRAM
- There is no good guidance for when hyper-threading will help