IBM Power8 CPUs - Overview

Each compute node has two sockets, with 10 cores per socket => 20 cores per node.

On summitdev, nodes are in SMT8 mode => 8 hardware threads per core => 160 hw threads per node.

Linux treats each hardware thread as a logical cpu.

Placement and binding of threads is very important for good performance.

There is a significant NUMA nature to memory access : memory local to your socket is best.

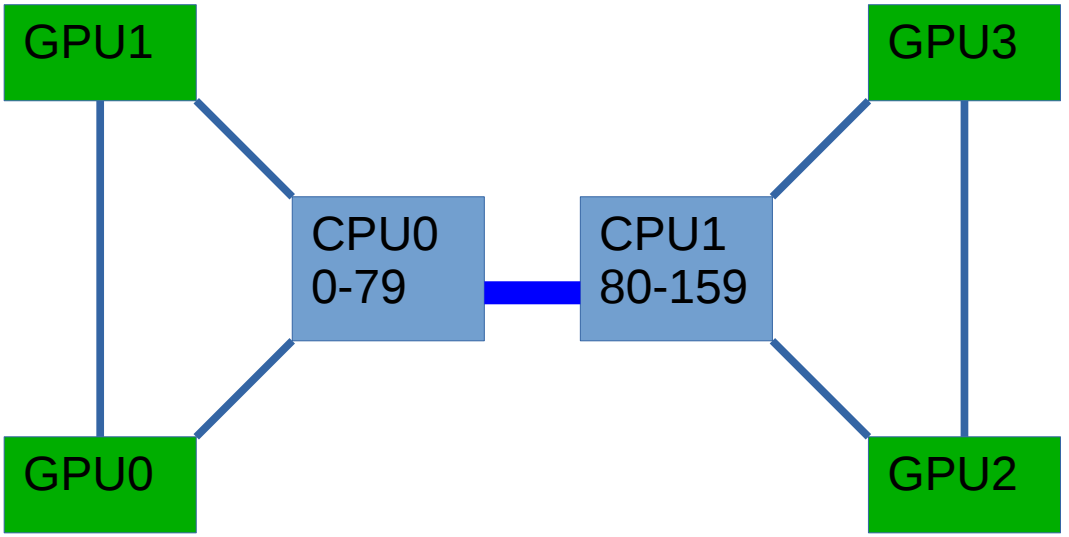
On summitdev, each compute node has 256 GB main memory.

There are four Pascal GPUs per node (devices 0,1 on socket 0, devices 2,3 on socket 1).

Some commands to run on the compute nodes :

```
$ numactl --hardware  (lists logical cpus and numa domains)

$ cat /proc/meminfo   (lists host memory)

$ cat /proc/cpuinfo   (minimal info on CPUs … more about frequency later)

$ nvidia-smi -q       (information about the NVIDIA GPUs)

$ nvidia-smi topo -m  (shows GPU/CPU topology and affinity information)
```

Sketch of Power8 Nodes on summitdev



Each CPU socket has 10 cores, 80 hardware threads.

GPUs 0,1 are connected by NVLINK to CPU0, and have affinity for logical cpus 0-79.
GPUs 2,3 are connected by NVLINK to CPU1, and have affinity for logical cpus 80-159.

Each Pascal GPU has 16 GB memory, peak dp Flops ~5 TF, stream triad bw ~520 GB/sec.

Total host memory = 256 GB, peak dp Flops ~560 GF, stream triad bw ~180 GB/sec.

Power8 frequency can be adjusted by the Linux kernel, ~3.5 GHz is typical for HPC workloads.
GPU clocks can be controlled via nvidia-smi.

Power8 CPU Details

64 KB L1 data-cache per core, 3-5 cycles latency load to use.

32 KB L1 instruction cache per core.

512 KB L2 cache per core, ~12 cycles latency.

8 MB L3 cache per core (NUCA architecture), ~27 cycles latency.

Two independent fixed-point units (FXU).

Two independent load-store units (LSU) plus two more load units.

Two independent floating-point units (VSU) (vector/scalar units).

Max double-precision floating point issue rate is 4 fmadds per cycle.

Max single-precision floating-point issue rate is 8 fmadds per cycle.

Max instruction completion rate is 8 per cycle.

Floating-point pipeline latency ~7 cycles.

