Optimizing resource utilization for Bioinformatics workflows on Discovery

Shobana Sekar, Ph.D.
Associate Bioinformatician
Research Computing, ITS
Northeastern University

Northeastern University
Research Computing

5/4/2021

Discovery

- Northeastern's HPC cluster – located in MGHPCC (Holyoke, MA)
  - 25,000+ CPU cores
  - over 200 GPUs
- Connected to Northeastern's network via 10 Gbps Ethernet (GbE)
- Active, archive, secure, and cloud storage solutions

LVX VERITAS VIRTVS

General terminology

- Compute nodes: servers that perform HPC calculations.
- Login nodes: servers for remote connection to the cluster - this is where you land upon logging into Discovery.
- Network: connection between nodes (LAN).
- Scheduler: manages HPC resources/job scheduling between users. Discovery uses the **Slurm** scheduler.

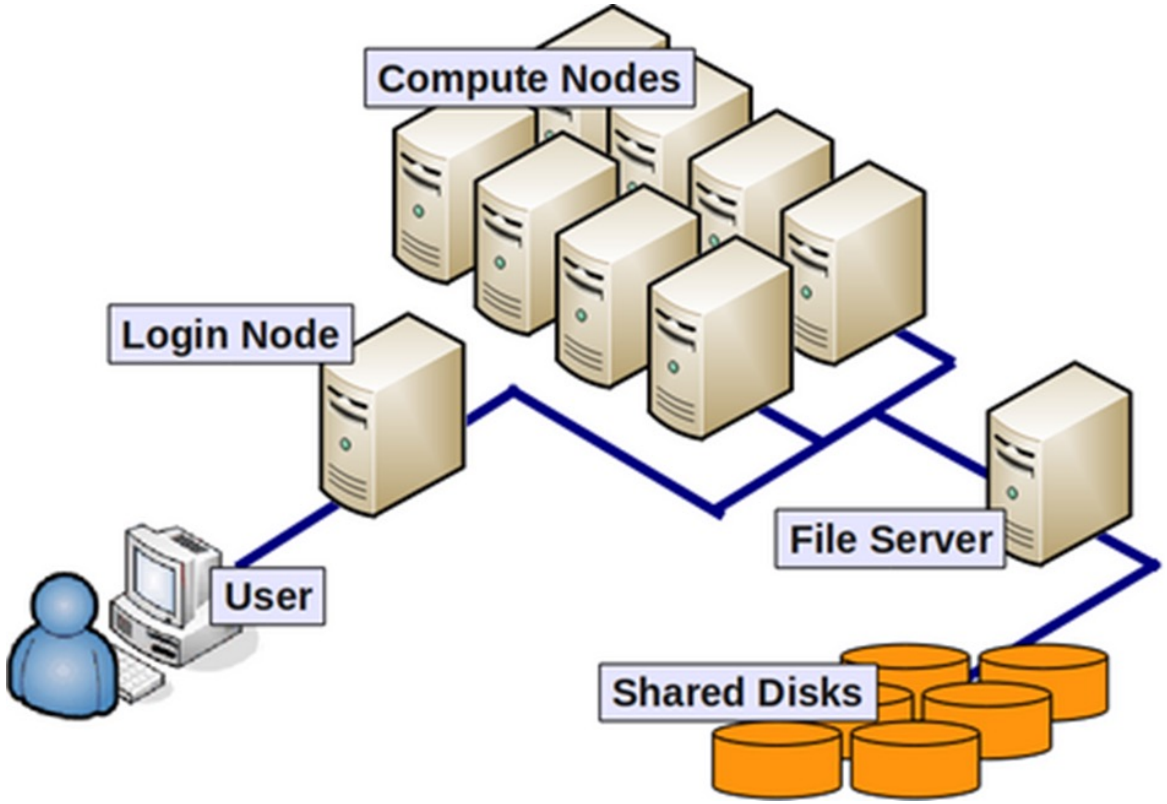


Image from:
https://wiki.rc.hms.harvard.edu/display/O2/O2+HPC+Cluster+and+Computing+Nodes+Hardware

General terminology

- Processor/CPU/Core: Central Processing Unit – a single processing component within a computer.
- Parallel Program: consists of many tasks that can be executed in parallel.

IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

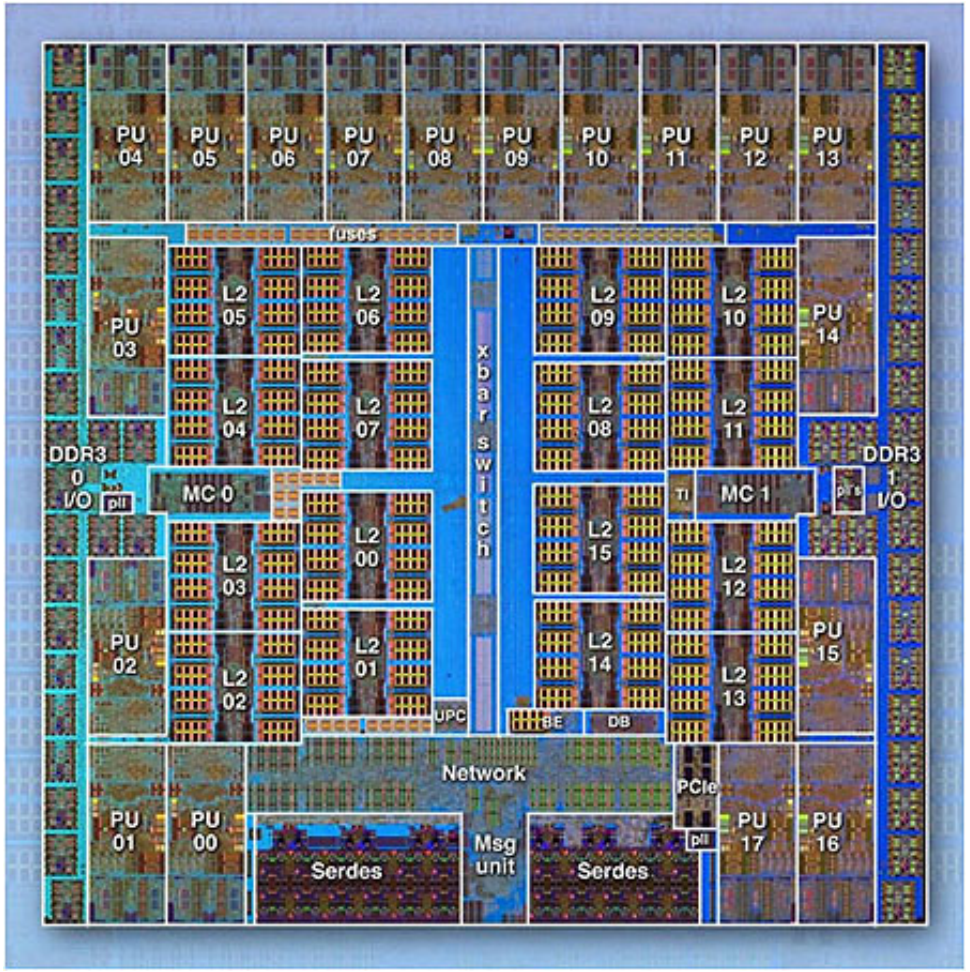


Image from the Lawrence Livermore National Laboratory Computing: https://computing.llnl.gov/

Best practices for efficient resource utilization

1. General partitions such as short, express, debug etc. are shared across all users, so **take advantage of your lab's partition** which is exclusive to your lab members (lotterhos)

2. **Perform benchmarking** i.e., run a few test jobs to assess resource utilization after they complete. Use the 'seff <jobid>' command for this on your test jobs

$ seff 18224522

Job ID: 18224522

Cluster: discovery

User/Group: s.sekar/users

State: COMPLETED (exit code 0)

