Introduction to Bioinformatics II

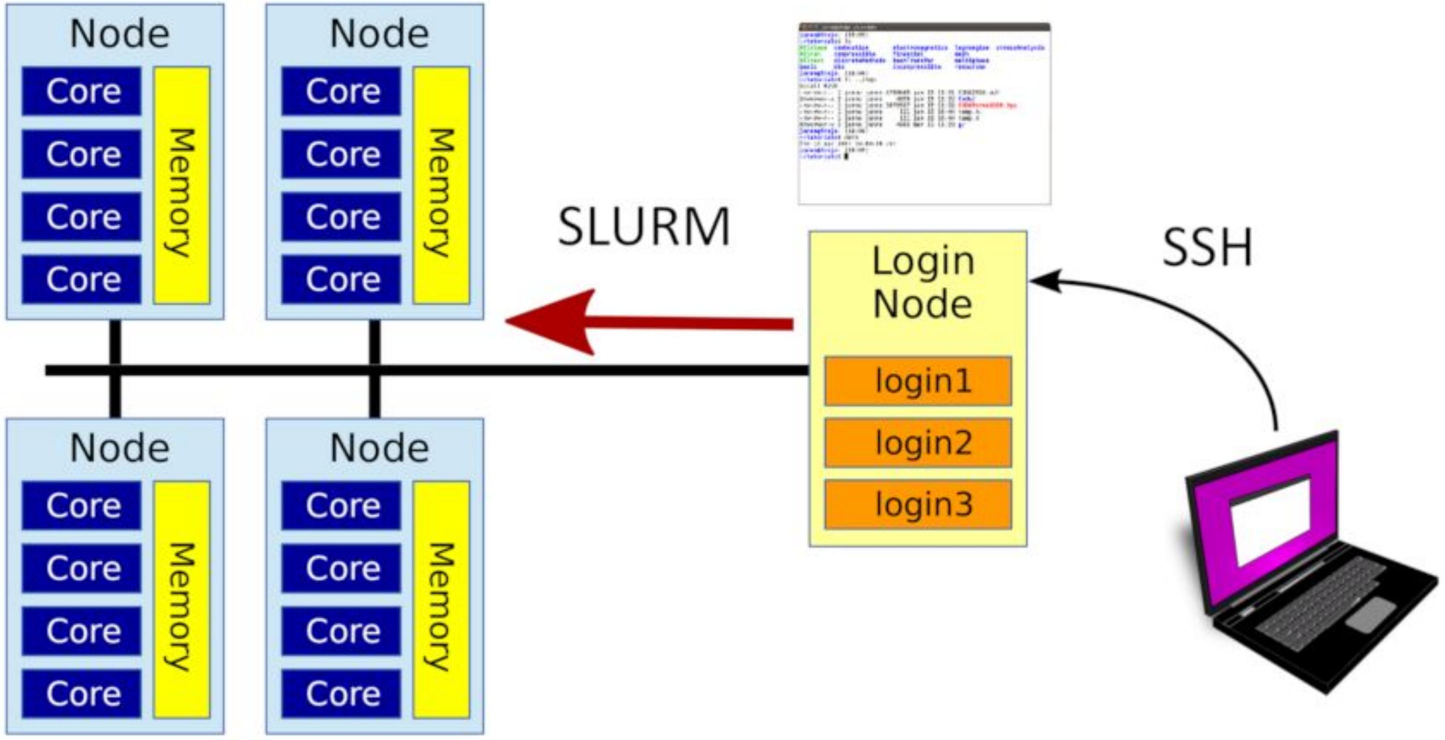
Cecile Cres & Anna Schrecengost

March 25, 2024

slurm
workload manager

Outline

- Definitions & slurm parameters
- Basics on parallel computing
- Different types of processes
    - Multithreaded processes
    - MPI processes
- Examples with python
- How to determine necessary resources
- Data storage on Unity
- Array jobs & job dependencies



https://groups.oist.jp/scs/use-slurm

Definitions

Name	Definition
node	One compute system with CPUs/cores, shared memory, and local storage
core	A single computation unit capable of running one process or one thread. On slurm, core = cpu
cpu	For Slurm, same as a core
process	A single running binary which can have one or more threads
task	For slurm, same as a process
thread	A single stream of execution. Runs on one core, belongs to a process
job	One request for resources on a cluster
partition	A group of nodes that can be used together
resource	Any hardware or software resource managed by Slurm (e.g. node, core, GPU, memory)

Important Slurm parameters

Option		Description
--ntasks	-n	Number of tasks or processes you wish to start, which use at least one core
--cpus-per-task	-c	Number of cores each task will use
--nodes	-N	Minimum number of nodes, each of which needs to run at least one task
--mem		Memory per node. Split among tasks on the same node, shared between cores in a single task
--mem-per-cpu		Memory per core. Total allocated memory per task is this value multiplied by the number of cores
--time	-t	The amount of time that you will need
--partition	-p	The partition that you wish to use
--gres		Specifies special resources that you want to use on a node (e.g. GPU units)
---job-name	-J	Give the job a name

