

AI × HPC at OIST: Your Co-Pilot for Scientific Computing

David Soto - SCDA (Network Team)



OKINAWA INSTITUTE OF SCIENCE AND TECHNOLOGY GRADUATE UNIVERSITY

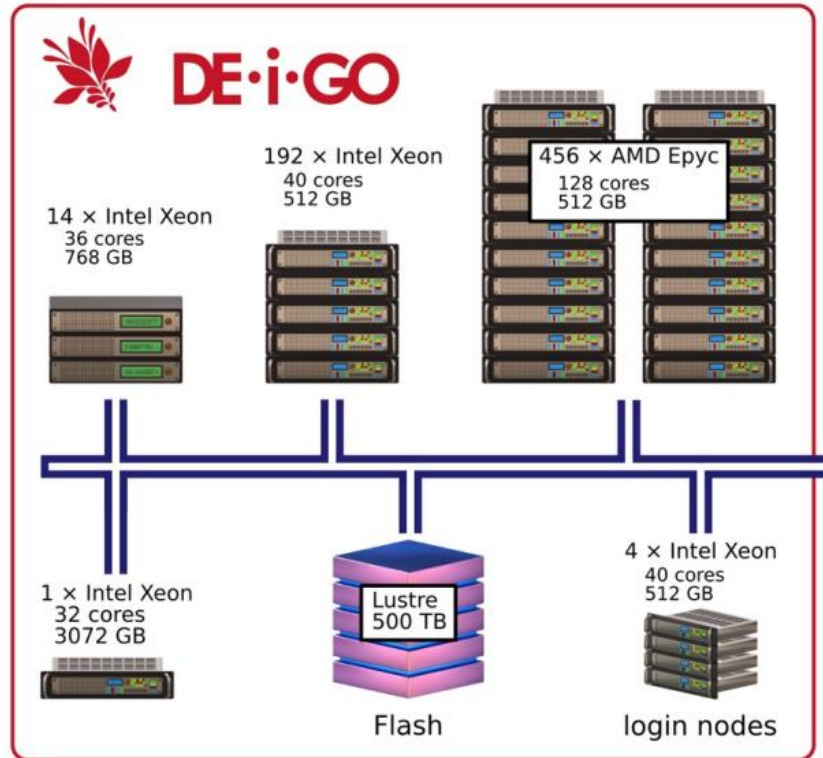
沖縄科学技術大学院大学



Agenda

- HPC at OIST
- Where AI fits in the HPC workflow
- AI as an HPC co-pilot
 - Case 1: Prompts - Ask AI instead of memorizing everything
 - Case 2: Choosing the correct resources
 - Case 3: Slurm script generator
 - Case 4: Debugging
 - Case 5: Profiling
 - Case 6: Data movement & Storage
 - Case 7: Experiment reproducibility
- Live Demo
- What you should NOT to do with AI and HPC

