PS High-Performance Computing 2024/2025
Assignment 1

Stefan Wagner

October 6, 2024

The goal of this assignment is to get you acquainted with working on a distributed memory cluster as well as obtaining, illustrating, and interpreting measurement data.

Exercise 1 (1 Point)

Description

This exercise consists in familiarizing yourself with SLURM job submission.

You received user credentials for the LCC3 cluster. If you did not change the default password, do so **immediately**. You are responsible for this account during this semester.

You can find information about LCC3 at https://www.uibk.ac.at/zid/systeme/hpc-systeme/leo3/ and information about SLURM job submission at https://www.uibk.ac.at/zid/systeme/hpc-systeme/common/tutorials/slurm-tutorial.html.

Please run any benchmarks or heavy CPU loads only on the compute nodes, not on the login node. If you want to do some interactive experimentation, use an interactive job as outlined in the tutorial. Make sure to stop any interactive jobs once you are done.

Tasks

- Study how to submit jobs in SLURM, how to check their state and how to cancel them.

  In order to start a SLURM job the following command is used:

```
sbatch [options] [job_script.slurm [ job_script_arguments ...]]
```

  To view the state of the SLURM job the following commands can be used:

```
suq
squeue
squeue -u $USER
```

  To cancel a SLURM job the following command can be used:

```
scancel <JOBID>
```

  As shown in Figure 1 the command `sbatch job.slurm` starts the corresponding `job.slurm` file which is also depicted in the figure. The commands `squ` and `squeue -u $USER` list all jobs of the current user. However the output of the two commands differs slightly. Additionally, the command `squeue` lists the jobs from all users. As can be seen in Figure 1 the command `scancel 123707` cancels the currently running job with the JOBID == 123707.

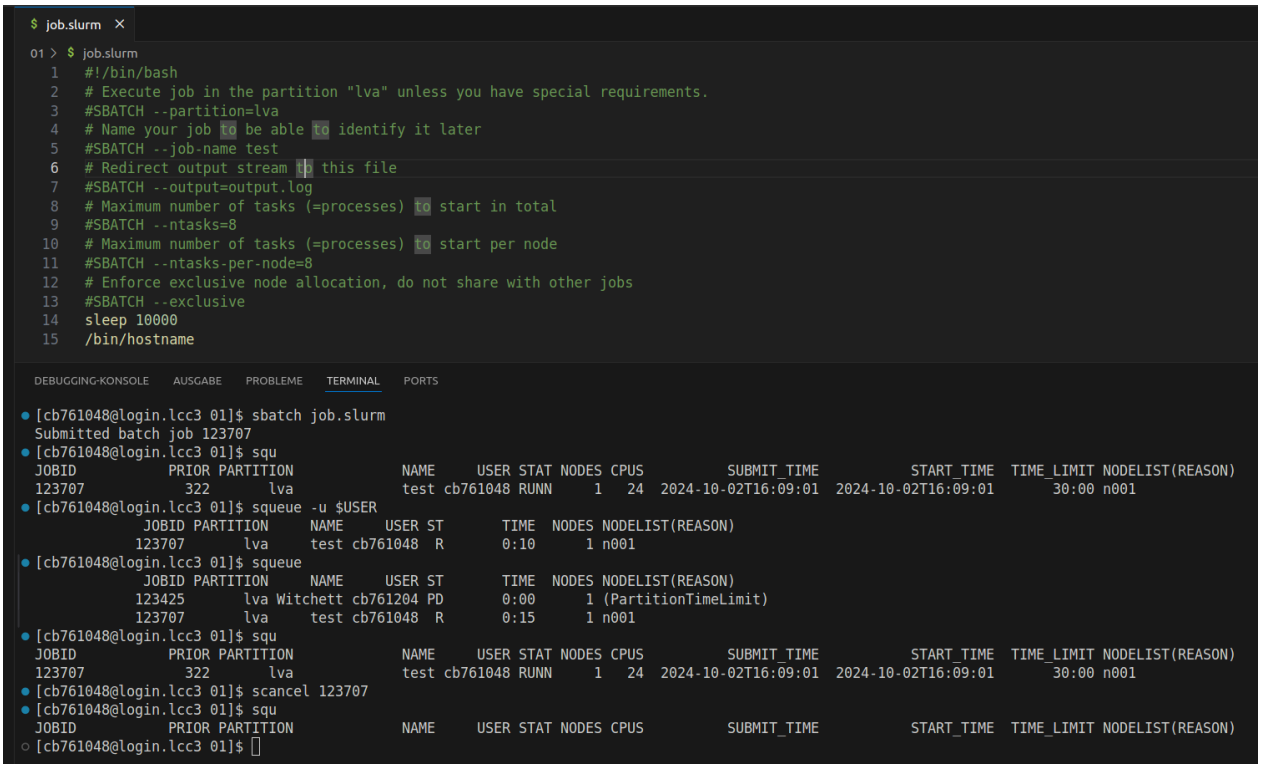

Figure 1: Example Execution of a SLURM Job

- Prepare a submission script that starts an arbitrary executable, e.g. `/bin/hostname`

  See Figure 1.

- In your opionion, what are the 5 most important parameters available when submitting a job and why? What are possible settings of these parameters, and what effect do they have?

  – `--ntasks=ntasks`

  The `--ntasks` parameter specifies the specific number of CPU tasks (or processes) for a SLURM job. For example, with `--ntasks=8`, SLURM will reserve 8 tasks, which can be used to run 8 instances of an MPI program, each representing an individual MPI process. Additionally, the environment variable `SLURM_NTASKS` is set to the specified number of tasks.

  The parameter is important because it ensures that the appropriate number of tasks or processes are allocated for parallel computing jobs, such as those using MPI.

  – `--array=m-n[:step][%maxrunning]`

  The parameter `--array` provides a simple way to parallelize jobs by submitting multiple independent job instances, called *array tasks*, within one job submission. This will start `n-m+1` tasks, each with a unique ID between `m` and `n` (inclusive). For example, `--array=1-5` would start 5 tasks with IDs 1, 2, 3, 4 and 5. The optional `step` option defines the increment between task IDs. For example, `--array=1-10:2` would start tasks with IDs 1, 3, 5, 7 and 9. The optional `maxrunning` option limits the number of tasks that can run simultaneously. For example, `--array=0-9%4` would run a total of 10 tasks but limit the number of simultaneously running tasks to 4 at any given time. Additionally, instead of specifying a range of task IDs, it is also possible to provide a comma-separated list of specific task IDs. For example, `--array=1,3,5,7` would only run tasks with the specified IDs. However, the task IDs must be between 0 and 75.000.