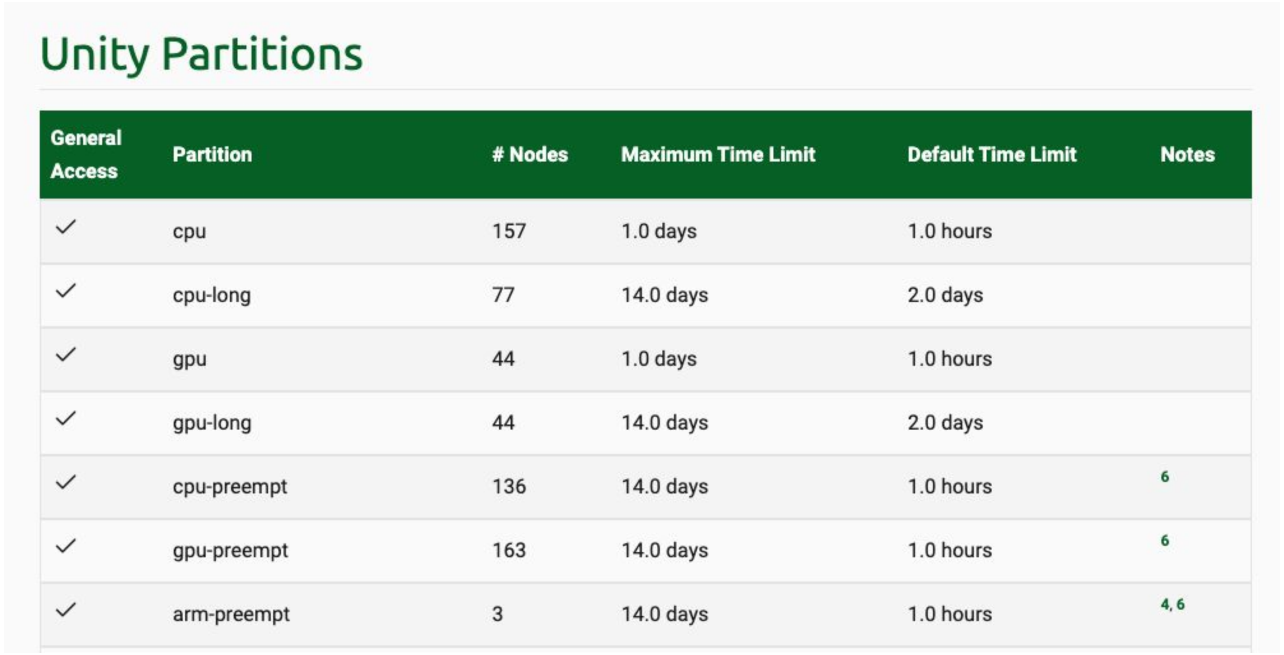
Slurm environmental parameters

Variable	Descrition
SLURM_JOB_ID	ID of job
SLURM_JOB_NODELIST	List of nodes allocated to job
SLURM_JOB_NUM_NODES	Total number of nodes in the job's resource allocation
SLURM_NTASKS	Number of tasks required
SLURM_CPUS_PER_TASK	Number of CPUs/cores requested per task
SLURM_ARRAY_TASK_ID	Job array ID (index) number

Slurm creates environmental variables when you submit jobs that you can use in your job script

Call them using e.g. $SLURM_CPUS_PER_TASK

Unity partitions (partial list)



Unity Partitions

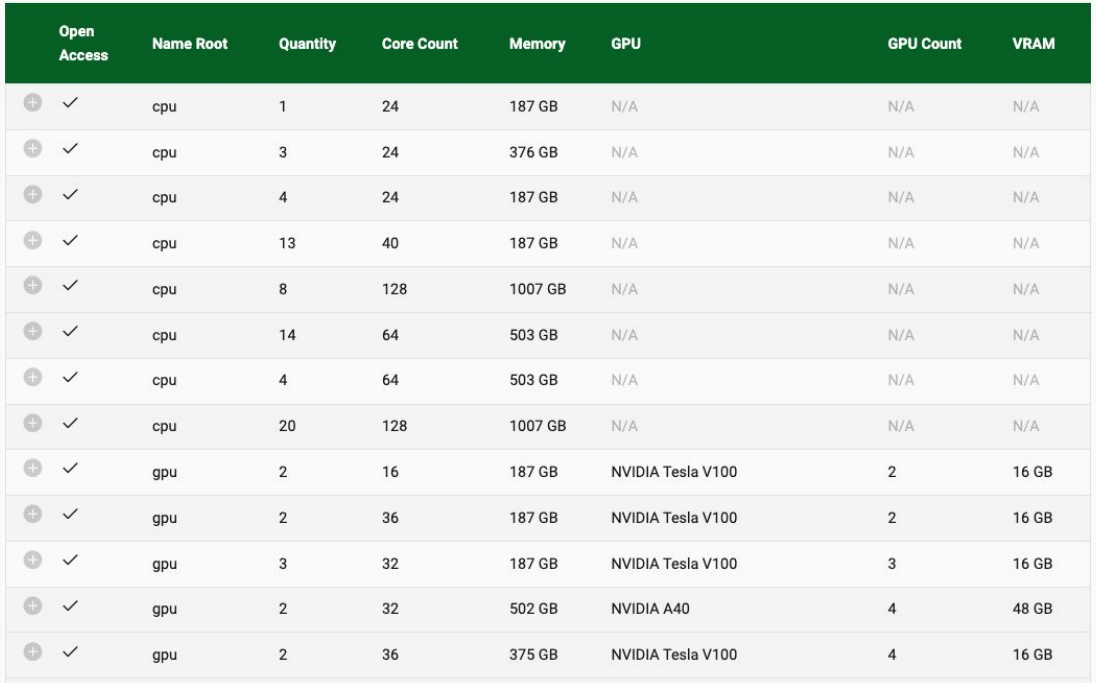
General Access	Partition	# Nodes	Maximum Time Limit	Default Time Limit	Notes
✓	cpu	157	1.0 days	1.0 hours	
✓	cpu-long	77	14.0 days	2.0 days	
✓	gpu	44	1.0 days	1.0 hours	
✓	gpu-long	44	14.0 days	2.0 days	
✓	cpu-preempt	136	14.0 days	1.0 hours	6
✓	gpu-preempt	163	14.0 days	1.0 hours	6
✓	arm-preempt	3	14.0 days	1.0 hours	4, 6

```
--partition  cpu,cpu-long
```

Specifying more than one partition: slurm will assign your job to the first available

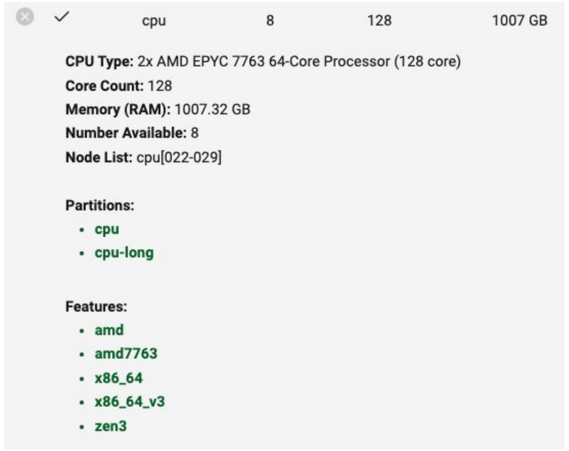
Jobs in the preempt partition start very quickly but can be preempted (stopped) after 2 hours - either make sure job has checkpoints or needs less than 2 hours

Unity nodes (partial list)



