Introduction to Job Scheduling and Parallel Performance

Ian Percel

University of Calgary, Research Computing Services

September 16, 2021

Outline

Where we are going

- Introduce some of the basic ideas of parallel computing (Job Schedules, Distributed Memory and Data Locality)
- Develop ideas about job scheduling workflows and resource usage analysis

Where we are going

By the end of this training, you should be able to

- Select a relevant parallel model for your computation
- Identify the resources required for your computation
- Write a slurm script for submission to the job scheduler

Models of Parallel Computing

Parallelism Broadly

Parallel work has a long history in practical human endeavours. Rigorous reasoning about it dates back to roughly the 1950s with the development of job-shop and logistical scheduling theories and their implementation on and for parallel computers. [8] [5]

- Coordinated construction of large structures [1] (e.g. Hadrian's Wall or CCC projects)
- Multi-channel pipetting [2]
- 12 bakers with 12 ovens
- Parallel computing [1]

Working on modern computing clusters (like ARC or Compute Canada sites) involves multiple kinds of resources and scales of parallelism. Disentangling them will require some discussion of parallel models and resource mappings.

Parallel Computing Strategies

One of the main routes to expanding available computational resources is by coordinating a computation across multiple processors/memory allocations.

1. Serial Computation
2. Shared Memory Parallelism
3. Distributed Memory Parallelism
4. Job-Level Parallelism
5. GPU Parallelism
6. Hybrid Models

We will emphasize distributed memory computation, job-level parallelism, and GPU parallelism as they are closely related parallel schemes.

Parallel Computing and Data Movement

Computations typically consist of two stages:

1. getting data to the processing unit
2. executing the computation

The speed with which a device can execute computations is the more commonly quoted number (e.g. 14 TFLOPS) and describes operations carried out per second under ideal conditions. However, that is only one of two rate limiting factors. [6]

Parallel Computing and Data Movement

A central concept in the analysis of parallel algorithms is *data locality* and it is usually characterized through the *compute-to-memory-access ratio* [6]

Data Locality

Moving data takes time and causes delays as it is fetched to participate in a computation performed by a processing unit. As such, the fewer reads that are required per computation, the less that data movement will contribute to limiting the speed of the computation.

Parallel Computing and Data Movement

Some reads are unavoidable and as a result *memory bandwidth* plays a key role in the total computational throughput of the device (e.g. 1000 GB/s) [6]

Data Movement Rate

The memory bandwidth can be divided by the size of the numbers that you are operating on (e.g. 4 bytes for single precision) to obtain how many numbers can be brought into the processor per second. Multiplying by the compute-to-memory-access ratio yields the maximum rate at which computations can execute based on data input requirements and data load speed.

Computational Throughput

Although processor speed, number of cores/threads/streaming processors, cpu memory size, storage I/O bandwidth, and memory bandwidth all play important roles in computational performance, it is their composite that determines the success or failure of a large scale calculation.

Successful strategies for high-performance computing prioritize *throughput* rather than *latency*.

Any individual delay is acceptable if the whole system can be made to complete faster as a result. Conversely, a small gain at one scale of computation at the expense of worse scalability is usually a bad trade in the long run.

Computational Throughput

Understanding how to get a large computation to run effectively on parallel resources requires an industrial engineering kind of thinking. Chemical reactors or job shops make a good metaphor for this kind of analysis.

We will approach this using task graphs.

A Parallelizable Example

Suppose we have 400 distinct 3-dimensional images (encoded as NIFTI files) to which we are going to apply a gaussian filter for denoising to each image. We can easily solve this for a single image using scipy. [4] The individual images are about 30MB decompressed and consist of roughly 90k elements, so processing a single image is fairly fast.

A Parallelizable Example Program

for each n in 1 to 400:

```
#read data from storage (task R_n)
myimg = Image("sample_n.nii.gz")
noisy_array = myimg.data

#processing (task P_n)
denoised_array = gaussian_filter(noisy_array, 2)

#writing data to storage (task W_n)
outImage = Image(denoised_array, name="denoised_n")
outImage.save(filename="denoised_n.nii.gz")
```

A Parallelizable Example Diagrammed

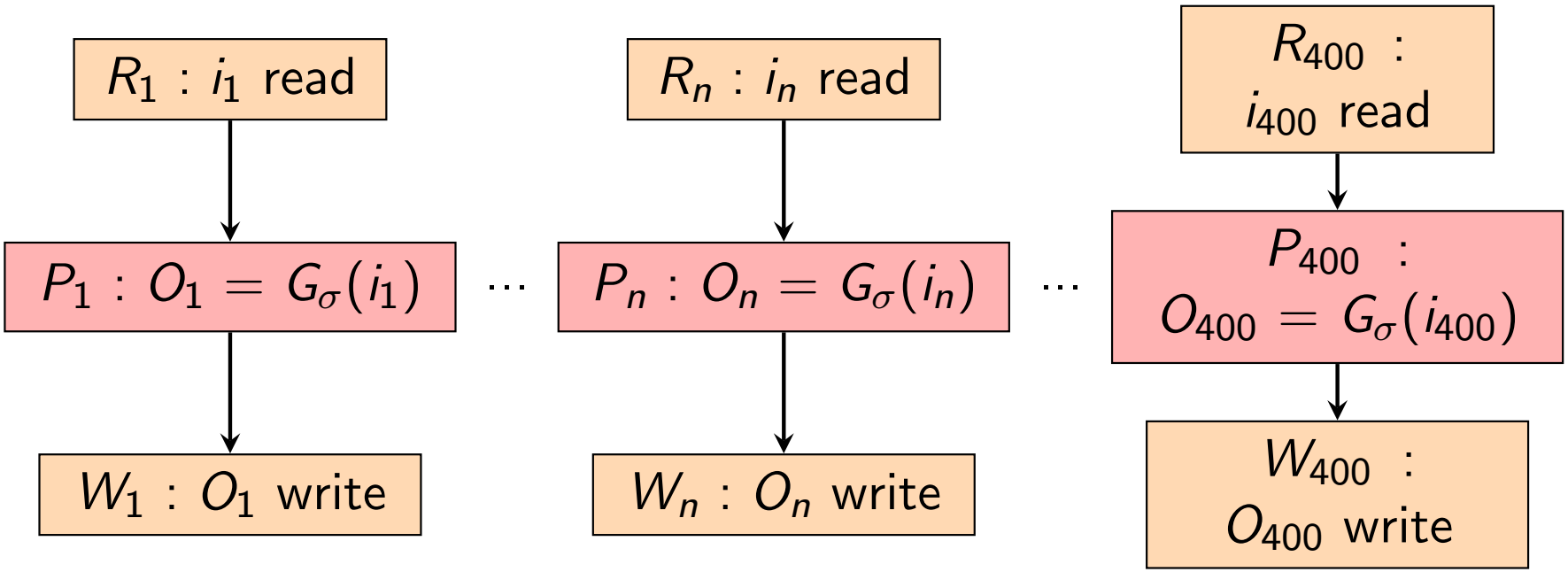


Figure 1: Coarse Task Graph for Gaussian Denoising of 400 Images [9] [5]

Serial Computation

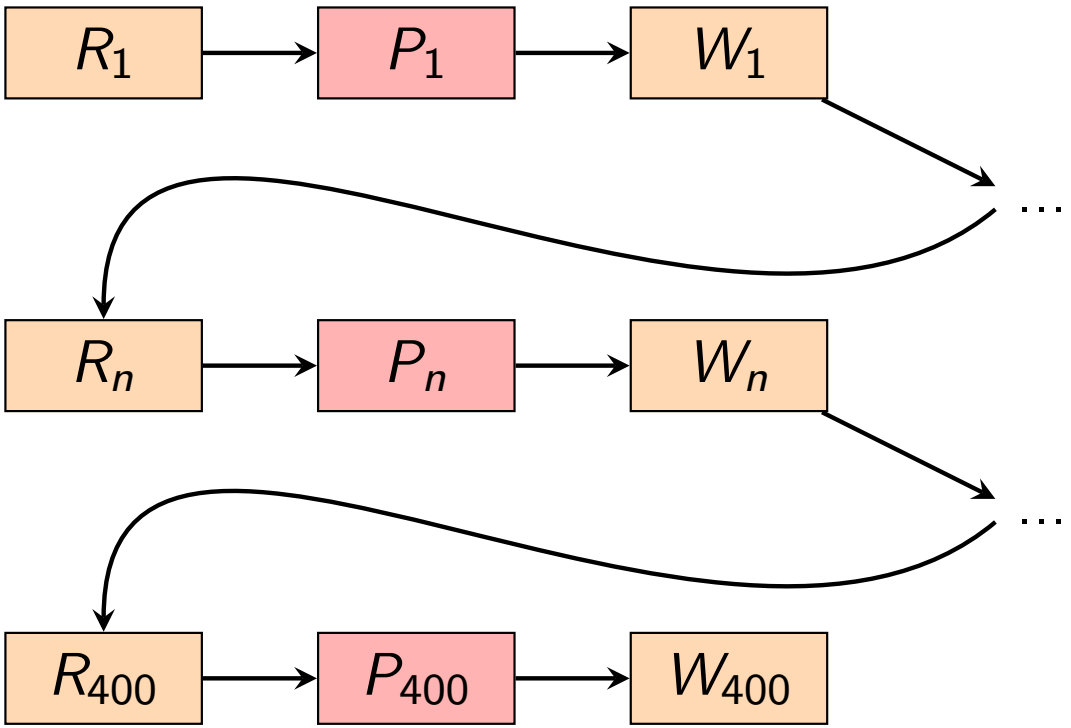


Figure 2: Serial Computation [9] [5] [7]