The following environment variables are set at runtime for each task and can be accessed in the SLURM script:

* `SLURM_ARRAY_TASK_COUNT`: Specifies the total number of tasks in the job array.
* `SLURM_ARRAY_TASK_ID`: Specifies the unique ID of the current task within the range n-m.
* `SLURM_ARRAY_TASK_MAX`: Specifies the highest task ID in the job array.
* `SLURM_ARRAY_TASK_MIN`: Specifies the lowest task ID in the job array.
* `SLURM_ARRAY_TASK_STEP`: Specifies the step size, or increment value, between task IDs.

This parameter is important and useful for tasks that can run independently but are similar in nature, such as processing different data files or running simulations with different input parameters. For example, the environment variables like `SLURM_ARRAY_TASK_ID` can be used to run different simulations or processes within each task of a job array, allowing for efficient parallelization of tasks with varying inputs or configurations.

– `--time=[[days-]hours:]minutes[:seconds]`

The `--time` parameter can be used to specify the maximal wall clock time limit for a job. It is possible to specify the time in various formats: just minutes (e.g. `--time=minutes`), days and hours (e.g. `--time=days-hours`) or a more detailed format with days, hours, minutes and seconds (e.g. `--time=[[days-]hours:]minutes[:seconds]`). However, on lcc3, the `seconds` parameter does not actually set the seconds but always rounds the minutes up to the next full minute.

When the time limit exceeds, each task of each job is sent a `TERM` (short for *terminate* and also known as `SIGTERM` in Unix-based systems) signal. This gives the opportunity to catch the `TERM` signal, allowing to perform a controlled shutdown before being forcefully stopped. If the tasks have not been terminated after 30 seconds then a `KILL` (e.g. `SIGKILL`) signal is sent to forcibly terminate the process.

This parameter is important because it allows you to configure the runtime of your jobs. On lcc3, the default runtime is set to 30 minutes, so for longer jobs, adjusting this setting is essential. Additionally, setting a shorter runtime can be beneficial, as it enables the scheduler to optimize resource allocation more effectively, potentially leading to an earlier start for the jobs.

– `--mail-type=[TYPE|ALL|NONE]` & `--mail-user=email_address`

The `--mail-type` and `--mail-user` parameters can be used to send email notifications. The `--mail-type` parameter specifies the events that trigger an email, while the `--mail-user` parameter defines the recipient. By default, `--mail-user` sends emails to the local user who submitted the job. However, on lcc3, this does not send emails to your personal email account, so it is recommended to explicitly specify the email address to ensure notifications are received.

The possible event types of the `--mail-type` parameter are presented in the following. It is possible to define multiple event types by using a comma separated list.

* `NONE`: No email is sent.
* `BEGIN`: An email is sent at the beginning of each job.
* `END`: An email is sent at the end of each job.
* `FAIL`: An email is sent if the job fails.
* `REQUEUE`: An email is sent when the job is requeued, e.g. after a node failure.
* `STAGE_OUT`: An email is sent when the job is staged out. This means that when the job completes, regardless of whether it succeeds or fails, the data from the burst buffers (a type of high-speed storage) is transferred back to the permanent storage (see https://slurm.schedmd.com/burst_buffer.html).
* `TIME_LIMIT`: An email is sent when the job exceeds its time limit.

* `TIME_LIMIT_50`: An email is sent when the job reaches 50% of its time limit.
* `TIME_LIMIT_80`: An email is sent when the job reaches 80% of its time limit.
* `TIME_LIMIT_90`: An email is sent when the job reaches 90% of its time limit.
* `ARRAY_TASKS`: An email is sent for `BEGIN`, `END`, `FAIL` of each task within a job array. However, using this feature with a large job array is not recommended, as it can result in an overwhelming number of emails being generated.
* `ALL`: Includes `BEGIN`, `END`, `FAIL`, `REQUEUE`, `STAGE_OUT`.

These parameters are important because they notify users of key job events, ensuring they stay informed about job progress or failures without manual monitoring. This allows for timely responses and helps manage long-running or critical jobs efficiently.

- How do you run your program in parallel? What environment setup is required?

  Assumptions: Access to a compute cluster, like lcc3.

  First, we need to prepare a job script (see Figure 2) that includes the necessary parameters, such as the number of tasks (`--ntasks`) and nodes (`--nodes`), and specify the program we want to run in parallel. For example, in the case of MPI, we might use `mpirun`, `mpiexec` or `srun` to start the program.

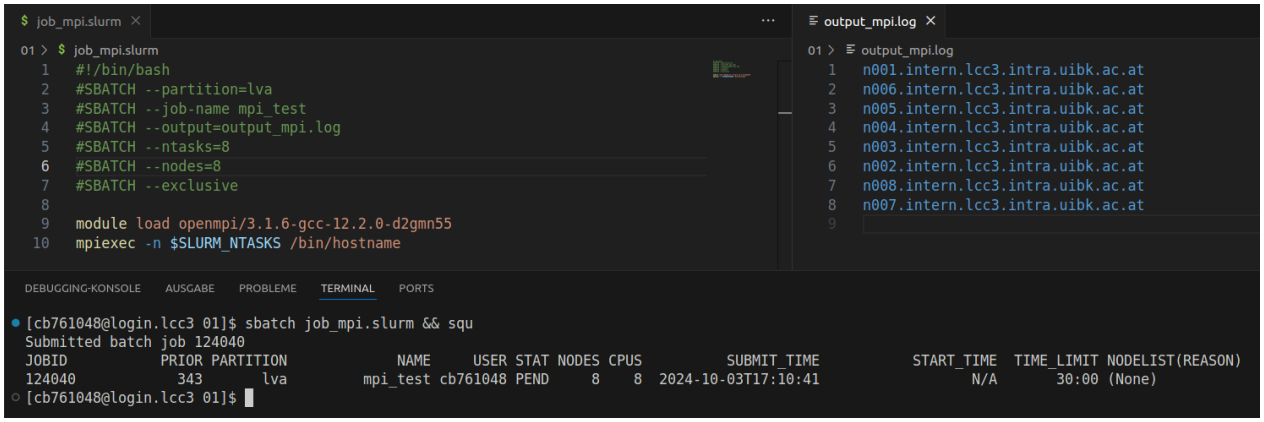

Figure 2: Example

Secondly, we need to load appropriate modules or set environment variables relevant for the parallel program we want to run. As shown in Figure 2 we load the `openmpi` module to access the **mpiexec** command.

