Launching Jobs in Parallel on ARC Clusters

Matthew Brown/Ayat Mohammed/Chris Kuhlman/Sarah Ghazanfari/Sofia Lima
Advanced Research Computing
Information Technology
Virginia Tech

Summer 2024 ARC workshop Series

- **Introductory / Orientation Workshops (2 hours each):**

~~*- Advanced Research Computing (ARC) Overview*~~
~~*- Connect to ARC systems and run your first jobs*~~
~~*- Get your software/code to run on ARC*~~

- **Special Topics (2 hours each):**

-Launching in Parallel

-Monitoring Resource Utilization and Job Efficiency

SignIn:https://docs.google.com/document/d/1y4Ib1M9hOKZsTDxIGUiMdvejl7L0N-1Ddc2ePqDUKYQ/edit?usp=share_link

Expectations

- This is an informal workshop
- Mostly informational about Advanced research computing at VT
- I want to hear your questions
- Welcome to use chat to ask questions + some time at the end
- Feedback needed to help improve future workshops

Launching in Parallel on ARC Clusters

Description

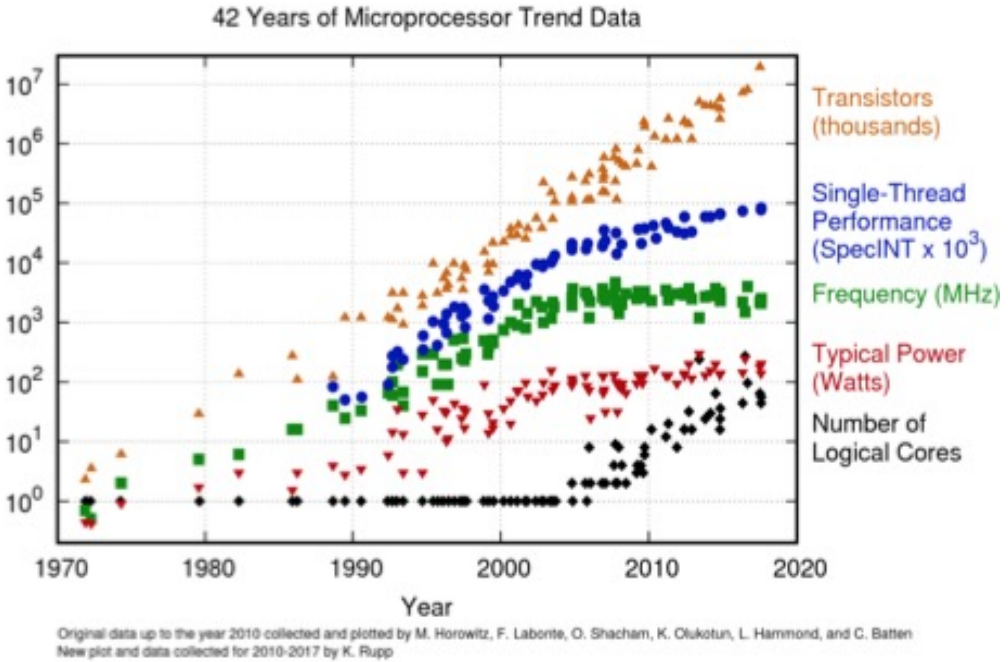
- Prepare you with the essential knowledge to harness the power of parallel computing on ARC clusters
- The details of parallel job execution
- Efficiently distribute computational workloads and maximize the utilization of ARC clusters
- Attendees who already have ARC accounts are invited to follow along with the demonstrations if desired

Outline:

- MPIRUN vs. SRUN
- SRUN for resource detection and binding
- GNU parallel for load balancing
- Monitoring performance and binding

Parallelism is the New Moore's Law

- Power and energy efficiency impose a key constraint on design of micro-architectures
- Clock speeds have plateaued
- Hardware parallelism is increasing rapidly to make up the difference



"Pleasingly Parallel"

- Computations are independent and can be executed simultaneously
- Examples: Parameter sweeps, numerical integration, BLAST searches
- Parallelization approaches and tools:
  - At BASH script level: GNU/parallel, srun
  - Matlab "parfor" to replace certain "for" loops
  - FORTRAN/C codes on data structure operations: OpenMP (threads) and/or MPI (ranks/tasks/processes)

