Use the UMassCS Swarm2 cluster efficiently for your research!

Keen Sung

February 5, 2019
Objectives

• Learn the basic architecture of swarm

• Walk through how to parallelize and run a job

• Tips for optimizing

• Checkpointing

• Troubleshooting
Swarm v Gypsum

**Swarm**
- 50 nodes
  - 56 cores
  - 128 GB RAM
- Total
  - 2800 cores
  - 6.4TB RAM

**Gypsum**
- 100 nodes
  - 4 GPU (25: M40, 75: TITAN X)
  - 24 cores
  - 256 GB RAM
- 53 nodes
  - 8 GPU (1080Ti)
  - 48 cores
  - 384 GB RAM
- Total
  - 4944 cores
  - 30.72TB RAM
  - 824 GPU
Clarifying ambiguous terminology

- **swarm**
- **node**
  - CPU/Socket
  - 128MB RAM
  - 500GB SSD
- **Socket**
  - Race gate on the motherboard for one physically packaged processor (each of which can contain one or more cores)
- **Core**
  - A complete private set of registers, execution units, and retirement queues needed to execute programs
- **Threads**
  - One or more hardware contexts within a single core. Each thread has attributes of one core, managed & scheduled as a single logical processor by the OS

https://slurm.schedmd.com/mc_support.html
Clarifying ambiguous terminology

#!/bin/bash
#SBATCH -N 1
#SBATCH -c 2
cpus per task
#SBATCH -n 1
#SBATCH --mem=1G
num tasks

https://slurm.schedmd.com/mc_support.html
SLURM

• Queuing and scheduling system
• Tries to account for fairness
  • Priority queue based on a fairness score calculated by current and historical usage of CPU or RAM by you and your group, and the age of submission

Leonenkov and Zhumatiy (2015)
Introducing New Backfill-based Scheduler for SLURM Resource Manager
Resource Accounting and Limits

**Swarm**

Disk space:
- /home (10GB)
- /work1 (2TB)

User limits:
- 2240/2800 CPU limit
- 5.01.0/6.4 TB RAM limit
- 10GB/allocated core (10GB/2 CPU)
Remember these rules

• **DO NOT** run anything on the head node --- always use srun or sbatch for anything computationally intensive

• **DO NOT** overallocate **time**, **memory**, or **CPU**

• **CHECK** your own jobs

**BE RESPECTFUL!**
First, you need an account

• Step 1. Get an account by having your advisor email CSCF
• Step 2. Log in with your CICS account

```
$ ssh ksung@swarm2.cs.umass.edu
ksung@swarm2.cs.umass.edu's password: hunter2
Welcome to Bright release         7.3

Based on CentOS Linux 7
ID: #000002

Use the following commands to adjust your environment:

'module avail' - show available modules
'module add <module>' - adds a module to your environment for this session
'module initadd <module>' - configure module to be loaded at every login

-------------------------------------------------------------

[ksung@swarm2 ~]$```

SLURM commands

sbatch --- run an sbatch formatted file (normal way to run something)

srun --- run a command with specified resources. If within an sbatch file, it must be less than or equal to sbatch allocation. By default, the sbatch allocation will be used

squeue --- look at all submitted jobs by all users
Let’s get something running!

- Example can be found in my home directory:
  /home/ksung/resources
    /process_example

Note: copy the whole directory to your own home directory before testing
Let’s get something running!

**Goal:** parallelize the file on the right

**Method 1:** Make it runnable with command line arguments

**Method 2:** Parallelize it with a python library

```python
# process_serial.py

import data

def compute(x):
    results_list = []
    for i in range(int(1e7)):
        v = data[(x+i)%len(data)]
        result = v * v
        results_list.append(result)
    return sum(results_list)

# 1.86 sec baseline
all_results = []
for _ in range(100):
    all_results = compute(int(rand()*1e7))
# 10.31 sec for 1 run / 119096k RSS
with open('results','w') as results_file:
    results_file.writelines([str(r) for r in all_results])
```

from random import random

# baseline 8088k RSS
with open('data','r') as data_file:
    data = data_file.readlines()
    # 801988k RSS, 184MB filesize
Let’s get something running!

generate.py (generate example data to work with
--- shown here for replicability)