Finally, we can start multiple instances of a program with the **mpiexec** command as shown in Figure 2. In this simple example we execute the `/bin/hostname` command in 8 different parallel processes.

Finally, we can run multiple instances of a program using the `mpiexec` command, as demonstrated in Figure 2. In this example, we execute the `/bin/hostname` command in 8 parallel processes, each on a separate CPU core and a separate node.

Exercise 2 (2 Points)

This exercise consists in running an MPI microbenchmark in order to examine the impact of HPC topologies on performance.

Description

The OSU Micro-Benchmarks suite holds multiple benchmarks that measure low-level performance properties such as latency and bandwidth between MPI ranks (=processes). Specifically, for this exercise, we are interested in the point-to-point ones, which exchange messages between 2 MPI ranks.

Tasks

- Download and build the OSU Micro-Benchmarks available at http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-5.8.tgz. Do not forget to set the compiler parameters for `configure`, e.g. `./configure CC=mpicc CXX=mpic++ ...`

  To download the OSU Micro-Benchmarks the following command can be used:

```
wget http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-5.8.tgz
```

  To build the benchmarks the following commands were used:

```
tar -xvzf osu-micro-benchmarks-5.8.tgz
cd osu-micro-benchmarks-5.8/
module load openmpi/3.1.6-gcc-12.2.0-d2gmn55
./configure CC=mpicc CXX=mpic++
./make
```

  The `module load openmpi/3.1.6-gcc-12.2.0-d2gmn55` command is needed in order to access the `mpicc` and `mpic++` compiler wrappers which are needed during compilation. The command `./configure CC=mpicc CXX=mpic++` sets the `C` compiler to `mpicc` (`CC=mpicc`) and the `C++` compiler to `mpic++` (`CXX=mpic++`). `mpicc` and `mpic++` are wrappers around standard `C` (e.g. `gcc`) and `C++` (e.g. `g++`) compilers, provided by the MPI implementation, in this case, `OpenMPI`. They ensure that the code is compiled by forwarding it to the `C` and `C++` compilers, while also linking the necessary MPI libraries and setting the required include directories for MPI-specific header files.

- After building, submit SLURM jobs that run the `osu_latency` and `osu_bw` executables.

  In order to submit SLURM jobs that run the `osu_latency` and `osu_bw` executables, the following SLURM script can be used:

```
#!/bin/bash
#SBATCH --partition=lva
#SBATCH --job-name osu_test
#SBATCH --output=output.log
#SBATCH --exclusive
#SBATCH --time 10

#SBATCH --ntasks=2

# load modules
module load openmpi/3.1.6-gcc-12.2.0-d2gmn55

# run programs
mpiexec ./osu-micro-benchmarks-5.8/mpi/pt2pt/osu_latency
mpiexec ./osu-micro-benchmarks-5.8/mpi/pt2pt/osu_bw
```

- Create a table and figures that illustrate the measured data and study them. What effects can you observe?

  In order to evaluate the results we first need to understand the benchmarks. Both executables are `Host-based, Point-to-Point MPI Benchmarks`.

  - **Host-based**: "The communication and performance measurements are carried out using the host machine's CPU (not specialized hardware like GPUs). This means that the benchmarks are executed on the CPU of the machine (host), measuring the performance of CPU-to-CPU communication over the network. The data transfers are managed by the CPU rather than offloaded to any specialized hardware (though network interfaces like InfiniBand or Ethernet may still be involved in the actual communication)." (source ChatGPT).
    - **Point-to-Point**: "The benchmarks measure direct communication between two MPI processes (one sender and one receiver). One process sends a message, and the other process receives it." (source ChatGPT).
      - **osu_latency - Latency Test**: "The latency tests are carried out in a ping-pong fashion. The sender sends a message with a certain data size to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Many iterations of this ping-pong test are carried out and average one-way latency numbers are obtained. Blocking version of MPI functions (MPI_Send and MPI_Recv) are used in the tests." (see OSU Micro-Benchmarks Definition).
      - **osu_bw - Bandwidth Test**: "The bandwidth tests are carried out by having the sender sending out a fixed number (equal to the window size) of back-to-back messages to the receiver and then waiting for a reply from the receiver. The receiver sends the reply only after receiving all these messages. This process is repeated for several iterations and the bandwidth is calculated based on the elapsed time (from the time sender sends the first message until the time it receives the reply back from the receiver) and the number of bytes sent by the sender. The objective of this bandwidth test is to determine the maximum sustained date rate that can be achieved at the network level. Thus, non-blocking version of MPI functions (MPI_Isend and MPI_Irecv) are used in the test." (see OSU Micro-Benchmarks Definition).

OSU MPI Latency & Bandwidth Test v5.8		
Size (bytes)	**Latency (us)**	**Bandwidth (MB/s)**
0	0.16	
1	0.16	9.58
2	0.16	18.95
4	0.16	37.71
8	0.16	77.10
16	0.16	153.21
32	0.20	292.77
64	0.21	592.45
128	0.32	748.35
256	0.35	1374.72
512	0.38	2419.53
1024	0.46	4022.96
2048	0.61	6169.27
4096	0.93	8493.67
8192	1.60	10761.16
16384	2.90	6558.41
32768	4.74	9266.49
65536	8.08	11622.75
131072	16.52	12506.14

262144	24.62	11606.09
524288	47.59	11414.04
1048576	90.66	11631.93
2097152	177.40	11739.32
4194304	765.60	11803.31