MPIRUN vs. SRUN

- MPI
  - Is a standardized interface for inter-process communication
  - Several implementations (Intel MPI, OpenMPI, MPICH, MVAPICH, IBM)
  - The startup mechanism is linked to the MPI library
  - Startup commands may be called mpirun, mpiexec or something else
- srun
  - srun is the standard SLURM command to start an MPI program
  - It is well integrated with SLURM
  - It automatically uses the allocated job resources: node list, tasks, logical cores per task
  - It chooses an optimal CPU binding for the tasks on an allocated host

MPIRUN vs. SRUN

- HPL is a computing benchmark

1. Pure MPI (1 process per core)

- Jobs in this case should typically be requested with –ntasks-per-node=128 (if you want full node performance)
- Intel, using mpirun. We use an environment variable to make sure that MPI processes are laid out in order and not moved around by the operating system:

module reset; module load HPL/2.3-intel-2019b *#intel*
mpirun -genv I_MPI_PIN_PROCESSOR_LIST=0-127 xhpl

- gcc, using mpirun. We use OpenMPI's mapping and binding functionality to assign the processes to consecutive cores:

module reset; module load HPL/2.3-foss-2020a *#gcc*
mpirun --map-by core --bind-to core -x OMP_NUM_THREADS=1 xhpl

- Intel or gcc, using srun: **srun --cpu-bind=cores xhpl**

MPIRUN vs. SRUN

1. `git clone https://github.com/AdvancedResearchComputing/examples.git`
2. `cd examples/`
3. `cd hpl/`
4. `cd mpi/`
5. `sbatch -A personal hpl_mpi.sh`

MPIRUN vs. SRUN(Contd.)

2. Hybrid MPI+OpenMP (1 MPI process/L3 cache):
   - Start one MPI process per L3 cache (every 4 cores)
   - Jobs in this case should typically be requested with -ntasks-per-node=32 -cpus-per-task=4 so that Slurm knows how many processes.

- Intel: We use environment variables to tell mpirun to start a process on every fourth core and use 4 OpenMP (MKL) threads per process:

mpirun -genv I_MPI_PIN_PROCESSOR_LIST="$(seq -s , 0 4 127)" -genv I_MPI_PIN_DOMAIN=omp -genv OMP_NUM_THREADS=4 -genv OMP_PROC_BIND=TRUE -genv OMP_PLACES=cores xhpl

- gcc: We use OpenMPI's mapping and binding functionality to assign the processes to L3 caches: **mpirun --map-by ppr:1:L3cache --bind-to l3cache -x OMP_NUM_THREADS=4 xhpl**

MPIRUN vs. SRUN (Contd.)

- Intel or gcc, using srun: **srun --cpu-bind=cores xhpl**

- Intel or gcc, using Slurm's srun launcher. We use a cpu mask to tell Slurm which cores each process should have access to. (0xF is hexadecimal for 15, or 1111 in binary, meaning access should be allowed to the first four cores. 0xF0 is 11110000 in binary, meaning access should be allowed to the second set of four cores. The list continues through 11110000…..0000, indicating that the last process should have access to cores 124-127.)

Not recommended

srun –cpu-bind=mask_cpu=0xF,0xF0,0xF00,0xF000,0xF0000,0xF00000,0xF000000,0xF0000000,0xF00000000,0xF000000000,0xF0000000000,0xF00000000000,0xF000000000000,0xF0000000000000,0xF00000000000000,0xF000000000000000,0xF0000000000000000,0xF00000000000000000,0xF000000000000000000,0xF0000000000000000000,0xF00000000000000000000,0xF000000000000000000000,0xF0000000000000000000000,0xF00000000000000000000000,0xF000000000000000000000000,0xF0000000000000000000000000,0xF00000000000000000000000000,0xF000000000000000000000000000 xhpl

MPIRUN vs. SRUN (Contd.)