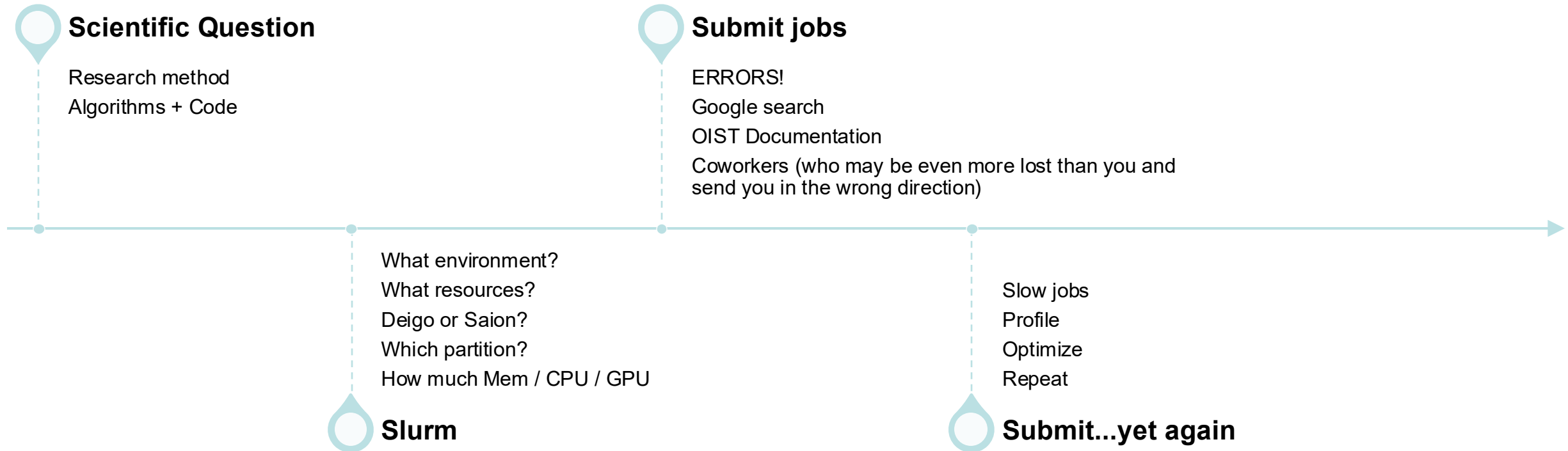
HPC at OIST



Deigo: around 66,000 CPU Cores
Saion: over 150 GPUs

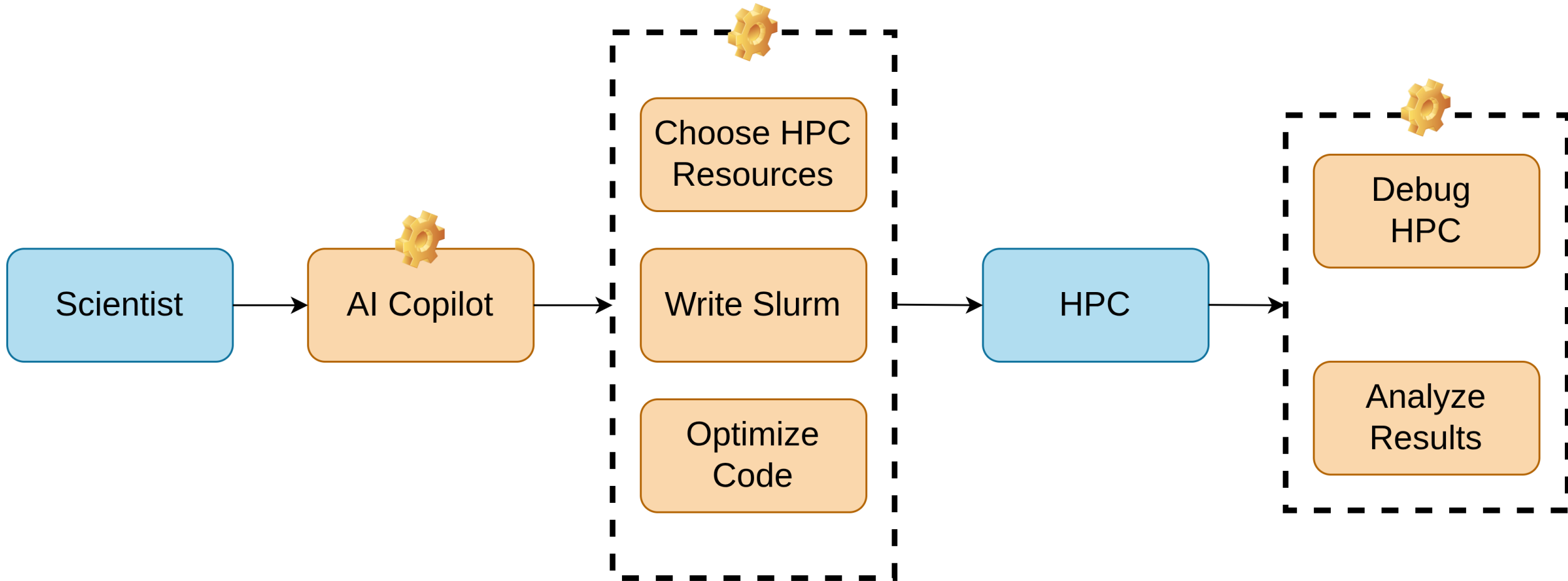
**The challenge for many researchers isn't necessarily access to compute.
It's knowing how to use that compute efficiently.**

Where AI fits in the HPC workflow



A scientist may spend very little time actually running the scientific calculation, but a surprising amount of time figuring out how to run it.

Where AI fits in the HPC workflow



AI can assist across the entire HPC workflow

AI as your HPC Co-Pilot

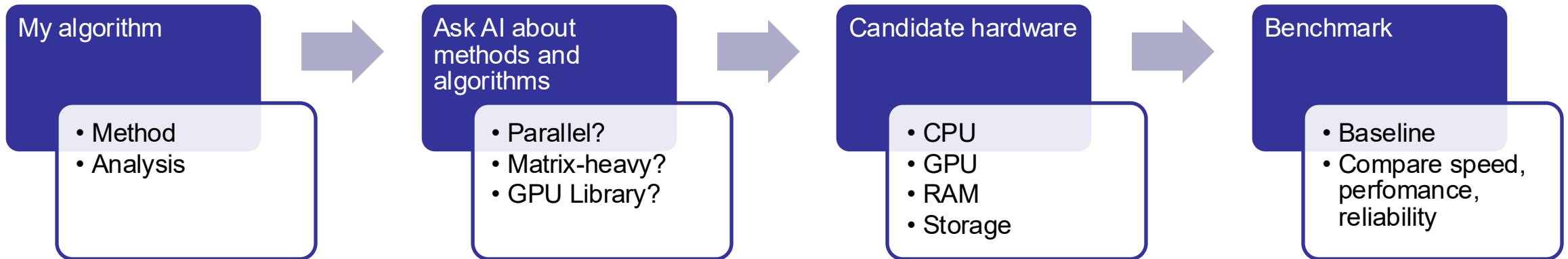
Case 1: Prompts - Ask AI instead of memorizing everything

- Example Prompts
 - "What is a Slurm partition?"
 - "What's the difference between srun and sbatch?"
 - "How do I request 16 CPUs and 64 GB RAM?"
 - "Should I use Deigo or Saion?"
 - "Explain this Slurm script line by line"
 - "What does --gres=gpu:1 mean?"

AI should explain the answer, not just give the command.

AI as your HPC Co-Pilot

Case 2: Choosing the correct resources Should I use Deigo (CPU) or Saion (GPU)



AI recommends. Benchmarking decides.

AI as your HPC Co-Pilot

Case 2: Choosing the correct resources

Should I use Deigo (CPU) or Saion (GPU)?

- **Ask AI: "I am using OIST HPC Deigo to work with a NumPy-based simulation with large matrix operations. Should I investigate Saion GPUs?"**

NumPy: library to simulate real-world systems, physics, or random events using fast array math.

Output from Copilot

Yes, it is worth investigating Saion, but whether it will actually speed up your NumPy simulation depends on how much of the workload can use GPUs.

At OIST, Deigo is the main CPU-oriented HPC cluster, while Saion is specifically designed for GPU-accelerated workloads and includes NVIDIA P100, V100, and A100 GPU nodes. oist +1

When Saion is likely a good idea

If your simulation spends most of its time in:

- Large dense matrix multiplication (`A @ B`)
- Linear algebra (`numpy.linalg`)
- FFTs
- Tensor operations
- Repeated array arithmetic on large arrays

then GPUs can provide substantial acceleration. Modern NVIDIA A100 GPUs on Saion were built precisely for these types of workloads. oist

For NumPy code, you would typically migrate to:

```
1 import cupy as cp
2
3 A = cp.random.rand(10000, 10000)
4 B = cp.random.rand(10000, 10000)
5 C = A @ B
```

CuPy is intentionally very NumPy-like, making many migrations relatively straightforward.

AI as your HPC Co-Pilot

Output from Copilot

Case 2: Choosing the correct resources

Should I use Deigo (CPU) or Saion (GPU)?

- **Ask AI: "I am using OIST HPC Deigo to work with a NumPy-based simulation with large matrix operations. Should I investigate Saion GPUs?"**

NumPy: library to simulate real-world systems, physics, or random events using fast array math.

