LSU INFORMATION TECHNOLOGY SERVICES

Distributed Workload

Wei Feinstein, Feng Chen and Le Yan

HPC User Services

LSU HPC & LONI

sys-help@loni.org

Louisiana State University

November, 2016

XSEDE Extreme Science and Engineering Discovery Environment

Overview

- Dilemma of running serial jobs on modern clusters
- Three parallel tools covered today
  - WQ

    Presenter: Wei Feinstein
  - GNU-PARALLEL

    Presenter: Feng Chen
  - SWIFT

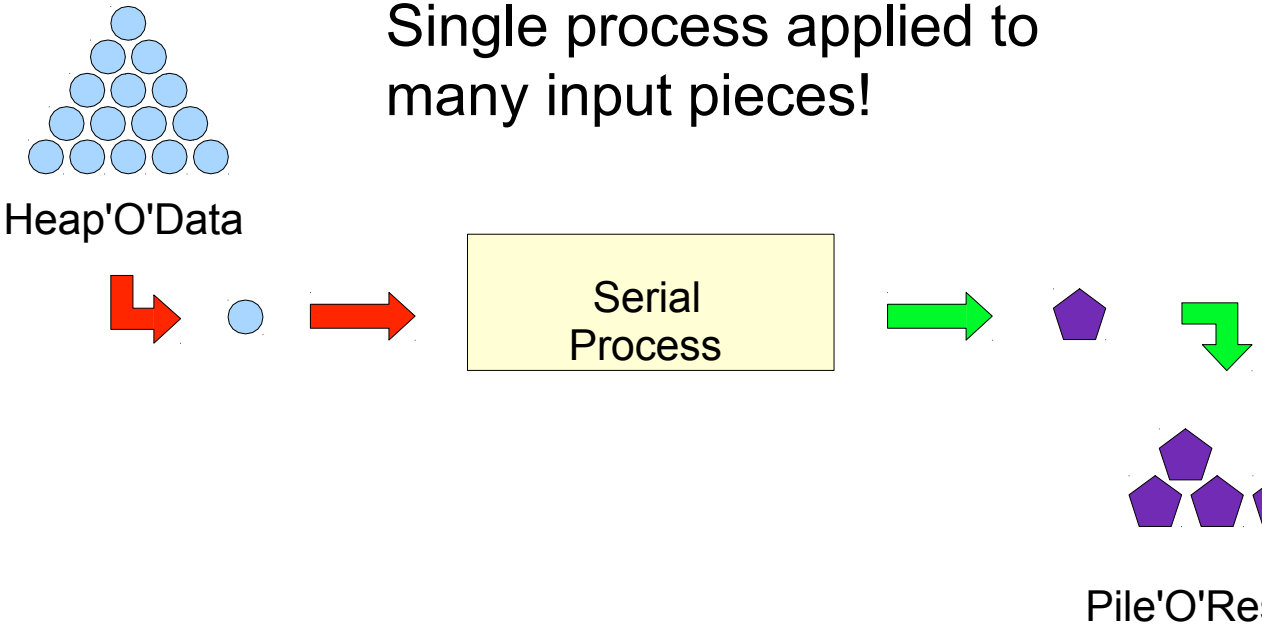
    Presenter: Le Yan

LSU HPC Environment

- Clusters tuned for large-scale parallel tasks.
- Full nodes assigned - access to 8, 16, 20, or even 48 cores per node, depending on system.

Q: How does one handle thousands of 1-core tasks without going crazy?

Problem Schematic



Single process applied to many input pieces!

Generates many output

Job submission examples used by some users

Running blastp with one core

blast-1core.qsub1

```
#! /bin/bash
#PBS -A hpc_hpcadmin3
#PBS -l nodes=1:ppn=1
#PBS -l walltime=00:20:00
#PBS –q single
#PBS –N blastp-worst

export DIR=/project/$USER/distribution_workload/blast
cd $DIR; mkdir output
blastp –query data/input1.faa –db db/xxx –out output/input1.out
```

blast-1core.qsub2

```
#! /bin/bash
#PBS -A hpc_hpcadmin3
#PBS -l nodes=1:ppn=1
#PBS -l walltime=00:20:00
#PBS –q single
#PBS –N blastp-worst

export DIR=/project/$USER/distribution_workload/blast
cd $DIR; mkdir output
blastp –query data/input2.faa –db db/xxx –out output/input2.out
```

Need to submit N jobs given N inputs

Running blastp with one node

blast-1node.qsub

```
#! /bin/bash
#PBS -A hpc_hpcadmin3
#PBS -l nodes=1:ppn=16
#PBS -l walltime=00:20:00
#PBS –q workq
#PBS –N blastp-better

export DIR=/project/$USER/distribution_workload/blast
cd $DIR; mkdir output
blastp –query data/input1.faa –db db/xxx –out output/input1.out  &
blastp –query data/input2.faa –db db/xxx –out output/input2.out  &
…
blastp –query data/input16.faa –db db/xxx –out output/input16.out  &

wait  #all the child processes to finish before terminating the parent process
```

- Only single node can be used
- Same command syntax with different inputs

Multi-Node Considerations

- The ***mother superior*** node is only one given all the job information, like environment variables, list of node names, etc.
- Start programs on other nodes with remote shell commands, like **ssh**.
- Assure all programs finish before script exits.

Running blastp with two nodes

blast-2nodes.qsub

```
#! /bin/bash
#PBS -A hpc_hpcadmin3
#PBS -l nodes=2:ppn=16
#PBS -l walltime=00:20:00
#PBS –q workq
#PBS –N blastp-2nodes
export DIR=/project/$USER/distribution_workload/blast
cd $DIR; mkdir output

# on mother superior (compute node1)
blastp –query data/input1.faa –db db/xxx –out output/input1.out &
…
blastp –query data/input1.faa –db db/xxx –out output/input16.out &
# on compute node2
ssh -n $HOST2 “cd $DIR; blastp –query data/input1.faa –db db/xxx –out
output/input1.out” &
…
ssh -n $HOST2 “cd $DIR; blastp –query data/input1.faa –db db/xxx –out
output/input16.out” &
wait  #all the child processes to finish before terminating the
parent process
```

What about with 10 nodes ??