- The results show the benefit of the hybrid MPI+OpenMP model and of MKL over OpenBLAS, particularly in the hybrid model

```
intel |     mpi   | mpirun | 2,944 GFlops/s
intel |     mpi   |   srun | 2,809 GFlops/s
  gcc |     mpi   | mpirun | 2,734 GFlops/s
  gcc |     mpi   |   srun | 2,659 GFlops/s
intel | mpi+omp   | mpirun | 3,241 GFlops/s
intel | mpi+omp   |   srun | 3,227 GFlops/s
  gcc | mpi+omp   | mpirun | 2,836 GFlops/s
  gcc | mpi+omp   |   srun | 2,845 GFlops/s
```

MPIRUN vs. SRUN (Contd.)

- How can I run multiple short, parallel tasks inside one job?

➢ An example structure:

```
# Specify the list of tasks
tasklist=task1 task2 task3
# Loop through the tasks
for tsk in $tasklist; do
  # run the task $tsk
  mpirun -np $SLURM_NTASKS ./a.out $tsk &
done
```

MPIRUN vs. SRUN (Contd.)

https://github.com/AdvancedResearchComputing/examples/tree/master/mpi

- **Using "mpi_quad.c"**
- **module load foss/2020b**
- **To compile: mpicc mpi_quad.c –o tc_nq_f20b_mpiquad**
- **salloc --nodes=4 --ntasks-per-node=16 --time=0:02:00 --account=personal**
- **To run: mpirun ./tc_nq_f20b_mpiquad**

- Compilation of programs may need additional
  - paths to header files (-I/path/to/inc)
  - paths to libraries (-L/path/to/lib)
  - library names (-ldepend for /path/to/lib/libdepend.so)

- Intel, GCC, NVHPC, etc. all use different options which are not cross compatible. Use manual and --help to investigate. Edit makefiles to customize for ARC software/versions

GNU/parallel for load balancing

- Multifunctional utility with lots of features and usages
- Great for passing arguments to repeated commands
- Much better for load balancing than BASH for loops
- Manual has lots of examples (man parallel)

Example:
Pass sequence of parameters to parallel executed code:

ls -d mw* | parallel tar -czf {}.tar.gz {}

- parallel + srun when running on several machines provides complimentary features of srun

srun features:

- knows about hosts allocated to job and requested task layout
- can control cpu-binding

GNU/parallel for load balancing

- How can I run multiple serial tasks inside one job?

```
# Define variables
  numtasks=16
  np=1
  # Loop through numtasks tasks
  while [ $np -le $numtasks ]
  do
     # Run the task in the background with input and output depending on the variable np
    ./a.out $np > $np.out &

    # Increment task counter
    np=$((np+1))
  done

  # Wait for all of the tasks to finish
  wait
```

GNU/parallel for load balancing

- Rewriting a for-loop and a while-read-loop

```
(for color in red green blue ; do
  for size in S M L XL XXL ; do
    echo $color $size
  done
done) | sort
```

- can be written like this:

```
parallel echo {1} {2} ::: red green blue ::: S M L XL XXL | sort
```

GNU/parallel for load balancing

- The Slurm Workload Manager is used in many clusters.
  Here is a simple example of using GNU parallel to call srun:

1. File pGNU:

```
#!/bin/bash
#SBATCH --account=personal
#SBATCH --partition=normal_q
#SBATCH --time 00:02:00
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --job-name GnuParallelDemo
#SBATCH --output gnuparallel.out
#module purge
#module load gnu_parallel
my_parallel="parallel --delay .2 -j $SLURM_NTASKS"
my_srun="srun --export=all --exclusive -n1"
my_srun="$my_srun --cpus-per-task=1 --cpu-bind=cores"
$my_parallel "$my_srun" echo This is job {} ::: {1..20}
```

2. Submit the job as:

```
sbatch pGNU
```

3. Open gnuparallel.out:

```
vi gnuparallel.out
```

GNU/parallel for load balancing

- Parallelizing rsync
- rsync is a great tool, but sometimes it will not fill up the available bandwidth. Running multiple rsync in parallel can fix this

```
cd src-dir
find . -type f | parallel -j10 -X rsync -zR -Ha ./{} infer1.arc.vt.edu:./JLAB/
```