Shared Memory Parallelism

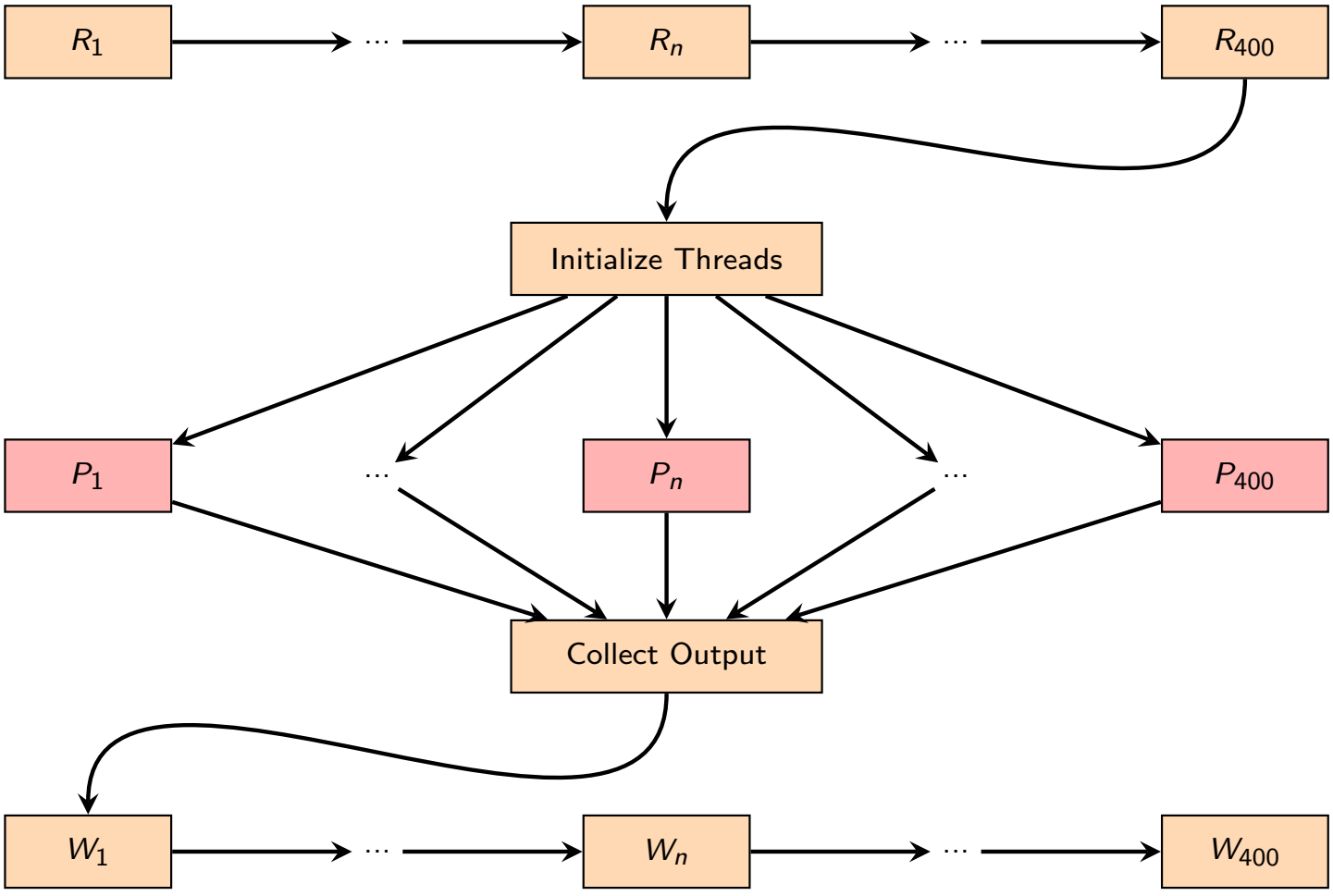


Figure 3: Shared Memory Computation with Serial Load/Write [9] [5] [7]

Distributed Memory Parallelism

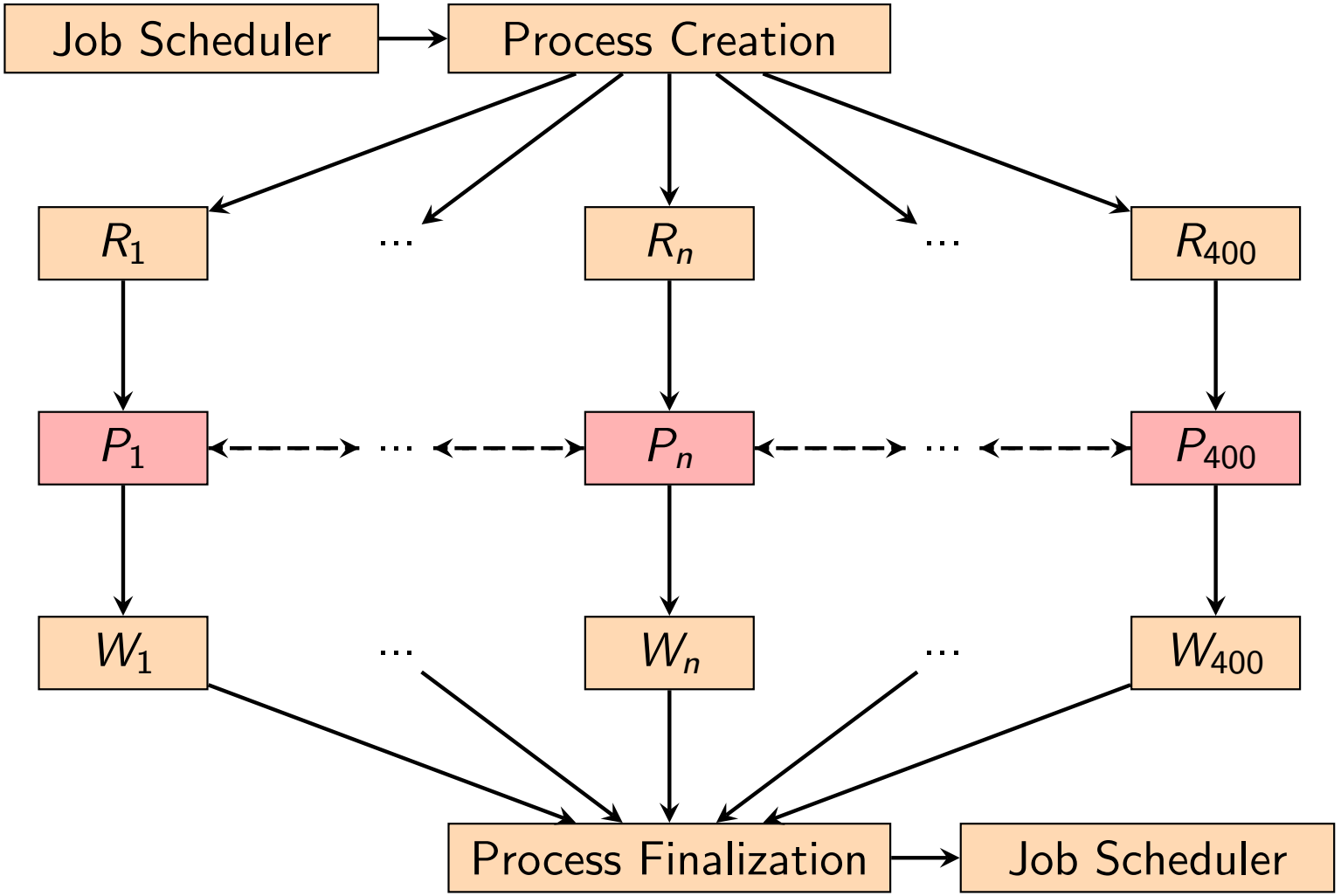


Figure 4: Distributed Memory Computation [9] [5] [7]

Job-Level Parallelism

Job-level parallelism is a simple type of distributed memory parallelism where work is broken up into batches and submitted as separate jobs to a job scheduler.