When Saion may not help much

GPU acceleration is often disappointing if your code is:

- Python-loop heavy
- Branch-heavy or irregular
- Memory-access dominated rather than compute dominated
- Frequently transferring data between CPU and GPU
- Running matrices that are not very large

In those cases, Deigo's large CPU node count may remain the better platform. Deigo provides many high-core-count AMD EPYC nodes with substantial memory and a high-speed InfiniBand network.

oist

A practical approach

I would recommend:

1. Profile the code on Deigo.
2. Determine where the runtime is spent.
3. If >70-80% of the time is in NumPy BLAS/LAPACK operations, test a CuPy version on Saion.
4. Benchmark with a representative problem size.

The key question is whether your simulation is **compute-bound** (good GPU candidate) or **memory/algorithm-bound** (less benefit).

AI as your HPC Co-Pilot

Output from Copilot

Case 2: Choosing the correct resources

Should I use Deigo (CPU) or Saion (GPU)?

- Ask AI for clarifications:

Can you clarify how can I perform this validation?

Absolutely. What I meant was:

Before investing time porting your simulation to GPUs, determine whether most of the runtime is already being spent inside highly optimized NumPy operations that have GPU equivalents.

If 70-80% of the runtime is inside functions such as:

- `numpy.matmul`
- `numpy.dot`
- `numpy.einsum`
- `numpy.linalg.solve`
- `numpy.linalg.inv`
- `numpy.linalg.eig`
- FFT operations

then replacing NumPy with CuPy on Saion has a good chance of producing significant speedups.

"If > 70-80% of the time is in NumPy BLAS/LAPACK operations, test a CuPy version on Saion."

NumPy: library to simulate real-world systems, physics, or random events using fast array math.



AI as your HPC Co-Pilot

"If > 70-80% of the time is in NumPy BLAS/LAPACK operations, test a CuPy version on Saion."

Step 1: Profile the application

The goal is to answer:

Where does the simulation actually spend its time?

Using cProfile

Run:

```
1 python -m cProfile -o profile.out simulation.py
```

Then:

```
1 import pstats
2
3 p = pstats.Stats("profile.out")
4 p.sort_stats("cumtime")
5 p.print_stats(30)
```

You might see something like:

```
1 ncalls tottime cumtime function
2 10      12.3     5400    numpy.matmul
3 20      5.1      1800    numpy.linalg.solve
4 100     2.5      200     load_input
```

Interpretation:

```
1 matmul: 5400 s
2 solve : 1800 s
3 other  : 400 s
```

Total:

```
1 7600 s
```

BLAS/LAPACK-related work:

```
1 5400 + 1800 = 7200 s
2
3 7200 / 7600 = 94.7%
```

This is an excellent GPU candidate.



Output from Copilot

AI as your HPC Co-Pilot

"If > 70-80% of the time is in NumPy BLAS/LAPACK operations, test a CuPy version on Saion."

Step 2: Measure kernel-level timing

Sometimes the profile output is difficult to interpret.

Add timing around major operations:

```
1 from time import perf_counter
2
3 t0 = perf_counter()
4 A = B @ C
5 print("matmul:", perf_counter() - t0)
6
7 t0 = perf_counter()
8 x = np.linalg.solve(A, b)
9 print("solve:", perf_counter() - t0)
```

After one run you might get:

```
1 matmul: 4200 sec
2 solve: 1100 sec
3 file I/O: 300 sec
4 other: 200 sec
```

Total:

```
1 5800 sec
```

Linear algebra percentage:

```
1 (4200 + 1100)/5800
2
3 ≈ 91%
```

Again, an excellent GPU candidate.

AI as your HPC Co-Pilot

"If > 70-80% of the time is in NumPy BLAS/LAPACK operations, test a CuPy version on Saion."

Step 3: Identify if NumPy is using BLAS/LAPACK

Check:

```
1 import numpy as np
2
3 np.show_config()
```

You may see:

```
1 openblas
```

or

```
1 mkl
```

Step 4: Look for Python loops

Suppose profiling shows:

1 Python loops	65%
2 NumPy operations	20%
3 File I/O	15%

Example:

```
1 for i in range(10000000):
2     result += expensive_function(i)
```

In that case, simply moving from NumPy to CuPy may not help much because the bottleneck is Python itself.

The ideal profile looks more like:

1 NumPy BLAS/LAPACK	80%
2 Python logic	5%
3 I/O	15%

AI as your HPC Co-Pilot

"If > 70-80% of the time is in NumPy BLAS/LAPACK operations, test a CuPy version on Saion."

A practical validation workflow

For your OIST workflow, I would do:

On Deigo

1. Profile the simulation.
2. Calculate percentage of runtime spent in:
 - matmul
 - dot
 - einsum
 - linalg
 - FFT

Decision

- ```
1 < 30% -> stay CPU-focused
2 30-70% -> benchmark carefully
3 > 70% -> strong GPU candidate
```

### On Saion

1. Port only the top one or two kernels to CuPy.
2. Benchmark them.
3. Estimate total runtime reduction.

For example:

| Operation | Current Runtime |
|-----------|-----------------|
| matmul    | 75 min          |
| solve     | 25 min          |
| Other     | 20 min          |

Output from Copilot

# AI as your HPC Co-Pilot

## Case 2: Choosing the correct resources

Should I use Deigo (CPU) or Saion (GPU)?

- **Ask AI: "I am using OIST HPC Deigo to work with a NumPy-based simulation with large matrix operations. Should I investigate Saion GPUs?"**

**NumPy:** library to simulate real-world systems, physics, or random events using fast array math.

## Output from Copilot

### OIST-specific suggestion

Since OIST provides the open **test-gpu** partition on Saion for experimentation, you can often port a small piece of the simulation and benchmark it before requesting access to larger GPU resources.

The test partition includes NVIDIA P100 GPU nodes specifically for development and testing. oist +1

### Rule of thumb

| Workload                                         | Likely best system |
|--------------------------------------------------|--------------------|
| Pure NumPy matrix algebra on very large matrices | Saion GPU          |
| NumPy + heavy Python loops                       | Deigo CPU          |
| Very large memory requirements (> GPU memory)    | Deigo CPU          |
| CuPy, PyTorch, JAX, TensorFlow                   | Saion GPU          |
| MPI scaling across many CPU nodes                | Deigo CPU          |

If you can share:

- matrix dimensions,
- approximate memory usage,
- key NumPy operations (`matmul`, `einsum`, `solves`, `FFTs`, etc.),
- and whether you're already using MPI,

I can give a much more concrete estimate of the speedup you might expect from Saion.