Table 1: Simple execution of benchmarks on one node

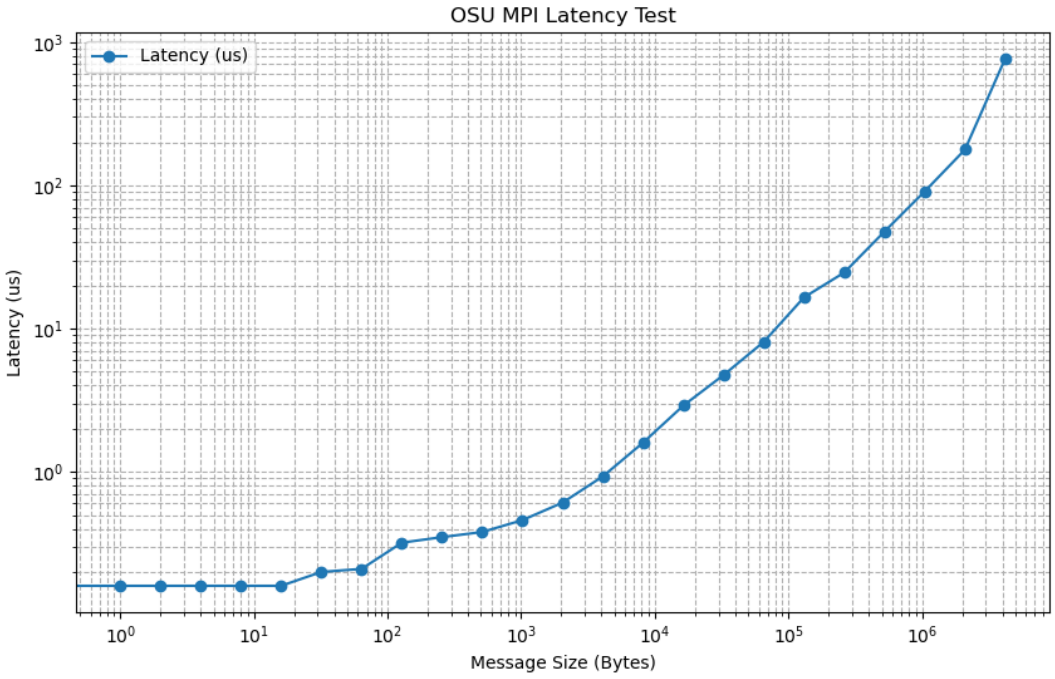

Figure 3: Simple execution of `osu_latency` on one node

Observations - Latency Test

- **Constant Latency for Small Messages**: For message sizes up to 16 bytes, the latency remains constant at 0.16 microseconds. This suggests that for very small messages, the communication overhead dominates the latency, and the size of the message itself does not significantly affect the time.
- **Latency Growth for Larger Messages**: For larger message sizes, we observe an exponential increase in latency. This increase starts to become more significant from around 128 bytes onwards, with latency reaching around 759 microseconds for the largest message size of 4MB.

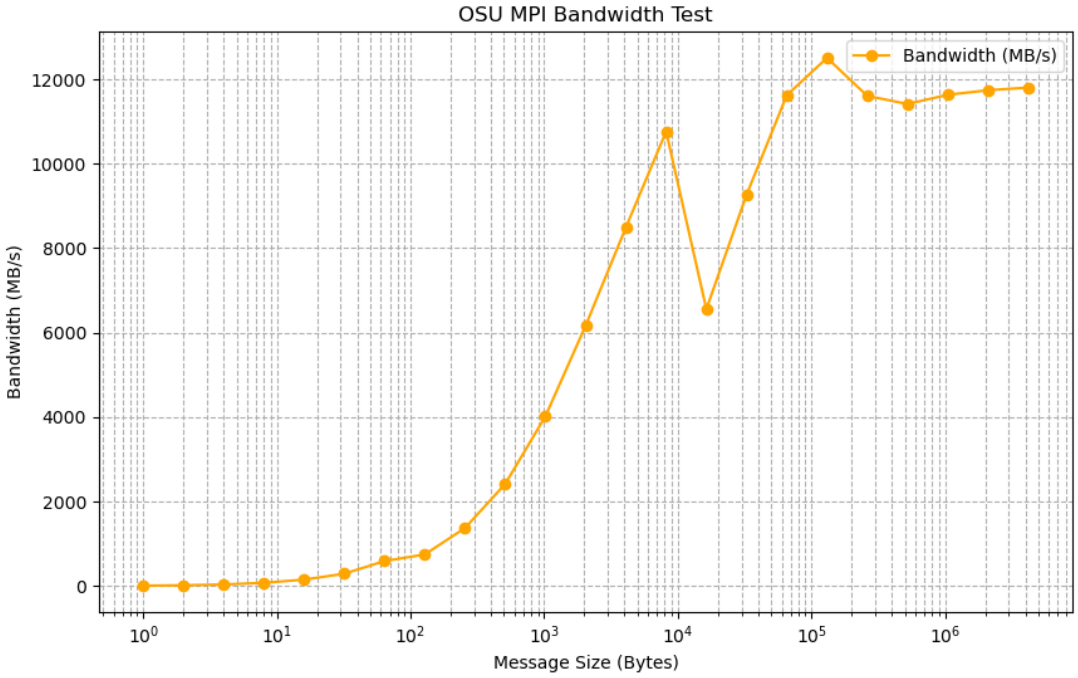

Figure 4: Simple execution of `osu_bw` on one node

Observations - Bandwidth Test

- **Low Bandwidth for Small Messages**: For very small message sizes the bandwidth remains quite low, starting from about 9.58 MB/s for a 1-byte message.
- **Exponential Increase for Mid-Sized Messages**: For mid-sized messages the bandwidth grows exponentially, peaking at around 12506.14 MB/s for messages around 130KB in size. However, there is a noticeable drop in bandwidth between 8KB and 65KB, where the performance drops before recovering. This drop could be attributed to system-specific factors such as caching, buffering or memory allocation limits, which momentarily affect the efficiency of data transfer in this size range.
- **Bandwidth Stabilization for Large Messages**: For message sizes above 130KB, the bandwidth experiences a slight drop but then stabilizes, fluctuating around 11500 MB/s. This suggests that increasing the message size beyond 130KB does not significantly improve bandwidth, as the system has likely reached its maximum throughput, constrained by the limitations of the hardware and network architecture.

- Find out more about the hardware that you're running on, e.g. by using `lstopo --of txt` (available via `module load hwloc`).

  Figure 5 shows the hardware architecture of a node on LCC3.