- no interprocess communication, tasks must be fully independent
- usually applied to data parallelism (e.g. 1 serial job per input file)
- parallelism managed by job scheduler (job arrays)

```
sbatch myJob1.slurm
...
sbatch myJob400.slurm
```

Job-Level Parallelism

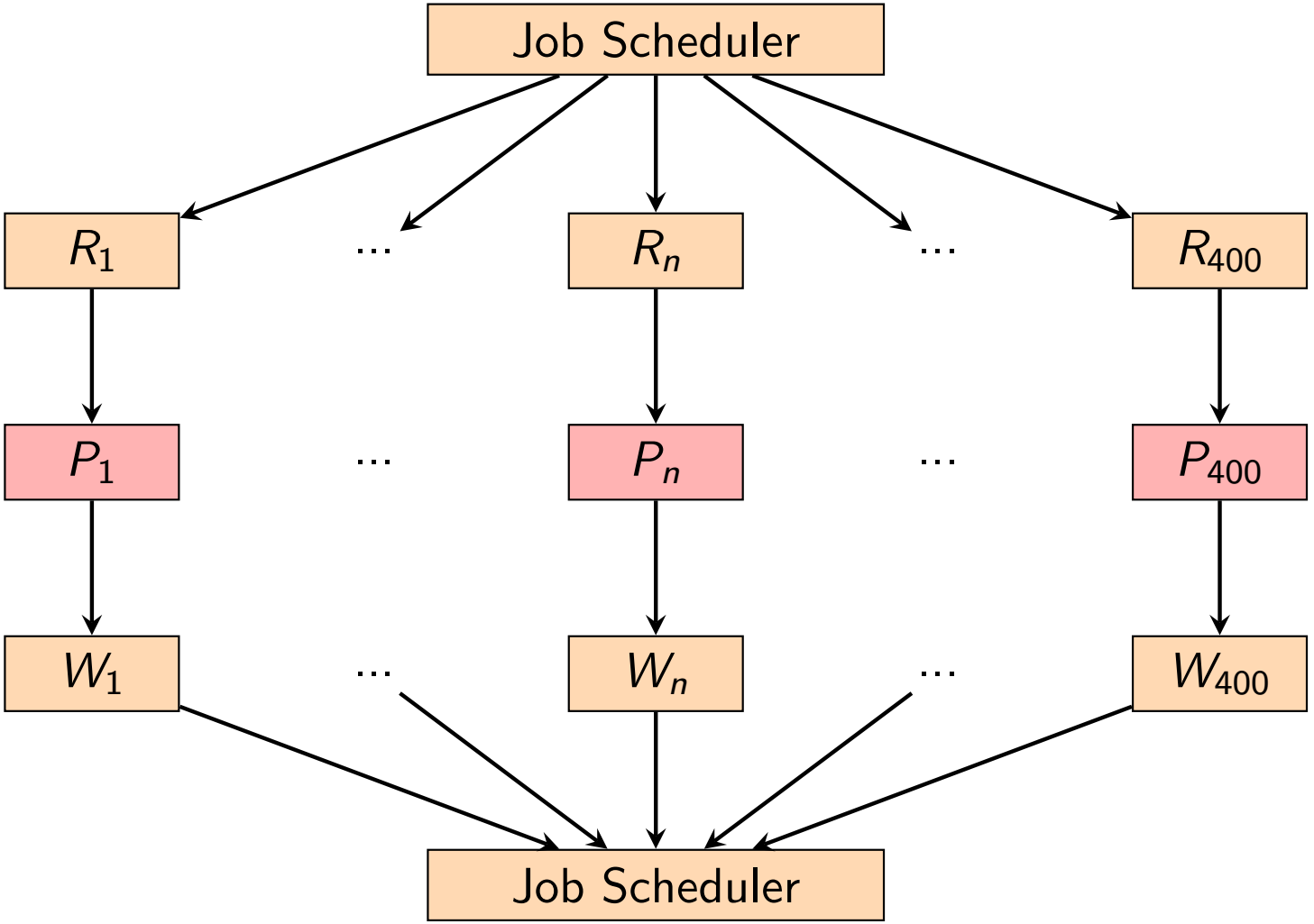


Figure 5: Job Parallel Computation [9] [5] [7]

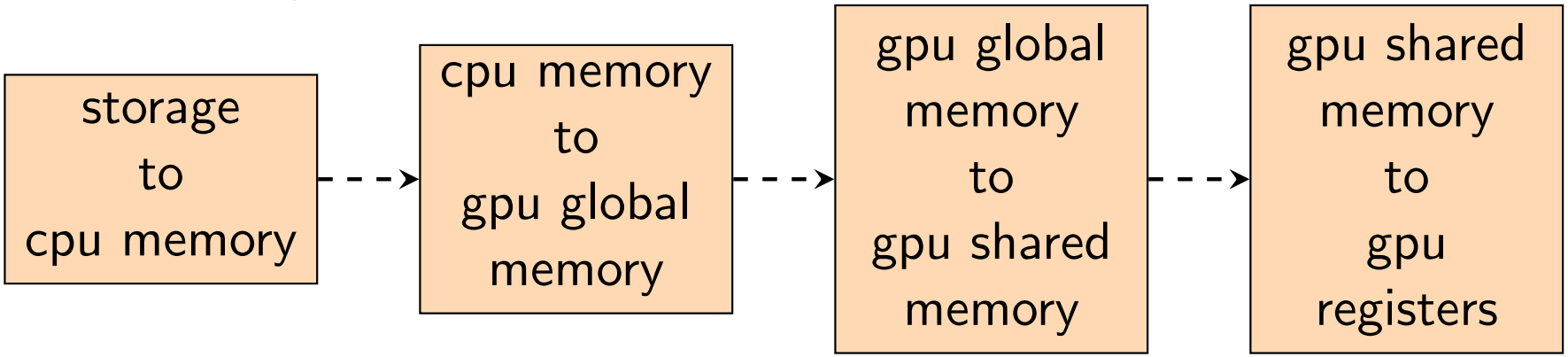
GPU Parallelism

GPU parallelism leverages a Graphical Processing Unit to manage a very large number of threads for vectorizing operations.

- Common applications: linear algebra, convolutional neural network training, and ray tracing
- can link to a standard library (e.g. cuBLAS or cuDNN) to achieve good performance and memory management
- more relevant to fine grained parallelism among operations (e.g. parallelizing the convolution itself)

GPU Parallelism and Data Path Optimization

Performance considerations for GPU parallelism are similar to those for distributed memory parallelism. However, the task graphs must explicitly account for a more complex data path from storage to the streaming processors



The deeper layers of this sequence of transfers are generally left to data loader and optimizer libraries to handle as they require a detailed knowledge of CUDA C. However, the first two must be understood by any user of a GPU.

A Parallelizable Example for GPU Data Handling

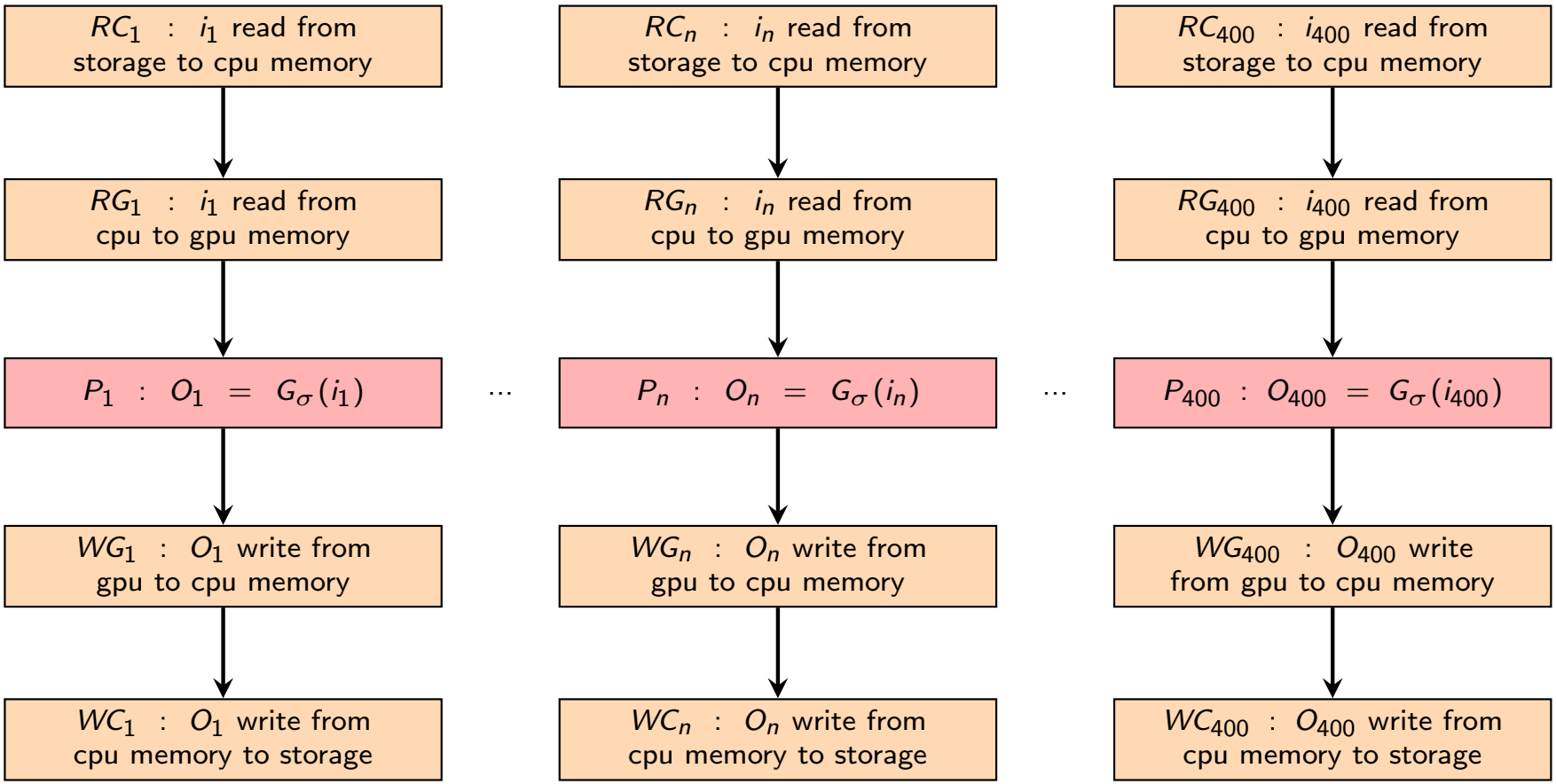


Figure 6: GPU Task Graph for Gaussian Denoising of 400 Images [9] [5]