Desired Solutions

- Avoid detailed scripting requirements - but allow flexibility and adaptability.
- Minimize customization and do most things automagically.
- Make method batch environment aware, particularly wallclock time constraints.

Tools

Three parallel tools covered today

- WQ
- GNU-PARALLEL
- SWIFT

Three examples used in the tutorial

- Serial job (blastp using signle core)

```
blastp –query xxx.fna –db db/xxx -num_threads 1
```

- Multiple-threaded (blastp using multiple cores)

```
blastp –query xxx.fna –db db/xxx -num_threads 4
```

- MPI jobs (laplace)

```
mpirun –np 4 lap_mpi 4096 4096 2 2 0.002 2000 0 0
```

Outline of WQ Parallel

- Overview of WQ
- How to use WQ
  - Serial jobs
  - Multi-threaded jobs
  - MPI jobs

What Is WQ? (Working Queue)

Dispatcher/Worker model written in python

- Handles ***tasks*** – defined as the work necessary to process one input file.
- Multiple ***workers*** execute the tasks - one worker per simultaneous task on all nodes.
- ***Workers*** request a ***task*** from the ***Dispatcher***. ***Workers*** share task times with ***Dispatcher***.
- ***Dispatcher*** won't assign a new task if it estimates that insufficient time remains to complete it.

WQ Components

Files (server side):

- **wq.py** – A Python script that implements the dispatcher and workers
- **wq-pbs.sh** – A script manages job environment on all nodes.

Files (client side):

- **wq.pbs** – A PBS batch script template with a few user required variable settings.
- **wq.sh** – A user created script (could be a program) that accepts one input file name as it's only argument.
- **wq.list** – A user created file containing input file names, one per line (suggest using absolute path names).

WQ installed on HPC/LONI system

- Clusters using softenv

  ```
  +wq-272
  +Python-3.5.1-anaconda-3.4.0
  ```

  add the keys to ~/.soft

- Clusters using module

  ```
  module load wq/272
  ```

To do list using WQ

- Copy wq.pbs (job submission) and wq.sh (user application script) to your working directory

```
cp $WQ_PBS_TEMPLATE workdir/wq.pbs
cp $WQ_EXAMPLE_TASK workdir/wq.sh
```

- Add executable permission: chmod +x wq.pbs wq.sh

- Tailor wq.pbs / wq.sh to your own applications
  - wq.pbs → wq-blastp.pbs
  - wq.sh → wq-blastp.sh

- Generate input.lst/wq.lst, where each line is used as the input to wq.sh.

Serial Job (WQ/serial/wq-blastp.sh)

Serial Job (WQ/serial/wq-blastp.pbs)

```
#! /bin/bash
########Begin WQ prologue section. ########
#PBS -A hpc_hpcadmin3
#PBS -l nodes=2:ppn=16
#PBS -l walltime=00:20:00
#PBS –q workq
#PBS -N WQ_BLAST_Serial

# Set up for 16 workers/node. 1 core/worker on a 16-core node (ppn/WPN)
WPN=16
# Set the working directory. Could use $PBS_O_WORKDIR
export DATADIR=/project/$USER/distribution_workload/blast
export WORKDIR=/project/$USER/distribution_workload/WQ/serial
# Name of the file containing the list of input files:
FILES=${DATADIR}/input.lst
# Start with task 1 (first line):
START=1
# Name of the task script:
TASK=${WORKDIR}/wq-blastp.sh
# Turn verbosity off(0)/one(1):
VERBOSE=0

######### Begin WQ epilogue section.#########
wq-pbs.sh $0 $WPN $WORKDIR $FILES $START $TASK $VERBOSE $1
```

wq-pbs.pbs : Epilogue Section

```
#### Begin WQ epilogue section. ####

wq-pbs.sh $0 $WPN $WORKDIR $FILES $START $TASK $VERBOSE $1
```

- Serious magic happens in the Epilogue
- Does some sanity checking of settings. Determines if running as mother superior.
- Mother superior starts dispatcher, and it's workers.
- Compute nodes start their workers.
- All workers start the ***request - execute*** cycle until walltime runs out or there are no more tasks to assign.

Serial Job (WQ/serial/wq-blastp.sh)

```
#! /bin/bash
# BLASTP uses OpenMP, so we'll set up # thread/core allowed for workers.
# WQ_CPT=ppn/WPN(cores/task)
export OMP_NUM_THREADS=${WQ_CPT}

mkdir -p $WORKDIR/output
# Treat the first command line argument as the input file pathname.
FILE=$1
BASE=`basename ${FILE}`

# Build the rather complex command line, including numactl string.
CMD="${WQ_NUMACTL} blastp -num_threads ${OMP_NUM_THREADS} "
CMD="${CMD} -db ${DATADIR}/db/img_v400_PROT.00"
CMD="${CMD} -query ${FILE}"
CMD="${CMD} -out ${WORKDIR}/output/${BASE}.out"
CMD="${CMD} -outfmt 7 -max_target_seqs 1"

# For testing purposes, use "if false". For real runs, use "if true":
if true ; then
   eval "${CMD}"
else
   echo "Would have executed the following command:"
   echo "${CMD}"
fi
```

wq-blastp.sh and input.list

wq-blastp.sh represents an actual shell script, program. If it works manually, it should function correctly when called by a worker.

```
$./wq-blastp.sh test1.faa
```

input.list: contains input file names, one per line.

```
$cat input.lst
/project/$USER/distribution_workload/blast/data/test1.faa
/project/$USER/distribution_workload/blast/data/test2.faa
/project/$USER/distribution_workload/blast/data/test3.faa
…
```

For a really large number of input files, generate it with the find command:

```
$cd blast/data
$find `pwd` -name '*.faa'-print > input.lst
```

Multi-Threaded Job (WQ/multi_threads/wq-blastp.pbs)