# AI as your HPC Co-Pilot

## Case 2: Choosing the correct resources

Should I use Deigo (CPU) or Saion (GPU)?

- **Ask AI: "What would I need to benchmark before moving the workload from Deigo to Saion?"**

Before moving from **Deigo** → **Saion**, I'd benchmark enough to answer one question: **"Will the GPU speed up my actual simulation, not just one matrix multiplication?"**

Create a table like:

| Metric                         | Value |
|--------------------------------|-------|
| Total runtime                  |       |
| Matrix sizes                   |       |
| Peak RAM                       |       |
| % time in NumPy kernels        |       |
| % time in Python               |       |
| CPU utilization                |       |
| Scaling with threads           |       |
| Expected GPU memory needed     |       |
| Data transferred per iteration |       |

# AI as your HPC Co-Pilot

## Output from Copilot

### Case 2: Choosing the correct resources

Should I use Deigo (CPU) or Saion (GPU)?

- **Ask AI: "What would I need to benchmark before moving the workload from Deigo to Saion?"**

#### A practical decision rule

Move to Saion if:

- ✓ Most runtime is in matrix algebra (`matmul`, `einsum`, `linalg`, FFT)
- ✓ Working data fits in GPU memory
- ✓ Little CPU ↔ GPU transfer is required
- ✓ CPU scaling on Deigo has plateaued

Stay on Deigo if:

- ✓ Large-memory workload exceeds GPU memory
- ✓ Heavy Python/MPI orchestration dominates runtime
- ✓ Performance is limited by I/O or memory bandwidth rather than arithmetic throughput. oist +1

For a NumPy simulation, the single most useful benchmark is usually: **measure how much of the runtime is spent in `numpy.matmul`, `numpy.einsum`, and `numpy.linalg` operations.** If that number is above roughly 70%, testing a CuPy version on Saion is very likely worthwhile.

# AI as your HPC Co-Pilot

## Case 3: Slurm script generator

### • Scenario

- I have a Python simulation.
- Each simulation:
  - Takes approximately 2 hours.
  - Needs 8 CPU cores and 20 GB RAM.
- I need to run it for 500 parameter combinations.
- How should I run this efficiently on OIST Deigo?

### • Ask the AI (before generating the script)

- Explain what HPC strategy you recommend and why.
- Now generate the Slurm script.

## Expected AI reasoning

500 independent simulations



Embarrassingly Parallel



Slurm job array



One script + parameter values

**This teaches reasoning before code generation.**

# AI as your HPC Co-Pilot

## Recommended HPC strategy

### Workload characteristics

- 500 independent parameter combinations
- Each run:
  - 2 hours
  - 8 CPU cores
  - 20 GB RAM

These are what HPC users call **embarrassingly parallel** jobs because each simulation is independent and does not need to communicate with the others.

### Why a Slurm job array?

A job array allows Slurm to launch many copies of the same job, each with a different parameter set.

Benefits:

- Simple to manage
- Efficient scheduling
- Only one submission command
- Easy failure recovery
- Maximizes cluster throughput

Instead of:

```
1 simulation_1
2 simulation_2
3 ...
4 simulation_500
```

submitted manually, you submit:

```
1 sbatch --array=0-499 run_sim.sh
```

and Slurm handles the rest.

## Output from Copilot



# AI as your HPC Co-Pilot

## Resource estimate

Each simulation needs:

- 1 8 cores
- 2 20 GB RAM
- 3 2 hours

Total compute:

- 1  $500 \times 2$  hours
- 2 = 1000 CPU-hours of wall time
- 3
- 4 Using 8 cores each:
- 5  $500 \times 16$  core-hours
- 6 = 8000 core-hours

The actual elapsed time depends on how many array tasks run simultaneously.

For example:

## 20 concurrent jobs

- 1 20 simulations at once
- 2  $500 / 20 = 25$  waves
- 3
- 4  $25 \times 2$  h = 50 h

## 50 concurrent jobs

- 1 50 simulations at once
- 2  $500 / 50 = 10$  waves
- 3
- 4  $10 \times 2$  h = 20 h

## 100 concurrent jobs

- 1 100 simulations at once
- 2  $500 / 100 = 5$  waves
- 3
- 4  $5 \times 2$  h = 10 h

The appropriate concurrency depends on your allocation and current cluster load.

## Output from Copilot

# AI as your HPC Co-Pilot

## Organize parameter sets

Create a file:

```
1 params.txt
```

Example:

```
1 0.1 10
2 0.1 20
3 0.1 30
4 ...
```

with 500 lines.

Each array task reads one line.

## Slurm array script

```
1 #!/bin/bash
2 #SBATCH --job-name=sim500
3 #SBATCH --partition=compute
4 #SBATCH --time=02:30:00
5 #SBATCH --cpus-per-task=8
6 #SBATCH --mem=20G
7 #SBATCH --array=0-499%50
8 #SBATCH --output=logs/%A_%a.out
9 #SBATCH --error=logs/%A_%a.err
10
11 module purge
12
13 # Activate environment
14 source ~/venvs/myenv/bin/activate
15
16 # Get parameter set corresponding to this task
17 PARAMS=$(sed -n "$((SLURM_ARRAY_TASK_ID+1))p" params.txt)
18
19 # Run simulation
20 python simulation.py $PARAMS
```

## Output from Copilot

# AI as your HPC Co-Pilot

Example:

```
1 0.1 10
2 0.1 20
3 0.1 30
4 ...
```

with 500 lines.

Each array task reads one line.