GPU Memory Parallelism Tasks

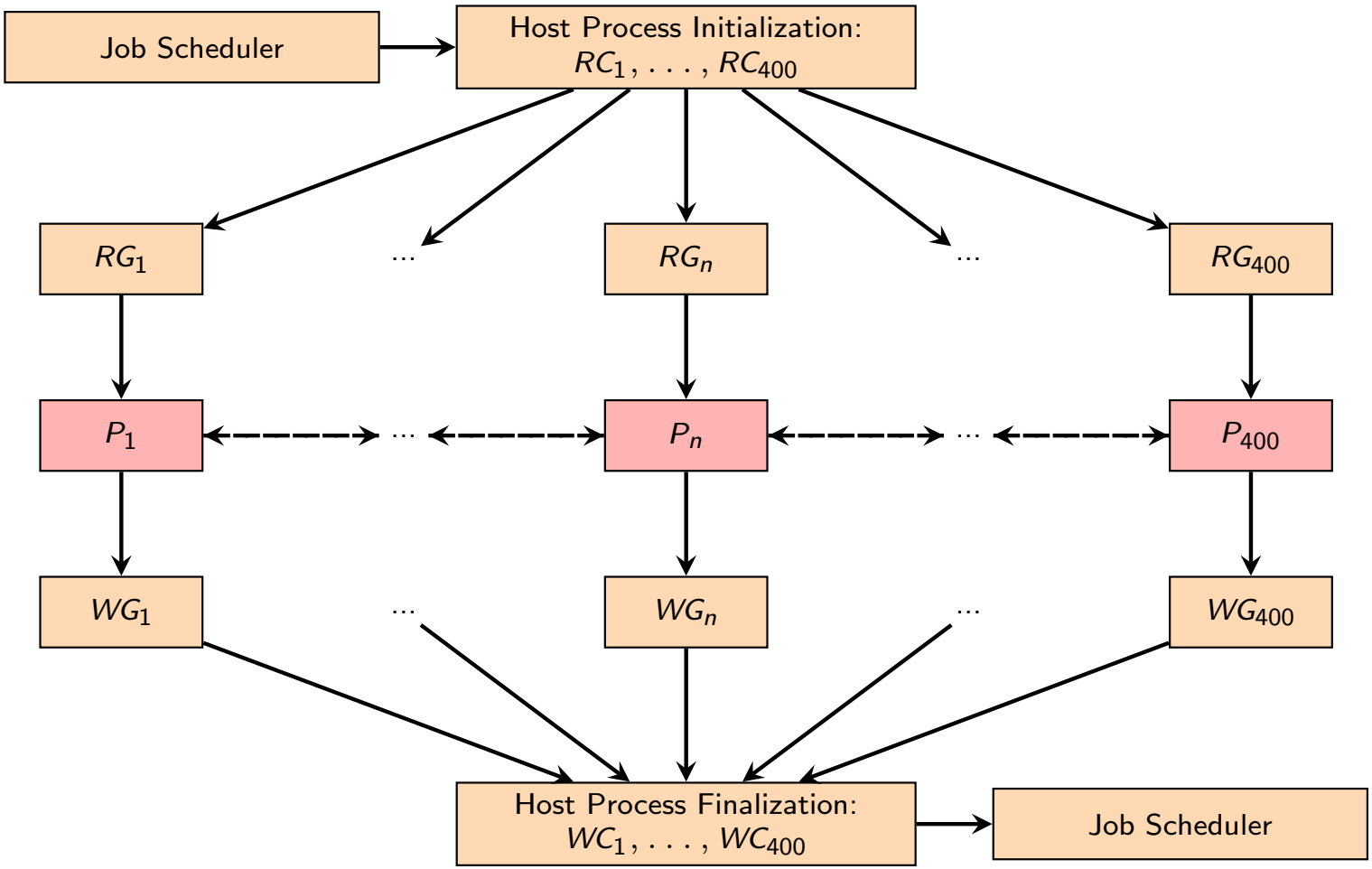


Figure 7: GPU Parallel Computation [6]

Job Scheduling Workflows

Understanding a Job Script

A Job script has three required components

1. `#!/bin/bash` or an equivalent indication of where to find the bash interpreter
2. a resource request (i.e. your contract with the scheduler)
3. a shell script (i.e. instructions in the form of bash commands for what to do when the job starts)

Sample Job Script: myJob.slurm

```
#!/bin/bash
#SBATCH --partition=cpu24
#SBATCH --job-name=threaded1-1
#SBATCH --output=/home/ian.percel/dsptest/outputs/slurm/slurm-%x-%j.out
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=11000M
#SBATCH --time=1:0:0

module load fsl/6.0.4

SCRIPT=~/dsptest/scripts/threaded_smoothing1.py
OUTPUT=~/dsptest/outputs/slurm/python-$SLURM_JOB_NAME-$SLURM_JOB_ID.out
touch $OUTPUT

\time python $SCRIPT ~/dsptest/data ~/dsptest/outputs/threaded/thread1/file1 1 1 >> $OUTPUT
```

Deriving a Resource Request from a Parallel Model

The first step in developing a job script is to create a prior estimate for the resource requirements. This guides your tests and can be inferred from parallel model being used in your code.

Parallel Type	OS Model	SLURM request
Serial	1 thread in 1 process	1 nodes 1 ntasks-per-node 1 cpus-per-task
Shared Memory	n threads in 1 process	1 nodes 1 ntasks-per-node n cpus-per-task
Dist. Memory	1 thread each in n processes distributed over k nodes	k nodes n/k ntasks-per-node 1 cpus-per-task

Computational Contexts

Job:

Managed by Scheduler
Can include many processes
Can include many threads
Can span many nodes

Large time cost to start

Process:

Started as part of one job
Can include many threads
Exists on one node
Memory isolated
Communicate with messages

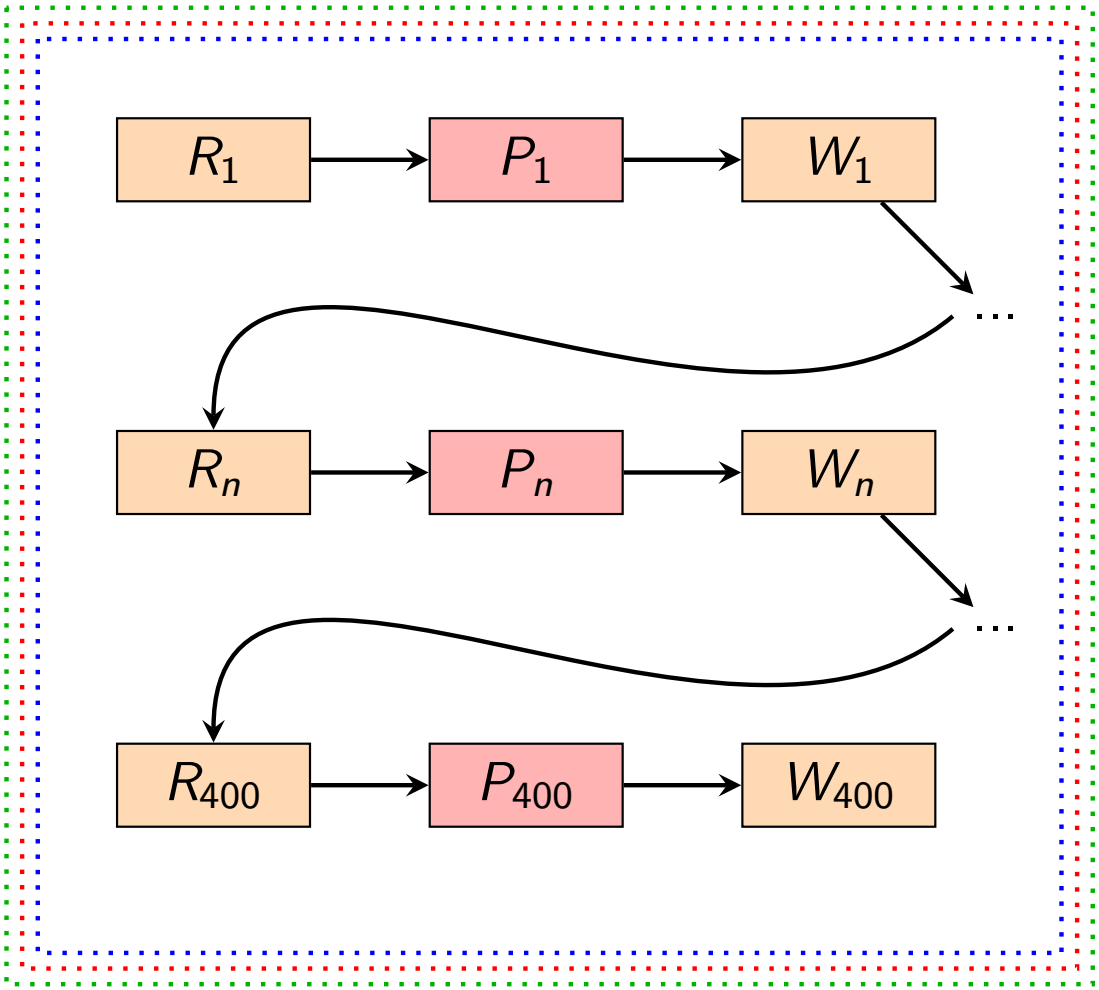
Large time and memory cost

Thread:

Started as part of one process
Exists on one node
Memory shared
Communicate through memory

Small time and memory cost

Serial Computation



1 single-threaded process

```
--nodes=1
--ntasks-per-node=1
--cpus-per-task=1
--mem=100
```

Figure 8: Serial Computation in Resources

Shared Memory Parallelism

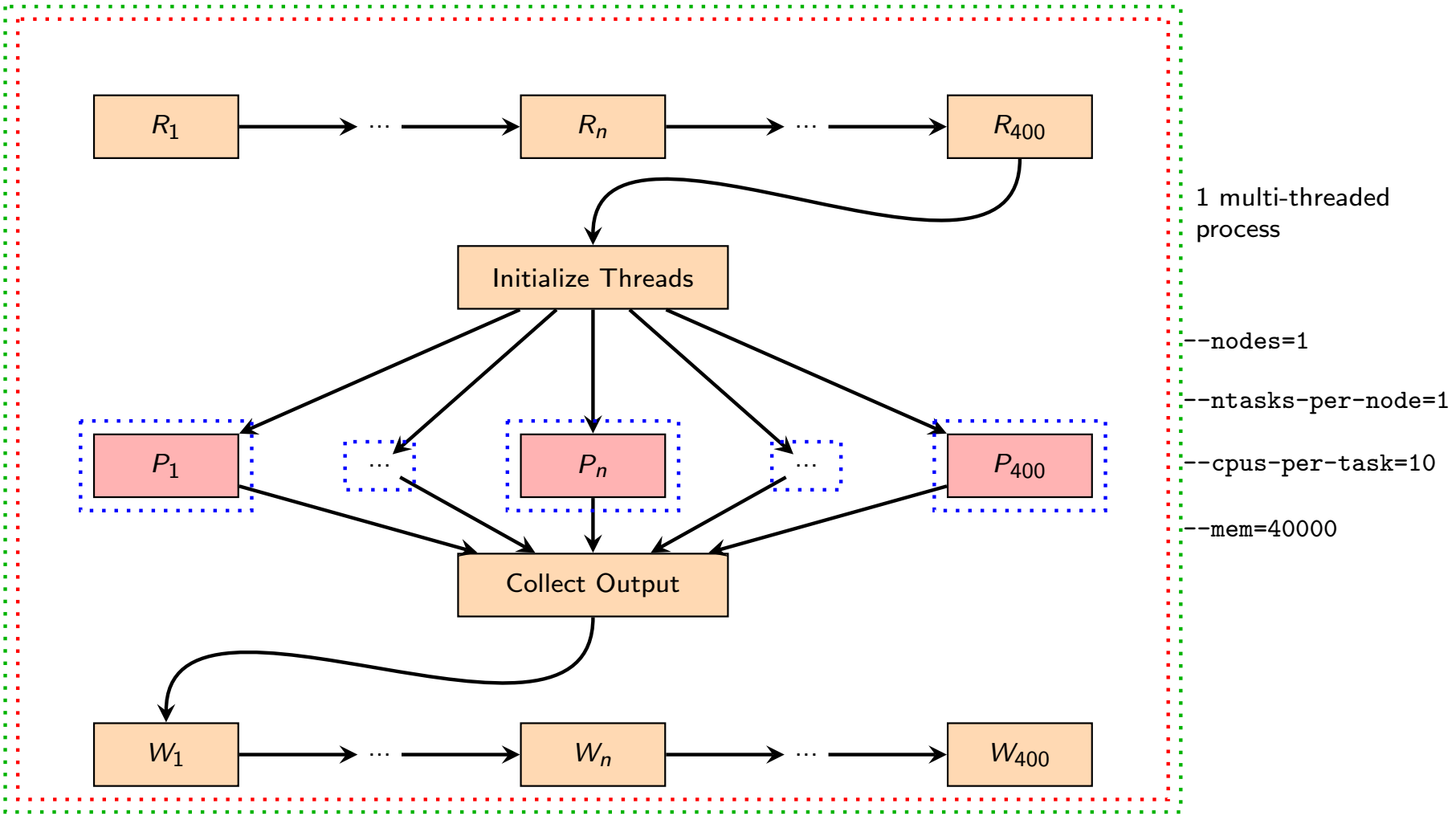
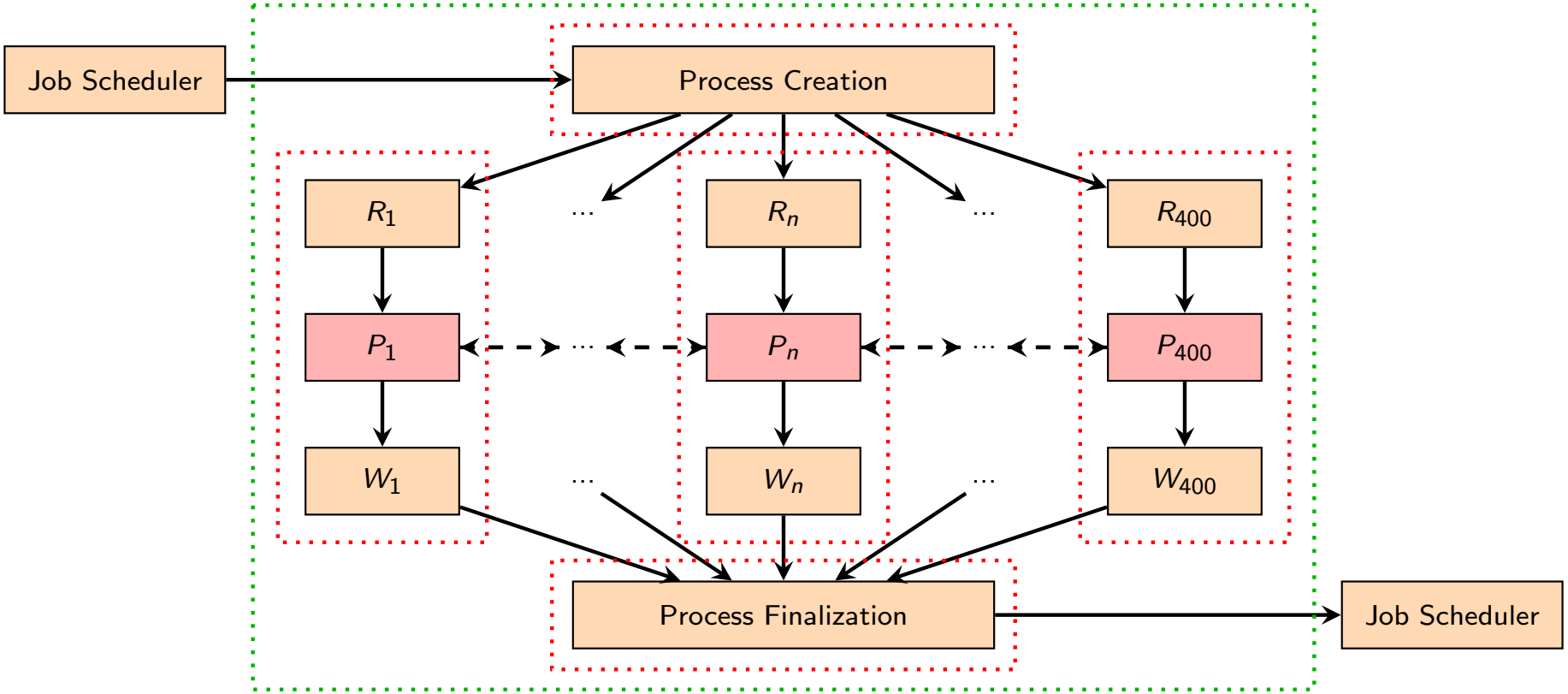