Multi-threaded Job (WQ/multi_threads/wq-blastp.pbs)

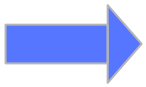
```
#! /bin/bash
########Begin WQ prologue section. ########
#PBS -A hpc_hpcadmin3
#PBS -l nodes=2:ppn=16
#PBS -l walltime=00:20:00
#PBS –q workq
#PBS -N WQ_BLAST_multiThreads

# Set up for 8 workers/node. 2 cores/worker on a 16-core node (ppn/WPN)
WPN=8
# Set the working directory. Could use $PBS_O_WORKDIR
export DATADIR=/project/$USER/distribution_workload/blast
export WORKDIR=/project/$USER/distribution_workload/WQ/multi_threads
# Name of the file containing the list of input files:
FILES=${DATADIR}/input.lst
# Start with task 1 (first line):
START=1
# Name of the task script:
TASK=${WORKDIR}/wq-blastp.sh
# Turn verbosity off(0)/one(1):
VERBOSE=0

######### Begin WQ epilogue section.#########
wq-pbs.sh $0 $WPN $WORKDIR $FILES $START $TASK $VERBOSE $1
```

Multi-threaded Job (WQ/multi_threads/wq-blastp.sh)

```
#! /bin/bash
# BLASTP uses OpenMP, so we'll set up # thread/core allowed for workers.
# WQ_CPT=ppn/WPN (cores/task)
export OMP_NUM_THREADS=${WQ_CPT}

mkdir -p $WORKDIR/output
# Treat the first command line argument as the input file pathname.
FILE=$1
BASE=`basename ${FILE}`

# Build the rather complex command line, including numactl string.
CMD="${WQ_NUMACTL} blastp -num_threads ${OMP_NUM_THREADS} "
CMD="${CMD} -db ${DATADIR}/db/img_v400_PROT.00"
CMD="${CMD} -query ${FILE}"
CMD="${CMD} -out ${WORKDIR}/output/${BASE}.out”
CMD="${CMD} -outfmt 7 -max_target_seqs 1"

# For testing purposes, use "if false". For real runs, use "if true":
if true ; then
   eval "${CMD}"
else
   echo "Would have executed the following command:"
   echo "${CMD}”
fi
```

MPI Job (WQ/mpi/wq-blastp.pbs)

MPI Job (WQ/mpi/wq-laplace.pbs)



```
#! /bin/bash
########Begin WQ prologue section. ########
#PBS -A hpc_hpcadmin3
#PBS -l nodes=2:ppn=16
#PBS -l walltime=00:20:00
#PBS –q workq
#PBS -N WQ_laplace_mpi

# Set up for 4 workers/node. 4 cores/worker on a 16-core node (ppn/WPN)
WPN=4
# Set the working directory. Could use $PBS_O_WORKDIR
export DATADIR=/project/$USER/distribution_workload/laplace
export WORKDIR=/project/$USER/distribution_workload/WQ/mpi
# Name of the file containing the list of input files:
FILES=${DATADIR}/input.lst
# Start with task 1 (first line):
START=1
# Name of the task script:
TASK=${WORKDIR}/wq-laplace.sh
# Turn verbosity off(0)/one(1):
VERBOSE=0

######### Begin WQ epilogue section.#########
wq-pbs.sh $0 $WPN $WORKDIR $FILES $START $TASK $VERBOSE $1
```

MPIJob (WQ/mpi/wq-laplace.sh)

```
#! /bin/bash
FILE=$1
BASE=`basename ${FILE}`
mkdir -p $WORKDIR/output
#Generate HOSTLIST
HOSTNAME=`uname -n`
HOSTLIST=""
for i in `seq 1 ${WQ_CPT}`; do
  HOSTLIST="${HOSTNAME},${HOSTLIST}"
done
HOSTLIST=${HOSTLIST%,*}

param=`cat ${FILE}`
CMD=" ${WQ_NUMACTL} mpirun"
CMD="${CMD} -host ${HOSTLIST}  -np ${WQ_CPT}  ${DATADIR}/lap_mpi  ${param}"
CMD=" ${CMD} > $WORKDIR/output/${BASE}.out "

# For testing purposes, use "if false", and each call will generate
if true; then
  eval "${CMD}"
else
  echo "${CMD}"
fi
```

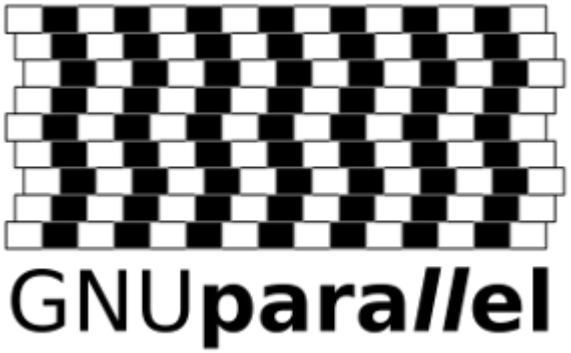
Distributed Workload using GNU Parallel

Feng Chen
HPC User Services
LSU HPC & LONI
sys-help@loni.org

Louisiana State University
Baton Rouge
November 02, 2016

GNU Parallel

- **GNU parallel is a shell tool for executing jobs in parallel using one or more computers (*compute nodes*).**
- **A job can be a *single command or a small script* that has to be run for each of the lines in the input.**
- **The typical input is a *list of files*, a list of hosts, a list of users, a list of URLs, or a list of tables.**
- **See more at: https://www.gnu.org/software/parallel/**



Outline

- **GNU Parallel Syntax**
- **Introducing Running 3 Types of jobs:**
  - Serial Jobs
    - ✓ Run each blast job in serial
  - Multi-Threaded Jobs
    - ✓ Run each blast job using 2 threads
  - Small MPI Jobs
    - ✓ Run each MPI Laplacian solver using 4 processes