Basic CPU performance measurements - summitdev

Measured double-precision Flops using dgemm() from IBM ESSL library: per-node performance

```
summitdev-r0c1n02 reports 507.916 GFlops
summitdev-r0c2n02 reports 507.689 GFlops   (507GF ~ 20*8*3.5GHz*0.90)
summitdev-r0c0n10 reports 507.013 GFlops
```

Measured single-precision Flops using sgemm() from IBM ESSL library: per-node performance

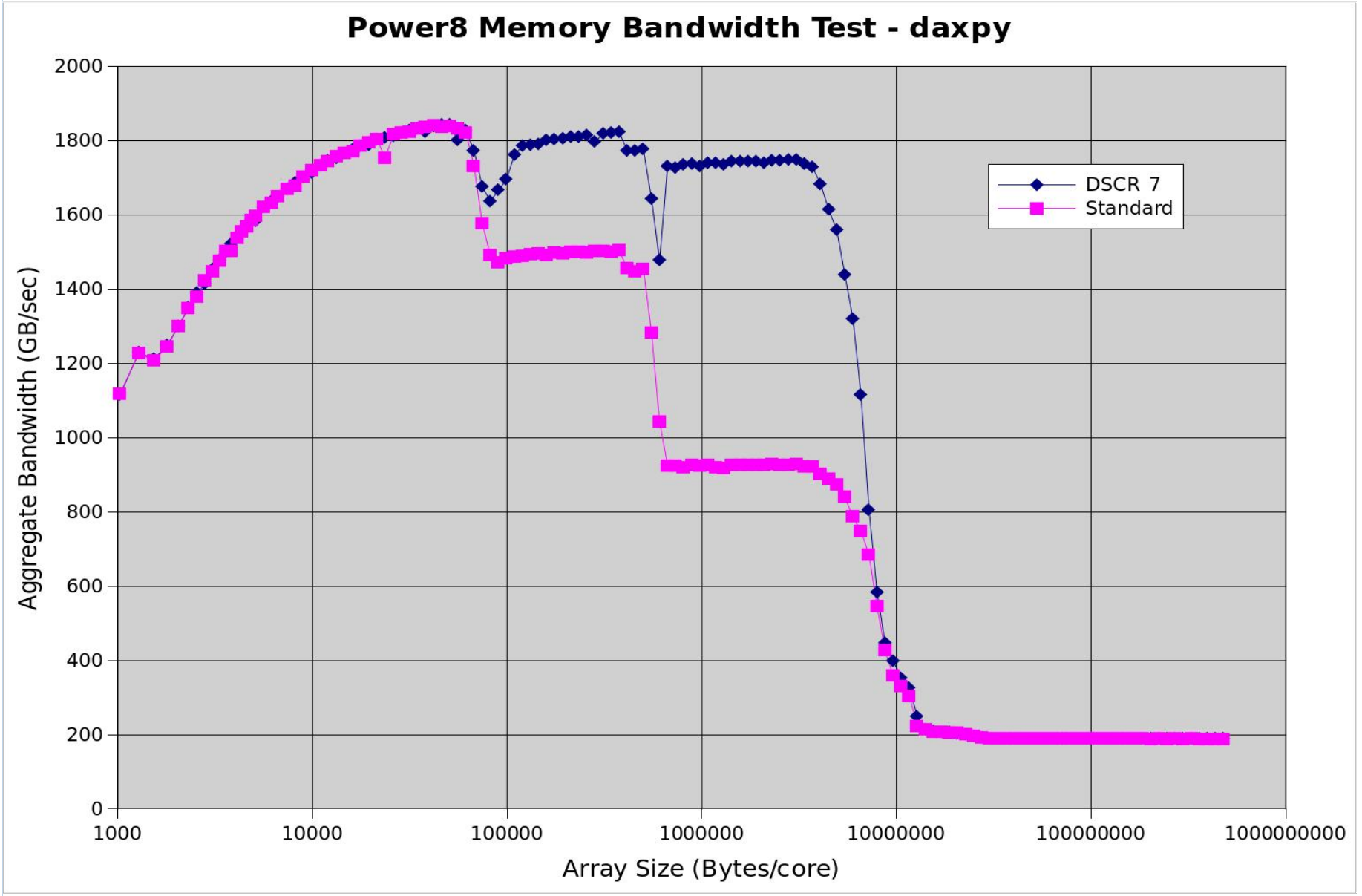
```
summitdev-r0c2n02 reports 962.680 GFlops
summitdev-r0c0n06 reports 962.092 GFlops   (962GF ~ 20*16*3.5GHz*0.86)
summitdev-r0c2n07 reports 962.073 GFlops
```