Figure 9: Shared Memory Computation in Resources

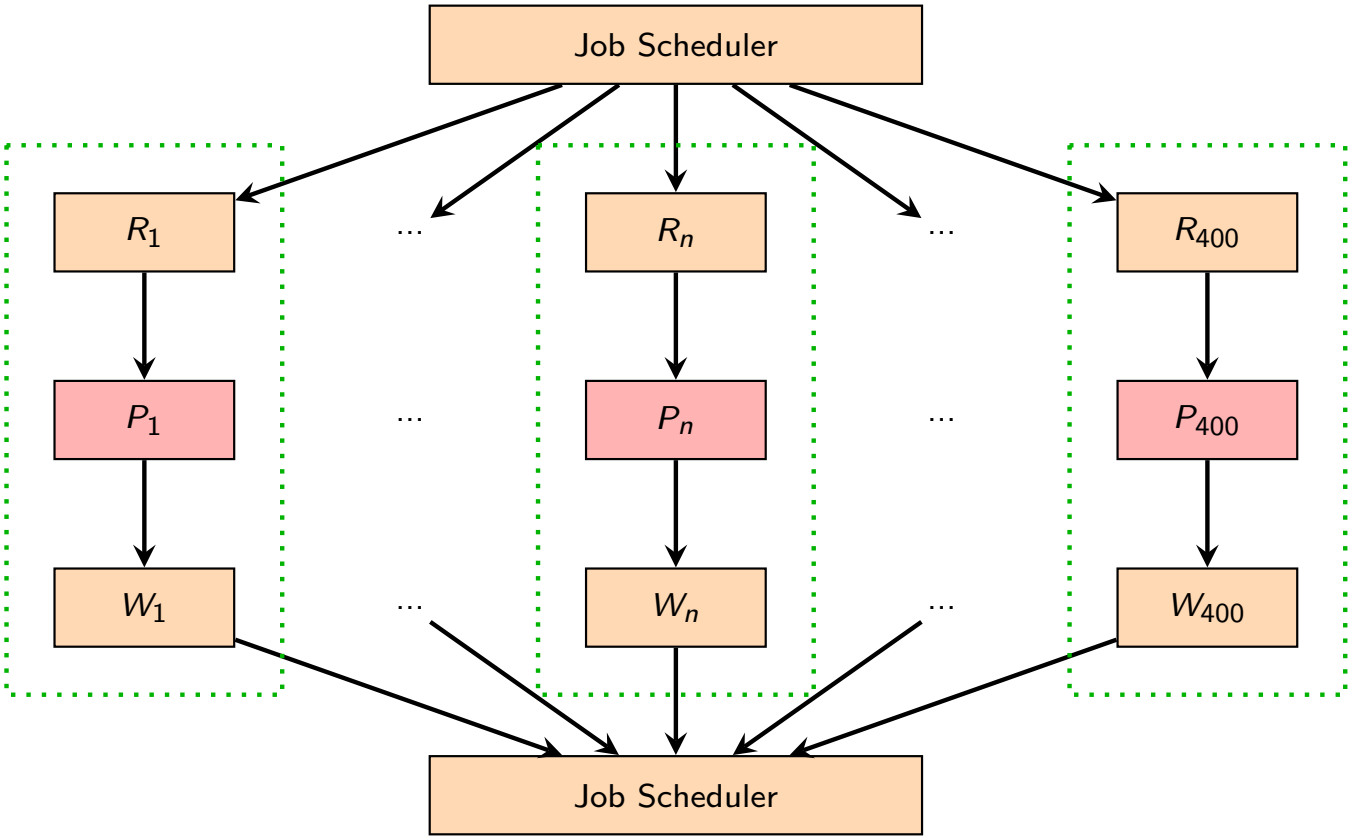
Distributed Memory Parallelism




```
--nodes=1 --ntasks-per-node=10 --cpus-per-task=1 --mem-per-cpu=100
--nodes=2 --ntasks-per-node=5 --cpus-per-task=1 --mem-per-cpu=100
```

Figure 10: Distributed Memory Computation in Resources

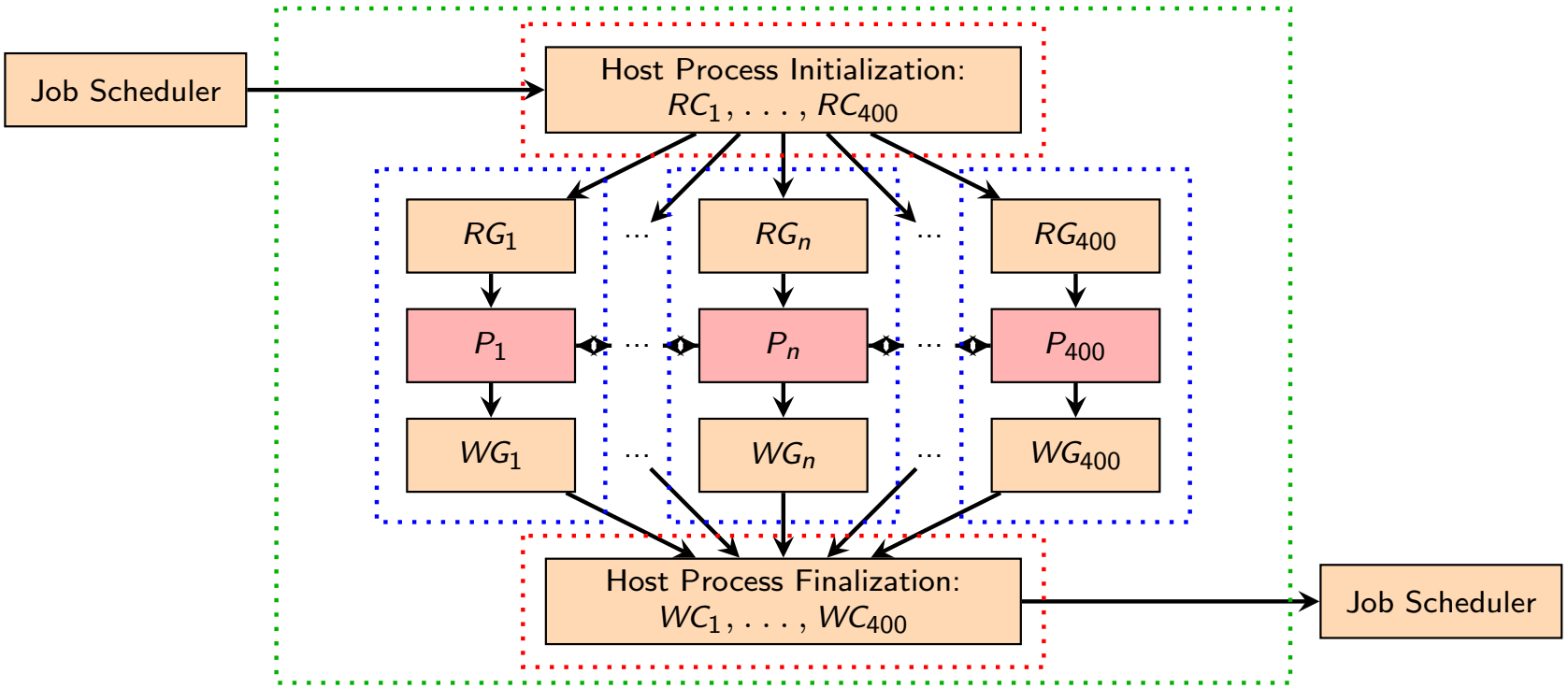
Job-Level Parallelism




```
--nodes=1 --ntasks-per-node=1 --cpus-per-task=1 --mem-per-cpu=100
```

Figure 11: Job Level Parallelism in Resources

GPU Memory Parallelism Tasks




```
--nodes=1 --ntasks-per-node=1 --cpus-per-task=4 --gres=gpu:1 --mem=100000
```

Figure 12: GPU Resources [6]

Job Script Development Work 1

The mappings explained so far should provide a preliminary guess for the resource request. (the first half of a job script) Let's use them to formulate a proposed request for the serial case.

1. Login to TALC
2. Examine the directory `dsptest`. What directories are in it?
3. Under `scripts/`, examine the contents of `serial_smoothing1.py` (you may want to `cat` it to the screen for future reference)
4. Which parallel model does this code most closely correspond to?
5. How many images coexist in memory at the same time in the full code? Propose a best guess as well as an upper bound.
6. Use `ls -lh` on the `dsptest/data` directory to see the distribution of file sizes. What is the rough maximum? What is the rough median?
7. How much memory might you need per image processed in the worst case? What about the best?
8. Write out a full request for yourself using `#SBATCH` directives
9. Examine the file `serial_smoothing1.slurm` and modify the partition, CPU request, and memory request to reflect your initial guess. (we will return to the time request shortly)
10. What do the other fields mean? (try using the `man sbatch` and searching for the parameter names)

Job Script Development 1: serial_smoothing1.slurm

```
#!/bin/bash
#SBATCH --partition=???
#SBATCH --job-name=serial1
#SBATCH --output=/home/???/dsptest/outputs/slurm/slurm-%x-%j.out
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=?
#SBATCH --cpus-per-task=?
#SBATCH --mem=???M
#SBATCH --time=1:0:0
```

assuming that we are going to use this for testing on a subset, we can start from an extreme overestimate of the single image memory utilization (say 3*file size) and then multiply this by 10 to cover a few cases that we can build a scaling estimate from even if no cleanup happens within the script's lifetime. Bear in mind the memory available on the partition that you choose.(`https://rcs.ucalgary.ca/TALC_Cluster`)

Interactively Testing for Correctness

Before submitting a job script, it is important to test your code for errors. This can be most easily done in an interactive setting, although some features will need to be tested with small batch job.