Adding GNU Parallel to Environment

- **On SuperMike2:**

```
[fchen14@mike421 ~]$ softenv -k gnuparallel
SoftEnv version 1.6.2
...
    +gnuparallel-20161022-gcc-4.4.6
                                    @types: Application/Tools @name: gnuparallel
                                      @version: 20161022 @build: gnuparallel-
                                      20161022-gcc-4.4.6 @internal: @external:
                                      https://www.gnu.org/software/parallel/
                                      @about: GNU parallel is a shell tool for
                                      executing jobs in parallel using one or
                                      more computers.
[fchen14@mike421 ~]$ soft add +gnuparallel-20161022-gcc-4.4.6
[fchen14@mike421 ~]$ parallel --version
GNU parallel 20161022
...
```

- **Or add** `+gnuparallel-20161022-gcc-4.4.6` to `~/.soft` then "`resoft`"

Distributed Workload using GNU Parallel

GNU Parallel Syntax

GNU Parallel Syntax

- **Reading commands to be run in parallel from an input file:**

```
parallel [OPTIONS] < CMDFILE
```

- **Reading command arguments on the command line:**

```
parallel [OPTIONS] COMMAND [ARGUMENTS] ::: ARGLIST
```

- **Reading command arguments from an input file:**

```
parallel [OPTIONS] COMMAND [ARGUMENTS] :::: ARGFILE
```

ARGLIST from command line

- **parallel [OPTIONS] COMMAND [ARGUMENTS] ::: ARGLIST**
- **Examples:**

```
[fchen14@mike421 ~]$ parallel echo ::: A B C
A
B
C
[fchen14@mike421 ~]$ parallel echo ::: `seq 1 3`
1
2
3
[fchen14@mike421 ~]$ parallel echo ::: {A..Z}
A
B
...
Z
[fchen14@mike421 test]$ ls -1 | parallel echo
2013-06-18.tgz
backups.sh
bigmem_test.pbs
...
```

ARGLIST from file

- **parallel [OPTIONS] COMMAND [ARGUMENTS] ::: ARGFILE**

```
[fchen14@mike421 blast]$ cd /project/fchen14/gpar/blast
[fchen14@mike421 blast]$ cat input.lst | head
data/test10.faa
data/test11.faa
...
[fchen14@mike421 blast]$ head input.lst -n 5 | parallel echo
data/test10.faa
data/test11.faa
data/test12.faa
data/test13.faa
data/test14.faa
[fchen14@mike421 blast]$ parallel echo :::: input.lst
data/test10.faa
data/test11.faa
data/test12.faa
...
```

Replacement Strings

- '{}' returns a full line read from the input source.

```
[fchen14@mike421 blast]$ parallel echo {} ::: data/test1.faa
data/test1.faa
```

- '{/}' removes everything up to and including the last forward slash:

```
[fchen14@mike421 blast]$ parallel echo {/} ::: data/test1.faa
test1.faa
```

- '{//}' returns the directory name of input line.

```
[fchen14@mike421 blast]$ parallel echo {//} ::: data/test1.faa
data
```

- '{.}' removes any filename extension:

```
[fchen14@mike421 blast]$ parallel echo {.} ::: data/test1.faa
data/test1
```

- '{/.}' returns the basename of the input line without extension. It is a combination of {/} and {.}:

```
[fchen14@mike421 blast]$ parallel echo {/.} ::: data/test1.faa
test1
```

- See "man parallel" for more detailed explanation.

Replacement String Example

- **Print the full path of the input file, and then print the desired output file name, e.g.:**
  - Input file: `data/test1.faa`
  - Output file name: `output/test1.out`

```
# Process data/test1.faa and send result to output/test1.out
$ parallel echo {} output/{/.}.out ::: data/test1.faa
data/test1.faa output/test1.out
```

Parallelize Job Script

- **GNU parallel is often called as this:**

```
cat input_file | parallel command
parallel command ::: foo bar
```

- **If command is a script, parallel can be combined into a single file so this will run the script in parallel:**

```
parallel [OPTIONS] script [ARGUMENTS] ::: ARGLIST
```

  - or

```
parallel [OPTIONS] script [ARGUMENTS] :::: ARGFILE
```

- **See next slide for example...**

Parallize Script Example

- **This is the script we want to parallize "cmd_ex.sh":**

```
#!/bin/bash
# print the input, on which host, which working directory
echo "This script uses input: $1 on $HOSTNAME:$PWD"
```

- **Parallize the script using ARGLIST from command line:**

```
[fchen14@mike421 misc]$ parallel --wd /project/fchen14/gpar/misc ./cmd_ex.sh ::: A B C
This script uses input: A on mike421:/project/fchen14/gpar/misc
This script uses input: B on mike421:/project/fchen14/gpar/misc
This script uses input: C on mike421:/project/fchen14/gpar/misc
```

- **Parallize the script using ARGFILE:**

```
[fchen14@mike421 misc]$ cat argfile
A
B
C
[fchen14@mike421 misc]$ parallel --wd /project/fchen14/gpar/misc ./cmd_ex.sh :::: argfile
This script uses input: A on mike421:/project/fchen14/gpar/misc
This script uses input: B on mike421:/project/fchen14/gpar/misc
This script uses input: C on mike421:/project/fchen14/gpar/misc
```

- **Can parallize Python/Perl scripts, see "man parallel" for details**

Common OPTIONS --jobs (-j)

- **--jobs N (-j N)**
  - Number of jobslots on each machine (node). Run up to N jobs in parallel. 0 means as many as possible. Default is 100% which will run one job per CPU core on each machine.
  - On HPC/LONI clusters, **N** is number of jobslots per node.
  - Make sure you use GNU Parallel version **>=20161022** to avoid a "Max jobs to run" bug

```
[fchen14@mike421 test]$ parallel --version
GNU parallel 20161022
...
```

- **-j +N**
  - Add N to the number of CPU cores. Run this many jobs in parallel.
- **-j -N**
  - Subtract N from the number of CPU cores. Run this many jobs in parallel. If the evaluated number is less than 1 then 1 will be used.