Stream benchmark : host = summitdev-r0c0n15

```
The total memory requirement is   35.8 GB
You are running each test  20 times
Function  Rate (GB/s) Avg time Min time Max time
 Copy:    149.08      0.1738   0.1542   0.1751
 Scale:   147.87      0.1748   0.1575   0.1777
 Add:     182.53      0.2119   0.2026   0.2140
 Triad:   177.66      0.2179   0.2062   0.2266
 Load:    143.20      0.2714   0.2464   0.3175
```

Power8 double-precision SIMD width is two : max dp Flops = #cores * 8 * Freq.
Power8 single-precision SIMD width is four : max sp Flops = #cores * 16 * Freq.
Normal frequency is ~3.5 GHz for dgemm/sgemm => ~90% of peak Flops is achieved.

For memory bandwidth : max read bw = 8*19.2 = 153.6 GB/sec per node
max write bw = 8*9.6 = 76.8 GB/sec per node
max triad bw = 230.4 GB/sec (8*19.2 read + 8*9.6 write) per node



daxpy : y(i) = a*x(i) + y(i)

Basic GPU health check on summitdev

```
export OMP_NUM_THREADS=10; export OMP_PLACES={0:10:8}
./health  (code executes on socket 0)

found 4 GPU devices on host summitdev-r0c0n13
checking device 0 = Tesla P100-SXM2-16GB ...
host to device transfer rate from pinned   memory = 32.35 GB/sec
host to device transfer rate from pageable memory = 14.46 GB/sec
device to host transfer rate from pinned   memory = 34.01 GB/sec
device to host transfer rate from pageable memory = 10.57 GB/sec
GPU daxpy bandwidth = 503.35 GB/sec
GPU dgemm TFlops = 4.848

checking device 1 = Tesla P100-SXM2-16GB ...
host to device transfer rate from pinned   memory = 32.34 GB/sec
host to device transfer rate from pageable memory = 14.72 GB/sec
device to host transfer rate from pinned   memory = 34.01 GB/sec
device to host transfer rate from pageable memory = 6.04 GB/sec
GPU daxpy bandwidth = 504.59 GB/sec
GPU dgemm TFlops = 4.955

checking device 2 = Tesla P100-SXM2-16GB ...
host to device transfer rate from pinned   memory = 29.75 GB/sec
host to device transfer rate from pageable memory = 17.36 GB/sec
device to host transfer rate from pinned   memory = 21.85 GB/sec
device to host transfer rate from pageable memory = 6.31 GB/sec
GPU daxpy bandwidth = 502.11 GB/sec
GPU dgemm TFlops = 5.002

checking device 3 = Tesla P100-SXM2-16GB ...
host to device transfer rate from pinned   memory = 29.72 GB/sec
host to device transfer rate from pageable memory = 17.16 GB/sec
device to host transfer rate from pinned   memory = 21.87 GB/sec
device to host transfer rate from pageable memory = 6.27 GB/sec
GPU daxpy bandwidth = 502.07 GB/sec
GPU dgemm TFlops = 4.932
```

Programming Environment on summitdev

The summitdev system uses the modules infrastructure :

$ module list (shows the modules that are loaded in your environment)
$ module avail (shows the available modules)
$ module add pgi/16.10 (loads the current PGI compiler, with some side effects)
$ module unload module_name (unloads the specified module, with some side effects)

Your choice of modules affects PATH, LD_LIBRARY_PATH, and other env variables.

Default user limits are different on the login node and the compute nodes (check ulimit -a).
Stack and core-file size are limited on the login node, but not on the compute nodes.

Use the login node to edit, compile, and submit jobs … run all jobs on compute nodes.