```python
from random import random

def generate_data_file():
    with open('data', 'w') as data_file:
        for _ in range(int(1e7)):
            data_file.write(str(random()) + '
')
generate_data_file()
```
Let's get something running --- profiling

Use `srun` and `time` to test and profile the script

```
from random import random
```

```
0.04 sec runtime at 90% CPU
8M memory
```
Let's get something running!

process_serial.py

```python
from random import random

with open('data', 'r') as data_file:
    data = data_file.readlines()
```

1.75 sec runtime at 97% CPU

803M memory
from random import random

# baseline 8088k RSS
with open('data', 'r') as data_file:
    data = data_file.readlines()
# 801988k RSS, 184MB filesize

def compute(x):
    results_list = []
    for i in range(int(1e7)):
        v = data[(x+i)%len(data)]
        result = v * v
        results_list.append(result)
    return sum(results_list)

# 1.86 sec baseline
all_results = []
for _ in range(100):
    all_results = compute(int(rand()*1e7))
# 10.31 sec for 1 run / 1119096k RSS
with open('results', 'w') as results_file:
    results_file.writelines([str(r) for r in all_results] )
```python
# baseline 801988k RSS
with open('data', 'r') as data_file:
    data = data_file.readlines()
    # 801988k RSS, 184MB filesize

def compute(x):
    results_list = []
    for i in range(int(1e7)):
        v = data[(x+i)%len(data)]
        result = v * v
        results_list.append(result)
    return sum(results_list)

# 1.86 sec baseline
all_results = []
for _ in range(100):
    all_results = compute(int(rand() * 1e7))
    # 10.31 sec for 1 run / 1119096k RSS
    with open('results', 'w') as results_file:
        results_file.writelines([str(r) for r in all_results])
```

```python
from random import random
import random, sys
run_number = sys.argv[1]
with open('data', 'r') as data_file:
    data = data_file.readlines()

def compute(x):  # profile this
    results_list = []
    for i in range(int(1e7)):
        v = data[(x+i)%len(data)]
        result = v * v
        results_list.append(result)
    return sum(results_list)

with open('results' + str(run_number), 'w') as results_f:
    results_f.write(str(compute(int(rand() * 1e7))) + '\n')
```

```
process_cmd.py
process_serial.py
```
# Anatomy of an sbatch file

```bash
#!/bin/bash
#SBATCH -j process_test  # name
#SBATCH -N 1             # number of nodes
#SBATCH -n 1             # number of tasks
#SBATCH -c 2             # number of cpus per task
#SBATCH --mem=1G         # memory per node
#SBATCH --mem-per-cpu=1G # memory per cpu
#SBATCH -a 0-99          # array
#SBATCH -t 00:01         # time allocated
#SBATCH -e process_test.err # error output file
#SBATCH -o process_out.out # stdout file

srun process.py ${SLURM_ARRAY_TASK_ID}
```

More info: https://slurm.schedmd.com/sbatch.html
#!/bin/bash
#SBATCH -j process_test
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -c 2
#SBATCH --mem=1G
#SBATCH -a 0-99
#SBATCH -e process_test.err
#SBATCH -o process_out.out

srun process.py ${SLURM_ARRAY_TASK_ID}

---

```python
from random import random
import random, sys

run_number = sys.argv[1]

with open('data', 'r') as data_file:
    data = data_file.readlines()

def compute(x):
    # profile this
    results_list = []
    for i in range(int(1e7)):
        v = data[(x+i)%len(data)]
        result = v * v
        results_list.append(result)
    return sum(results_list)

with open('results' + str(run_number), 'w') as results_f:
    results_f.write(
        str(compute(int(rand() * 1e7))) + '
    ))
```
run.sb

#!/bin/bash
#SBATCH -j process_test
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -c 2
#SBATCH --mem=1G
#SBATCH -a 0-99
#SBATCH -e process_test.err
#SBATCH -o process_out.out

srun process.py ${SLUBATCH_ARRAY_TASK_ID}

process_cmd.py