Common OPTIONS `--slf`

- **`--slf filename (--sshloginfile filename)`**
  - File with sshlogins. The file consists of sshlogins on separate lines. Empty lines and lines starting with '#' are ignored.
  - Look at "`man parallel`" for more detailed explanation.
  - A typical example on HPC/LONI clusters while running batch jobs:

    `--slf $PBS_NODEFILE`
  - Recall what is inside `$PBS_NODEFILE`?

```
[fchen14@mike421 laplace]$ cat $PBS_NODEFILE
mike421
mike421
...
mike421
mike429
mike429
...
mike429
```

16 repeats (cores) on mike421

16 repeats (cores) on mike429

Common OPTIONS `--wd`

- **`--wd mydir (--workdir mydir)`**
  - Designate the working directory of your commands.
  - A typical value can be `$PBS_O_WORDIR`

Common OPTIONS `--progress`

- **`--progress`**
  - Show progress of computations.
  - List the computers involved in the task with number of CPU cores detected and the max number of jobs to run.
  - After that show progress for each node: number of running jobs, number of completed jobs, and percentage of all jobs done by this computer.
  - Example:

```
[fchen14@mike421 ~]$ parallel --progress echo ::: A B C

Computers / CPU cores / Max jobs to run
1:local / 16 / 3

Computer:jobs running/jobs completed/%of started jobs/Average seconds to complete
local:3/0/100%/0.0s A
local:2/1/100%/1.0s B
local:1/2/100%/0.5s C
local:0/3/100%/0.3s
```

- **See also `--bar`**

Common OPTIONS --joblog

- **--joblog logfile**
  - Creates a record for each completed subjob to be written to LOGFILE, with info on how long they took, their exit status, etc.
  - Can be used to identify failed jobs, e.g.:

```
[fchen14@mike421 misc]$ parallel --joblog logfile exit ::: 1 2 0 0
[fchen14@mike421 misc]$ cat joblog
cat: joblog: No such file or directory
[fchen14@mike421 misc]$ cat logfile
Seq     Host    Starttime       JobRuntime      Send    Receive Exitval Signal  Command
1       :       1477514132.358      0.019       0       0       1       0       exit 1
2       :       1477514132.375      0.003       0       0       2       0       exit 2
3       :       1477514132.376      0.002       0       0       0       0       exit 0
4       :       1477514132.377      0.003       0       0       0       0       exit 0
```

Common OPTIONS `--timeout`

- **`--timeout secs`**
  - Time out for command. If the command runs for longer than secs seconds it will get killed.
  - If secs is followed by a % then the timeout will dynamically be computed as a percentage of the median average runtime. Only values > 100% will make sense.

  - Useful if you know the command has failed if it runs longer than a threshold.

Distributed Workload using GNU Parallel

Serial Jobs Example

Distribute Serial Jobs - Run blast

```
#!/bin/bash
#PBS -l nodes=2:ppn=16
#PBS -l walltime=1:00:00
#PBS -A hpc_hpcadmin3
#PBS -q workq
#PBS -N gpl_ser_blast
#PBS -o gpl_ser_blast.out
#PBS -e gpl_ser_blast.err
JOBS_PER_NODE=16
export WDIR=/project/$USER/distribution_workload/blast
cd $WDIR
# create a folder output
mkdir -p output
# the parallel command
parallel --progress \              # shows the progress
         --joblog logfile \        # specify a job logfile
         -j $JOBS_PER_NODE \       # specify the jobs per node
         --slf $PBS_NODEFILE \     # distribute workload among allocated nodes
         --workdir $WDIR \         # specify the working directory of the script
         $PBS_O_WORKDIR/cmd_ser_blast.sh {} {/.} :::: input.lst
         # script_to_parallize input output :::: ARGFILE
```

```
[fchen14@mike421 blast]$ head input.lst
data/test10.faa
data/test11.faa
data/test12.faa
data/test13.faa
data/test14.faa
data/test15.faa
...
```

The script to Run blast

```
#!/bin/bash
FILE=$(eval echo $1) # process the input $1 when there is bash variable
echo "using input: $FILE"
TIC=`date +%s.%N`
blastp -query $FILE -db db/img_v400_PROT.00 -out output/$2.out \
       -outfmt 7 -max_target_seqs 100 -num_threads 1
TOC=`date +%s.%N`
J1_TIME=`echo "$TOC - $TIC" | bc -l`
echo "This serrun took=$J1_TIME sec using $FILE on $HOSTNAME"
```

$1 from {}

$2 from {/.}

Distributed Workload using GNU Parallel

Multi-Threaded Example

Distribute Multi-Threaded Jobs - blast

- **Distribute Multi-Threaded jobs is very similar to the pure serial job example, the only difference is** `JOBS_PER_NODE`**…**
  - `JOBS_PER_NODE=CPU_CORES_PER_NODE / NUM_THREADS_PER_JOB`
- **If each job uses 2 threads, each node on SuperMike2 has 16 cores, then**
  - `JOBS_PER_NODE=16/2=8`
- **PBS script (`#PBS comments` omitted):**

```
JOBS_PER_NODE=8
export WDIR=/project/$USER/distribution_workload/blast
cd $WDIR
mkdir -p output
NTHREADS=2
parallel --progress \
         -j $JOBS_PER_NODE \
         --slf $PBS_NODEFILE \
         --workdir $WDIR \
         $PBS_O_WORKDIR/cmd_mt_blast.sh {} {/.} $NTHREADS \
         :::: input.lst
```

changes needed compared to serial version

The script to Run Multi-Threaded blast

```
#!/bin/bash
# This is the script we want to parallize and distribute
echo "using input: $1, output $2.out"
FILE=$(eval echo $1)
echo "using input: $FILE"
TIC=`date +%s.%N`
blastp -query $FILE \
       -db db/img_v400_PROT.00 \
       -out output/$2.out  \
       -outfmt 7 -max_target_seqs 100 \
       -num_threads $3
TOC=`date +%s.%N`
J1_TIME=`echo "$TOC - $TIC" | bc -l`
echo "This serrun took=$J1_TIME sec using $FILE with $3 threads on $HOSTNAME"
```

