Network performance in distributed training: Maximizing GPU utilization on OpenShift

October 16, 2025

Tanya Osokin, Kevin Pouget, Michey Mehta

Related topics: Artificial intelligence, Containers, Kubernetes

Related products: Red Hat OpenShift

Table of contents:

We compared two IBM Cloud GPU clusters—one with NVIDIA L40S GPUs and one with H100 GPUs—head-to-head to see what really drives distributed training performance.

The key finding is that for distributed training, the choice of network architecture is the most significant factor in performance, far outweighing the capabilities of the default container networking. Our tests conclusively show that using the standard Red Hat OpenShift pod network for internode communication creates a severe performance bottleneck that prevents expensive GPU resources from being fully utilized.

For the L40S cluster, leveraging secondary Virtual Network Interface Cards (vNICs) provided a significant performance advantage over the default pod network, with this gap widening at scale to a peak performance increase of 132% in the 8-node test. For the more powerful H100 cluster, the impact was even more stark: switching from vNICs to a high-throughput single root input/output virtualization (SR-IOV) network yielded a 3x increase in training throughput. While enabling remote direct memory access (RDMA) over SR-IOV provided only a marginal performance increase for the model size tested, it is a critical technology for larger models where CPU network management becomes a bottleneck.

bottleneck.

Our primary recommendations are as follows:

- High-performance networking is essential for scalable distributed training on OpenShift.
- For clusters with top-tier GPUs like the H100, SR-IOV NICs with higher bandwidth are required to prevent the network from becoming the primary bottleneck.
- For clusters with mid-range GPUs like the L40S, secondary vNICs provide a cost-effective solution for achieving efficient scaling.

Investing in the appropriate network infrastructure is critical to maximizing the return on investment in GPU hardware and ensuring cost-effective, high-performance AI operations at scale.

Technical background

The distributed training process in this report is based on a common methodology for training large language models. While our specific tests used the open source InstructLab project for its training script and efficient data processing, the findings on network performance are broadly applicable to any distributed training framework.

The core training methodology is built on PyTorch's Fully Sharded Data Parallel (FSDP) feature. FSDP is a data parallelism technique that enables the training of very large models by sharding the model's parameters, gradients, and optimizer states across all available GPUs in the cluster. By ensuring each GPU only holds a fraction of the total model, FSDP makes it possible to train models that would otherwise be too large to fit into a single GPU's memory.

This sharding strategy, however, makes the training process highly dependent on the speed and efficiency of the underlying network. GPUs must constantly communicate to exchange the necessary parameters during the forward and backward passes, so the choice of network architecture becomes a critical factor in overall FSDP performance.

Hardware we used

Our testing utilized two distinct IBM Cloud GPU clusters to understand how different hardware configurations respond to network optimization.

2xL40S cluster workers

The gx3-48x240x2l40s virtual instance in IBM Cloud features 48 vCPUs, 240 GiB of RAM, and 2 NVIDIA L40S GPUs with 48 GB of memory each. It offers a high-speed network connection with a bandwidth of 100 Gbps per vNIC, making it ideal for high-performance computing tasks such as AI, machine learning, and large-scale simulations. This configuration provides a powerful blend of CPU, GPU, and memory resources for demanding workloads.

In our tests, we scaled the system from 1 to 8 nodes to evaluate how the performance changes with increased computational resources. This approach allowed us to analyze the system's scalability and understand the impact of node scaling on overall performance.

8xH100 cluster nodes

The gx3d-160x1792x8h100 virtual instance in IBM Cloud is equipped with 160 vCPUs, 1,792 GiB of RAM, and 8 NVIDIA H100 GPUs with 80 GB of memory each. It provides a high-speed network connection with a bandwidth of 200 Gbps per vNIC and up to 8 SR-IOV NIC with 400 Gbps each, making it well-suited for extremely demanding workloads such as advanced AI, deep learning, and high-performance computing. This configuration delivers substantial computational power with an impressive GPU setup for large-scale simulations and data-intensive tasks.