Compilers : clang, gnu, pgi, ibm xl … nvcc … a lot of options to consider

gcc, g++, gfortran : default version is 4.8.5;
Some options to be aware of : -mcpu=power8 , -Ofast, -fopenmp
The gnu compilers are the default for nvcc, but you can control that with the -ccbin option.

pgcc, pgc++, pgf90 : most recent version is 16.10 … your best option for OpenACC support.
Example invocation : pgcc -c -mp -acc -ta=tesla:cc60 -Minfo=accel -fast nbody1.c

xlc_r, xlC_r (13.1.5), xlf2008_r (15.1.5) : has OpenMP 4.5 features : -qsmp=omp -qoffload
Some options to know : -qversion, -qhot, -qsimd=auto, -qlist, -qipa, -qcuda, ...
IBM XL compilers often generate the most efficient CPU code. CUDA Fortran is supported.
Two XL Fortran porting issues : -qextname to add an underscore; -WF,-DABC,-DXYZ for cpp.

clang, clang++, xlflang : OpenMP 4.5 features : -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda

nvcc : -arch=sm_60 option for Pascal GPUs; can specify compiler using -ccbin

Some Compiler Documentation – Useful Links

IBM XL C/C++ 13.1.5 pdf files :
http://www-01.ibm.com/support/docview.wss?uid=swg27048883

IBM XL Fortran 15.1.5 pdf files :
http://www-01.ibm.com/support/docview.wss?uid=swg27048884

PGI Documetation for OpenPOWER and NVIDIA Processors … pdf files :
https://www.pgroup.com/resources/docs-openpower.htm

GNU compilers : https://gcc.gnu.org/

The clang compiler : http://clang.llvm.org/

OpenMP specifications : http://www.openmp.org/

MPI on summitdev

IBM Spectrum MPI … based on openmpi, with IBM additions.

The most common compiler scripts are : mpicc, mpicxx, mpif90 … but there are many others:

```
mpiCC, mpic++, mpicc, mpicxx, mpif77, mpif90, mpifort
mpipgic++, mpipgicc, mpipgifort, mpixlC, mpixlc, mpixlf
```

The mpi* scripts use compilers controlled by env variables: OMPI_CC, OMPI_CXX, OMPI_FC. Your choice of modules will set the associated OMPI_* env variables.

Example : you have XL compilers loaded, but you want “mpicc” to use GNU compilers:

```
$ export OMPI_CC=gcc;  mpicc -show   => gcc is now being used
```

Note that the OMPI_* env variables take precedence for all of the mpi* compile scripts. For example, if you “module add clang/2.9.cuda8”, mpixlc -show => you have the clang compiler.

You can check what the mpicc compile script is doing : mpicc -show (same for mpicxx, mpif90).