- `https://rcs.ucalgary.ca/index.php/Jupyter_Notebooks`
- `salloc --partition=cpu24 -N1 -n1 -c1 --mem=1000 --time=1:0:0`
- Either of these interfaces can be used to test code generally. If your script depends on a library in a particular module, the `salloc` option is more appropriate

Job Script Development Work 2

1. Use `salloc` to request 1GB of memory and 1 core for 1 hour
2. Using `module load <moduleName>` load the fsl/6.0.4 module to get access to a python environment with fslpy
3. start a python interpreter on the compute node
4. test the steps inside the main loop within the serial code for a single file like `sampleT1_1.nii.gz`
5. exit the interpreter and then end the interactive job with `exit`
6. How long did this take to run for the load, process and write tasks for a single image?
7. How long will the overall job likely take per image? (aim for an upper bound)
8. Update your slurm script resource request to reflect this amount of time (with some good margin since this is such a small number)

Job Script Development 2: salloc

```
[username@arc ~]$ salloc --partition=cpu24 -N1 -n1 -c1 --mem=1000 --time=1:0:0
salloc: Pending job allocation 8430460
salloc: job 8430460 queued and waiting for resources
salloc: job 8430460 has been allocated resources
salloc: Granted job allocation 8430460
salloc: Waiting for resource configuration
salloc: Nodes cn056 are ready for job
[username@cn056 ~]$ module load fsl/6.0.4
[username@cn056 ~]$ python
.
.
.
```

See `https://rcs.ucalgary.ca/Running_jobs#Job_Design_and_Resource_Estimation` for another detailed example but the simple python script used in the computational model at the start of the presentation should help you test the code in the loop.

Job Script Development Work 3

To create a working sbatch script, we will need to combine a resource request with a bash script that calls the appropriate software (in our case, a python script passed to an interpreter with arguments)

1. Examine the file `serial_smoothing1.slurm` that you have been modifying
2. What do the other request fields mean? (try using the `man sbatch` and searching for the parameter names)
3. What do the environment variables mean? (try searching the sbatch documentation)
4. What do you think this script does in terms of reading file locations and writing outputs?
5. Are there any paths you might have to change to make the script work? (make this change now)
6. Return to the `serial_smoothing1.py` script and determine what the parameters passed to the script do.
7. What fields would you change to increase the number of files processed?
8. What fields would you change to alter the output file names and paths so that your work stays organized?

Job Script Development 3: serial_smoothing1.slurm

```
...
#SBATCH --job-name=serial1
#SBATCH --output=/home/???/dsptest/outputs/slurm/slurm-%x-%j.out
...
SCRIPT=~/dsptest/scripts/serial_smoothing1.py
OUTPUT=~/dsptest/outputs/slurm/python-$SLURM_JOB_NAME-$SLURM_JOB_ID.out
touch $OUTPUT

\time python $SCRIPT ~/dsptest/data ~/dsptest/outputs/serial/1 1 >> $OUTPUT
```

Job Submission Work 4

1. Submit your completed job script to the scheduler using the command `sbatch ~/dsptest/scripts/serial_smoothing1.slurm`
2. Record the JobID that you received
3. Run the command `squeue -j JobID` where JobID is the number that you just received
4. Inspect the ST and NodeList(Reason) fields to determine if the job is running yet. When it completes it will disappear from the queue
5. Check that the job succeeded by running the command `sacct -j JobID`

Job Submission Work 4: Sample SLURM Interaction

```
$vim myJob.slurm
$sbatch myJob.slurm
Submitted batch job 7853383
$squeue -j 7853383
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           7853383 cpu24 example1 ian.perc PD       0:00      1 (Priority)
           ...
$squeue -j 7853383
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           7853383 cpu24 example1 ian.perc R       1:01      1 b14
           ...
$squeue -j 7853383
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
$sacct -j 7853383
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
7853383         example1      cpu24       ian.perc         24  COMPLETED      0:0
7853383.batch        batch                  ian.perc         24  COMPLETED      0:0
7853383.extern      extern                  ian.perc         24  COMPLETED      0:0
```

Job Monitoring Commands and Options

- `squeue` provides information on pending or running jobs that relate to the scheduling process
- `sacct -j JobID -o JobID,JobName,MaxRSS,Elapsed,State` will also provide job completion information on running or completed jobs
- once the job has started, you can use `arc.job-info` to measure the performance of the job
- this is a snapshot but it can be run repeatedly to produce a time series for further analysis

Job Monitoring Commands and Options

- The `slurm-%j.out` file (which we have renamed with the `--output` option) records any job output that would have printed to the screen (as well as any errors)
- Code that runs as part of the job can include logging that is directed to separate files
- In our case, this takes the form of a redirect >> for the output associated with python print statements inside the script
- Apart from being useful for performance analysis, when a job has stalled it can often be detected from a combination of state and logging outputs

(note, log outputs may lag behind the actual computation by as much as minutes)

Job Monitoring Commands and Options

`--mail-user` and `--mail-type` options can be combined with job dependencies to provide robust notifications about the completion times for long running jobs or collections of jobs. With mail-type set to BEGIN and END, one would get 2 emails with subjects like

```
Slurm Job_id=8442969 Name=matmul_test02032021.slurm Began,
Queued time 00:00:05
```

and

```
Slurm Job_id=8442969 Name=matmul_test02032021.slurm Ended,
Run time 00:08:02, COMPLETED, ExitCode 0
```

Job Monitoring Work 1

1. For the job that just ran, check its completion again with the `sacct` command including the `-o` option introduced in the last slide
2. Look in the `dsptest/outputs/slurm` directory and (based on the format specified in the slurm script) identify the outputs from the job script and from the python script
3. What commands in the python script produce the timing of the different tasks? What do these numbers mean?
4. Where did the output of `\time` go? Can you find the CPU efficiency from the GNU time output?
5. Where did the processed NIFTI file go? How big is it compared to the original?
6. What commands in the slurm script set the output directories for the different files?

Job Monitoring Work 1: solution

```
[ian.percel@talc dsptest]$ ls outputs/slurm
python-serial1-1291.out  slurm-serial1-1291.out
```

the python* file holds the timing information and the slurm* file holds the CPU% information

```
[ian.percel@talc dsptest]$ ls outputs/serial/1
denoisedT1_1.nii.gz
```

The `OUTPUT` environment variable determines where the detailed time logging data goes, the `--output` SBATCH directive determines where the output of GNU time and error goes, and the second argument to the python script determines where the processed file goes.

Sources of Performance Information

- In `sacct`, the MaxRSS field is an estimate to the maximum memory required for the job
- In `sacct`, the Elapsed field is the total time taken for the job
- If timing statements are included in the script, these can be used to obtain more fine-grained information about the cause for growth in different total resource requests (e.g. time spent reading and writing data as opposed to processing it)
- If `\time` is used to call a command in the shell script an estimate to average CPU utilization on a single node can be obtained for that operation
- `arc.job-info` can be used to produce a time series of resource utilization

Scaling Experiments and Analysis of Variance

In practice, it is rarely possible to estimate real job requirements from measurement of a single test job. Sources of uncertainty include:

- Large real job size requiring extrapolation fo requirements from small tests
- Variance across hardware
- Variance across datasets
- Intrinsic variance where random numbers are involved

Often this drives a need to sample different configurations and estimate the growth in memory and time required using statistical methods

Serial Data

threads	tasks	jobID	mem	mem-h	time	rtime	ptime	wtime	CPUeff	rtime/task	ptime/task	wtime/task	wtime/ptim	mem/task
1	1	8721687	178676	0.178676	39	0.45	0.44	1.53	8%	0.45	0.44	1.53	3.47727273	178.676
1	10	8721762	180580	0.18058	51	5.49	4.4	13.4	44%	0.549	0.44	1.34	3.04545455	18.058
1	20	8721793	180844	0.180844	77	11.6	8.8	29.1	57%	0.58	0.44	1.455	3.30681818	9.0422
1	40	8721798	235304	0.235304	128	23.4	17.5	60.3	67%	0.585	0.4375	1.5075	3.44571429	5.8826
1	100	8721801	236888	0.236888	290	57.5	44	157.9	73%	0.575	0.44	1.579	3.58863636	2.36888
1	200	8721803	242324	0.242324	511	102	88.25	294.9	82%	0.51	0.44125	1.4745	3.34164306	1.21162
1	300	8721806	242328	0.242328	740	140.25	133.55	439.6	86%	0.4675	0.44516667	1.46533333	3.29165107	0.80776
1	400	8721807	244320	0.24432	1042	213	179	621.5	82%	0.5325	0.4475	1.55375	3.47206704	0.6108

Multi-threaded Data

threads	tasks	JobID	mem	mem-h	time	rtime	ptime	wtime	CPUeff	rtime/task	ptime/task	wtime/task	wtime/ptim	mem/task	State
1	1	8724526	178292	0.178292	34	0.45	0.23	0.88	5%	0.45	0.23	0.88	3.82608696	0.178292	C
1	10	8724554	1192832	1.192832	48	3.6	2.43	9	29%	0.36	0.243	0.9	3.7037037	0.1192832	C
1	40	8724588	4578244	4.578244	91	13.7	9.27	34.64	59%	0.3425	0.23175	0.866	3.73678533	0.1144561	C
1	100	8724615	11350976	11.350976	168	34.56	23.45	88	77%	0.3456	0.2345	0.88	3.75266525	0.11350976	C
1	400	8724627	18626076	18.626076	142										OOM
1	400	8725851	45206212	45.206212	610	146.5	92	344.4	84%	0.36625	0.23	0.861	3.74347826	0.11301553	C
2	10	8725871	1197176	1.197176	42	3.7	1.18	10.9	33%	0.37	0.118	1.09	9.23728814	0.1197176	C
2	40	8725872	4579064	4.579064	85	18.8	4.7	34.8	62%	0.47	0.1175	0.87	7.40425532	0.1144766	C
2	100	8725873	11351664	11.351664	171	43	11.8	89	76%	0.43	0.118	0.89	7.54237288	0.11351664	C
2	400	8725874	45205092	45.205092	577	151.25	46.7	352	89%	0.378125	0.11675	0.88	7.53747323	0.11301273	C