In our tests, we scaled from 1 to 2 nodes, with each node equipped with 8 NVIDIA H100 GPUs. This configuration provided substantial computational power per node, allowing us to evaluate how performance changes when doubling the number of nodes, with each node capable of handling complex, GPU-intensive workloads due to the high-performance H100 GPUs.

Test details

Our performance testing was conducted with reproducibility and transparency in mind, under CI automation. In the subsections below we detail the key aspects of the test environment, the test harness and the training job.

Testing environment

All the tests ran on an OpenShift cluster with the following platform versions:

- OpenShift version: 4.18.16
- OpenShift AI version: 2.19.0

Test automation harness

The entire test workflow was automated using TOPSAIL, an open source toolbox for orchestrating complex test scenarios on OpenShift. The `fine_tuning` project within TOPSAIL was used to manage all aspects of the test execution, including preparing the environment, preprocessing the dataset, and dynamically generating and deploying all necessary Kubernetes resources. This includes the PyTorchJob manifest, the config.json hyperparameter file, and the run_ilab.sh entrypoint script detailed below. This automation ensures a consistent and repeatable testing methodology across all configurations.

Training job specification

The distributed training jobs were orchestrated using the `PyTorchJob` custom resource. The following is a summary of the key configuration parameters for an 8-node `ilab` job example:

```
apiVersion: kubeflow.org/v1
kind: PyTorchJob
metadata:
```

```
  name: ilab
  namespace: fine-tuning-testing
spec:
  pytorchReplicaSpecs:
    Master:
      replicas: 1
      restartPolicy: Never
      template:
        spec:
          containers:
          - name: pytorch
            image: registry.redhat.io/rhelai1/instructlab-nvi
            command:
            - bash
            - /mnt/entrypoint/run_ilab.sh
            resources:
              limits:
                nvidia.com/gpu: "2"
              requests:
                cpu: "1"
                memory: 10Gi
                nvidia.com/gpu: "2"
            # ... and other configurations
    Worker:
      replicas: 7
      restartPolicy: Never
      template:
        spec:
          containers:
          - name: pytorch
            image: registry.redhat.io/rhelai1/instructlab-nvi
            command:
            - bash
            - /mnt/entrypoint/run_ilab.sh
            resources:
              limits:
                nvidia.com/gpu: "2"
              requests:
                cpu: "1"
                memory: 10Gi
```

```
            nvidia.com/gpu: "2"
        volumeMounts:
        - mountPath: /dev/shm
          name: shm-volume
        # ... and other volume mounts
      volumes:
      - name: storage-volume
        persistentVolumeClaim:
          claimName: fine-tuning-storage
      - name: shm-volume
        emptyDir:
          medium: Memory
          sizeLimit: 20Gi
      # ... and other volumes
```

Copy snippet

This configuration established a distributed training cluster with one Controller and seven Worker replicas. Each replica pod was provisioned with 2 NVIDIA GPUs, 1 CPU core, and 10 Gi of system memory. A key architectural choice was the use of a 20 Gi in-memory volume (`/dev/shm`) to facilitate high-performance inter-process communication, essential for distributed training.

The core of the training process was executed via `torchrun` . The following parameters remained constant across all tests to ensure a consistent baseline:

- `--model_name_or_path=granite-3.1-8b-starter-v2`
- `--num_epochs=1`
- `--data_path=data.jsonl`

Key variables that were adjusted for each specific test scenario included the number of nodes (`--nnodes`), the number of processes per node (`--nproc_per_node`), and the maximum batch length (`--max_batch_len`).

The model was trained on a dataset containing 9,872 tokenized instruction-response samples with average sequence length of 1,483

tokens per sample.

Network configurations

This study evaluated performance across several distinct network architectures to determine their impact on internode communication, a critical factor in distributed training throughput. The subsections below describe the four different network configurations.