	Open Access	Name Root	Quantity	Core Count	Memory	GPU	GPU Count	VRAM
⊕	✓	cpu	1	24	187 GB	N/A	N/A	N/A
⊕	✓	cpu	3	24	376 GB	N/A	N/A	N/A
⊕	✓	cpu	4	24	187 GB	N/A	N/A	N/A
⊕	✓	cpu	13	40	187 GB	N/A	N/A	N/A
⊕	✓	cpu	8	128	1007 GB	N/A	N/A	N/A
⊕	✓	cpu	14	64	503 GB	N/A	N/A	N/A
⊕	✓	cpu	4	64	503 GB	N/A	N/A	N/A
⊕	✓	cpu	20	128	1007 GB	N/A	N/A	N/A
⊕	✓	gpu	2	16	187 GB	NVIDIA Tesla V100	2	16 GB
⊕	✓	gpu	2	36	187 GB	NVIDIA Tesla V100	2	16 GB
⊕	✓	gpu	3	32	187 GB	NVIDIA Tesla V100	3	16 GB
⊕	✓	gpu	2	32	502 GB	NVIDIA A40	4	48 GB
⊕	✓	gpu	2	36	375 GB	NVIDIA Tesla V100	4	16 GB

Click ⊕ to get more info about the nodes:



⊗	✓	cpu	8	128	1007 GB

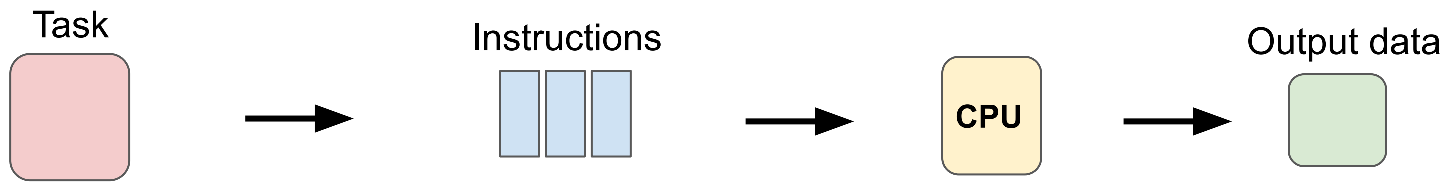
CPU Type: 2x AMD EPYC 7763 64-Core Processor (128 core)
Core Count: 128
Memory (RAM): 1007.32 GB
Number Available: 8
Node List: cpu[022-029]

Partitions:
- cpu
- cpu-long

Features:
- amd
- amd7763
- x86_64
- x86_64_v3
- zen3

Parallel computing vs serial computing

Serial computing



Calculate size of a genome

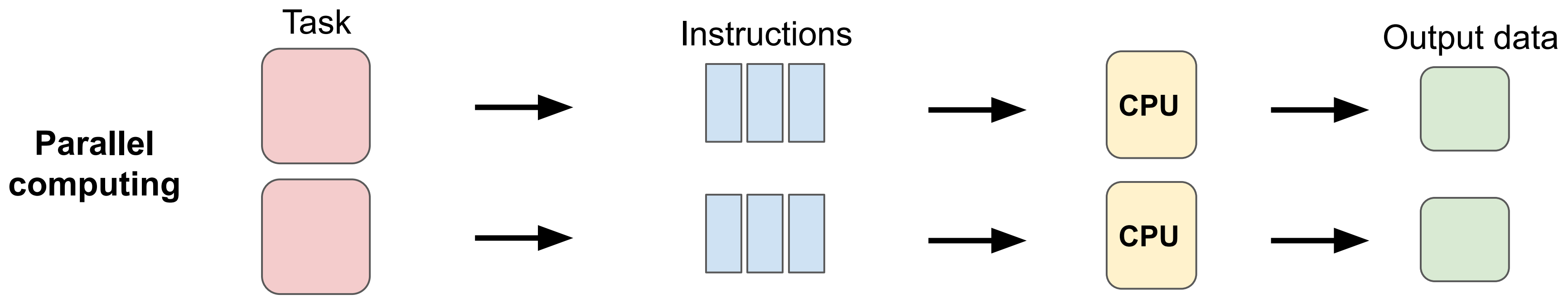
1. Open FASTA file
2. Store DNA sequence into a string
3. Calculate size of string

Output file with genome accessing ID and size

- Instructions are executed in a sequence
- Single CPU core / processor
- High workload per processor

CPU core: a compute unit of a Central Processing Unit (CPU) capable of running processes. Also referred to as a **processor**.
Process: an instance of a program.

Parallel computing vs serial computing

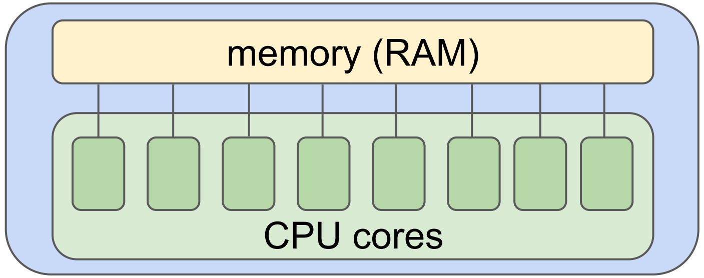


Calculate the size of genomes in multiple FASTA files (example of data parallelism)

- Instructions are executed in parallel
- Multiple CPU cores
- Low workload per processor

Parallel computing: distributed vs shared memory

Shared memory architecture

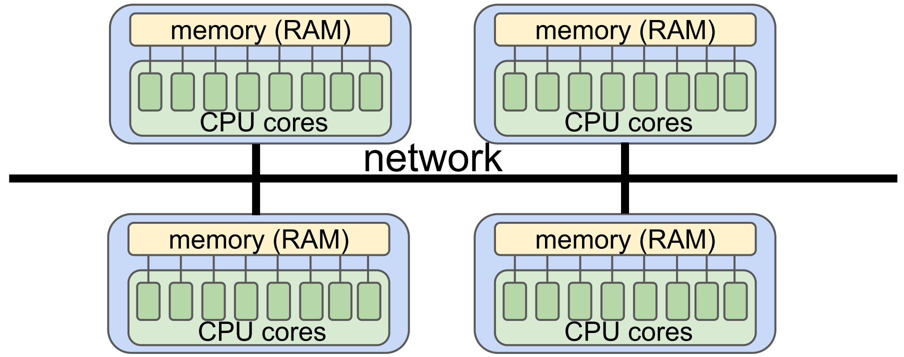


- Multiple processors share access to a global memory space
- Shared memory programming (or multithreading)
  - Threads are sub processes within a process
- Python multiprocessing library

Process: an instance of a program.
Threads: program flows with each flow processing its own subset of data or its own set of instructions.
RAM (Random Access Memory): memory space where files and programs are loaded to run.