$1 from {}

$2 from {/.}

$3 from $NTHREADS, the only change compared to serial

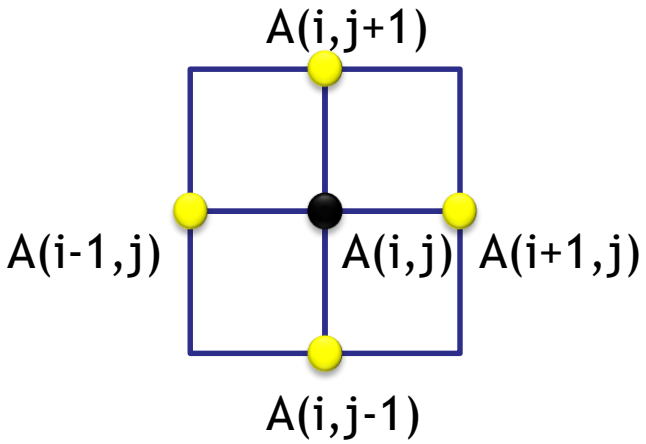
Distributed Workload using GNU Parallel

Multi-Process (MPI) Example

Distribute MPI Jobs - Laplace Solver

- **This section describes how to distribute small MPI jobs.**
- **Example problem - Laplacian Solver**
  - Solves a 2D Laplacian equation on a 2D grid
  - Use 4096x4096 2D grid, run 2000 iterations

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$



Graphical representation for Jacobi iteration

Array: told

	1.0	1.0	1.0	1.0	1.0	1.0	
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	

$t_{1,1} = 0.25 \times (1.0 + 1.0 + 0.0 + 0.0) = 0.5$

$dt = \max(t_{i,j} - told_{i,j}) = 0.5 - 0 = 0.5$

Array: t (tnew)

	1.0	1.0	1.0	1.0	1.0	1.0	
1.0	0.5	0.375	0.344	0.336	0.334	0.333	0.0
1.0	0.375	0.188	0.133	0.117	0.113	0.112	0.0
1.0	0.344	0.133	0.066	0.046	0.04	0.038	0.0
1.0	0.336	0.117	0.046	0.023	0.016	0.013	0.0
	0.0	0.0	0.0	0.0	0.0	0.0	

PBS Script for Distributing MPI Jobs

```
#!/bin/bash
#PBS -l nodes=2:ppn=16
#PBS -l walltime=1:00:00
#PBS -A hpc_hpcadmin3
#PBS -q workq
#PBS -N gpl_mpi
```

```
[fchen14@mike421 laplace]$ cat input.lst
/project/$USER/distribution_workload/laplace/input/input1
/project/$USER/distribution_workload/laplace/input/input2
/project/$USER/distribution_workload/laplace/input/input3
...
[fchen14@mike421 laplace]$ cat \
/project/$USER/distribution_workload/laplace/input/input1
4096 4096 2 2 0.08 20000 0 0
```

```
JOBS_PER_NODE=4 # uses 4 jobs per node, on SuperMike2: 16/4=4
echo "JOBS_PER_NODE="$JOBS_PER_NODE
NPROCS=4        # uses 4 mpi processes per MPI job
WDIR=/project/$USER/distribution_workload/laplace
cd $WDIR
parallel --progress \
         -j $JOBS_PER_NODE \
         --slf $PBS_NODEFILE \
         --workdir $WDIR \
         $PBS_O_WORKDIR/cmd_mpi.sh {} $NPROCS :::: input.lst
         # script_name input num_mpi_process
```

The Script to Run MPI Laplacian Solver

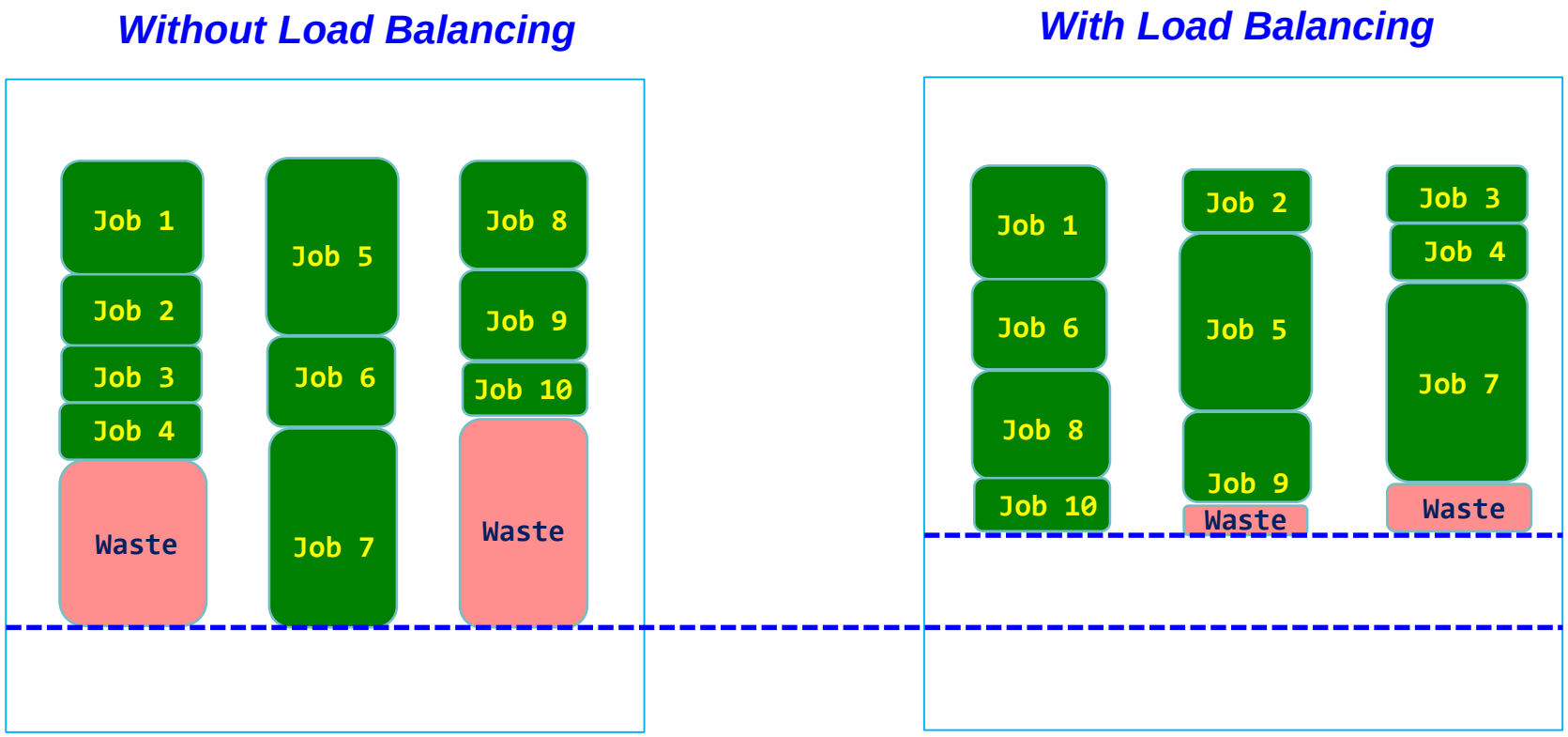
```
#!/bin/bash
echo "using input: $1"
TIC=`date +%s.%N`
## get an input from $1 (input.lst)
FILE=$(eval echo $1)
param=`cat ${FILE}`
echo param=$param
# get number of mpi process from $2 ($NPROCS)
mpirun -np $2 ./lap_mpi $param
TOC=`date +%s.%N`
J1_TIME=`echo "$TOC - $TIC" | bc -l`
echo "This mpirun took=$J1_TIME sec on $HOSTNAME"
```