## Slurm array script

```
1 #!/bin/bash
2 #SBATCH --job-name=sim500
3 #SBATCH --partition=compute
4 #SBATCH --time=02:30:00
5 #SBATCH --cpus-per-task=8
6 #SBATCH --mem=20G
7 #SBATCH --array=0-499%50
8 #SBATCH --output=logs/%A_%a.out
9 #SBATCH --error=logs/%A_%a.err
10
11 module purge
12
13 # Activate environment
14 source ~/venvs/myenv/bin/activate
15
16 # Get parameter set corresponding to this task
17 PARAMS=$(sed -n "${SLURM_ARRAY_TASK_ID+1}p" params.txt)
18
19 # Run simulation
20 python simulation.py $PARAMS
```

- **Does it run?** Yes.
- **Is it efficient?** Not so much.
  - Tells you to create a file with 500 parameter values. By hand....apparently. No thank you.
  - Better to create values programmatically in the code and send SLURM\_ARRAY\_TASK\_ID to the script directly.
  - Uses --array=0-499, but then has to do a bash calculation  $\$((SLURM\_ARRAY\_TASK\_ID+1))$ . Better to use array=1-500 and pass the task ID directly.
  - Not aware of working directories (/flash, /work). A beginner would run this from /home and fill up the space pretty quickly.

**You need to understand what AI tells you, and how it actually applies to your situation, or you may be doing things subtly wrong or less optimal.**

# AI as your HPC Co-Pilot

## Case 4: Debugging

- **Your job failed. Ask AI why.**

*\*Fictional error\**

*slurmstepd: error:  
Detected 1 oom-kill event(s)  
in StepId=123456.batch*

*Some of your processes may have been killed  
by the cgroup out-of-memory handler.*



# AI as your HPC Co-Pilot

## Case 4: Debugging

- **Ask AI:**
  - “Explain this error in simple terms. What should I inspect before increasing my memory request?”
  - “What commands or measurements would help me determine whether the problem is genuinely insufficient memory?”

**Error -> AI Interpretation -> Hypotheses -> Measure -> Diagnose -> Fix**

# AI as your HPC Co-Pilot

This error means your job used more memory than Slurm allocated to it, and the Linux kernel's Out-Of-Memory (OOM) protection terminated one or more processes to protect the node.

## What the message means

```
1 slurmstepd: error:
2 Detected 1 oom-kill event(s)
3 in StepId=123456.batch
4
5 Some of your processes may have been killed
6 by the cgroup out-of-memory handler.
```

Breaking it down:

- **oom-kill event(s)**: The operating system detected that your job exceeded its memory limit.
- **cgroup out-of-memory handler**: Slurm places jobs inside a memory-controlled cgroup. When the job exceeds its assigned memory, the kernel kills processes within that job.
- **Detected 1 oom-kill event(s)**: At least one process in your batch job was terminated because it ran out of memory.

## Why it happened

A common scenario is:

```
1 #SBATCH --cpus-per-task=8
2 #SBATCH --mem=20G
```

but during execution the program actually used:

```
1 23 GB
2 25 GB
3 40 GB
```

of RAM.

Once memory usage exceeded the 20 GB limit, the OOM handler intervened and killed one or more processes.

Possible causes include:

- Reading a larger dataset than expected.
- Creating very large NumPy arrays.
- Memory leaks.
- Multiple worker processes each allocating significant memory.
- Temporary copies of arrays during matrix operations.
- Underestimating memory requirements.

## Output from Copilot

# AI as your HPC Co-Pilot

## How to confirm

Check the job statistics:

```
1 sacct -j 123456 --format=JobID,State,Elapsed,MaxRSS,ReqMem
```

Example:

```
1 JobID State Elapsed MaxRSS ReqMem
2 123456 FAILED 01:45:10 24G 20G
```

This would clearly show:

```
1 Requested memory: 20 GB
2 Peak memory: 24 GB
```

You can also get detailed information:

```
1 seff 123456
```

Example:

```
1 Memory Utilized: 24.1 GB
2 Memory Efficiency: 120%
```

**Tools suggestions are correct,  
but reporting presentation is not.  
\*AI is not aware of our internal settings\***

**Output from Copilot**

# AI as your HPC Co-Pilot

## Real output of sacct and seff from Deigo

```
[david-soto@deigo-login2 Demo]$ sacct -j 45526940
```

| JobID        | JobName    | Partition | Account   | AllocCPUS | State     | ExitCode |
|--------------|------------|-----------|-----------|-----------|-----------|----------|
| 45526940_0   | python-ar+ | compute   | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_0.+ | batch      |           | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_0.+ | extern     |           | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_1   | python-ar+ | compute   | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_1.+ | batch      |           | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_1.+ | extern     |           | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_2   | python-ar+ | compute   | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_2.+ | batch      |           | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_2.+ | extern     |           | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_3   | python-ar+ | compute   | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_3.+ | batch      |           | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_3.+ | extern     |           | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_4   | python-ar+ | compute   | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_4.+ | batch      |           | scicomsec | 8         | COMPLETED | 0:0      |
| 45526940_4.+ | extern     |           | scicomsec | 8         | COMPLETED | 0:0      |

```
[david-soto@deigo-login2 Demo]$ seff 45526940
```

Job ID: 45526940  
Array Job ID: 45526940\_99  
Cluster: deigo  
User/Group: david-soto/scicomsec  
State: COMPLETED (exit code 0)  
Nodes: 1  
Cores per node: 8  
CPU Utilized: 00:00:07  
CPU Efficiency: 29.17% of 00:00:24 core-walltime  
Job Wall-clock time: 00:00:03  
Memory Utilized: 1.23 MB  
Memory Efficiency: 0.01% of 8.00 GB (8.00 GB/node)

# AI as your HPC Co-Pilot

## How to fix it

### Option 1: Request more memory

If the job genuinely needs more memory:

```
1 #SBATCH --mem=32G
```

or

```
1 #SBATCH --mem=40G
```

### Option 2: Reduce memory usage

Examples:

- Process data in chunks.
- Avoid unnecessary array copies.
- Use `float32` instead of `float64` when appropriate.
- Free large objects when no longer needed.

```
1 del large_array
```

### Option 3: Check parallel workers

Sometimes memory scales with worker count.

For example:

```
1 8 workers × 4 GB each
2 = 32 GB
```

even though the main process appears small.

