OSU Micro-Benchmark (OMB) Latency/Bandwidth Evaluation using ReFrame in UniLu HPC

Fabio, Shahmeer, Wei

MHPC

2025-05-22

Outline

Outline

1. 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.

1. Approach

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.

1. Approach

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.

1. Approach

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 scripts

1. Approach

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.

Outline

2. Methodology

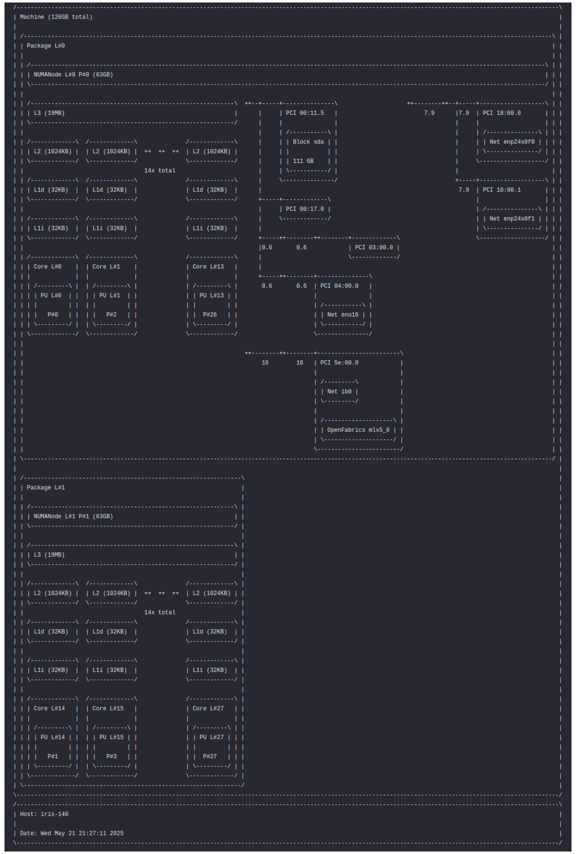

Figure 1: System topology of Iris cluster

2. Methodology

Figure 2: System topology of Aion cluster

2. 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 bandwidth, with a message size of 1MB (1048576 Bytes)

2. Methodology

Metrics	Description
generic	Compiled from source
easybuild	Compiled with EasyBuild
eessi	Binaries loaded from the EESSI distribution

2. Methodology

Metrics	Iris CPU binding strategy	Aion CPU binding strategy	Description
intra_numa	`map_cpu:0,1`	`map_cpu:0,1`	both processes are running on the same NUMA node
cross_numa	-	`map_cpu:0,16`	both processes are running on the same physical socket but different NUMA nodes
cross_socket	`map_cpu:0,14`	`map_cpu:0,64`	both processes are running on the same compute node but different sockets

2. Methodology

inter_node	`--cpu-bind=cores`	`--cpu-bind=cores`	the 2 processes are running on different nodes

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.

2. Methodology

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

```json
// Aion's env_vars
"env_vars": [
      ["intra_numa", "map_cpu:0,1"],
      ["cross_numa", "map_cpu:0,16"],
      ["cross_socket", "map_cpu:0,64"],
],
"env_vars": [
      ["intra_numa", "map_cpu:0,1"],
      ["cross_socket", "map_cpu:0,14"],
]
```

Outline

3. Results

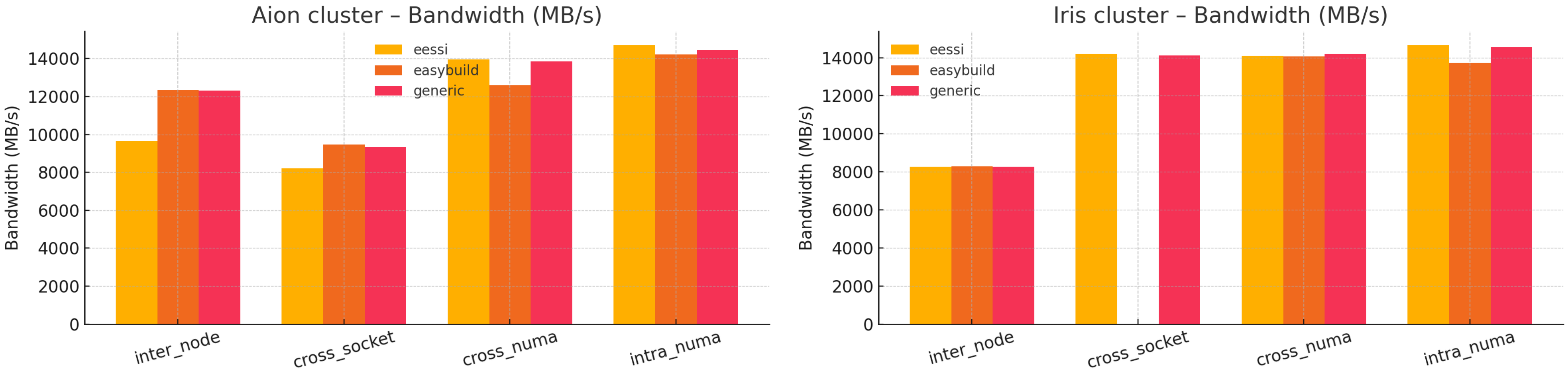

Figure 3: Bandwidth result in Aion & Iris cluster.