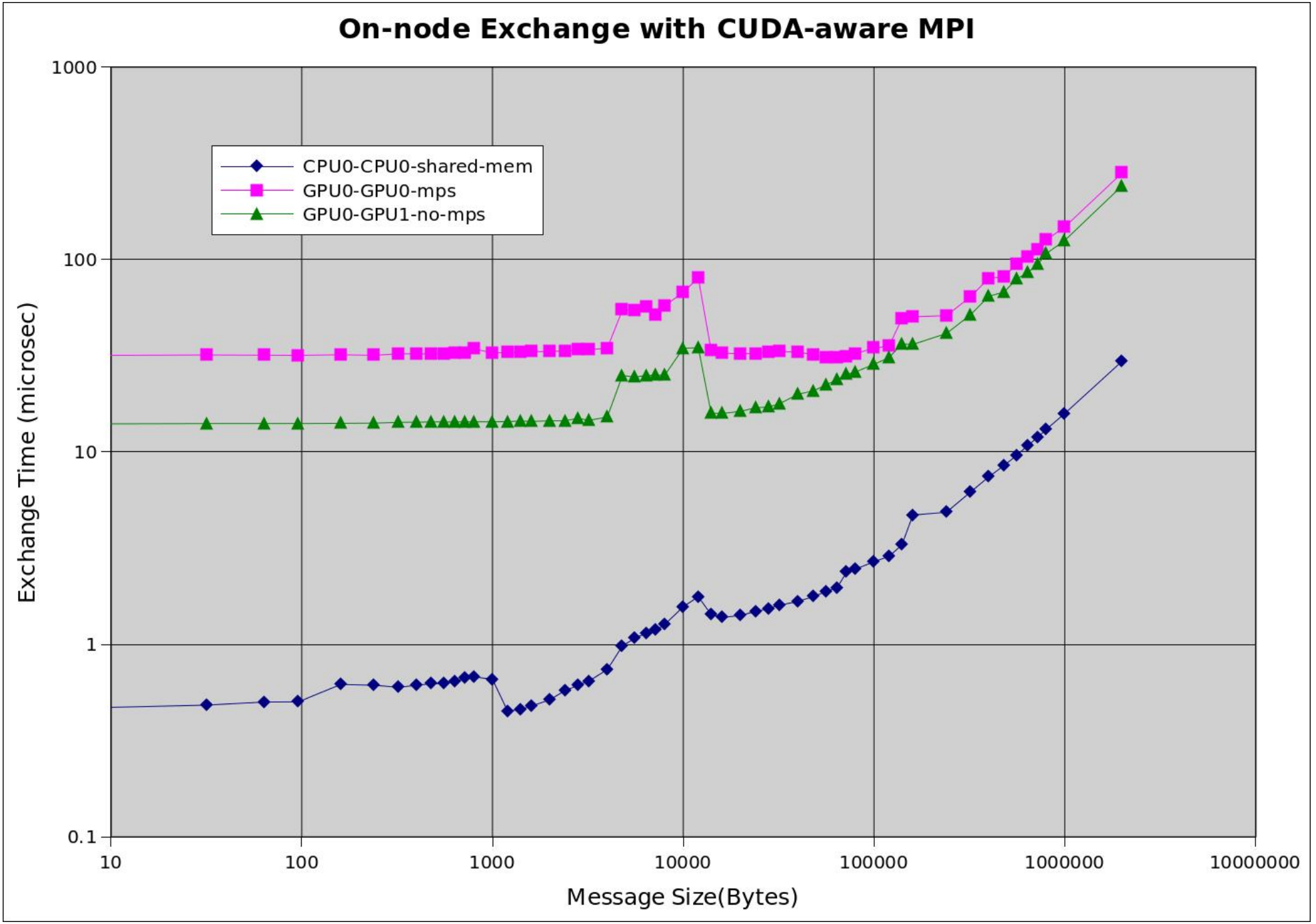
The “modules” approach sets PATH, LD_LIBRARY_PATH, and OMPI_* variables.

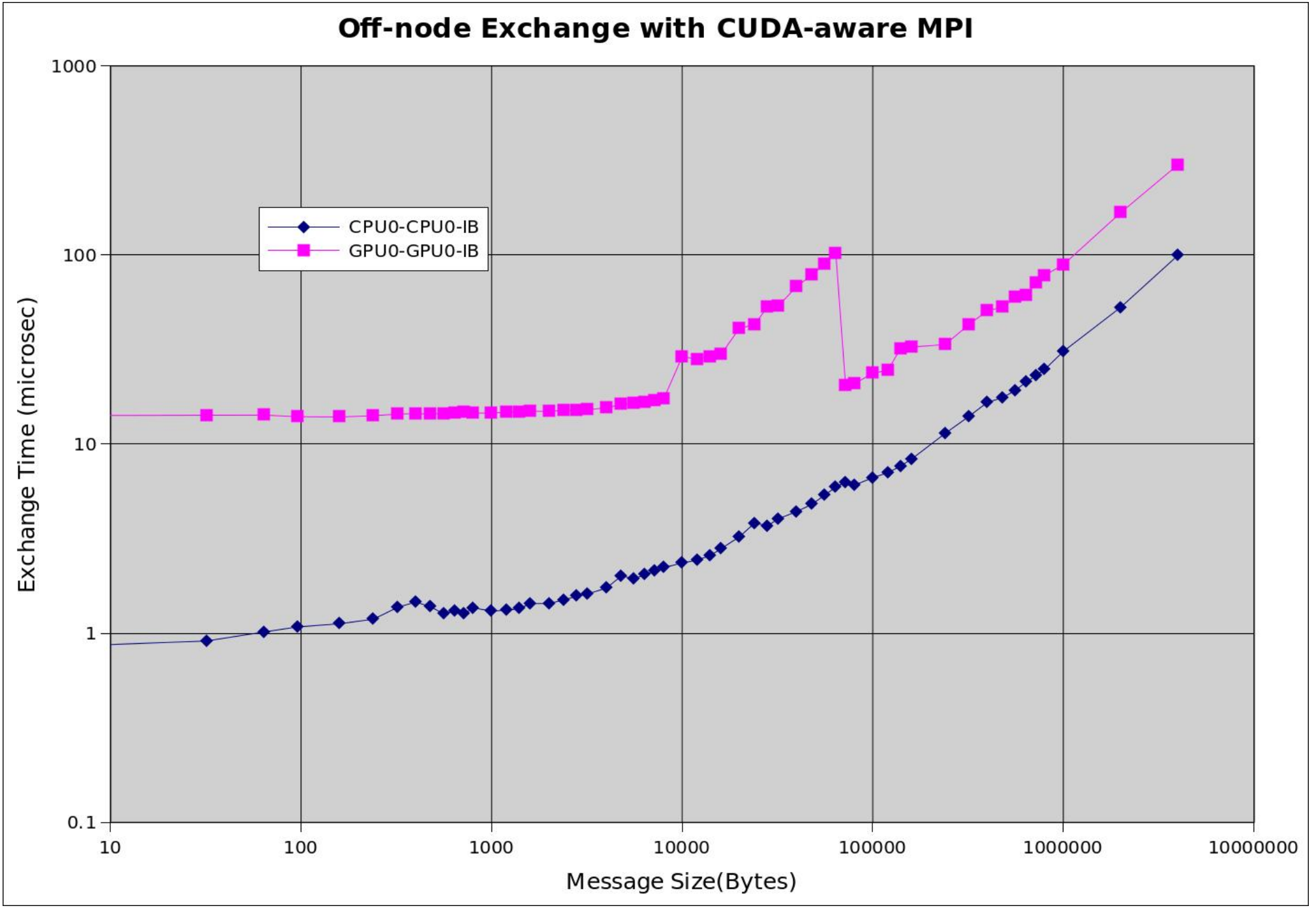
Most job scripts will use mpirun or mpiexec … look at output from mpirun --help .