Nodes: 1

Cores per node: 64

CPU Utilized: 22-04:58:07

CPU Efficiency: 90.82% of 24-10:51:44 core-walltime

Job Wall-clock time: 09:10:11

Memory Utilized: 86.83 GB

Memory Efficiency: 17.37% of 500.00 GB

Best practices for efficient resource utilization

3. When submitting jobs, you can specify specific hardware features required using the **--constraint=** flag. You can list out the available features and their corresponding information as below:

$ sinfo -p short --Format=nodes,cpus,memory,nodeai,features

NODES	CPUS	MEMORY	NODES(A/I)	AVAIL_FEATURES
175	24	109000+	144/9	lenovo,rapl,haswell
429	28	256000+	317/41	broadwell
20	128	512000	19/0	zen2,ib
4	16	384000	4/0	sandybridge,largemem
1	20	128000	1/0	ivybridge
128	56	186000	116/1	ib,cascadelake
2	224	3094534	2/0	cascadelake,ib

Specify one of this in the --constraint option

- However, remember that using a constraint can mean that you may wait longer for your job to start, as the scheduler will need to find and allocate the appropriate hardware that you have specified for your job.

source: https://rc-docs.northeastern.edu/en/latest/hardware/hardware_overview.html#hardware-overview

Best practices for efficient resource utilization

4. **Parallelize** using the following strategies:

a) Use **threads to parallelize your job**, if and when supported by the program you are running. The number of threads specified should be the same as the number of cores requested. This method parallelizes instructions on the same data