Parallel computing: distributed vs shared memory

Distributed memory architecture



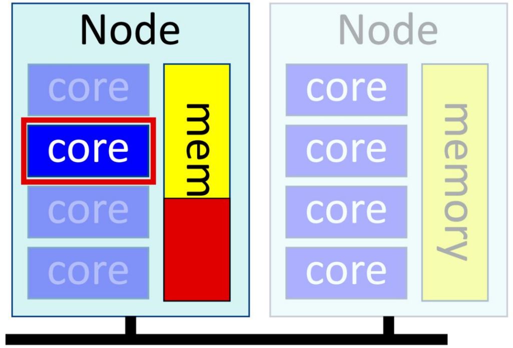
- Collection of serial computers (compute nodes) working together
- Each node has rapid access to its own local memory and access to the memory of other nodes via some sort of communications network
- Data are exchanged between nodes as messages over the network
- Message Passing Interface (MPI): message passing library for parallel programs

Parallel computing: distributed vs shared memory

	Distributed memory architecture	Shared memory architecture
Advantages	● Memory is scalable with the number of processors ● Each processor can rapidly access its own memory ● Costly	● Memory space accessible to all processors ● Data sharing between tasks is fast due to the proximity of memory to CPUs
Disadvantages	● The programmer is responsible for many of the details associated with data communication between processors. ● Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.	● Lack of scalability between memory and CPUs

Relevant Slurm parameters

input
CPU core
output



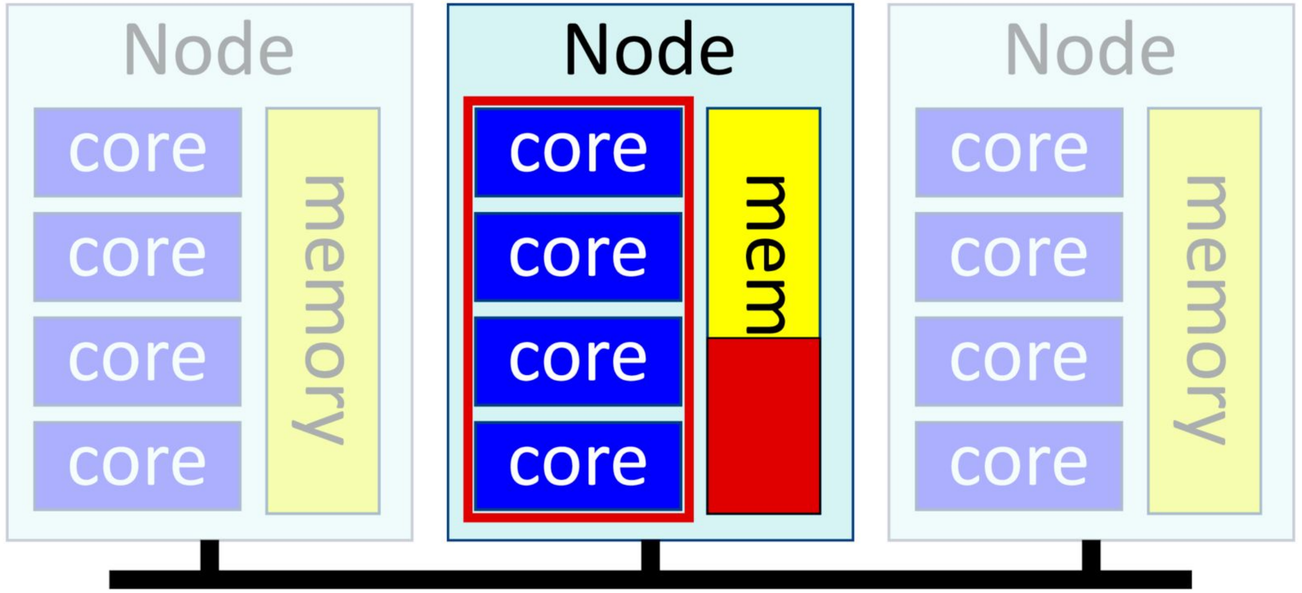
```
#!/bin/bash

#SBATCH --ntasks=1, -n=1
#SBATCH --cpus-per-task=1, -c=1
#SBATCH --mem=<X>
```

Here -n and -c are Slurm defaults

Always specify memory

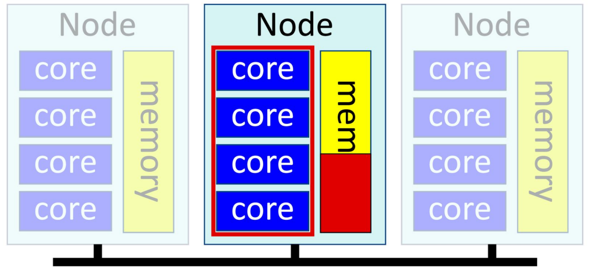
--mem=total memory for job

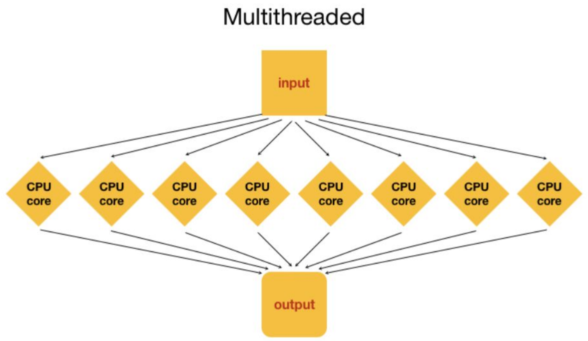
Single process, multiple threads - multithreading



https://groups.oist.jp/scs/use-slurm

Multithreading in bioinformatics





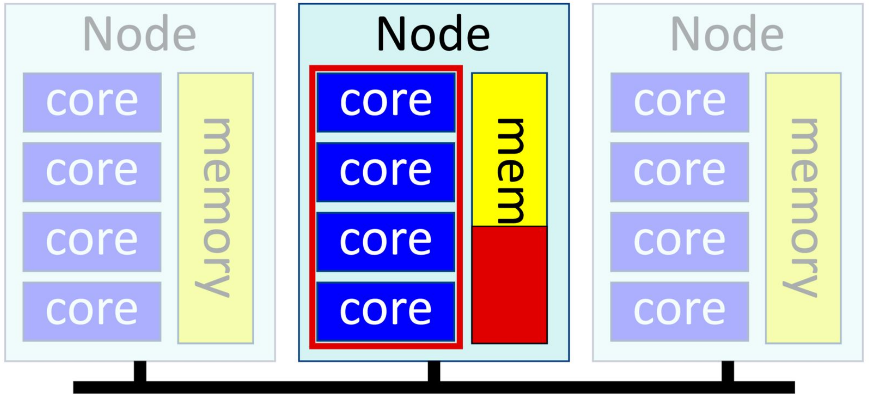
Many bioinformatics tasks are “embarrassingly parallel” - e.g. many sequence processing steps can be run independently for each sample: array jobs