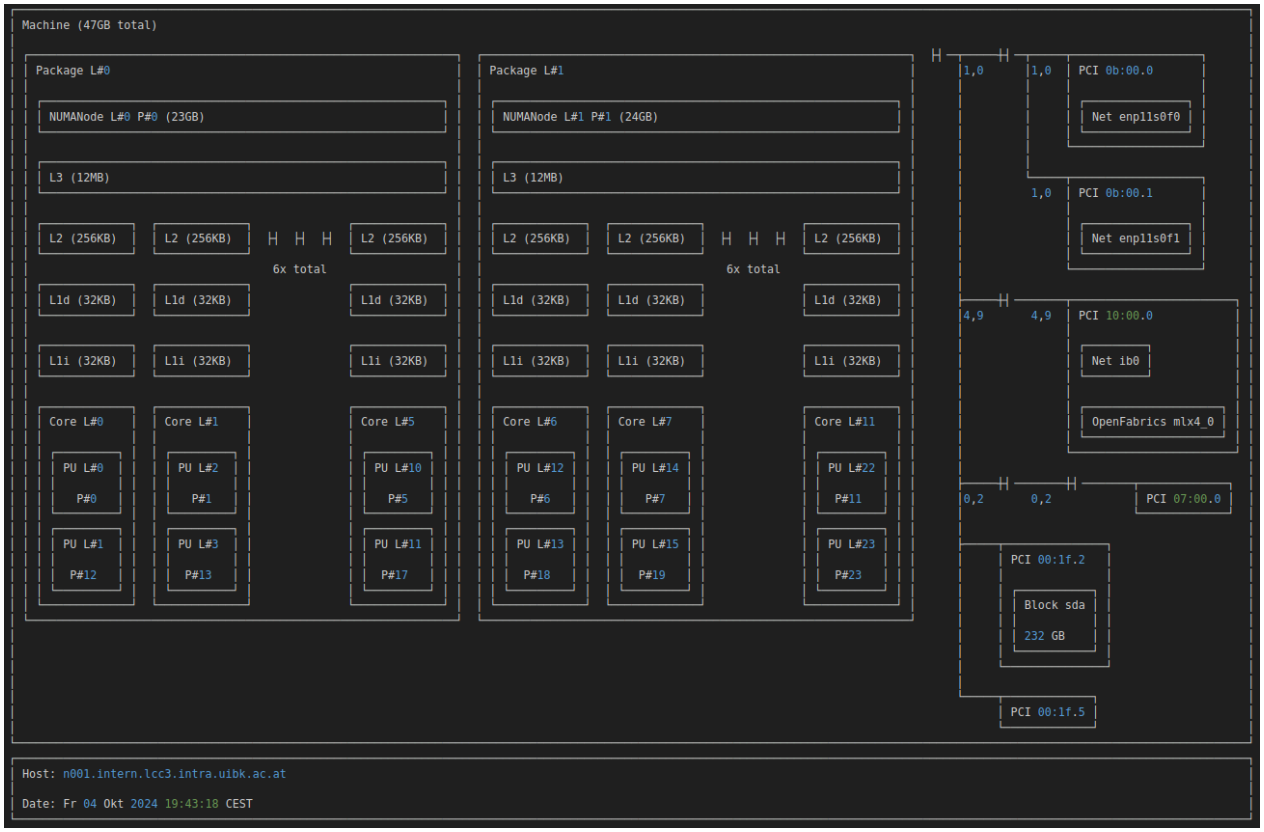

Figure 5: LCC3 Node: Hardware Information

Hardware Details (per cluster)

- 8 machines
- 96 cores or 192 threads for the whole cluster

Hardware Details (per machine)

- 47 GB of total memory
- Two CPU packages (sockets) with six core CPUs per package
- Two NUMA nodes with 23GB and 24GB respectively per machine
- 6 physical cores with 2 threads (processing units or PUs) per core, making 12 threads per NUMA node and 24 threads for the entire machine.
- L1d and L1i (32KB each) and L2 (256KB) per core.
- L3 cache (12MB) is shared by all cores in each CPU package.
- Ethernet (enp11s0f0 and enp11s0f1) and InfiniBand (ib0) network interfaces per machine.
- sda storage device (232 GB) per machine.

Modify your experiment such that the 2 MPI ranks are placed on

- **different cores of the same socket**
  To execute the benchmarks on different cores within the same socket, it's sufficient to specify `#SBATCH --ntasks=2`, as LCC3 by default assigns one node for two tasks, and `mpiexec` automatically maps these tasks to two separate cores on a single socket. However, for complete assurance in rank placement, you can explicitly use the options `--bind-to core` and `--map-by core` with the `mpiexec` command to ensure the MPI ranks are properly allocated (refer to the mpiexec(1) man page for more details).

```
#!/bin/bash
#SBATCH --partition=lva
#SBATCH --job-name osu_different_cores_of_same_socket
#SBATCH --output=logs/01_osu_different_cores_of_same_socket.log
#SBATCH --exclusive
#SBATCH --time 10

#SBATCH --ntasks=2
#SBATCH --nodes=1

# load modules
module load openmpi/3.1.6-gcc-12.2.0-d2gmn55

# run programs
mpiexec --display-map --bind-to core --map-by core ../osu-micro-benchmarks-5.8/mpi/pt2pt/osu_latency
mpiexec --display-map --bind-to core --map-by core ../osu-micro-benchmarks-5.8/mpi/pt2pt/osu_bw
```

- **different sockets of the same node**
  To place the MPI ranks on different sockets within the same node, you need to specify the options `--bind-to core` and `--map-by socket` in the `mpiexec` command. These options ensure that each rank is allocated to a different socket (refer to the mpiexec(1) man page for more details).

```
#!/bin/bash
#SBATCH --partition=lva
#SBATCH --job-name osu_different_sockets_of_same_node
#SBATCH --output=logs/02_osu_different_sockets_of_same_node.log
#SBATCH --exclusive
#SBATCH --time 10

#SBATCH --ntasks=2
#SBATCH --nodes=1

# load modules
module load openmpi/3.1.6-gcc-12.2.0-d2gmn55

# run programs
mpiexec --display-map --bind-to core --map-by socket ../osu-micro-benchmarks-5.8/mpi/pt2pt/osu_latency
mpiexec --display-map --bind-to core --map-by socket ../osu-micro-benchmarks-5.8/mpi/pt2pt/osu_bw
```

- **different nodes**
  To place the MPI ranks on different nodes, simply specify `#SBATCH --nodes=2`, as LCC3 by default allocates the tasks to separate nodes. However, to ensure that no two tasks are placed on the same node, you can additionally specify `#SBATCH --ntasks-per-node=1`, which explicitly prohibits multiple tasks from being assigned to a single node (refer to the mpiexec(1) man page for more details).

```
#!/bin/bash
#SBATCH --partition=lva
#SBATCH --job-name osu_different_nodes
#SBATCH --output=logs/03_osu_different_nodes.log
#SBATCH --exclusive
```

```
#SBATCH --time 10

#SBATCH --ntasks=2
#SBATCH --ntasks-per-node=1
#SBATCH --nodes=2

# load modules
module load openmpi/3.1.6-gcc-12.2.0-d2gmn55

# run programs
mpiexec --display-map ../osu-micro-benchmarks-5.8/mpi/pt2pt/osu_latency
mpiexec --display-map ../osu-micro-benchmarks-5.8/mpi/pt2pt/osu_bw
```

- Amend your table and figures to include these additional measurements. What effects can you observe? How can you verify rank placement without looking at performance?