Distributed Workload with GNU Parallel

Load Balancing

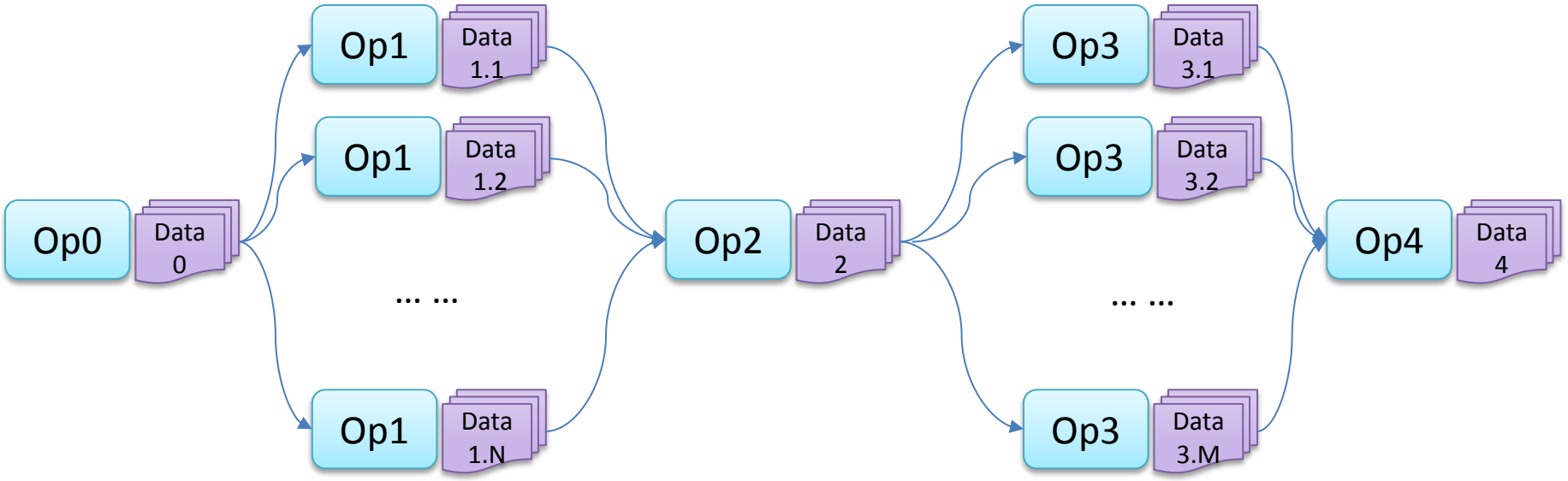
Load Balancing in GNU Parallel

- **GNU Parallel spawns the next job when one finishes - keeping the CPUs active and thus saving time.**



Parallel Workload Processing with Swift

Describing Workflows



- To describe a workflow, one needs to
  - Define data objects
  - Define applications (operations)
  - Define procedures

What Is Swift?

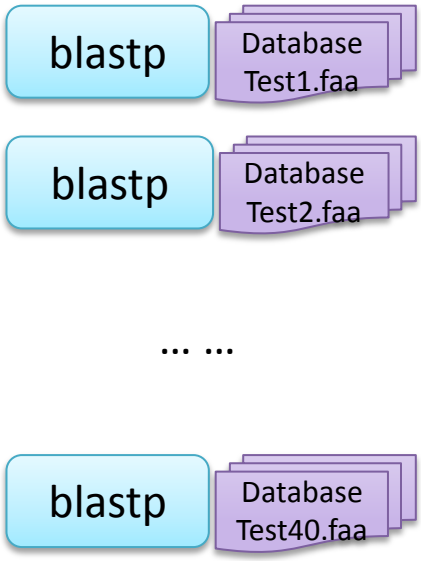
- Data-flow oriented parallel scripting language
- Swift allows users to
  - Define their workflows in a Swift script
  - Then execute it
- Swift supports
  - Dataset typing and mapping
  - Dataset iteration
  - Conditional branching
  - Procedural composition

Running Swift on Super Mike 2

- Add the softenv key "`+swift-0.96.2`" to your .soft file
- Start an interactive job session
- Run the "`swift_setup.sh`" script (important!)
- Run the Swift script