e.g., bwa aligner supports threads using -t parameter

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=8
#SBATCH --time=4:00:00
#SBATCH --job-name=MyJobName
#SBATCH --mem=100G
#SBATCH --partition=lotterhos
bwa mem -t 8 reference.fa read1.fq > aligned.sam
```

b) **Splitting the jobs** across chromosomes or samples if applicable. This method parallelizes/splits data, running the same instruction on each chunk of the data.

Note: Unless your tool is configured to support MPI (Message Passing Interface), always specify nodes = 1. If using a number higher than 1, the program should be configured to communicate between nodes, if not the additional nodes may just remain idle.

Best practices for efficient resource utilization

5. Whenever possible, try to **run individual tasks in a pipeline separately** – this acts as a 'checkpointing' technique, i.e., you do not need to re-run the entire pipeline when one step fails, but re-run only the step that failed.

   e.g., a typical WGS analysis workflow includes the following steps:

   FASTQC > Alignment > Mark duplicates > Filtering > Merging > Variant Calling

   Performing one step at a time helps you troubleshoot errors in specific steps in a more efficient manner

Best practices for pipeline automation

- Once you have verified all pipeline steps work well individually, you can also employ automation methods by using job arrays/job dependency techniques.
- A few best practices for pipeline automation:

  i. Each job within the pipeline should run sanity checks (ensure all intermediate results from previous steps were correctly produced and are available for analysis of the current step).

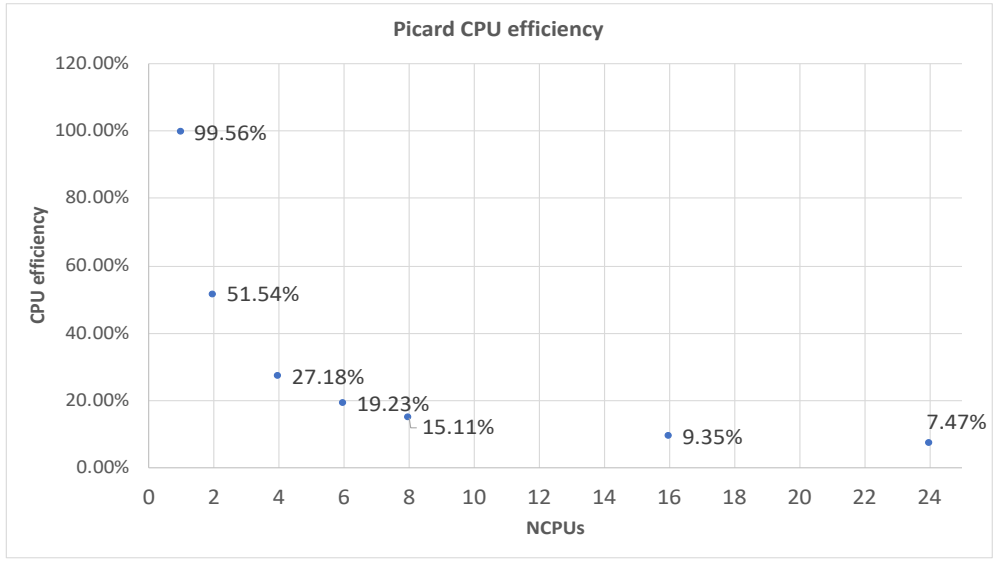
  ii. If one job fails, the following pipeline jobs should all terminate.

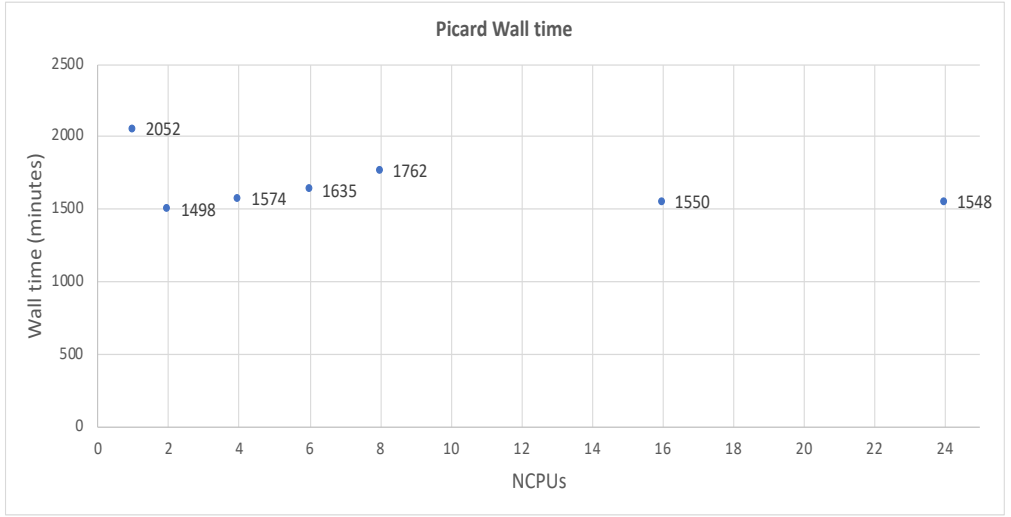
  iii. Keep track of the current step/iteration values, file names, directory names and other important parameters of the current state of the pipeline by producing informative log files.

Examples

Picard MarkDuplicates

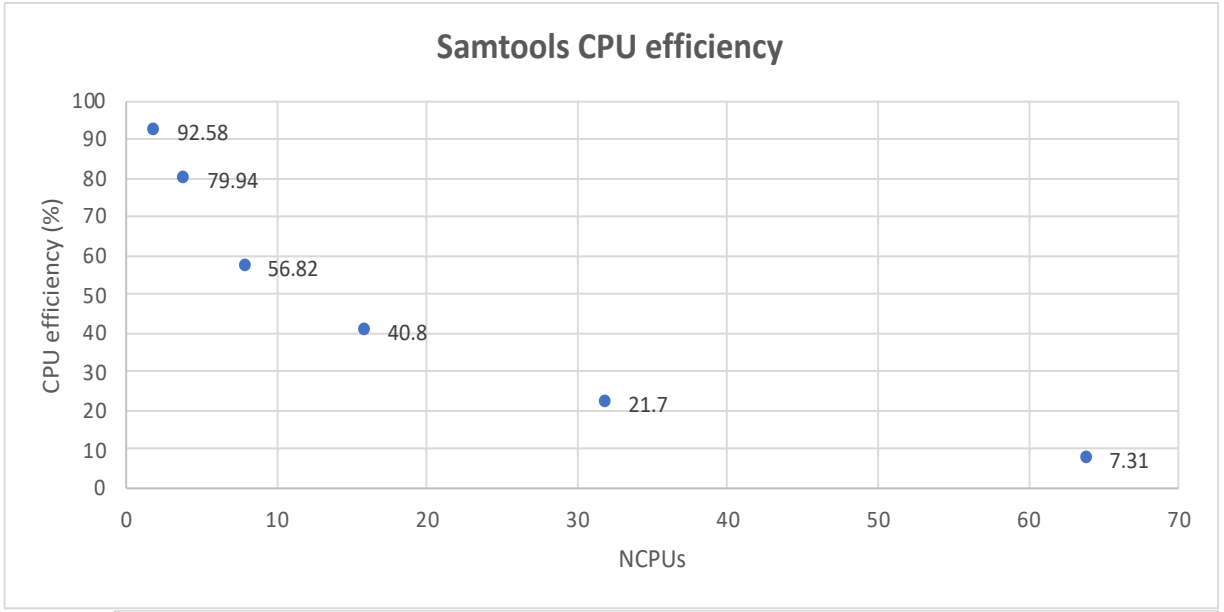
- Picard MarkDuplicates does not support threads, hence, best CPU efficiency achieved using 1 CPU

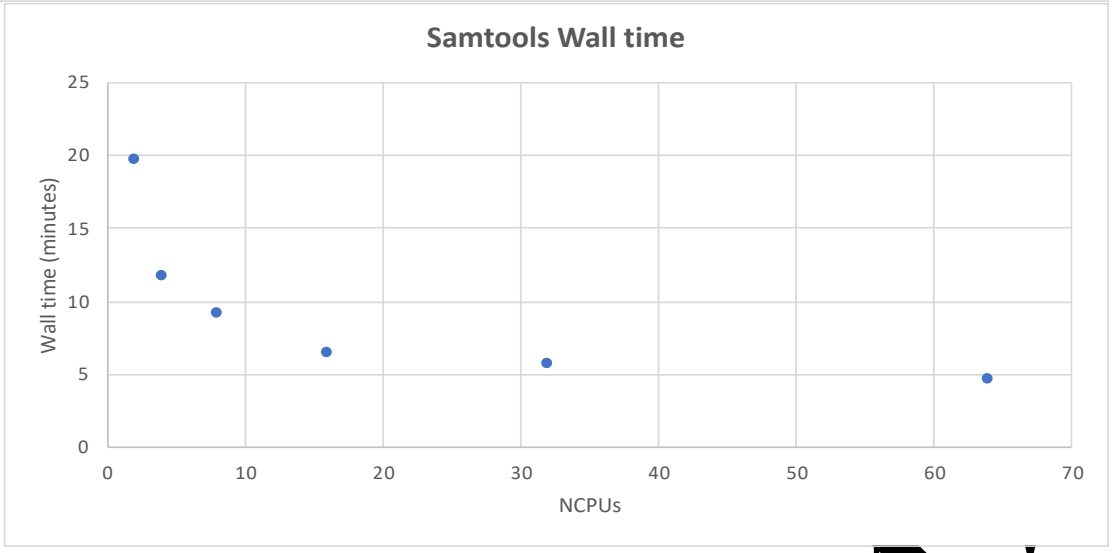




Samtools merge

- Tested merge functionality of samtools with 2,4,8,16, 32 and 64 CPUs on 5 bams
- Best efficiency achieved when using 2 CPUs