Pod network

Description: The default OpenShift software-defined network (OVN), where each pod receives an IP address and communicates over a virtualized overlay network.

Performance characteristics: The bandwidth of the pod network is not fixed; it is limited by the underlying physical network and is reduced by the overhead of the software-defined network (SDN) encapsulation protocols. This introduces latency and consumes CPU cycles, making it unsuitable for high-throughput/low latency distributed workloads.

vNIC

Description: A virtualized network interface that allows a Virtual Machine (VM) to interact with the network as if it had its own dedicated hardware, abstracting the physical Network Interface Card(NIC) using the virtio protocol.

Performance characteristics: The bandwidth of a vNIC is capped by the instance profile. In our tests, this was **100 Gbps** for the L40S nodes and **200 Gbps** for the H100 nodes. While faster than the pod network, it still carries virtualization overhead from the hypervisor.

SR-IOV

Description: SR-IOV is a Peripheral Component Interconnect Express (PCIe) specification that lets a single physical NIC expose multiple Virtual Functions (VFs) directly to guest VMs, removing hypervisor switching

Functions (VFs) directly to guest VMs, removing hypervisor switching overhead.

Performance characteristics: SR-IOV provides near-line-rate performance. For the H100 cluster, this meant accessing the full bandwidth of the physical **400 Gbps** NICs, resulting in a dramatic increase in throughput.

SR-IOV with RDMA

Description: This combines SR-IOV with RDMA, enabling network interfaces to directly access the memory of other nodes without involving the CPU. This zero-copy data transfer mechanism is designed for maximum throughput and minimal latency.

Performance characteristics: The bandwidth is the same as the underlying SR-IOV interface (for example, **400 Gbps**), but RDMA further reduces latency and CPU utilization by offloading data transfer operations from the CPU, which is critical for workloads with very large data transfers.

Network configurations available in L40S clusters

- Pod network
- vNICs on different subnets

Network configurations available in H100 clusters

- Pod network
- vNICs
- SR-IOV NICs
- SR-IOV NICs with RDMA

IBM Cloud NIC creation procedures

This section outlines the steps for creating and attaching various network interface cards in IBM Cloud, covering both standard vNICs and high-performance SR-IOV NICs.

Creating a subnet

Before a vNIC can be created, it must be associated with a subnet. In the IBM Cloud console, this process begins under the **Virtual Private Cloud (VPC) Infrastructure** section by navigating to **Subnets** and selecting **Create** (Figure 1). You must then provide a unique name for the subnet, select the parent VPC, and choose a specific location (zone).

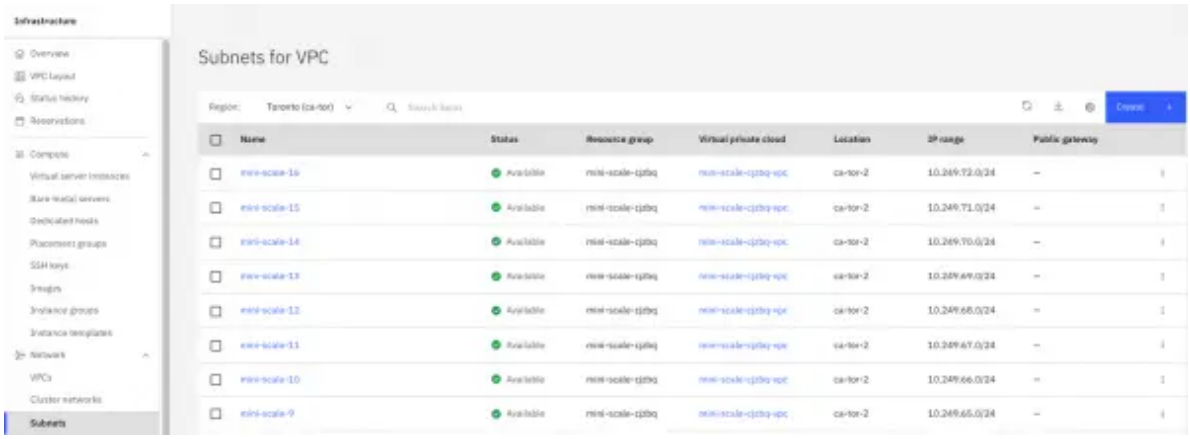

Figure 1: Attaching a vNIC to a Virtual Server Instance.

