Practical Tips for Running on SUMMIT

David Appelhans

ORNL SUMMIT Training Workshop
Feb 11, 2019
OUTLINE

- Talk is a collection of hands on experience in readying applications for SUMMIT.

- Lightweight MPI profiler.

- Useful jsrun/MPI flags.

- LSF+jsrun example submission scripts
  - 1 rank per socket, core, or GPU example scripts.
  - Using multiple ranks per GPU with MPS.

- Profiling of 1 rank with nvprof at scale.

- Checking on job submission
  - Useful lsf commands.
  - ssh to compute node, get stack traces, nvidia smi.
**Best Practice: Lightweight MPI Tracing**

A lightweight MPI tracing library can easily be linked against at runtime\(^1\).

- No performance penalty—we ran this library with all CORAL benchmarks during acceptance.
- Very useful in identifying node imbalance at scale.
- How to use it
  - Official version shipped with SMPI:
    ```
    export OMPI_LD_PRELOAD_POSTPEND=$OLCF_SPECTRUM_MPI_ROOT/lib/libmpitrace.so
    ```

It writes several profile files with format of mpi_profile.jobid.rank

\(^1\)Vampire is another available profiling tool
# Using the MPI Profiler

Can see where time is spent, calculate achieved communication bandwidth, average message size, etc.

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>calls</th>
<th>avg. bytes</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_rank</td>
<td>320850</td>
<td>0.0</td>
<td>0.010</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>4</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>84</td>
<td>540688.0</td>
<td>0.036</td>
</tr>
<tr>
<td>MPI_Send_init</td>
<td>168</td>
<td>8404992.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Recv_init</td>
<td>168</td>
<td>8404992.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>84</td>
<td>540688.0</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>3672</td>
<td>0.0</td>
<td>18.112</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>42</td>
<td>0.0</td>
<td>0.007</td>
</tr>
<tr>
<td>MPI_Start</td>
<td>3588</td>
<td>0.0</td>
<td>0.038</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>24</td>
<td>37.2</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>411</td>
<td>0.0</td>
<td>34.302</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>1</td>
<td>4.0</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>696</td>
<td>20.0</td>
<td>8.602</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>1</td>
<td>4.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Gatherv</td>
<td>2</td>
<td>288.0</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Total communication time = 61.108 seconds.
Total elapsed time = 239.719 seconds.
User CPU time = 2211.329 seconds.
System time = 372.080 seconds.
Max resident set size = 37583.938 MBytes.
IDENTIFYING WORK IMBALANCE

Ranks taking the most time in computation spend the least amount of time in MPI calls because all other ranks have been waiting for them. Example rank 0 output info:

<table>
<thead>
<tr>
<th>Histogram of times spent in MPI</th>
<th>Histogram of times spent in MPI (Balanced Run)</th>
</tr>
</thead>
<tbody>
<tr>
<td>time—bin</td>
<td>time—bin</td>
</tr>
<tr>
<td>ranks</td>
<td>ranks</td>
</tr>
<tr>
<td>30.231</td>
<td>18.413</td>
</tr>
<tr>
<td>32.611</td>
<td>19.261</td>
</tr>
<tr>
<td>34.991</td>
<td>20.110</td>
</tr>
<tr>
<td>37.371</td>
<td>20.958</td>
</tr>
<tr>
<td>39.751</td>
<td>21.807</td>
</tr>
<tr>
<td>42.131</td>
<td>22.655</td>
</tr>
<tr>
<td>44.511</td>
<td>23.504</td>
</tr>
<tr>
<td>46.891</td>
<td>24.352</td>
</tr>
<tr>
<td>49.271</td>
<td>25.201</td>
</tr>
<tr>
<td>51.651</td>
<td>26.049</td>
</tr>
<tr>
<td>54.030</td>
<td>26.898</td>
</tr>
<tr>
<td>56.410</td>
<td>27.746</td>
</tr>
<tr>
<td>58.790</td>
<td>28.595</td>
</tr>
<tr>
<td>61.170</td>
<td>29.444</td>
</tr>
<tr>
<td>63.550</td>
<td>30.292</td>
</tr>
</tbody>
</table>
**USING THE MPI PROFILER**

Rank 0 summary file also provides rank to node correlation:

<table>
<thead>
<tr>
<th>taskid</th>
<th>hostname</th>
<th>cpu</th>
<th>comm(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>290</td>
<td>sierra3358</td>
<td>88</td>
<td>52.04</td>
</tr>
<tr>
<td>291</td>
<td>sierra3358</td>
<td>128</td>
<td>52.29</td>
</tr>
<tr>
<td>292</td>
<td>sierra3800</td>
<td>0</td>
<td>43.48</td>
</tr>
<tr>
<td>293</td>
<td>sierra3800</td>
<td>40</td>
<td>30.23</td>
</tr>
<tr>
<td>294</td>
<td>sierra3800</td>
<td>88</td>
<td>54.34</td>
</tr>
<tr>
<td>295</td>
<td>sierra3800</td>
<td>128</td>
<td>51.69</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
USEFUL JSRUN FLAGS

- Send kill signal to processes on a MPI failure (helps kill your job instead of hanging it).
  ```
  jsrun -X 1 <further commands>
  ```

- Prepend the rank id to the output
  ```
  jsrun --stdio_mode prepended
  ```

- If Spectrum MPI arguments are needed (e.g. async argument when using 1-sided communication)
  ```
  jsrun --smiargs="--async"
  ```
EXAMPLE SUBMISSION SCRIPTS

• ORNL training material is very good (link + visualizer).