- Copied over data to /tmp to see if it helps with speeding up – no difference observed





Freebayes variant calling

Tests performed:

1. Freebayes serial on 1 chr
   - Started tests with 1 node, 1 CPU and 2G memory --> gradually increased these across different iterations
   - Cancelled due to memory exceeded issue
2. Freebayes serial on 5MB regions of 1 chr
   - Timed out after 24 hrs when submitted on short partition

Freebayes variant calling

Tests performed:

3. Freebayes-parallel on 1 chr
   - 1 node, 128 CPUs, --constraint=zen2
   - Failed after 2.5 days due to memory exceeded issue
   - At this point, also fixed an issue with the command line of freebayes-parallel - freebayes command was running on entire ref due to incorrect parameter placement
4. Freebayes-parallel on 100kb chunks on 1 chr
   - Finished in ~36 hours for all chrs in total
   - Used a combination of different architectures, 64 CPUs and ~200 GB memory when submitting for each chr

Freebayes variant calling

- Successful job examples

Using zen2 constraint with 64 cpus

$ seff 18224522

Job ID: 18224522

Cluster: discovery

User/Group: s.sekar/users

State: COMPLETED (exit code 0)

Nodes: 1

Cores per node: 64

CPU Utilized: 22-04:58:07

CPU Efficiency: 90.82% of 24-10:51:44 core-walltime

Job Wall-clock time: 09:10:11

Memory Utilized: 86.83 GB

Memory Efficiency: 17.37% of 500.00 GB

Using zen2 constraint with 128 cpus

$ seff 18190049

Job ID: 18190049

Cluster: discovery

User/Group: schaal.s/users

State: COMPLETED (exit code 0)

Nodes: 1

Cores per node: 128

CPU Utilized: 22-07:19:31

CPU Efficiency: 74.31% of 30-00:23:28 core-walltime

Job Wall-clock time: 05:37:41

Memory Utilized: 48.43 GB

Memory Efficiency: 9.69% of 500.00 GB

Using cascadelake constraint with 64 cpus

$ seff 18224528

Job ID: 18224528

Cluster: discovery

User/Group: s.sekar/users

State: COMPLETED (exit code 0)

Nodes: 1

Cores per node: 64

CPU Utilized: 28-01:07:38

CPU Efficiency: 91.62% of 30-14:41:04 core-walltime

Job Wall-clock time: 11:28:46

Memory Utilized: 86.34 GB

Memory Efficiency: 2.86% of 2.95 TB

> When running test jobs, you may allocate some extra memory as buffer to gauge how much memory will actually end up being used. Use --mem=0 in your slurm job header to allocate all memory of the node for your job.

CAUTION - Only do this for test jobs, not your actual run on all samples.

Array jobs