```python
from random import random
import random,sys

run_number = sys.argv[1]

with open('data','r') as data_file:
    data = data_file.readlines()

def compute(x):
    # profile this
```

```python

ts_f:  

str(compute(int(rand()*1e7)))+'
'
```

```python
```
#!/bin/bash
#SBATCH -j process_test
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -c 2
#SBATCH --mem=1G
#SBATCH -a 0-99
#SBATCH -e process_test.err
#SBATCH -o process_out.out

srun process.py ${SLURM_ARRAY_TASK_ID}
Post-hoc profiling (throttled result)
Hyperthreading

• Non-MKL benchmark

[k sung@swarm2 ~] $ sacct -j 9826135 -o MaxRSS,TotalCPU,CPUtilime,Elapsed
MaxRSS TotalCPU CPUtilime Elapsed
---------- ---------- ---------- ----------
00:09.680 00:00:26 00:00:13
729K 00:09.680 00:00:26 00:00:13

[k sung@swarm2 ~] $ sacct -j 9826136 -o MaxRSS,TotalCPU,CPUtilime,Elapsed
MaxRSS TotalCPU CPUtilime Elapsed
---------- ---------- ---------- ----------
00:35.875 00:00:38 00:00:19
729K 00:35.875 00:00:38 00:00:19

• MKL benchmark

[k sung@swarm2 benches] $ sacct -j 9825685 -o MaxRSS,CPUtilime,TotalCPU,Elapsed
MaxRSS CPUtilime TotalCPU Elapsed
---------- ---------- ---------- ----------
00:03:06 03:00.324 00:01:33
00:07:46 07:42.980 00:01:33

[k sung@swarm2 benches] $ sacct -j 9825304 -o MaxRSS,CPUtilime,TotalCPU,Elapsed
MaxRSS CPUtilime TotalCPU Elapsed
---------- ---------- ---------- ----------
00:07:46 07:42.965 00:01:32
00:35.875 00:00:38 00:00:19
Hyperthreading

• Users can only book one whole core at a time (two threads with hyperthreading)

• Forcing your program to use both threads will probably not significantly increase your efficiency. It will however look like you’re using only 50% of CPU

• Take advantage of libraries (like numpy) that optimize for hyperthreads! Python on swarm is compiled with Intel MKL support for hyperthreading. Anaconda’s release should come with it, too.
Using a library is usually better

**Pros**
- Don’t reinvent the wheel
- Can save memory and time
- Can consolidate (reduce) results more easily

**Cons**
- Libraries are language dependent
- It is sometimes harder to implement
Method 2 - multiprocessing library

```python
from random import random
from multiprocessing import Pool

# baseline 80188k RSS
with open('data','r') as data_file:
    data = data_file.readlines()
# 801988k RSS, 184MB filesize

def compute(x):
    results_list = []
    for i in range(int(1e7)):
        v = data[(x+i)%len(data)]
        result = v * v
        results_list.append(result)
    return sum(results_list)

def compute(x):
    # profile this
    results_list = []
    for i in range(int(1e7)):
        v = data[(x+i)%len(data)]
        result = v * v
        results_list.append(result)
    return sum(results_list)

p = Pool(20)
all_results = p.map(compute, [int(random()*1e7) for _ in range(100)]
with open('results','w') as results_file:
    results_file.writelines([str(r) for r in all_results])
```

Save memory with multiprocessing!
#!/bin/bash
#SBATCH --job-name process_test
#SBATCH --n 1
#SBATCH --c 8
#SBATCH --mem=2G
#SBATCH --error process_test.err
#SBATCH --output process_out.out

srun -c 8 python process_multi.py

(throttled result)
Don’t commit these sins

• **DO NOT** run anything on the head node --- always use srun or sbatch for anything computationally intensive

• **DO NOT** overallocate **time, memory, or CPU**

• **CHECK** your own jobs

**BE RESPECTFUL!**
Other tips

• Minimize reads and writes to disk

• Write fault-tolerant code
  • Save “state” often so that code can restart if it fails for any reason

• Make your program as fragmentable as possible. It is easier to schedule a high number of low resource jobs than a lower number of resource intensive jobs
Checkpointing with DMTCP

• Example can be found in my home directory:
  /home/ksung/resources
    /dmtcp_example

Note: copy the whole directory to your own home directory before testing
Checkpointing with DMTCP (experimental)

- Any job with more than one node will be buggy
- Saves memory state to filesystem