To enhance network connectivity for an existing Virtual Server Instance (VSI), additional vNICs can be created and attached. The process begins by navigating to the specific VSI within the IBM Cloud console (Figure 2).

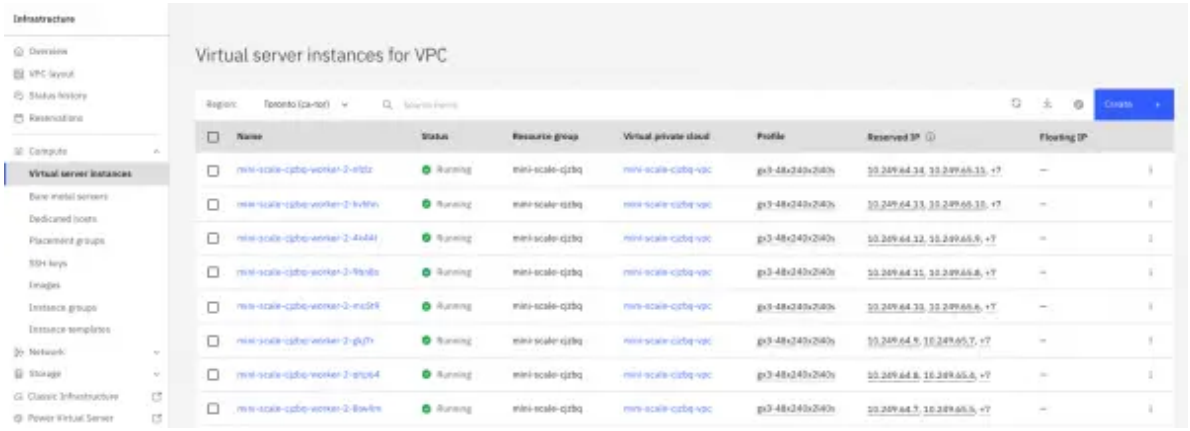

Figure 2: Viewing the virtual server instances in the IBM Cloud console.

In the instance's details page (Figure 3), select the **Network interfaces** tab on the **Networking** page, which lists all currently attached NICs. Here, you can initiate the creation of a new interface.