Example: Multithreaded BLAST

- 40 blastp tasks
  - Each requires two files



```
$ cat blast.swift

type file;

app (file o) blast (file faa, string db, int outfmt, int max_seqs, int nthreads)
{
  blastp "-query" filename(faa) "-db" db "-outfmt" outfmt "-max_target_seqs" max_seqs "-
num_threads" nthreads stdout=filename(o);
}

int nsim     = toInt(arg("nsim","1"));
int outfmt   = toInt(arg("outfmt","7"));
int max_seqs  = toInt(arg("max_seqs","100"));

file sims[];

foreach i in [1:nsim] {
  file simout <single_file_mapper; file=strcat("output/sim_",i,".out")>;
  file faafile <single_file_mapper; file=strcat("blast/data/test",i,".faa")>;
  string dbfile = "db/img_v400_PROT.00";
  simout = blast(faafile, dbfile, outfmt, max_seqs, 1);
  sims[i] = simout;
}
```

```
$ swift blast_multi_thread.swift -nsim=40
Swift 0.96.2 git-rev: b9611649002eecd640fc6c58bbb88cb35ce03539 heads/release-0.96-swift 6287
RunID: run004
Progress: Mon, 31 Oct 2016 15:28:19-0500
Progress: Mon, 31 Oct 2016 15:28:20-0500  Selecting site:32  Submitting:7  Active:1
Progress: Mon, 31 Oct 2016 15:28:25-0500  Selecting site:32  Submitting:5  Submitted:1
Active:2
…
Progress: Mon, 31 Oct 2016 15:28:30-0500  Selecting site:32  Submitted:1  Active:7
Progress: Mon, 31 Oct 2016 15:28:31-0500  Selecting site:32  Active:8
Progress: Mon, 31 Oct 2016 15:28:35-0500  Selecting site:31  Active:8  Finished
successfully:1
…
Progress: Mon, 31 Oct 2016 15:30:14-0500  Active:2  Finished successfully:38
Progress: Mon, 31 Oct 2016 15:30:17-0500  Active:1  Stage out:1  Finished successfully:38
Progress: Mon, 31 Oct 2016 15:30:32-0500  Stage out:1  Finished successfully:39
Final status: Mon, 31 Oct 2016 15:30:50-0500  Finished successfully:40
```

Data Types

- Primitive types: int, float, string, boolean
- Composite types: similar to other languages

```
type dataset {
  int rows;
  string name;
}
dataset mydata;
```

- Marker type: a type without definition
  - Swift will treat variables of this type as individual opaque objects.

```
type imagefile;
Imagefile myimage;
```

Arrays

- Arrays can be declared using the [ ] suffix
- Swift arrays can be associative (with keys of primitive types other than integer)

```
type image;
image myimages[];
myimages[1] = "sim1.jpg";

type file;
file[string] objdata;
objdata["obj_a"] = "a.dat";
```

Mappers

- Swift mappers provide a mechanism to specify the layout of mapped datasets on disk.
  - `single_file_mapper`: map to one single file
  - `simple_mapper`: map files into an array by prefix, suffix and/or pattern
  - `filesys_mapper`: similar to `simple_mapper` (will look at files on the file system)

```
type image;
image myimages[] <simple_mapper;prefix="sim",suffix=".jpg", padding=2>;
# myimages[0] -> "sim00.jpg"
# myimages[1] -> "sim01.jpg"
# myimages[2] -> "sim02.jpg"
…
```

Procedures

- Two types of procedures
  - Compound procedure – Composed of Swift operations and procedures
  - Application procedure - how to execute an external program
- Swift script procedures can take multiple inputs and produce multiple outputs
  - Inputs are specified to the right of the function name, and outputs are specified to the left.
- Procedures can be invoked like functions in other languages

```
# Compound procedure
(int out1, int out2) myproc (int in1, int in2) {
  out1 = in1+in2; out2 = in1*in2;
}

# Application procedure with the "app" keyword
app (file out) count (file in) {
  wc -l @in stdout=@out
}

# Invocation
file f <"myfile.txt">; file r <"result.txt">;
r = count(f);
```

Loop Constructs

- `foreach`: executed in parallel
- `iterate`: executed in sequence (should not be used unless absolutely necessary)

```
# A simple foreach that will be executed in parallel
foreach i in (1:10) {
  results[i] = myapp(i);
}

# foreach can loop over both elements and indices of an array
file textfiles[] <filesys_mapper; suffix=".txt">;
file results[];

foreach file,i in textfiles {
  results[i] = count(file);
}
```

Swift Execution Model

- Implicitly parallel - a procedure or expression will be executed when all of its parameters have been assigned value
- In another word, all code executes in parallel unless there are variables shared between the code that cause sequencing

```
# A valid execution order is:
# A1 S(x) A2 S(y)

(file a, file b) A() {
  a = A1();
  b = A2();
}
file x, y, s, t;
(x,y) = A();
s = S(x);
t = S(y);
```

Putting Everything Together

```
$ cat blast.swift

type file;

app (file o) blast (file faa, string db, int outfmt, int max_seqs, int nthreads)
{
  blastp "-query" filename(faa) "-db" db "-outfmt" outfmt "-max_target_seqs" max_seqs "-
num_threads" nthreads stdout=filename(o);
}
```

Define the application "blast"

```
int nsim     = toInt(arg("nsim","1"));
int outfmt   = toInt(arg("outfmt","7"));
int max_seqs  = toInt(arg("max_seqs","100"));
```

Define command line arguments

```
file sims[];

foreach i in [1:nsim] {
  file simout <single_file_mapper; file=strcat("output/sim_",i,".out")>;
  file faafile <single_file_mapper; file=strcat("blast/data/test",i,".faa")>;
  string dbfile = "db/img_v400_PROT.00";
  simout = blast(faafile, dbfile, outfmt, max_seqs, 1);
  sims[i] = simout;
}
```

Run the tasks in parallel

Further reading

- Swift user guide (0.96)
  - http://swift-lang.org/guides/release-0.96/userguide/userguide.html
- Swift tutorial
  - http://swift-lang.org/swift-tutorial/doc/tutorial.html