## Output from Copilot

# AI as your HPC Co-Pilot

## Case 5: Profiling

- Assume we have a job with heavy I/O file workload:
  - Multi-dimensional grid datasets (NetCDF, HDF5HDF5)
  - Large-scale checkpointing (dumps gigabytes of memory state to disk so the job can restart if it fails).
  - DNA/RNA sequencing (massive amounts of reads/writes against DBs)
  - Deep Learning / AI Training (large-scale transformers or computer vision models)

# AI as your HPC Co-Pilot

## Case 5: Profiling

- **Fictional Job profile:**
  - CPU utilization 24%
  - Memory utilization 38%
  - I/O wait 41%
  - Walltime 03:12:15
  - Requested CPUs 64

- **Ask AI:**

Interpret this profile

Hypotheses for the low CPU

Suggest an experiment to distinguish them

### Interpretation of the profile

Your metrics suggest that the job is **not CPU-bound**.

| Metric             | Value  | Interpretation                                                     |
|--------------------|--------|--------------------------------------------------------------------|
| CPU Utilization    | 24%    | Most CPU cores are idle much of the time                           |
| Memory Utilization | 38%    | Memory capacity is not limiting performance                        |
| I/O Wait           | 41%    | Processes spend a significant fraction of time waiting for storage |
| Requested CPUs     | 64     | The application is not effectively using all allocated cores       |
| Walltime           | 3h 12m | Runtime is being extended by waiting rather than computation       |

**Performance Evidence**

# AI as your HPC Co-Pilot

The most striking metric is:

```
1 I/O wait = 41%
```

This means CPUs are frequently stalled waiting for data to arrive from storage rather than performing computations.

A rough interpretation is:

```
1 24% computing
2 41% waiting for I/O
3 35% other states
4 ``
```

So requesting 64 CPUs does not appear to be providing proportional benefit at the moment.

## Hypothesis 1: Storage I/O bottleneck

The application may be:

- Reading many small files
- Continuously loading datasets
- Writing large output files
- Frequently checkpointing

In this scenario:

```
1 CPU -> request data
2 CPU -> waits
3 CPU -> waits
4 CPU -> waits
5 CPU -> resumes
6 ``
```

Symptoms:

- High %wa (I/O wait)
- Low CPU utilization
- Increasing CPUs does not improve runtime

## Output from Copilot

# AI as your HPC Co-Pilot

## Hypothesis 2: Parallel scaling is poor

You requested:

```
1 64 CPUs
```

but the code may only effectively use a small fraction of them.

Examples:

- Serial sections of code
- Lock contention
- Synchronization overhead
- MPI processes waiting for each other
- OpenMP threads sitting idle

If the application only uses ~15 effective cores:

```
1 15 / 64 ≈ 23%
```

which is remarkably close to your observed CPU utilization.

## Hypothesis 3: Workload is memory-latency or bandwidth limited

Even though total memory utilization is only 38%, the code may spend time waiting on memory accesses.

Typical cases:

- Large sparse matrices
- Random access patterns
- Graph algorithms
- Irregular NumPy indexing

Symptoms:

- CPUs appear underutilized
- Little benefit from additional threads
- Runtime dominated by data movement rather than arithmetic

## Output from Copilot

# AI as your HPC Co-Pilot

## Hypothesis 4: Too many CPUs requested

Some applications perform worse at high core counts.

For example:

```
1 8 cores -> 4 hours
2 16 cores -> 2.5 hours
3 32 cores -> 2.0 hours
4 64 cores -> 1.9 hours
```

After a certain point, additional cores mostly wait for work.

In this case:

- Scheduler gives 64 CPUs
- Application can only use 20-30 efficiently
- CPU utilization drops

## Hypothesis 5: MPI imbalance

If this is an MPI job, some ranks may be doing much more work than others.

A typical pattern:

```
1 Rank 0: busy
2 Rank 1: busy
3 Rank 2: waiting
4 ...
5 Rank 63: waiting
```

Overall CPU utilization looks low because many ranks spend time waiting at communication barriers.

## Output from Copilot

# AI as your HPC Co-Pilot

- **Experiments to distinguish the different Hypotheses:**
  1. Run from fast scratch storage
  2. CPU scaling study
  3. Monitor I/O throughput (e.g. `iostat -x 5` or High Sci).
  4. Profile CPU usage per thread or process

## Most likely explanation

Given the profile:

```
1 CPU Utilization: 24%
2 Memory Utilization: 38%
3 I/O Wait: 41%
4 Requested CPUs: 64
```

my leading hypothesis is:

The job is primarily I/O-bound and secondarily overprovisioned on CPU cores.

The fastest way to verify this is:

1. Run the same workload from high-performance scratch storage.
2. Repeat the run with 16, 32, and 64 CPUs.
3. Compare walltimes.

Those two experiments usually reveal very quickly whether the bottleneck is storage access or poor parallel scaling.