3. Results

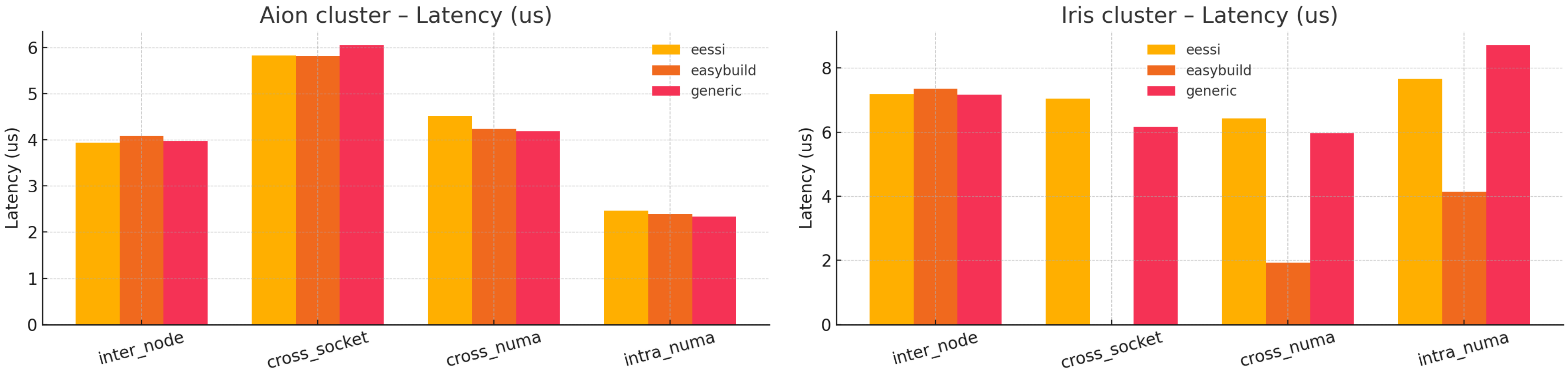

Figure 4: Latency result in Aion & Iris cluster.

3. Results

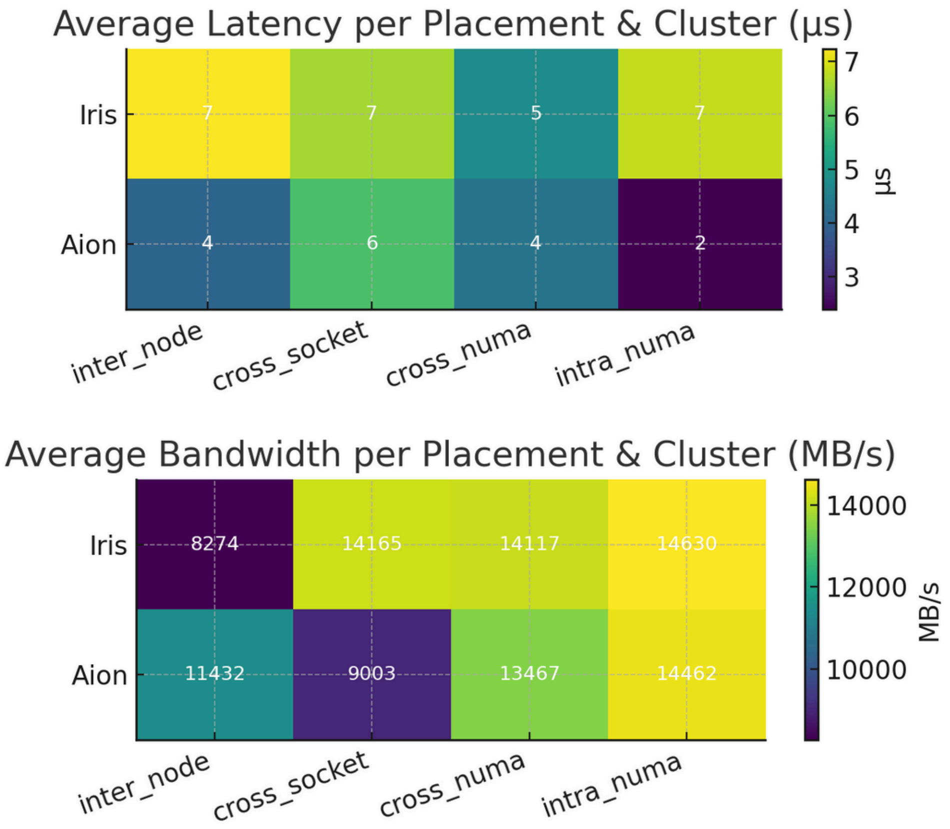

Figure 5: Average Bandwidth & Latency

3. Results

From Figure 3 and Figure 4 we can observation:

1. Iris excels in local bandwidth (especially for cross_socket/cross_numa).
2. Aion shows lower latency, likely due to faster CPU scheduling or better NUMA balancing.

Figure 5 shows that for latency:

1. Iris: Averages 7 μs across most placements.
2. Aion: Significantly faster—down to 2.3 μs in intra_numa.

Also for bandwidth:

1. Iris dominates across all placements, with cross_numa/cross_socket 14.1 GB/s+
2. Aion achieves similar intra_numa bandwidth, but underperforms in inter_node (11.4 GB/s)

3. Results

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

```sh
--- 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
=================================
```

3. Results

Some Clues:

- The error only occurs occasionally.
- The problem only affects the EasyBuild binary.
- `omb_ddt_assign()` is where the crash occurs: this function may use data-dependent logic or SIMD instructions.

Reasonable Explanation:

If the binary was compiled on a node with newer instruction sets, but later executed on an older Iris node without such capabilities, this would result in a SIGILL (illegal instruction) error like the one shown.

Outline

4.1 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).

4.2 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.

4.3 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.