```
Start a job: sbatch slurm_launch.job
Continue a job: sbatch slurm_rstr.job
```

• /home/ksung/dmtcp_example
DMTCP

slurm_launch.out

```bash
[kunig@swarm2 dmtpc2]$ tail -30 dmtpc.out
swarm001 19
swarm001 19
swarm001 19
swarm001 19
swarm001 20
swarm001 20
swarm001 20
swarm001 20
swarm002 20
swarm002 20
swarm002 20
swarm002 20
swarm001 21
swarm001 21
swarm001 21
swarm001 21
swarm002 21
swarm002 21
swarm002 21
swarm002 21
swarm001 22
swarm001 22
swarm001 22
swarm001 22
swarm002 22
swarm002 22
swarm002 22
swarm002 22
swarm001 23
swarm001 23
swarm001 23
swarm001 23
swarm002 23
swarm002 23
swarm002 23
swarm002 23
swarm001 24
swarm001 24
swarm001 24
swarm001 24
swarm002 24
swarm002 24
swarm002 24
swarm002 24
```
DMTCP

Excerpt from slurm_launch.job

```bash
# 1. Start DMTCP coordinator
start_coordinator -i 10 # ... <put dmtcp coordinator options here>

# 2. Launch application
# 2.1. If you use mplex/mpirun to launch an application, use the following command line:
$ dmtcp_launch --rm mplex <mpi-options> ./<app-binary> <app-options>
# 2.2. If you use PMIx to launch an application, use the following command line:
$ srun dmtcp_launch --rm ./<app-binary> <app-options>
# Note: PMIx is not supported yet.
# 2.3. If you use the Stampede supercomputer at Texas Advanced Computing Center (TACC), use ibrun command to launch the application (---rm is not required):
# $ ibrun dmtcp_launch ./<app-binary> <app-options>

# dmtcp_launch --rm mpirun --mca btl self_tcp ./<your binary>
ibrun dmtcp_launch --rm python count.py
```

Excerpt from slurm_rstr.job

```bash
# 1. Start DMTCP coordinator
start_coordinator # -i 120 ... <put dmtcp coordinator options here>

# 2. Restart application

/bin/bash ./dmtcp_restart_script.sh -h $DMTCP_COORD_HOST -p $DMTCP_COORD_PORT
```

# If you use the Stampede supercomputer at Texas Advanced Computing Center (TACC), add the --hostfile option:
```
# /bin/bash ./dmtcp_restart_script.sh -h $DMTCP_COORD_HOST -p $DMTCP_COORD_PORT
# --hostfile $HOSTFILE
```
Troubleshooting

- Memory error

slurmstepd: error: Step 9829757.0 exceeded memory limit (126337 > 102400), being killed
slurmstepd: error: *** STEP 9829757.0 ON swarm001 CANCELLED AT 2019-02-02T23:35:55 ***
slurmstepd: error: Exceeded job memory limit
srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
srun: error: swarm002: task 7: Killed
srun: error: swarm001: tasks 0-1,3: Killed
Troubleshooting

• Time expiry error
  • SIGTERM 32 sec before SIGKILL

```python
import signal
import sys
from time import sleep

def sigterm_handler(_signo, _stack_frame):
    print('sorry', flush=True)
    for i in range(1000):
        print(i, flush=True)
        sleep(1)
    sys.exit(0)

signal.signal(signal.SIGTERM, sigterm_handler)
sleep(600)
```

```
ksung@swarm2 dmtcp2]$ srun -t 00:00:01 python term_test.py
srun: Force Terminated job 9844110
srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
slurmstepd: error: *** STEP 9844110.0 ON swarm002 CANCELLED AT 2019-02-05T14:07:34 DUE TO TIME LIMIT ***
sorry
0
.
.
.
28
29
srun: error: swarm002: task 0: Killed
```
Troubleshooting

• Allocation error --- your allocation doesn’t make sense

• Assoc Limit --- you or your group is currently already maxing out your resource limit

• Resource --- you are first in line but there are not enough resources for your job

• Priority --- you are waiting for the first in line (Resource) to be scheduled
Troubleshooting
/home/ksung/blame.py