## Output from Copilot

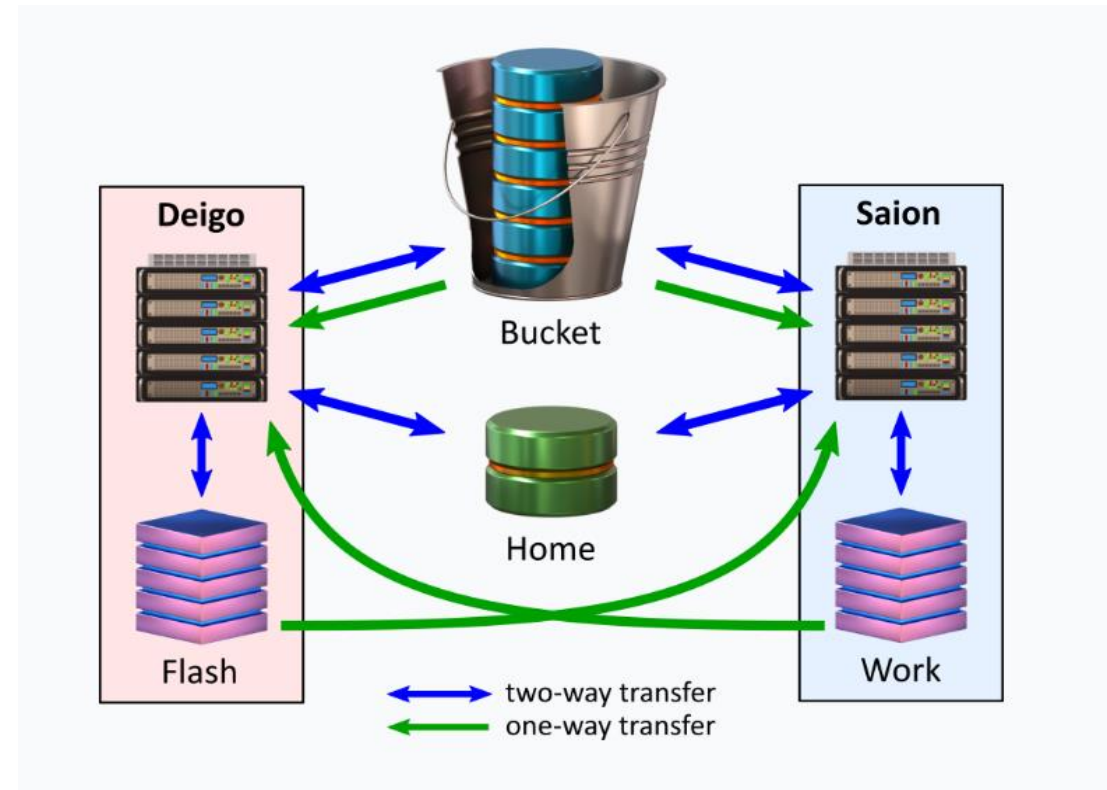
# AI as your HPC Co-Pilot

## Case 6: Data movement & Storage

- An HPC job isn't just CPU and GPU.
- Bucket vs Flash vs Work
- Research data on Bucket while using cluster local storage for reading/writing
- Workflows that move CPU-bound work to Deigo and GPU-bound work to Saion

## Ask AI

“Given this workflow, explain where I should place my input data, temporary files, and final results, and why.”



**Data movement can dominate the runtime**

# AI as your HPC Co-Pilot

## Case 7: AI for experiment reproducibility

- Researchers often remember the scientific result but forget exactly how they generated it.
- AI can help capture the computational context.

### Ask AI:

- “Turn this Slurm script, parameter file and experiment description into a reproducibility record.”

### AI is not the source of truth.

The actual **job configuration**, **code version**, **environment** and **data** remain the source of truth.

**AI can turn an HPC experiment into a reproducible record**

```
</> YAML

experiment:
 code_version: abc123

system: Deigo
partition: compute

resources:
 cpus: 32
 memory: 128G
 time: 04:00:00

software:
 python: ...
 gcc: ...

parameters:
 temperature: 300
 pressure: 1.0

input:
 dataset: experiment_2026_08
```

# Live Demo

- Scenario
  - “I have 100 independent Python simulations. Each needs 8 CPU cores and about 30 minutes. I want to run them on Deigo.”
- Ask AI
  - “What HPC strategy should I use?”
  - “Explain why a job array may be appropriate.”
  - “Create an OIST Deigo Slurm script for this workflow. Use placeholders where OIST-specific values need verification.”
  - “Review your own script. What could be wrong or inefficient?”
  - Compare the generated script with the actual OIST documentation.

# Live Demo

```
#!/bin/bash
#SBATCH --job-name=python-array-demo
#SBATCH --array=0-99
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --mem=8G
#SBATCH --time=01:00:00

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export MKL_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export OPENBLAS_NUM_THREADS=${SLURM_CPUS_PER_TASK}

python simulation.py --case ${SLURM_ARRAY_TASK_ID}
```

- **Does it run by default?** No
  - *sbatch: error: Batch job submission failed: No partition specified or system default partition.*
- **Does it mentioned anything about working directories?** No

# Live Demo

```
#!/bin/bash
#SBATCH --job-name=python-array-demo
#SBATCH --array=0-99
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --mem=8G
#SBATCH --time=01:00:00

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export MKL_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export OPENBLAS_NUM_THREADS=${SLURM_CPUS_PER_TASK}

python simulation.py --case ${SLURM_ARRAY_TASK_ID}
```

- **Adding partition=compute**
  - *Does it run now? Yes*
  - *Any notifications or outputs? No*
  - *Manually check with sacc or seff*
  - *Resources were used efficiently?*

```
[david-soto@deigo-login3 Demo]$ seff 45670133
Job ID: 45670133
Array Job ID: 45670133_99
Cluster: deigo
User/Group: david-soto/scicomsec
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 8
CPU Utilized: 00:00:05
CPU Efficiency: 12.50% of 00:00:40 core-walltime
Job Wall-clock time: 00:00:05
Memory Utilized: 1.28 MB
Memory Efficiency: 0.02% of 8.00 GB (8.00 GB/node)
```