But many bioinformatics problems cannot be divided into separate tasks but still need to be run in parallel across several cores due to large size of dataset and computational resources required

Most bioinformatics software can utilize multiple cores – look for parameters including the words “threads” or “cores”

https://hbctraining.github.io/Intro-to-shell-flipped/lessons/08_HPC_intro_and_terms.html

Multithreading in Slurm



```
#!/bin/bash

#SBATCH --ntasks=1, -n=1
#SBATCH --cpus-per-task=<n>, -c=<n>
#SBATCH --mem=<X>
```

--mem = memory per node = total memory for job

Make sure that you are also requesting the same amount of cores/threads in the code that you are running!

There is a practical limit on how many cores a given application can use, be careful not to ask for more than you can use and test to find out how many are effective for your job

Example batch script

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=11
#SBATCH --mem=60GB
#SBATCH --mail-user=myemail@school.edu
#SBATCH --mail-type=ALL
#SBATCH –time=10:00:00 # ten hours
#SBATCH –partition=cpu
#SBATCH –job-name=example_job
#SBATCH –output=%x_%j.out

module load raxml/8.2.12

raxmlHPC-PTHREADS-SSE3
-T "$SLURM_CPUS_ON_NODE"
-m GTRGAMMAI -s aln.fasta -n T1 -p 12345 -# 200
```

Always specify memory

Include if you want email updates for your job

Allocate enough time for job and specify partition that can provide enough

Give job a memorable name

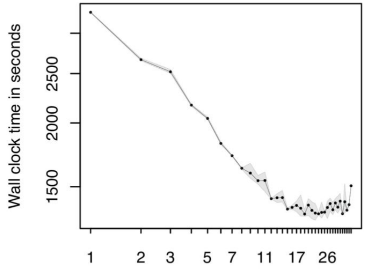
Rename standard output files based on job name (%x) and job ID (%j)

Here -T = number of threads for software to run. Slurm environmental parameter for **--cpus-per-task**

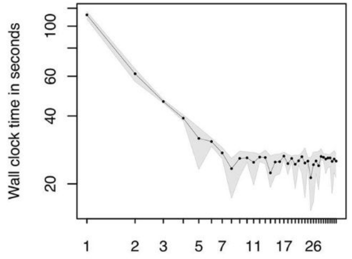
Scalability of bioinformatics tools with increasing cores

MSA

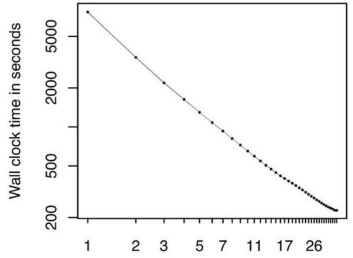
ClustalOmega



MAFFT

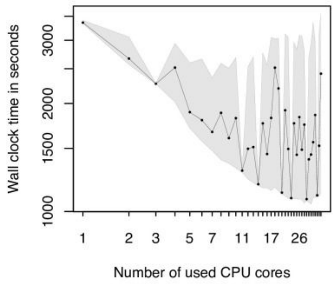


SINA

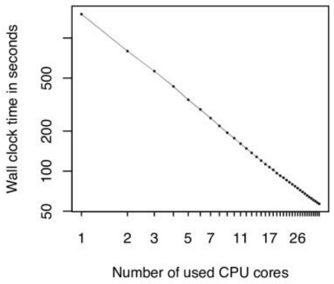


Short read alignment

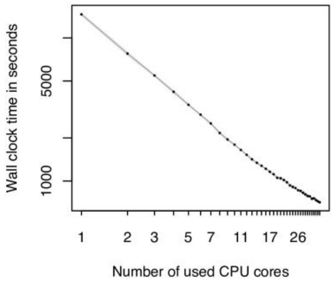
Bowtie2_build



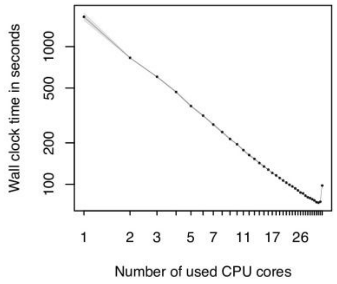
Bowtie2_align



BBMap

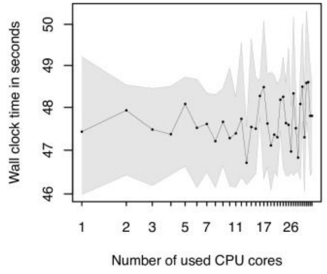


BWA_MEM

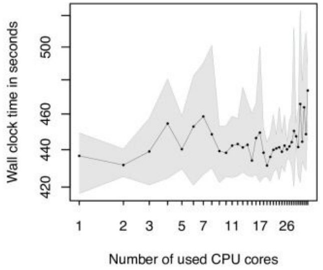


De novo sequence assembly

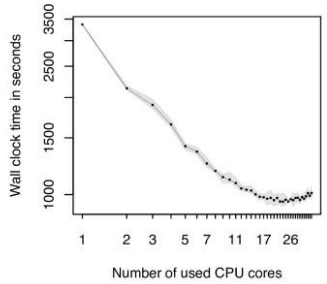
Velveth



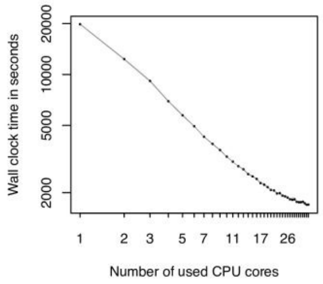
Velvetg



IDBA



SPAdes



https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8323933/

Python library for parallel computing: multiprocessing

multiprocessing is bound to a single compute node

Parallelism using Pool

- Pool maps a target function to a list of values
- By default, Pool will have one process for each CPU core requested (num_processes)
- After the execution of the code, Pool returns the output in the form of a list
- Pool only keeps processes under execution into memory
- Method preferred if dealing with a lot of data