Verify Rank Placement

To verify the rank placement the `-- display-map` option can be specified in the `mpiexec` command. As illustrated in Figure 6, MPI ranks can be placed in various configurations, such as on different hardware threads within a single core, on separate cores within the same socket or across different sockets (refer to the mpiexec(1) man page for more details).

```
#!/bin/bash
#SBATCH --ntasks=2

module load openmpi/3.1.6-gcc-12.2.0-d2gmn55
mpiexec --display-map --bind-to hwthread --map-by hwthread /bin/hostname
mpiexec --display-map --bind-to core --map-by core /bin/hostname
mpiexec --display-map --bind-to core --map-by socket /bin/hostname
```

```
Data for JOB [33330,1] offset 0 Total slots allocated 2

 ========================   JOB MAP   ========================

 Data for node: n001    Num slots: 2    Max slots: 0    Num procs: 2
    Process OMPI jobid: [33330,1] App: 0 Process rank: 0 Bound: socket 0[core 0[hwt 0]]:[B./../../../../..][../../../../../..]
    Process OMPI jobid: [33330,1] App: 0 Process rank: 1 Bound: socket 0[core 0[hwt 1]]:[.B/../../../../..][../../../../../..]

 =============================================================
n001.intern.lcc3.intra.uibk.ac.at
n001.intern.lcc3.intra.uibk.ac.at

 Data for JOB [33354,1] offset 0 Total slots allocated 2

 ========================   JOB MAP   ========================

 Data for node: n001    Num slots: 2    Max slots: 0    Num procs: 2
    Process OMPI jobid: [33354,1] App: 0 Process rank: 0 Bound: socket 0[core 0[hwt 0-1]]:[BB/../../../../..][../../../../../..]
    Process OMPI jobid: [33354,1] App: 0 Process rank: 1 Bound: socket 0[core 1[hwt 0-1]]:[../BB/../../../..][../../../../../..]

 =============================================================
n001.intern.lcc3.intra.uibk.ac.at
n001.intern.lcc3.intra.uibk.ac.at

 Data for JOB [33356,1] offset 0 Total slots allocated 2

 ========================   JOB MAP   ========================

 Data for node: n001    Num slots: 2    Max slots: 0    Num procs: 2
    Process OMPI jobid: [33356,1] App: 0 Process rank: 0 Bound: socket 0[core 0[hwt 0-1]]:[BB/../../../../..][../../../../../..]
    Process OMPI jobid: [33356,1] App: 0 Process rank: 1 Bound: socket 1[core 6[hwt 0-1]]:[../../../../../..][BB/../../../../..]

 =============================================================
n001.intern.lcc3.intra.uibk.ac.at
n001.intern.lcc3.intra.uibk.ac.at
```

Figure 6: Verify rank placement: Output of `--display-map`

Results

The results of a **single** execution are shown in Table 2 and Figures 7 and 8. These results clearly demonstrate that as the physical distance between processing units increases - whether comparing different cores, sockets, or nodes - there is a corresponding rise in latency and a decrease in bandwidth. This trend is especially noticeable for larger message sizes, where the performance differences become more pronounced.

OSU MPI Latency & Bandwidth Test v5.8						
Size (bytes)	**Latency (us)**			**Bandwidth (MB/s)**		
	diff. cores	diff. sockets	diff. nodes	diff. cores	diff. sockets	diff. nodes
0	0.16	0.33	1.69			
1	0.16	0.33	1.66	9.40	8.55	3.43
2	0.16	0.33	1.64	18.78	16.72	6.98
4	0.16	0.33	1.64	37.71	33.75	13.92
8	0.16	0.32	1.64	73.51	67.82	27.87
16	0.16	0.32	1.64	151.45	140.17	55.58
32	0.20	0.50	1.65	292.15	262.76	112.29
64	0.21	0.51	1.78	597.30	538.02	206.18
128	0.32	0.72	2.46	740.36	398.84	297.01
256	0.35	0.75	2.61	1337.48	739.23	565.16
512	0.37	0.81	2.89	2318.61	1367.13	1080.82
1024	0.46	0.98	3.44	3971.29	2089.75	1781.19
2048	0.60	1.22	4.49	6151.07	3112.88	2707.53
4096	0.93	1.65	6.61	8550.82	4504.37	2935.42
8192	1.59	2.62	8.46	10634.56	5539.04	3117.07
16384	2.91	4.32	11.13	6531.85	6365.13	3195.78
32768	4.78	7.52	16.24	9290.99	8845.70	3235.72
65536	8.21	13.73	26.33	11692.11	10653.97	3263.89
131072	16.80	26.85	45.94	12520.65	11431.07	3279.12
262144	24.29	25.13	83.93	11286.33	10694.44	3286.80
524288	47.65	49.08	161.21	11152.50	10251.13	3293.96
1048576	91.14	93.67	315.64	11513.14	10701.20	3295.65
2097152	181.65	184.99	624.53	11768.97	11158.62	3395.57
4194304	739.75	395.83	1243.29	11331.54	10388.49	3396.35

