Approach

According to the project requirements, it is clear that we need to compile and load the OSU Micro-Benchmark in three different ways, and then benchmark it on four system architectures across two different clusters.

Fetching the binary files is not particularly difficult, but using the ReFrame framework to handle everything from pulling source code, to compilation, to execution, and finally report generation is an interesting challenge.

Of course, we could split this into two steps: compile first, then hard-code the compiled binary paths into the ReFrame test. However, this approach lacks elegance.

Since benchmarking involves different code sources, different system topology parameters, different clusters, and different performance metrics, we explored ReFrame's documentation and eventually arrived at a clean and extensible solution for injecting these parameters:

Different code sources	Use fixtures to automate fetching and compiling from source
Different topology	Inject custom topology parameters using custom-defined `env_vars`
Different clusters	Configure cluster-specific logic via ReFrame's system configuration
Different metrics	Split different metrics into separate test files

As a result, the final solution automatically handles the cluster-specific logic within the code. We only need to run a single command to execute the full benchmark on both Iris and Aion clusters.

Methodology

All tests are based on OSU Micro-Benchmarks, specifically:

- `osu_latency` for measuring latency, with a message size of 8192 Bytes
- `osu_bw` for measuring bndwidth, with a message size of 1MB (1048576 Bytes)

Each test uses **2 MPI processes**, with the **CPU binding strategy automatically configured** based on the system architecture:

For example, `map_cpu:0,1` corresponds to the `intra_numa` placement.

Specifically, we define the placement info in `env_vars` of config file for different systems(Aion and Iris).

The ReFrame framework is used to standardize the testing workflow, including module loading and launcher options (e.g., `srun`). Each test defines performance reference values and tolerance thresholds (e.g., ±10%) to evaluate whether the results meet expectations.

Finally, all test results are collected and visualized using **bar charts and heatmaps** for performance analysis.

Results

The detailed results of the final performance report can be found in the accompanying performance report file.

All tests ran smoothly on the Aion cluster. However, on the Iris cluster, the EasyBuild version of the binary occasionally exhibits instability：：

```
--- rfm_job.out ------ rfm_job.err (last 10 lines) ---
 4 0x000000000040440e _start()  ???:0
==================================
==== backtrace (tid:1240770) ====
```

```
 0 0x0000000000012d10 __funlockfile()  :0
 1 0x000000000040cf3b omb_ddt_assign()  ???:0
 2 0x00000000004037ac main()  ???:0
 3 0x000000000003a7e5 __libc_start_main()  ???:0
 4 0x000000000040440e _start()  ???:0
==================================
srun: error: iris-070: tasks 0-1: Illegal instruction (core dumped)
--- rfm_job.err ---
```

Key takeaways

Platform Recommendations:

- Aion is better suited for **low-latency computing** tasks (lower latency in intra- and cross-NUMA scenarios).
- Iris is better suited for **high-bandwidth communication** workloads (higher inter-node bandwidth observed).

Binary Selection Recommendations:

- Generic and EasyBuild*binaries show balanced and stable performance across scenarios.
- EESSI binaries may present compatibility issues, although they perform well in certain placement cases.

Insights for Benchmark Design:

- Benchmarks should thoroughly account for system **topology** (NUMA domains, socket layout, and inter-node communication).
- Compatibility of high-performance libraries must not be overlooked; for cross-platform deployment, locally built or EasyBuild-based binaries are preferred.