```
/home/ksung/blame.py

Percent of resources allocated per user

<table>
<thead>
<tr>
<th>user</th>
<th>alloc cpu</th>
<th>alloc mem</th>
<th>wasted cpu</th>
<th>wasted mem</th>
<th>num_nodes</th>
<th>cpus_per_node</th>
<th>mem_per_node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7.1%</td>
<td>12.5%</td>
<td>2.2%</td>
<td>11.2%</td>
<td>15</td>
<td>14.7</td>
<td>63.0G</td>
</tr>
<tr>
<td></td>
<td>4.2%</td>
<td>13.5%</td>
<td>1.9%</td>
<td>5.6%</td>
<td>14</td>
<td>9.1</td>
<td>73.1G</td>
</tr>
<tr>
<td></td>
<td>1.9%</td>
<td>1.3%</td>
<td>1.7%</td>
<td>1.3%</td>
<td>6</td>
<td>10.0</td>
<td>16.7G</td>
</tr>
<tr>
<td></td>
<td>1.0%</td>
<td>1.2%</td>
<td>0.8%</td>
<td>1.0%</td>
<td>1</td>
<td>30.0</td>
<td>93.0G</td>
</tr>
<tr>
<td></td>
<td>0.6%</td>
<td>0.1%</td>
<td>0.3%</td>
<td>0.1%</td>
<td>5</td>
<td>3.6</td>
<td>2.2G</td>
</tr>
<tr>
<td></td>
<td>0.1%</td>
<td>0.1%</td>
<td>0.1%</td>
<td>0.1%</td>
<td>1</td>
<td>2.0</td>
<td>4.0G</td>
</tr>
<tr>
<td></td>
<td>9.4%</td>
<td>3.8%</td>
<td>0.1%</td>
<td>1.0%</td>
<td>8</td>
<td>36.0</td>
<td>36.0G</td>
</tr>
<tr>
<td></td>
<td>3.2%</td>
<td>2.6%</td>
<td>0.8%</td>
<td>-0.6%</td>
<td>2</td>
<td>59.0</td>
<td>100.0G</td>
</tr>
</tbody>
</table>

Total swarm allocation

cpu  mem
0.27  0.35

Resources available

cpu  mem
2234  4985.2G
```

/home/ksung/sueff.py

```
/home/ksung/sueff.py

<table>
<thead>
<tr>
<th>user</th>
<th>cpu eff</th>
<th>mem eff</th>
<th>alloc cpu</th>
<th>alloc mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>69.76%</td>
<td>≤ 10%</td>
<td>228</td>
<td>945G</td>
</tr>
</tbody>
</table>
```
Usage history

**CPU (29%/50%)**

![CPU Usage History Graph](image1)

**Memory (11%/41%)**

![Memory Usage History Graph](image2)
Policy changes to expect in the near future

Motivation: increase swarm efficiency, use, fairness, and turnover

• Shorter defq time and more defq-only nodes

• Changes in fairness calculation
Commands you should use often

squeue -u <user>
sbatch <sbatch file>
srun time <executable>
sacct -j <JobID> -o Job,MaxRSS,TotalCPU,CPUTime,Elapsed

blame (/home/ksung/resources/bin/blame)
sueff (/home/ksung/resources/bin/sueff)
List of resources

• /home/ksung/resources/install --- install dmtcp, sueff, and blame

Examples:
• /home/ksung/resources/dmtcp_example
• /home/ksung/resources/process_example

https://slurm.schedmd.com/sbatch.html
Summary

• **DO NOT** run anything on the head node --- always use `srun` or `sbatch` for anything computationally intensive

• Profile your program!
  • **DO NOT** overallocate **time**, **memory**, or **CPU**

• **CHECK** your own jobs when you run them

**BE RESPECTFUL!**

Install the tools:
```
$ /home/ksung/resources/install
```

Monitor the mailing list:
```plaintext
swarm-users@cs.umass.edu
```

Issues?
Email the mailing list or Keen: `ksung@cs.umass.edu`