- Adjust -j10 until you find the optimal number
- rsync -R will create the needed subdirectories, so all files are not put into a single dir. The ./ is needed so the resulting command looks similar to:

```
rsync -zR ././sub/dir/file fooserver:/dest-dir/
```

- The /./ is what rsync -R works on.
- If you are unable to push data, but need to pull them and the files are called digits.png (e.g. 000000.png) you might be able to do:

```
seq -w 0 99 | parallel rsync -Havessh fooserver:src/*{}.png destdir/
```

"Built-in" or library-based parallelism

MATLAB examples:

1. Built-in Arithmetic Uses Available Cores

module load tinkercliffs-rome/matlab/R2021a
srun –A personal --nodes=1 --ntasks=1 --cpus-per-task=32 --pty matlab -nosplash -nosoftwareopengl -sd `pwd`
>> N=25000; A=rand(N); B=rand(N); tic; A*B; toc;

2. parpool spawns workers to which parfor can farm out tasks

>> parpool(32);
>> tic; n=200; A=2000; a=zeros(1,n); parfor i=1:n; a(i)=max(abs(eig(rand(A)))); end; toc;

3. Using GPUs: gpuArray: A = gpuArray([1 0 1; -1 -2 0; 0 1 -1]); e = eig(A);

SRUN for resource detection and binding

- Job allocation on a DGX node:

```
--ntasks-per-node=32 --gres=gpu:2
```

- srun uses a subset here:

```
srun --ntasks=8 python mpi_scatter.py

$cat mpi_cupy.py

from mpi4py import MPI
import cupy as cp

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

dev = cp.cuda.Device(rank)
print("rank:",rank,'bus_id:', dev.pci_bus_id)
print(dev.mem_info)
```

- We need custom gpu-binding to get the separate MPI ranks to "see" different GPU devices:

```
srun --ntasks=2 --gpu-bind=single:1 python mpi_cupy.py
```

Monitoring performance and binding

- **OMP_PLACES** is employed to specify places on the machine where the threads are put. However, this variable on its own does not determine thread pinning completely, because your system still won't know in what pattern to assign the threads to the given places. Therefore, you also need to set **OMP_PROC_BIND**
- **OMP_PROC_BIND** specifies a binding policy which basically sets criteria by which the threads are distributed
- If you want to get a schematic overview of your cluster's hardware, e. g. to figure out how many hardware threads there are, type: **$ lstopo**
- STREAM is a memory bandwidth benchmark. To maximize bandwidth, we run in parallel with one process per L3 cache (cores 0, 4, …, 124).

Monitoring performance and binding

```
#Load the Intel toolchain
  module reset; module load intel/2019b

  #Tell OpenMP to use every 4th core
  export OMP_PROC_BIND=true
  export OMP_NUM_THREADS=32
  export OMP_PLACES="$( seq -s },{ 0 4 127 | sed -e 's/\(.*\)/\{\1\}/' )"

  #Compile
  icc -o stream.intel stream.c -DSTATIC -DNTIMES=10 -
DSTREAM_ARRAY_SIZE=2500000000 \
    -mcmodel=large -shared-intel -Ofast -qopenmp -ffreestanding -qopt-streaming-stores
always

#Run
./stream.intel
```

Monitoring performance and binding

- How can I monitor GPU utilization during my job?
  - Add a line like this to a batch script prior to starting a gpu workload:

```
nvidia-smi
--query-gpu
=timestamp,name,pci.bus_id,driver_version,temperature.gpu,utilization.gpu,utilization.memory,mem
ory.total,memory.free,memy.used --format=csv -l 3 > $SLURM_JOBID.gpu.log &
```

The `&` causes the query to run in the background and keep running until the job ends or this process is manually killed. The `> $SLURM_JOBID.gpu.log` causes the output to be redirected to a file whose name is the numerical job id followed by .gpu.log.

- The -l 3 is for the repeating polling interval. From the nvidia-smi manual:

`-l SEC, --loop=SEC`

- Continuously report query data at the specified interval, rather than the default of just once.