# Live Demo

- **Did it use the working directories correctly? No**
  - Script ran in the Compute nodes, but all outputs/results were stored in the Login node.

```
[david-soto@deigo-login3 Demo]$ ls
result_0.txt result_30.txt result_51.txt result_72.txt result_93.txt slurm-45670133_20.out slurm-45670133_41.out slurm-45670133_62.out slurm-45670133_83.out
result_10.txt result_31.txt result_52.txt result_73.txt result_94.txt slurm-45670133_21.out slurm-45670133_42.out slurm-45670133_63.out slurm-45670133_84.out
result_11.txt result_32.txt result_53.txt result_74.txt result_95.txt slurm-45670133_22.out slurm-45670133_43.out slurm-45670133_64.out slurm-45670133_85.out
result_12.txt result_33.txt result_54.txt result_75.txt result_96.txt slurm-45670133_23.out slurm-45670133_44.out slurm-45670133_65.out slurm-45670133_86.out
result_13.txt result_34.txt result_55.txt result_76.txt result_97.txt slurm-45670133_24.out slurm-45670133_45.out slurm-45670133_66.out slurm-45670133_87.out
result_14.txt result_35.txt result_56.txt result_77.txt result_98.txt slurm-45670133_25.out slurm-45670133_46.out slurm-45670133_67.out slurm-45670133_88.out
result_15.txt result_36.txt result_57.txt result_78.txt result_99.txt slurm-45670133_26.out slurm-45670133_47.out slurm-45670133_68.out slurm-45670133_89.out
result_16.txt result_37.txt result_58.txt result_79.txt result_9.txt slurm-45670133_27.out slurm-45670133_48.out slurm-45670133_69.out slurm-45670133_8.out
result_17.txt result_38.txt result_59.txt result_7.txt simulation.py slurm-45670133_28.out slurm-45670133_49.out slurm-45670133_6.out slurm-45670133_90.out
result_18.txt result_39.txt result_5.txt result_80.txt simulation_revised.slurm slurm-45670133_29.out slurm-45670133_4.out slurm-45670133_70.out slurm-45670133_91.out
result_19.txt result_3.txt result_60.txt result_81.txt simulation.slurm slurm-45670133_2.out slurm-45670133_50.out slurm-45670133_71.out slurm-45670133_92.out
result_1.txt result_40.txt result_61.txt result_82.txt slurm-45670133_0.out slurm-45670133_30.out slurm-45670133_51.out slurm-45670133_72.out slurm-45670133_93.out
result_20.txt result_41.txt result_62.txt result_83.txt slurm-45670133_10.out slurm-45670133_31.out slurm-45670133_52.out slurm-45670133_73.out slurm-45670133_94.out
result_21.txt result_42.txt result_63.txt result_84.txt slurm-45670133_11.out slurm-45670133_32.out slurm-45670133_53.out slurm-45670133_74.out slurm-45670133_95.out
result_22.txt result_43.txt result_64.txt result_85.txt slurm-45670133_12.out slurm-45670133_33.out slurm-45670133_54.out slurm-45670133_75.out slurm-45670133_96.out
result_23.txt result_44.txt result_65.txt result_86.txt slurm-45670133_13.out slurm-45670133_34.out slurm-45670133_55.out slurm-45670133_76.out slurm-45670133_97.out
result_24.txt result_45.txt result_66.txt result_87.txt slurm-45670133_14.out slurm-45670133_35.out slurm-45670133_56.out slurm-45670133_77.out slurm-45670133_98.out
result_25.txt result_46.txt result_67.txt result_88.txt slurm-45670133_15.out slurm-45670133_36.out slurm-45670133_57.out slurm-45670133_78.out slurm-45670133_99.out
result_26.txt result_47.txt result_68.txt result_89.txt slurm-45670133_16.out slurm-45670133_37.out slurm-45670133_58.out slurm-45670133_79.out slurm-45670133_9.out
result_27.txt result_48.txt result_69.txt result_8.txt slurm-45670133_17.out slurm-45670133_38.out slurm-45670133_59.out slurm-45670133_7.out
result_28.txt result_49.txt result_6.txt result_90.txt slurm-45670133_18.out slurm-45670133_39.out slurm-45670133_5.out slurm-45670133_80.out
result_29.txt result_4.txt result_70.txt result_91.txt slurm-45670133_19.out slurm-45670133_3.out slurm-45670133_60.out slurm-45670133_81.out
result_2.txt result_50.txt result_71.txt result_92.txt slurm-45670133_1.out slurm-45670133_40.out slurm-45670133_61.out slurm-45670133_82.out
[david-soto@deigo-login3 Demo]$
```

# Live Demo

- **Some simple improvements**

- Reduce CPU and RAM to half to test performance (CPU 8 -> 4) (RAM 8GB -> 4GB)
- Reduce requested time (1:00:00 -> 1:00)
- Create temporary work directories in the slurm script following SCDA documentation.  
tempdir=\$(mktemp -d /flash/MyunitU/myprog.XXXX)  
cd \$tempdir  
#Run script from here  
#Then copy results from flash to bucket and remove tempdir  
scp output.txt deigo:/bucket/MyunitU/  
rm -r \$tempdir
- If you want, add notifications for your job submission, ending, and in case of failure  
#SBATCH --mail-type=BEGIN,FAIL,END  
#SBATCH [--mail-user=mail@oist.jp](mailto:mail@oist.jp)

# What you should **NOT** do with AI and HCP

## 1. Trust AI generated Slurm scripts blindly

AI can produce syntactically correct scripts that are inappropriate for your workload:

- Waste cluster resources
- Increase queue time
- Violate allocation policies

**Always verify your actual requirements (CPU, RAM, Storage)**

# What you should **NOT** do with AI and HCP

## 2. Let AI estimate resources without measurements

"AI says I need 64 cores and 256GB RAM to run this"

Instead:

- Run smaller tests.
- Measure actual CPU and RAM utilization.
- Use profiling tools to calculate resource utilization from previous jobs (seffs, sacct).

**Resource requests should come from data, not guesses**

# What you should **NOT** do with AI and HCP

## 3. Submit hundreds of jobs generated by AI without checking

Sometimes AI suggests things like:

- Flood the scheduler and
- Create unnecessary load.
- Frustrate other users.

```
1 for i in {1..10000}
2 do
3 sbatch run.sh
4 done
```

**This way of submitting jobs is actually disallowed within the system, and will most likely fail due to enforced limits**



**Slurm arrays + Throttling > Massive independent jobs**

# What you should **NOT** do with AI and HCP

## 4. Assume AI understands your bottleneck

AI may suggest: "Use GPU"

When your job is actually:

- I/O bound.
- Memory bound.
- Serial

**In HPC, the most expensive words are often: "The AI said it would be faster." Always verify with an experiment.**

# What you should **NOT** do with AI and HCP

## 5. Treat AI as a replacement for official HPC documentation

AI is useful for:

- Explaining concepts.
- Drafting scripts.
- Suggesting tests.

Cluster specific details should come from internal documentation.

**Always follow internal policies first**

# What you should **NOT** do with AI and HCP

## **6. Run AI agents or tools directly on Login Nodes**

Login nodes are not intended for heavy computation. Use them for editing files, compiling code, and submitting/monitoring jobs.

**SDCA policies disallow running lots of processes (AI Agents) or functions that sweep through the entire filesystem.**

**Use login nodes for preparation and job submission.  
Use compute nodes for computation.**

# What you should **NOT** do with AI and HCP

**7. And you probably have heard this one a few times already:  
Do NOT put confidential research data into public AI systems**

Be very careful with:

- Unpublished results.
- Proprietary algorithms.
- Experiments datasets.
- Confidential information.

**Make sure your AI use complies with OIST's AI policies:  
<https://www.oist.jp/education/policies-regulations/gs-policies>**

# Q & A