Table 2: Example Execution of benchmarks on different cores, sockets and nodes

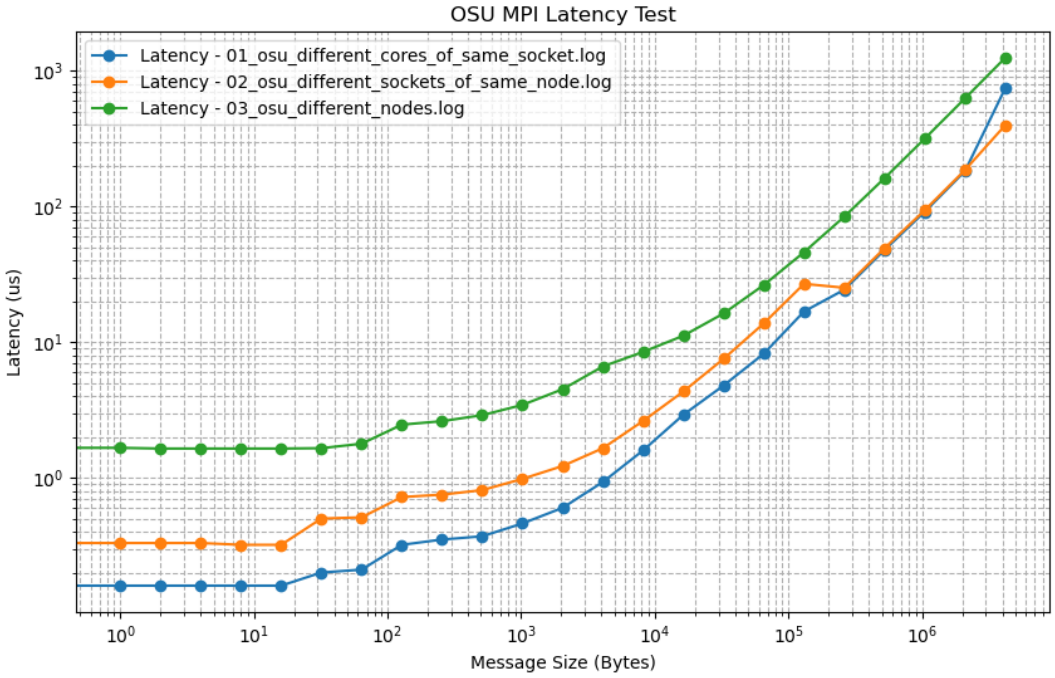

Figure 7: Example Execution of latency benchmark on different cores, sockets and nodes

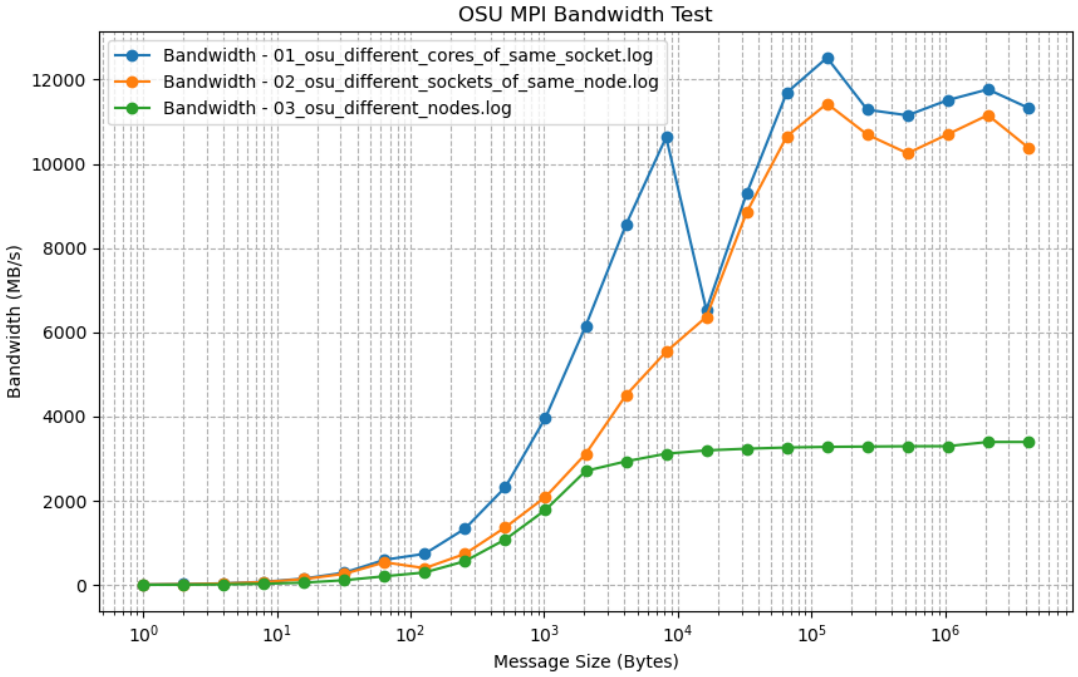

Figure 8: Example Execution of bandwidth benchmark on different cores, sockets and nodes

- How stable are the measurements when running the experiments multiple times?

  To evaluate the stability of the measurements, the results of 10 benchmark executions were aggregated, as shown in Table 3 and visualized in Figures 9 and 10. By averaging the results and calculating the standard deviation across these iterations, we can assess the consistency of the measurements.

  Table 3 presents the mean and standard deviation for both latency and bandwidth across different message sizes and processing configurations (cores, sockets, nodes). The low standard deviation values for smaller message sizes suggest that the system's performance is highly consistent when handling small amounts of data. However, as message sizes increase, some configurations show a slightly higher standard deviation.

  Overall, the results suggest that the system performs reliably across repeated experiments, with minimal performance fluctuations in most configurations.

OSU MPI Latency & Bandwidth Test v5.8												
Size (B)	**Latency (us)**						**Bandwidth (MB/s)**					
	diff. cores		diff. sockets		diff. nodes		diff. cores		diff. sockets		diff. nodes	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
0	0.16	0.0	0.33	0.01	1.68	0.01						
1	0.16	0.0	0.33	0.0	1.66	0.01	9.52	0.18	8.97	0.12	3.37	0.17
2	0.16	0.0	0.33	0.0	1.65	0.0	18.91	0.45	17.79	0.21	6.88	0.27
4	0.16	0.0	0.33	0.01	1.64	0.01	37.8	0.76	35.48	0.47	13.96	0.15
8	0.16	0.0	0.32	0.01	1.66	0.06	75.85	1.1	71.26	1.02	27.58	0.44
16	0.16	0.0	0.32	0.01	1.64	0.01	152.63	2.21	143.19	2.08	55.78	0.82
32	0.2	0.01	0.51	0.01	1.65	0.0	294.44	6.09	263.59	2.51	112.52	1.14
64	0.21	0.01	0.51	0.01	1.79	0.01	598.63	10.22	545.02	14.06	208.49	0.91
128	0.33	0.01	0.73	0.01	2.46	0.01	739.36	11.88	409.77	5.57	300.65	1.59
256	0.35	0.01	0.76	0.01	2.61	0.01	1353.86	22.48	774.21	8.34	567.04	4.34
512	0.38	0.01	0.81	0.01	2.9	0.01	2393.29	72.43	1427.11	16.64	1079.82	19.15
1024	0.46	0.01	0.98	0.01	3.45	0.01	3883.32	67.39	2143.03	16.16	1775.35	92.86
2048	0.61	0.01	1.22	0.0	4.49	0.01	6030.75	122.9	3253.78	9.35	2673.26	75.28
4096	0.93	0.02	1.64	0.0	6.6	0.01	8462.41	187.56	4741.98	53.1	2937.22	9.77
8192	1.6	0.03	2.63	0.02	8.47	0.01	10540.17	232.99	5853.13	48.2	3129.33	3.73
16384	2.91	0.06	4.31	0.02	11.16	0.08	6486.81	160.71	6442.35	107.88	3199.94	11.33
32768	4.74	0.1	7.31	0.15	16.24	0.05	9121.53	185.45	9165.39	87.54	3241.5	4.72
65536	8.29	0.19	13.4	0.09	26.31	0.06	11244.43	234.39	11533.44	261.16	3267.4	3.13
131072	16.43	0.32	26.82	0.25	45.93	0.09	11737.88	283.95	12387.63	375.77	3280.88	3.23
262144	25.07	0.47	24.51	0.24	84.13	0.27	10852.09	150.47	11477.48	90.04	3286.46	3.03
524288	49.66	0.97	47.6	0.22	161.28	0.11	10782.18	181.54	11341.7	79.53	3293.17	2.54
1048576	96.01	2.17	91.44	0.71	315.7	0.11	11247.33	186.65	11652.6	157.44	3294.63	2.84
2097152	204.53	12.84	178.33	1.28	624.56	0.04	11577.32	163.24	11882.87	104.36	3395.62	0.09
4194304	772.43	11.75	381.12	10.92	1243.28	0.15	10824.79	199.39	11613.41	386.08	3396.39	0.03