To enable CUDA-aware support, you must use : mpirun -gpu …

To label output by MPI rank : mpirun --tag-output …; or mpirun -stdio p …

To get some insight into the MPI components : ompi_info


On-node Exchange with CUDA-aware MPI
CPU0-CPU0-shared-mem
GPU0-GPU0-mps
GPU0-GPU1-no-mps
Exchange Time (microsec)
Message Size(Bytes)
1000
100
10
1
0.1
10
100
1000
10000
100000
1000000
10000000


Off-node Exchange with CUDA-aware MPI
CPU0-CPU0-IB
GPU0-GPU0-IB
Exchange Time (microsec)
1000
100
10
1
0.1
10
100
1000
10000
100000
1000000
10000000
Message Size(Bytes)

Binding Processes and Threads

It is highly recommended to bind processes and threads, and to verify binding.

For serial tests, use the taskset command : taskset -c 8 a.out (binds a.out to logical cpu 8).
Check the man page for taskset for more info.

For single-process OpenMP tests, use OMP_PLACES.
Suggested syntax : export OMP_PLACES={startproc:numthreads:stride}.
Example: one thread per core on all cores of socket 1: export OMP_PLACES={80:10:8}.

The mpirun command has affinity options, but they are not sufficient for many purposes.
The same is true for LSF … it can set an affinity mask, but it cannot enforce thread binding.
If you have a non-threaded code and you want one MPI rank per core, you can use
mpirun --bind-to core …

Spectrum MPI exports env variables OMPI_COMM_WORLD_LOCAL_RANK, and
OMPI_COMM_WORLD_LOCAL_SIZE. You can use these in a helper script.

```
#!/bin/bash
let cpus_per_rank=160/$OMPI_COMM_WORLD_LOCAL_SIZE)
let startproc=$cpus_per_rank*$OMPI_COMM_WORLD_LOCAL_RANK
let stride=cpus_per_rank/$OMP_NUM_THREADS
export OMP_PLACES={$startproc:$OMP_NUM_THREADS:$stride}
$@
```

mpirun --bind-to none -np NumRanks helper.sh your.exe [args]

Caveat: the mpirun command sets default affinity masks that can interfere with your attempt to control affinity with OMP_PLACES, so it is best to disable mpirun affinity.