```
import multiprocessing
import glob
import sys
from Bio import SeqIO

def get_genome_size(fa_file): # target function that takes a fasta file as input and returns a
  results = {}                    # dictionary mapping dna sequence id to size of dna sequence
  for record in SeqIO.parse(fa_file, "fasta"):
    results[record.id] = len(str(record.seq))
  return results

def get_all_genome_size(fasta_files, num_processes):
  with multiprocessing.Pool(num_processes) as pool:   # create  a Pool object with a fixed number of processes
    results = pool.map(get_genome_size_v1, fasta_files)   # call target function on all fasta files

if __name__ == "__main__":
  num_processes = int(sys.argv[1])
  fasta_files = glob.glob(os.path.join(os.getcwd(), "*.fna")) # get list of fasta files in working directory
  get_all_genome_size(fasta_files, num_processes)
```

Python library for parallel computing: multiprocessing

multiprocessing is bound to a single compute node

Parallelism using Process

- Specify function to execute via target and arguments via args
- By default, Process will have one process for each CPU core requested on node
- Process keeps all the processes in memory
- Method preferred if your task is I/O bound (task is limited by the speed at which it can perform input/output operations) or if you are working with small volume of data

```
import multiprocessing
import glob
from Bio import SeqIO

def get_genome_size(fa_file, results):
  p_results = {}
  for record in SeqIO.parse(fa_file, "fasta"):
    p_results[record.id] = len(str(record.seq))
  results[os.getpid()] = p_results

def get_all_genome_size(fasta_files, num_processes):
  with mp.Manager() as manager:  # create manager object to allow processes to manipulate python data structures
    results = manager.dict()  # create dictionary to share amongst processes
   # create list of Process objects
    processes = [mp.Process(target=get_genome_size, args=(fasta_files[i], results)) for i in range(len(fasta_files))]
    for p in processes:
      p.start()  # start the processes
    for p in processes:
      p.join() # join the processes, program will hang and  wait until  all the processes are done

if __name__ == "__main__":
  fasta_files = glob.glob(os.path.join(os.getcwd(), "*.fna"))# get list of fasta files in working directory
  get_all_genome_size(fasta_files, num_processes)
```

Python library for parallel computing: multiprocessing

multiprocessing is bound to a single compute node

Start an interactive session

Parallelism using Pool

```
$ salloc -p cpu -c 10 --mem=50G --time=02:00:00
$ python get_dna_size_pool.py 10
[{'NZ_CP009775.1': 5515525, 'NZ_CP009776.1': 115320,
'NZ_CP009777.1': 212192, 'NZ_CP009778.1': 15271},
{'NZ_CP007457.1': 2032698}, {'NZ_CP011698.1':
4105334},...
```

time to compute genome size for 503 fasta files: 1.133 seconds

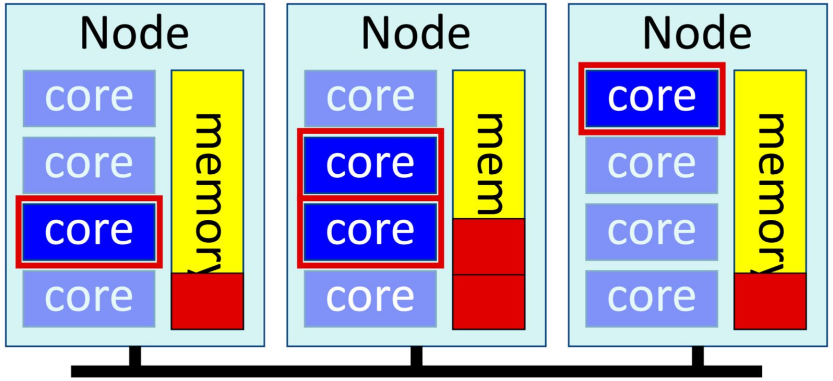
Parallelism using Process

```
$ salloc -p cpu -c 10 --mem=50G --time=02:00:00
$ python get_dna_size_process.py
{1: {'NZ_CP007457.1': 2032698}, 2: {'NZ_CP011698.1': 4105334}, 0:
{'NZ_CP009775.1': 5515525, 'NZ_CP009776.1': 115320,
'NZ_CP009777.1': 212192, 'NZ_CP009778.1': 15271},...
```

time to compute genome size for 503 fasta files: 2.880 seconds

Use **psutil** (python system and process utilities) python library (documentation) to retrieve statistics on running processes and system utilization (CPU, memory,…)
Python **multiprocessing** library documentation

Multiple processes, single thread - MPI



1. Load MPI and software
2. Run with srun

```
#!/bin/bash

#SBATCH --nodes=<n>, -N=<n>
#SBATCH --ntasks=<m>, -n=<m>
#SBATCH --cpus-per-task=<1>, -c=<1>
#SBATCH --mem=<X>

module load intel-oneapi-mpi/2021.6.0
module load raxml/8.2.11

srun raxmlHPC-MPI <arguments>
```

How to estimate resources needed?

- Finding the amount of memory and number of cores to request requires testing and experiments
- Databases → best to have jobs that allocate enough memory to store the databases
  - Program will read from disk (I/O bound) if not enough memory
- Use `seff <job id>` to check the memory and CPU usage after the job is done

Example with BLAST+:

- Sequences stored in .nsq files
- Ncbi nt database: 689GB

```
#!/bin/bash
#SBATCH --cpus-per-task=11 # number of threads + main thread
#SBATCH --mem=200G

module load blast-plus/2.13.0+py3.8.12

blastn -query gene.faa -db /datasets/bio/ncbi-db/nt -out blastn.out
-num_threads 10 -max_target_seqs 1
```

High performance storage

- /scratch/workspace → high performance VAST storage → place to read/write data
- /project → NESE → place to store data that is not actively used
- Typical workflow:
  - Transfer data from /project to /scratch/workspace
  - Run program to read and write data to /scratch/workspace
  - Transfer the final output back to /project
- Transfer data from /project to /scratch/workspace
  - Use rsync
  - If large volume of data/files (>10k) use tar
  - Use tmux (Unity documentation) to transfer data

GNU parallel

- free, open-source tool for running shell commands and scripts in parallel and sequence on a single node
- Best for workflows:
    - with tasks that do not use MPI
    - with similar tasks and no execution order requirements or data dependencies
- By default, GNU parallel runs one command per core

Example with serial tasks:

```
#!/bin/bash
#SBATCH -c 1
#SBATCH --mem=10G

module load sratoolkit/3.0.7
module load parallel/20210922

cat SRR.numbers | parallel fastq-dump --split-files --origfmt --gzip
```

GNU parallel

- Use the -j/--jobs option to specify a different number of parallel commands

Example with parallel tasks:

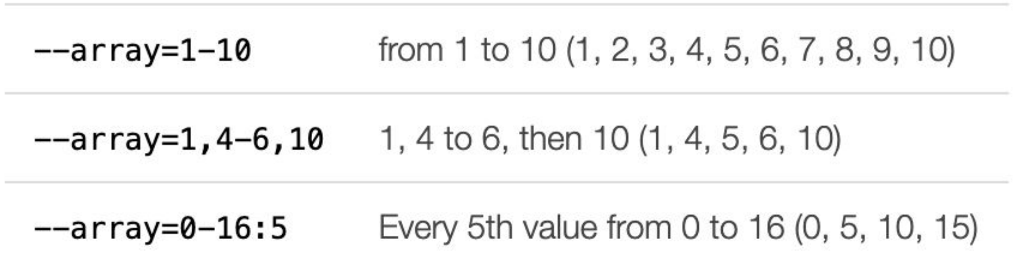
```
#!/bin/bash
#SBATCH -c 5
#SBATCH --mem=10G

module load sratoolkit/3.0.7
module load parallel/20210922

cat SRR.numbers | parallel -j 5 fastq-dump --split-files --origfmt --gzip
```

Array jobs

- Job arrays automate submission of multiple copies of a single template jobs
- Slurm creates and submits multiple jobs for you
- Limitations:
    - Consider what resources you will need for each array job
- Useful for many small tasks that can be run independently and simultaneously
- Useful when you have a collection of input files that are all analysed in the same way – e.g. processing sequencing files from many samples

Array jobs - basic options

- Use –array- flag in sbatch script to enable array processing
- Each array element is assigned an inclusive index:--array=0-10 submits 11 jobs
- Then reference array index in job script with the ${SLURM_ARRAY_TASK_ID} environmental variable



`--array=1-10`	from 1 to 10 (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
`--array=1,4-6,10`	1, 4 to 6, then 10 (1, 4, 5, 6, 10)
`--array=0-16:5`	Every 5th value from 0 to 16 (0, 5, 10, 15)

If end range with %<number>, Slurm will start at most <number> jobs at one time

Example array

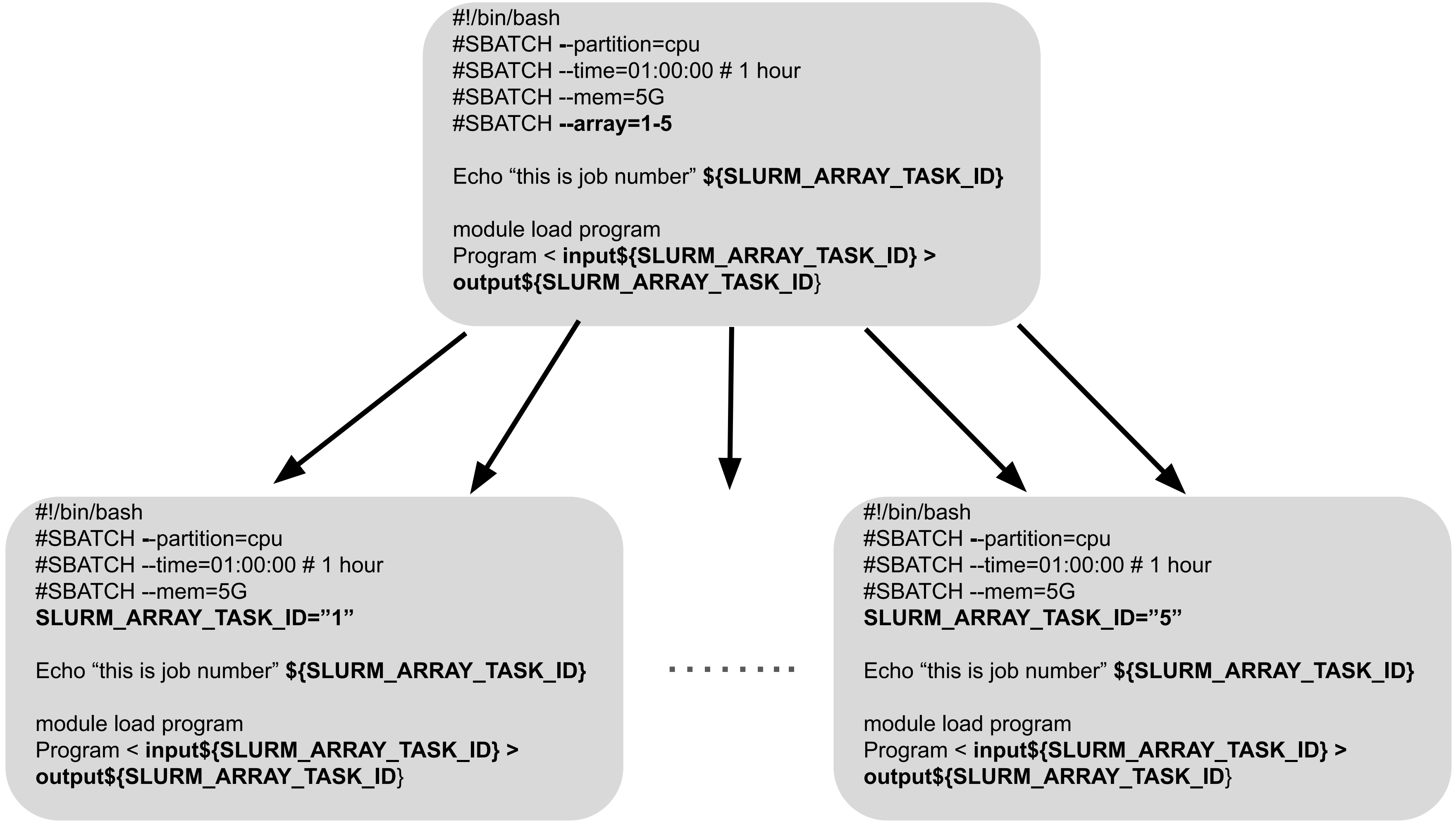
5 job array

Print the array task ID

Run program - takes input with task ID and outputs output with same task ID

Here you have input files named input_1.data … input_5.data

```
#!/bin/bash
#SBATCH --partition=cpu
#SBATCH --time=01:00:00 # 1 hour
#SBATCH --mem=5G
#SBATCH --array=1-5

Echo “this is job number” ${SLURM_ARRAY_TASK_ID}

module load program
program < input_${SLURM_ARRAY_TASK_ID}.data >
output_${SLURM_ARRAY_TASK_ID}
```