Table 3: Mean and standard deviation from 10 benchmark executions

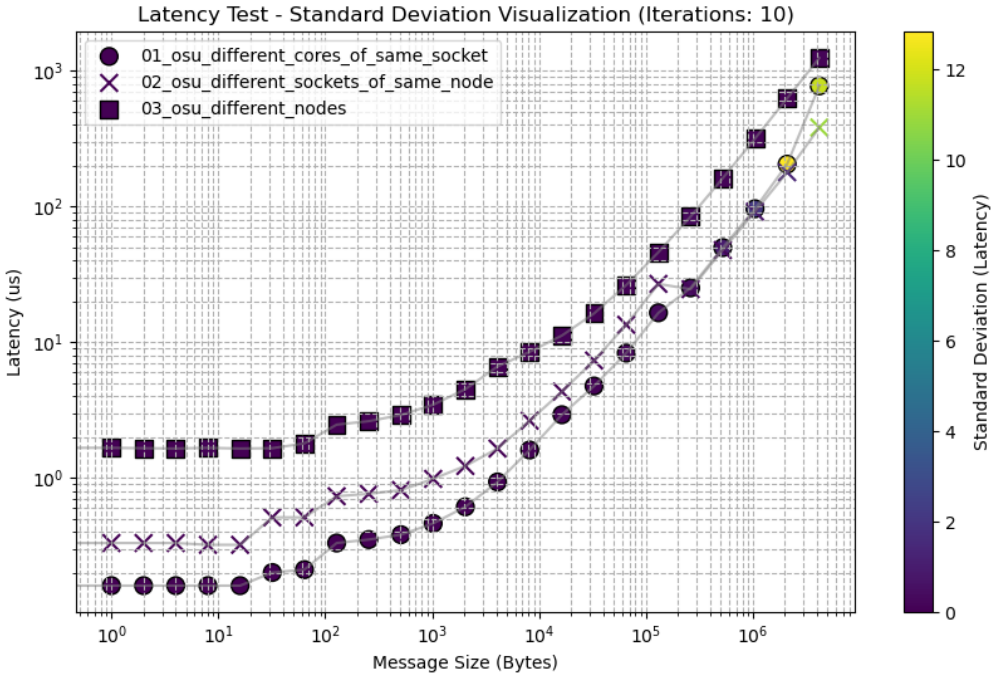

Figure 9: Visualization of standard derivation for multiple executions of the latency benchmark

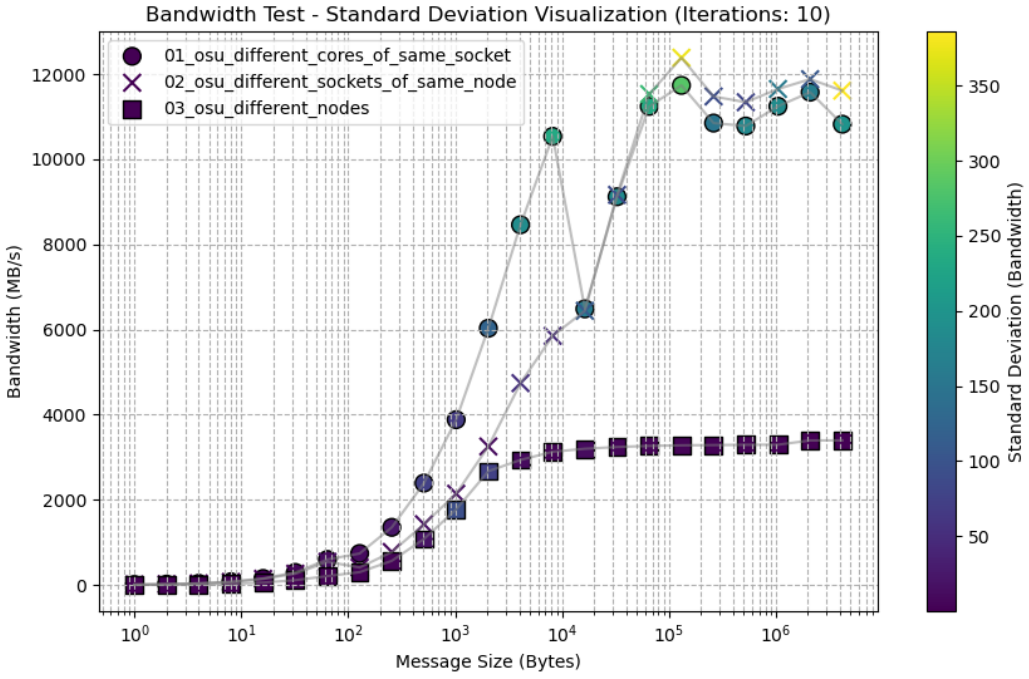

Figure 10: Visualization of standard derivation for multiple executions of the bandwidth benchmark

- Insert the measured time for latency (size 0) and bandwidth (size 1048576) into the comparison spreadsheet: https://docs.google.com/spreadsheets/d/1p6d9F12EtykmI2-7MnHkg0U15UAtaCvWz8Ip92ZEsWo