• Using scripts for lsf job submission and jsrun configuration allows repeatable experiments.

• See talk by Chris Fuson for detailed explanation of lsf and jsrun.

• Key to jsrun is deciding what a resource set will represent (e.g. 1 rs per socket, or per GPU).

• The following are some examples of common use cases. Links to full submission scripts are on each page.
1 Rand per Core (CLICKABLE LINK)

nodes=1

gpus_per_socket=3 # number of gpus to use per socket
application_cores =21 # cores available to the application per socket
threads_per_core=4 # Each core can go up to smt4 for 4 hardware threads.
# user sets rank_per_socket, calculate other quantities from this:
ranks_per_socket=21 # needs to be evenly divisible by gpus_per_socket (if using GPUs)

# calculated from input:
let num_sockets=2*$nodes
let cores_per_rank = $application_cores / $ranks_per_socket # avail cores divided into the ranks.
let cores_per_socket = $cores_per_rank*$ranks_per_socket # this is used cores per socket (not necessarily equal to application cores
let threads_per_rank = $threads_per_core*$cores_per_rank

... LSF submission stuff ...

jsrun --stdio_mode=prepend --D CUDA_VISIBLE_DEVICES \
  --OMP_NUM_THREADS=${threads_per_rank} \  
  --nrs ${num_sockets} \  
  --tasks_per_rs ${ranks_per_socket} \  
  --cpu_per_rs ${cores_per_socket} \  
  --gpu_per_rs ${gpus_per_socket} \  
  --bind=proportional--packed:${cores_per_rank} \  
  --d plane:${ranks_per_socket} \  
  ./ print -- affinity .sh
**1 RANK PER GPU** (CLICKABLE LINK)

```plaintext
nodes=1
gpus_per_socket=3 # number of gpus to use per socket (3 for summit, 2 for sierra)
gpus_per_rs=1 # 1 res set per GPU (one to one gpu to rs mapping).
threads_per_core=2 # Each core can go up to smt4 for 4 hardware threads.
ranks_per_rs=1 # If using more than 1 rank per gpu, need to enable mps through lsf.

# derived quantities:
let rs_per_socket=gpus_per_socket/gpus_per_rs
let ranks_per_socket=ranks_per_rs*rs_per_socket
# There are 21 (of 22) cores available to the application per socket (on Summit)
let cores_per_rank=21/ranks_per_socket # 21 avail cores divided into the ranks.
let cores_per_rs=cores_per_rank*ranks_per_rs
let nrs=2*rs_per_socket*nodes # total number of resource sets:
let threads_per_rank=threads_per_core*cores_per_rank

... LSF submission stuff ...

jsrun --stdio_mode=prepend -D CUDA_VISIBLE_DEVICES 
  -E OMP_NUM_THREADS=${threads_per_rank} 
  -nrs ${nrs} -n tasks_per_rs ${ranks_per_rs} 
  -cpu_per_rs ${cores_per_rank} 
  -gpu_per_rs ${gpus_per_socket} 
  -bind=proportional-packed:${cores_per_rank} 
  -d plane:${ranks_per_rs} 
  ./ print -affinity.sh
```
**Sharing GPU among 2 ranks**  
[Click for full example](#)

One resource set per GPU, multiple tasks per resource set.

```plaintext
# derived quantities:
let rs_per_socket = gpus_per_socket / gpus_per_rs
let ranks_per_socket = ranks_per_rs * rs_per_socket

# There are 21 (of 22) cores available to the application per socket (on Summit)
let cores_per_rank = 21 / ranks_per_socket # 21 avail cores divided into the ranks.
let cores_per_rs = cores_per_rank * ranks_per_rs
let nrs = 2 * $rs_per_socket * $nodes
let threads_per_rank = threads_per_core * cores_per_rank

# Must enable MPS in lsf submission:

```bash
#BSUB -alloc_flags gpumps
jsrun --stdio_mode=prepend -D
    CUDA_VISIBLE_DEVICES \n    -E OMP_NUM_THREADS=${threads_per_rank} \n    -n nrs ${nrs} \n    -t tasks_per_rs ${ranks_per_rs} \n    -d cpu_per_rs ${cores_per_rs} \n    -g gpu_per_rs ${gpus_per_rs} \n    -bind=proportional,packed=${cores_per_rank} \n    --thread bind
./print_affinity.sh
```

---

2 If specifying multiple flags, must do it in one line. E.g. #BSUB -alloc_flags “smt2 gpumps“
**CHECKING ON YOUR SUBMISSION**

- `bjobs` will show a list of your current jobs and their status.
- To check pending reason of a specific job, `bjobs -p <jobid>`
- Is the que busy/open? `bqueues`

```
[dappelh@login5.summit]$ bqueues
QUEUE_NAME  PRIO  STATUS       MAX JL/U JL/P JL/H NJOBS PEND RUN SUSP
  test        5     Open:Inact    45   -   -   -   0   0   0   0
 storage      4     Open:Inact    -   -   -   -   0   0   0   0
  tested      3     Open:Active   -   -   -   -   0   0   0   0
    new       2     Open:Inact    -   -   -   -   43  43   0   0
 batch       1     Open:Active   -   -   -   -   85  85   0   0
```

- What jobs are scheduled on the machine? `bjobs -u all`

```
[dappelh@login1.summit]$ bjobs -u all
 JOBID USER STAT SLOTS QUEUE START_TIME FINISH_TIME JOB_NAME
 220841 bykov RUN 86017 batch Nov 28 14:15 Nov 29 02:15 lsdalton
 220932 jaharris RUN 43    batch Nov 28 17:37 Nov 28 18:07 Not_Specified
 220933 ngawande RUN 2017    batch Nov 28 17:40 Nov 28 19:40 d48_c
 220908 chochia PEND —    tested — — Not_Specified
 220926 chochia PEND —    tested — — Not_Specified
```
CHECKING ON A RUNNING JOB

• *bpeek* will show you tail of standard out and standard error (jobid is optional):

  bpeek <jobid> | less

• Something seems wrong, can I check into a job further?

  bjobs -WP <jobid> | less

<table>
<thead>
<tr>
<th>JOBID</th>
<th>USER</th>
<th>STAT</th>
<th>QUEUE</th>
<th>FROM_HOST</th>
<th>EXEC_HOST</th>
<th>JOB_NAME</th>
<th>SUBMIT_TIME</th>
<th>%COMPLETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>220933</td>
<td>ngawand</td>
<td>RUN</td>
<td>batch</td>
<td>login1</td>
<td>batch1</td>
<td>d48_c</td>
<td>Nov 28 17:40</td>
<td>32.62% L</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g36n12</td>
<td>g36n12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Can ssh to a compute node (only while job is running)

  ssh g36n12
Troubleshooting a Job on a Compute Node

- Use gstack to get call stack of each thread (use top to get pid)

```
gstack <pid>
```

- See if the GPUs have something currently running:

```
nvidia-smi
```

Example idle GPUs:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.58 Driver Version: 396.58                                 |
+-----------------------------------------------------------------------------+
| GPU Name Persistence-M| Bus-Id Disp.A | Volatile Uncorr. ECC |                  |                     |
| Fan Temp Perf Pwr:Usage/Cap| Memory-Usage | GPU-Util Compute M. |
|=============================================================================|
| 0 Tesla V100-SXM2... On | 00000004:04:00.0 Off | 0 |
| N/A 34C P0 36W / 300W | 0MiB / 16128MiB | 0% E. Process |
+-----------------------------------------------------------------------------+
| Processes: GPU Memory |
| GPU PID Type Process name Usage |
|=============================================================================|
| No running processes found |
+-----------------------------------------------------------------------------+
```
Profiling at Scale

- Running `nvprof` on every rank of a large scale job will slow to a crawl.
- However, profiling just one of the ranks at scale with `nvprof`/visual profiler is possible.

- See talk later this week on detailed usage of `nvprof` and visual profiler as well as Score-P/Vampire.
SCRIPT FOR 1 RANK TO LAUNCH NVPROF

Have jsrun launch this script which executes profiler+application if rank matches profile rank, otherwise just launches application without profiling:

```bash
export PROFILE_RANK=1
export PROFILE_PATH="/gpfs/path_to_your_directory"
jsrun <flags> profile_helper.sh a.out
```

Contents of profile_helper.sh:

```bash
#!/bin/bash
if [ $PMIX_RANK == $PROFILE_RANK ]; then
nvprof -f -o $PROFILE_PATH "$@
else
"$@
fi
```
CLOSING REMARKS

Hopefully you have learned how to

• Use the lightweight mpi profiler even if not focusing on MPI performance.
• Use scripts for lsf/jsrun submissions.
• Check on your job submission after submitting.
• Launching a profiler at scale on a single MPI rank.

Questions?

David Appelhans - dappelh@us.ibm.com

Github script location: https://github.com/dappelha/summit-scripts