Multi-processing Data

procs	tasks	JobID	mem	mem-h	time	setup	execution		CPUeff	setup/task	exe/task	setup/proc	exe/proc	mem/task	mem/proc
1	1	8725915	0	0	32	1	1.5		6%	1	1.5	1	1.5	0	0
1	10	8725921	184572	0.184572	56	0.2	17.5		24%	0.02	1.75	0.2	17.5	18457.2	184572
1	40	8725922	185220	0.18522	105	0.2	65.5		50%	0.005	1.6375	0.2	65.5	4630.5	185220
1	100	8725923	186788	0.186788	189	0.14	158		68%	0.0014	1.58	0.14	158	1867.88	186788
1	400	8725924	194528	0.194528	650	0.14	618		79%	0.00035	1.545	0.14	618	486.32	194528
2	10	8725925	44684	0.044684	41	0.1	9.3		35%	0.01	0.93	0.05	4.65	4468.4	22342
2	40	8725926	328236	0.328236	67	0.1	34.75		81%	0.0025	0.86875	0.05	17.375	8205.9	164118
2	100	8725927	326464	0.326464	114	0.1	82		116%	0.001	0.82	0.05	41	3264.64	163232
2	400	8725928	349832	0.349832	339	0.1	306.7		153%	0.00025	0.76675	0.05	153.35	874.58	174916

Scaling Experiments Work 1

This project will explore the task of running on the full collection of 400 images using different parallel strategies, thread and process counts, and how performance and resource requirements scale as the problem grows.

1. Use `cp` to copy the slurm serial processing template that you have been working on to produce versions that will process 1, 10, 20, 40, 100, 200, 400 files. Make sure that the slurm and python output files go in the `outputs/slurm` directory with unique names and the processed data goes in appropriate `outputs/serial` subdirectories.
2. Make a table of outputs for total memory, CPU Efficiency, read time, processing time, write time, and total time for each size of job
3. How does memory usage grow with increasing numbers of images? How much memory would you need to request for 40000 images?
4. How does time grow with increasing numbers of images? How much time would you need to request for 40000 images?
5. What component of the time in the computation is dominant?

Scaling Experiments Work 2: Multithreading

1. Repeat your analysis for correctness and initial request in the multithreaded python script (`threaded_smoothing1.py`) and corresponding slurm script (`threaded_smoothing-thread1_file1.slurm`)
2. Use `cp` to copy the slurm threaded processing template to produce versions that will process 10, 40, 100, and 400 files using 1, 2, or 4 threads. Make sure that the slurm and python output files go in the `outputs/slurm` directory with unique names and the processed data goes in appropriate `outputs/threadparallel` subdirectories.
3. Make a table of outputs for total memory, CPU Efficiency, read time, processing time, write time, and total time for each size of job

Scaling Experiments Work 3: Multithreading

1. How does memory usage grow with increasing numbers of images? How much memory would you need to request for 40000 images?
2. How does time grow with increasing numbers of images? How much time would you need to request for 40000 images using 2 threads?
3. How much is it possible to benefit from increasing parallel processing for the convolution alone?

Scaling Experiments Work 4: Multiprocessing

1. Repeat your analysis for correctness and initial request in the multiprocessing python script (`process_smoothing1.py`) and corresponding slurm script (`process_smoothing-proc1_file1.slurm`)
2. Use cp to copy the slurm multiprocess processing template to produce versions that will process 10, 40, 100, and 400 files using 1, 2, or 4 processes. Make sure that the slurm and python output files go in the `outputs/slurm` directory with unique names and the processed data goes in appropriate `outputs/processparallel` subdirectories.
3. Make a table of outputs for total memory, CPU Efficiency, read time, processing time, write time, and total time for each size of job

Scaling Experiments Work 5: Multiprocessing

1. How does memory usage grow with increasing numbers of images? How much memory would you need to request for 40000 images?
2. How does time grow with increasing numbers of images? How much time would you need to request for 40000 images using 2 processes?
3. How long might it take with 24 processes?

Serial Computation

A Serial computation performs one operation at a time until all of them are complete.

```
for n in range(1,401):
  infile="sample_"+str(n)+".nii.gz"
  outfile="denoised_"+str(n)+".nii.gz"
  outname="denoised_"+str(n)
  myimg = Image(infile)
  noisy_array = myimg.data
  denoised_array = gaussian_filter(noisy_array, 2)
  outImage = Image(denoised_arrray, name=outname)
  outImage.save(filename=outfile)
```

Shared Memory Parallelism

Shared memory parallelism makes use of multiple processors accessing the same memory (i.e. on the same node) to divide up work using threads [3]

```
import concurrent.futures as cf
#define parallel executable function
def single_denoise(data):
  outname, outfile, img=data
  output=gaussian_filter(img, 2)
  return (outname, outfile, output)

def main():
  img_list=serial_input()
  out_list=parallel_execution(img_list)
  serial_output(out_list)
```

gaussian_filter calls gaussian_filter1d calls NI_Correlate1D defined in ni_filters.c This makes use of NPY_BEGIN_THREADS to disable the GIL . You can use this to check which python functions (in NumPy and SciPy) are capable of multithreading.

Shared Memory Parallelism

continued from previous slide:

```
#read input serially
def serial_input():
  ilist=[]
  for n in range(1,401):
    infile="sample_"+str(n)+".nii.gz"
    outfile="denoised_"+str(n)+".nii.gz"
    outname="denoised_"+str(n)
    myimg = Image(infile)
    noisy = myimg.data
    ilist.append((outname,outfile,noisy))
  return ilist
```

Shared Memory Parallelism

continued from previous slide:

```
#parallel execution on multiple threads
def parallel_execution(ilist):
  output_list=[]
  with cf.ThreadPoolExecutor(8) as executor:
    for idata in executor.map(single_denoise, ilist):
      output_list.append(idata)
  return output_list

#Write output list serially
def serial_output(output_list):
  for img_data in output_list:
    outImage = Image(img_data[2], name=img_data[0])
    outImage.save(filename=img_data[1])

if __name__== '__main__':
  main()
```

Distributed Memory Parallelism

Multiprocessing parallelism makes use of multiple processors accessing different memory and namespaces to divide up work that may involve complex overlap and communication. Each of the 400 gaussian filters in this case work independently but cannot be dispatched to different nodes without using MPI (e.g. the mpi4py module in Python). [3]

```
import concurrent.futures as cf
#define parallel function
def single_image(n):
  infile="sample_"+str(n)+".nii.gz"
  outfile="denoised_"+str(n)+".nii.gz"
  outname="denoised_"+str(n)
  myimg = Image(infile)
  noisy = myimg.data
  output=gaussian_filter(noisy, 2)
  outImage = Image(output, name=outname)
  outImage.save(filename=outfile)
  return n
```

Distributed Memory Parallelism

continued from previous slide:

```
#parallel part
def parallel_execution():
  with cf.ProcessPoolExecutor(10) as executor:
    for n in executor.map(single_image, range(1,401)):
      print(str(n)+" is complete")

def main():
  parallel_execution()

if __name__== '__main__':
  main()
```

Some Mathematical Considerations

We will call the order 3 tensors i_1 through i_{400}. We need to compute the convolution with the gaussian distribution [10]

$$\mathscr{F}(g) = \widehat{g}(k,\sigma) = \exp\left(\frac{-k^2}{2\sigma^2}\right) \tag{1}$$

$$O_n = \mathscr{F}^{-1}(\widehat{g}(k,\sigma)\mathscr{F}(i_n)) = g * i_n \equiv G_\sigma(i_n) \tag{2}$$

for each n.

The FFT version of this convolution and the sum of products version of the convolution have different properties when implemented as parallel algorithms and suggest different acceleration strategies. This is only relevant if each image is large enough to warrant parallel processing. In our example, which has about 90,000 elements, a mediocre implementation on a single core finishes in less than 1 second and the main task decomposition is over images. For larger data sets, finer-grained parallelism needs to be contemplated. There are many good texts on parallel FFT and GPU algorithms, which can be used to inform this analysis.

[1] *Solving Problems on Concurrent Processors: General Techniques and Regular Problems*, volume 1. Prentice Hall, 1988.

[2] Multichannel pipettes - the easy way to increase productivity, August 2005.

[3] Python concurrent futures, March 2021.

[4] Scipy ndimage, February 2021.

[5] E. G. Coffman. *Computer and Job-Shop Scheduling Theory*. Wiley Interscience, 1976.

[6] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 3rd edition, 2017.

[7] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2005.

[8] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 5 edition, 2016.

[9] Yves Robert. Task graph scheduling. In David Padua, editor, *Encyclopedia of Parallel Computing*. Springer, 2011.

[10] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag, 2011.