- For details on query options: `nvidia-smi --help-query-gpu`

Monitoring performance and binding

- Output from `nvidia-smi` run as above looks like this for a 2-gpu job (notice the different gpu identifier strings):

```
2021/10/29 16:36:30.047, A100-SXM-80GB, 00000000:CB:00.0, 460.73.01, 41, 0 %, 0 %, 81251 MiB, 81248 MiB, 3 MiB
2021/10/29 16:36:33.048, A100-SXM-80GB, 00000000:07:00.0, 460.73.01, 58, 16 %, 4 %, 81251 MiB, 66511 MiB, 14740 MiB
2021/10/29 16:36:33.053, A100-SXM-80GB, 00000000:CB:00.0, 460.73.01, 41, 0 %, 0 %, 81251 MiB, 81248 MiB, 3 MiB
2021/10/29 16:36:36.054, A100-SXM-80GB, 00000000:07:00.0, 460.73.01, 65, 98 %, 15 %, 81251 MiB, 66571 MiB, 14680 MiB
2021/10/29 16:36:36.055, A100-SXM-80GB, 00000000:CB:00.0, 460.73.01, 41, 0 %, 0 %, 81251 MiB, 81248 MiB, 3 MiB
2021/10/29 16:36:39.055, A100-SXM-80GB, 00000000:07:00.0, 460.73.01, 67, 100 %, 36 %, 81251 MiB, 66571 MiB, 14680 MiB
2021/10/29 16:36:39.056, A100-SXM-80GB, 00000000:CB:00.0, 460.73.01, 41, 0 %, 0 %, 81251 MiB, 81248 MiB, 3 MiB
2021/10/29 16:36:42.057, A100-SXM-80GB, 00000000:07:00.0, 460.73.01, 54, 10 %, 2 %, 81251 MiB, 66571 MiB, 14680 MiB
2021/10/29 16:36:42.058, A100-SXM-80GB, 00000000:CB:00.0, 460.73.01, 41, 0 %, 0 %, 81251 MiB, 81248 MiB, 3 MiB
2021/10/29 16:36:45.059, A100-SXM-80GB, 00000000:07:00.0, 460.73.01, 54, 0 %, 0 %, 81251 MiB, 66571 MiB, 14680 MiB
2021/10/29 16:36:45.060, A100-SXM-80GB, 00000000:CB:00.0, 460.73.01, 41, 0 %, 0 %, 81251 MiB, 81248 MiB, 3 MiB
2021/10/29 16:36:48.060, A100-SXM-80GB, 00000000:07:00.0, 460.73.01, 68, 100 %, 26 %, 81251 MiB, 66571 MiB, 14680 MiB
2021/10/29 16:36:48.061, A100-SXM-80GB, 00000000:CB:00.0, 460.73.01, 41, 0 %, 0 %, 81251 MiB, 81248 MiB, 3 MiB
2021/10/29 16:36:51.062, A100-SXM-80GB, 00000000:07:00.0, 460.73.01, 52, 20 %, 3 %, 81251 MiB, 66571 MiB, 14680 MiB
2021/10/29 16:36:51.063, A100-SXM-80GB, 00000000:CB:00.0, 460.73.01, 41, 0 %, 0 %, 81251 MiB, 81248 MiB, 3 MiB
2021/10/29 16:36:54.064, A100-SXM-80GB, 00000000:07:00.0, 460.73.01, 52, 0 %, 0 %, 81251 MiB, 66571 MiB, 14680 MiB
```

- Follow the output with `tail -f <jobid>.gpu.log`

Support, Consultation and Collaboration

ARC Helpdesk: https://arc.vt.edu/support

ARC Helpdesk GRAs work as a team to handle most incoming questions/problems.

"How do I setup SSH keys for authentication?" "What can I do to get my job to launch faster?" "Why did my job stop?"

"Is MATLAB available on Huckleberry?" "How can I share my files with my collaborator?"

Escalate to ARC Computational Scientists as needed.

Office Hours (https://arc.vt.edu/office-hours)

Thanks for watching and listening!

- ARC Website: www.arc.vt.edu
- My contact info: Ayat Mohammed
  maaayat@vt.edu