For a more elaborate example binding helper script : /ccs/home/walkup/bin/bind.sh

Routines to Set Affinity or Query Affinity

```
#define   _GNU_SOURCE
#include <sched.h>
int cpu = sched_getcpu();  // returns the logical cpu for the executing thread

#define _ GNU_ SOURCE
#include <pthread.h>
cpu_set_t cpuset;
int rc, cpu = 8;
pthread_t thread = pthread_self();
CPU_ZERO(&cpuset);
CPU_SET(cpu, &cpuset);
rc = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset);
…
rc = pthread_getaffinity_np(thread, sizeof(cpu_set_t), &cpuset);
// use : int CPU_ISSET(int cpu, &cpuset); to see if “cpu” is in the affinity mask
```

Note that using pthread_setaffinity_np() inside your application amounts to re-binding the thread. For OpenMP threads, it is preferable to use OMP_PLACES, because the OpenMP runtime can use that to get affinity right from the start, instead of re-binding later.

Example of using pthread_setaffinity_np(): /ccs/home/walkup/codes/bind/bindthreads.c

You can use the ps command to check affinity. Script “psbind” is listed below:

```
#!/bin/bash
if [ -z "$1" ]; then
  echo syntax: psbind executable_file
  exit
fi
for pid in $(pgrep $1); do ps -mo pid,tid,fname,user,psr -p $pid;done
```

LSF on summitdev

For LSF documentation, version 10.1 is currently on summitdev … web link :

http://www.ibm.com/support/knowledgecenter/SSWRJV_10.1.0/lsf_welcome/lsf_welcome.html
For pdf files, do a web search for : IBM Spectrum LSF Wiki

Essential LSF commands : bsub, bjobs, bkill, bpeek, bqueues, bmod, brsvs, bhosts

summitdev has man pages : man bkill (for example)

```
$ env | grep LSF
LSF_SERVERDIR=/opt/lsf/10.1/linux3.10-glibc2.17-ppc64le/etc
LSF_LIBDIR=/opt/lsf/10.1/linux3.10-glibc2.17-ppc64le/lib
LSF_BINDIR=/opt/lsf/10.1/linux3.10-glibc2.17-ppc64le/bin
LSF_ENVDIR=/opt/lsf/conf
```

Defaults for things like bjobs outputs are defined in the LSF configuration file (/opt/lsf/conf/lsf.conf).

My suggestion for bjobs output format (I put this in my .bash_profile) :

```
export LSB_BJOBS_FORMAT="jobid:7 queue:10 user:12 stat:7 slots:8 first_host:18 submit_time:13 start_time:13 run_time:15"
```

Simple LSF commands :

```
bjobs          : lists your jobs  (uses the format defined by LSB_BJOBS_FORMAT)
bjobs -uall    : lists all jobs
bjobs -l jobid : lists the details for a specified jobid
bkill jobid    : requests LSF to kill the specified job
bpeek jobid    : look at a snapshot of stdout/stderr
brsvs          : show reservations
```

Simple LSF job scripts

(1) Grab one interactive node for 10 minutes :

```
$ bsub –n 1 –Is –P project –W 10 –q interactive /bin/bash
```

After getting a login shell, you might want to source your .bash_profile file.

(2) Simple MPI job script … this script builds the batch.job file and submits the job.
If the script below is named run.sh, you submit the job by typing : run.sh [enter]

```
#!/bin/bash
nodes=4
ppn=20
let nmpi=$nodes*$ppn
#-------------------------------------
cat >batch.job <<EOF
#BSUB -o %J.out
#BSUB -e %J.err
#BSUB -R "span[ptile=$ppn]"              (sets the number of processes per node)
#BSUB -n $nmpi                           (sets the total number of processes)
#BSUB -q batch
#BSUB -P project
#BSUB -W 15
#-------------------------------------
export OMP_NUM_THREADS=1
ulimit -s 10240
mpirun –bind-to core -np $nmpi  p3d
EOF
#-------------------------------------
bsub  <batch.job
```

For LSF jobs, the -R option specifies resources, in this case -R “span[ptile=20]” => 20 MPI ranks/node.
It is not necessary to specify “-np 80” in the mpirun command … that is already specified by #BSUB -n.
I like to keep the “-np” option to make the mpirun command more clear to humans.

More LSF Job scripts

Same MPI example directly using the bsub command :

Job script batch.job, contents listed below:

```
#BSUB -o %J.out
#BSUB -e %J.err
#BSUB -R "span[ptile=20]"
#BSUB -n 80
#BSUB -q batch
#BSUB -P project
#BSUB -W 15
export OMP_NUM_THREADS=1
ulimit -s 10240
mpirun –bind-to core -np 80  p3d
```

```
$ bsub <batch.job
```

For each option specified by #BSUB, there is a corresponding command-line option for bsub.