```
#!/bin/bash

#SBATCH --job-name=DNAMethylation_trimStep
#SBATCH --mem=50Gb
#SBATCH --mail-user=downey-wall.a@northeastern.edu
#SBATCH --mail-type=FAIL
#SBATCH --partition=lotterhos
#SBATCH --nodes=1
#SBATCH --cpus-per-task=18
#SBATCH --array=1-32%2
#SBATCH --output=/work/lotterhos/2018OALarvae_DNAm/slurm_log/%j.out
#SBATCH --error=/work/lotterhos/2018OALarvae_DNAm/slurm_log/%j.err

## Script is used for trimming and performing initial qc on raw sequences.

source ~/miniconda3/bin/activate adw_20210415

cd /work/lotterhos/2018OALarvae_DNAm

VARNAME=`sed "${SLURM_ARRAY_TASK_ID}q;d" RAW/sample_files.txt`

i=$(ls RAW/${VARNAME}_R1*)
j=$(ls RAW/${VARNAME}_R2*)

trim_galore \
--paired \
--clip_r1 10 \
--clip_r2 10 \
--three_prime_clip_R1 10 \
--three_prime_clip_R2 10 \
--output_dir trimmed_files \
--fastqc_args "--outdir trimmed_files --threads 18" \
${i} \
${j} \
2> trimmed_files/${VARNAME}_stderr.log
```

- Run benchmarks to assess if 18 CPUs is most optimal
- --array=1-32%2 submits 2 jobs at a time, serially.
- If results are not dependent on each other, you can kick off upto 4 jobs at a time using the current 18 threads setting.
- If benchmarks reveal lower number of cores is more efficient, you can kick off more jobs at a time.
- For e.g., if 8 CPUs is more efficient, you can kick off 72/8 = 9 jobs at a time. (Total number of cores available in the lotterhos partition is 72)