```
#!/bin/bash
#SBATCH --partition=cpu
#SBATCH --time=01:00:00 # 1 hour
#SBATCH --mem=5G
#SBATCH --array=1-5

Echo “this is job number” ${SLURM_ARRAY_TASK_ID}

module load program
Program < input${SLURM_ARRAY_TASK_ID} >
output${SLURM_ARRAY_TASK_ID}
```

```
#!/bin/bash
#SBATCH --partition=cpu
#SBATCH --time=01:00:00 # 1 hour
#SBATCH --mem=5G
SLURM_ARRAY_TASK_ID=”1”

Echo “this is job number” ${SLURM_ARRAY_TASK_ID}

module load program
Program < input${SLURM_ARRAY_TASK_ID} >
output${SLURM_ARRAY_TASK_ID}
```

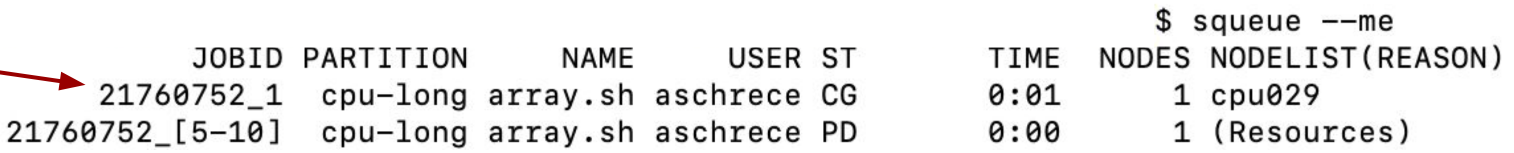
. . . .

```
#!/bin/bash
#SBATCH --partition=cpu
#SBATCH --time=01:00:00 # 1 hour
#SBATCH --mem=5G
SLURM_ARRAY_TASK_ID=”5”

Echo “this is job number” ${SLURM_ARRAY_TASK_ID}

module load program
Program < input${SLURM_ARRAY_TASK_ID} >
output${SLURM_ARRAY_TASK_ID}
```

Array jobs management

Individual running array job

Array jobs in queue



```
                                                                          $ squeue --me
          JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
     21760752_1  cpu-long array.sh aschrece CG       0:01      1 cpu029
21760752_[5-10]  cpu-long array.sh aschrece PD       0:00      1 (Resources)
```

- Cancel entire array using **`scancel JOBID`** (e.g. **`scancel 21760752`**)
- Cancel individual array using its index: **`scancel JOBID_ARRAYINDEX`** (e.g. **`scancel 21760752_1`**)
- Each array job is a copy of original template. Resource requests in the template are for each job, not the entire array.

Example array - BLAST

- Different approaches depending on how your data is structured
  - Refer to documentation for other examples

Generate list (input fasta file)

Count number of lines and use as length of array

```
$ ls -l *.fasta > input.txt
$ wc -l input.txt
23
```

Define input variable for BLAST

- Select each line in input.txt using array index

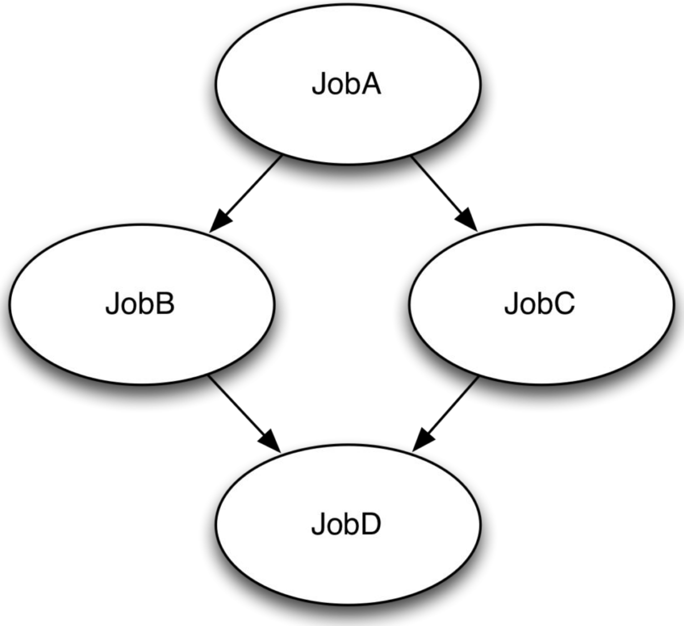
Run BLAST for each line in input

Specify number of threads

```
#!/bin/bash
#SBATCH --N 1
#SBATCH -c 12
#SBATCH --array=1-23

INPUT_FILE=$(sed -n
"${SLURM_ARRAY_TASK_ID}p" input.txt)

module load blast-plus/2.12.0
blastx -query "${INPUT_FILE}" -db nr -num_threads
"${SLURM_CPUS_ON_NODE}"
```

Job dependencies/chain jobs

- Submit a job only after another one finishes
  - Useful for workflows/pipelines, jobs with very long runtimes, jobs where some steps use CPUs and some GPUs, etc.
- Defer the start of a job until the specified dependencies have been satisfied
- Useful for workflows/pipelines or if you have really long running jobs
- Add SBATCH –dependency=<type>
  - Give Slurm a job ID or list of job IDs that you want to wait before the current job will run, and specify what condition you want to wat for



How to run job dependencies

`--dependency=<condition>:jobid:jobid:...`

<condition>=condition for job to run, **jobid** = list of job IDs job is waiting for

`--dependency=afterok:jobid`

Start job after an earlier job has finished successfully

e.g. want job2.sh to run after job1.sh:

```
$ sbatch job1.sh
Submitted batch job 14107426
$ sbatch --dependency=afterok:14107426 job2.sh
```

How to run job dependencies

`--dependency=afternotok:jobid`	Start job after an earlier job has failed
`--dependency=afterany:jobid`	Start a job after an earlier job has finished
`--dependency=after:jobid[+time]`	Start a job after an earlier job has started or been cancelled plus a delay given by [time]
`--dependency=aftercorr:jobid[+time]`	Matches array jobs with array jobs - when one job in an array ends successfully, the corresponding array job in the dependent jobs starts

Additional Resources

- Unity Onboarding video (Spring 2024)
- Snakemake workshop
- Intro to using GPUs on Unity
- AI lab workshops available this semester
- Unity community Slack
- More contact information

Next workshop

- Next workshop on Monday April 22 at 11am