I prefer scripts like the one on the previous chart for simpler job submission..

Interactive multi-node MPI jobs via LSF

You can use LSF to grab some number of nodes, and run multi-node MPI jobs interactively.

First do the one-time setup for ssh between nodes (cp id_rsa.pub authorized_keys; in your .ssh dir).

(1) grab Num nodes for 30 minutes (note the -n and ptile settings)

```
$ bsub -n Num -R "span[ptile=1]" -Is -P project -W 30 -q interactive /bin/bash
```

(2) make a hostfile to use with mpirun using $LSB_HOSTS

```
$ echo $LSB_HOSTS | sed 's/ /\n/g' > hf
```

(3) run your job interactively (note the --oversubscribe option)

```
$ mpirun --oversubscribe -hostfile hf -npernode 20 -np numRanks your.exe [args]
```

The “--oversubscribe” option is needed because LSF thinks you wanted just one “slot” per node.

The batch job submission method is more typical for MPI jobs, but you can use MPI interactively on the resources allocated to you by LSF if you want to.

For batch MPI jobs, you can ssh to the nodes used by your job while that job is active. This can be very useful for diagnostic purposes.

If you need to exclude a “bad” node : #BSUB -R “select[hname!='badnode']” …please notify the admins so the bad node can be taken out of the LSF queue.

Using NVIDIA MPS via LSF on summitdev

https://www.olcf.ornl.gov/kb_articles/summitdev-quickstart/#Batch_Scripts

MPS = multi-process service … designed to support multiple processes per GPU.

To request MPS, add a line to your LSF batch job script :

#BSUB -env "all,JOB_FEATURE=gpumps"

The “all” in the -env option is needed to reproduce your PATH (etc.) on compute nodes.

For MPS to work properly, you need to assign a device to each MPI rank. A block distribution of MPI ranks to devices normally best. Use a helper script or explicit calls in your code:

```
int ranks_per_node = get_ranks_per_node();
int local_rank = myrank%ranks_per_node;
CUDA_RC(cudaGetDeviceCount(&numDevices));
int myDevice = (local_rank * numDevices) / ranks_per_node;
CUDA_RC(cudaSetDevice(myDevice));
```

Example of a helper script to set a GPU device for each MPI rank (set_device.sh) :

```
#!/bin/bash
let ngpus=4
let product=$ngpus*$OMPI_COMM_WORLD_LOCAL_RANK
let mydevice=$product/$OMPI_COMM_WORLD_LOCAL_SIZE
export CUDA_VISIBLE_DEVICES=$mydevice
# run the program
"$@"

mpirun -np numProcs set_device.sh  your.exe [args]
```

You can also explicitly manage MPS yourself with a helper script to start/stop daemons.

Monitoring LSF job output

LSF on summitdev is configured with LSB_STDOUT_DIRECT=y (see /opt/lsf/conf/lsf.conf).

Job output/error “go directly to the destination file” … but NFS does not immediately update the output files on the login node.

The LSF method to see a current snapshot of your output : bpeek jobid

From the login node, the command “bpeek jobid” is basically equivalent to :

```
$ ssh  first_host  cat  $working_directory/jobid.out
```

While your job is running, you can ssh to the “first_host” and monitor your job from there. That will take out the NFS remote update issue. To identify the first host, you can use commands such as :

```
$ bjobs -l
```

I set LSB_BJOBS_FORMAT to include “first_host” in the default output from the “bjobs” command.

export LSB_BJOBS_FORMAT="jobid:7 queue:10 user:12 stat:7 slots:8 first_host:18 submit_time:13 start_time:13 run_time:15"

```
[summitdev-login1:~/codes/p3d] bjobs
JOBID   QUEUE      USER         STAT    SLOTS    FIRST_HOST         SUBMIT_TIME   START_TIME    RUN_TIME
45112   batch      walkup       RUN     80       summitdev-r0c0n17  Jan  8 10:50  Jan  8 10:50  21 second(s)
```

When your job is running, LSF also writes stdout/stderr to temporary files in your $HOME/.lsbatch directory. Those files will have the same NFS update issue.