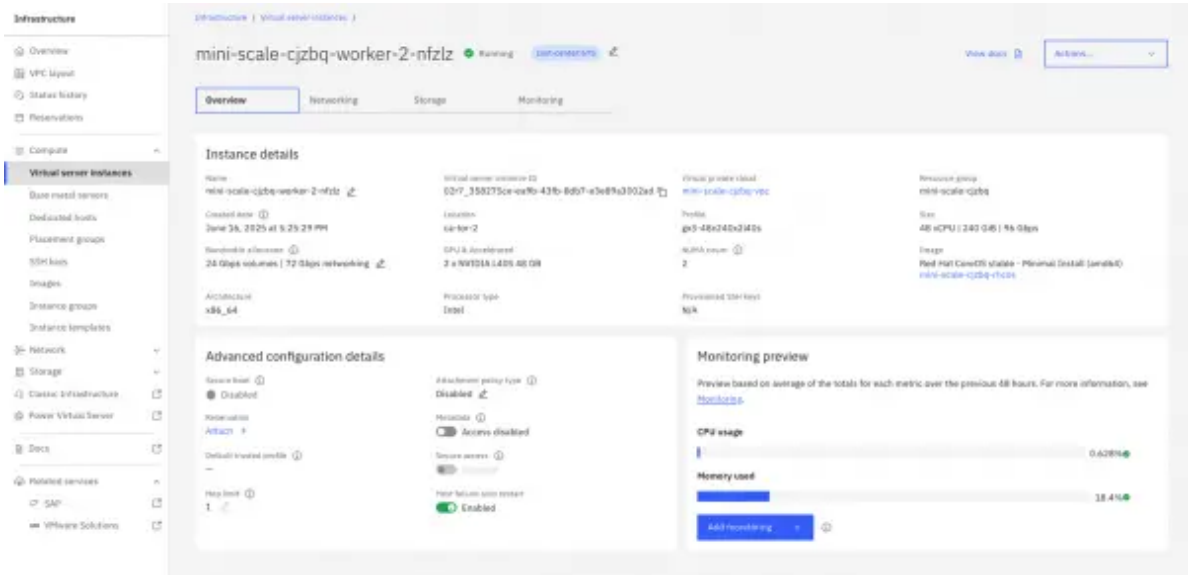

Figure 3: The virtual server instance details page overview.

The configuration requires specifying a name for the new vNIC and selecting a target subnet from an existing VPC. It's crucial that the chosen subnet is in the same zone as the VSI. Once the configuration is confirmed, the vNIC is provisioned and attached to the instance, appearing in the interface list and becoming available to the guest operating system for configuration. See Figure 4.

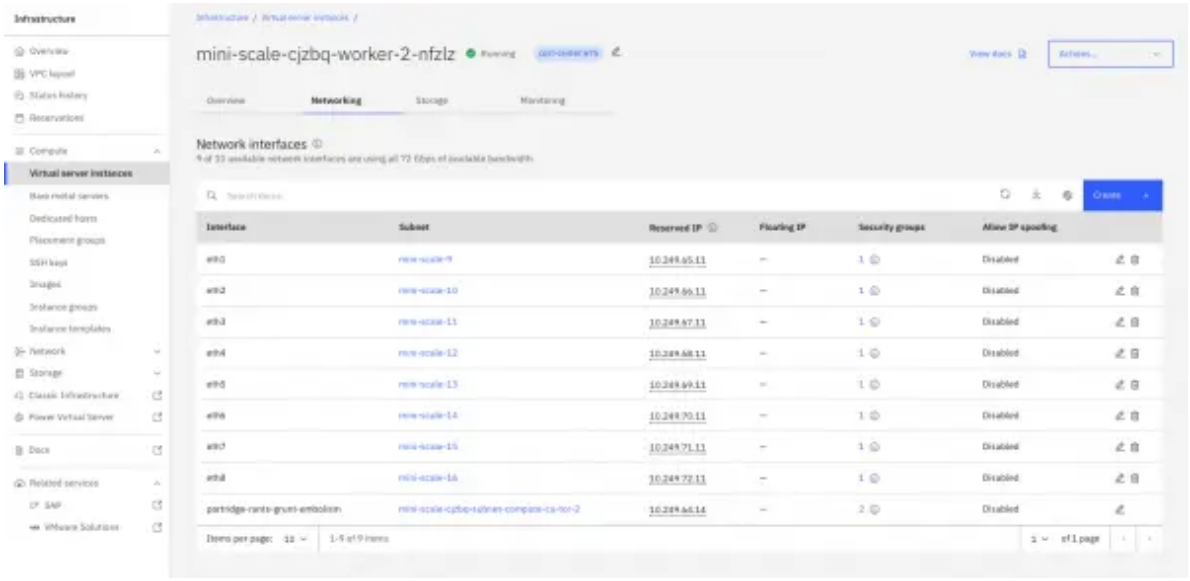

Figure 4: The interface list showing the new vNIC.

Procedure for adding SR-IOV NICs in IBM Cloud

For workloads that demand high performance and low latency, SR-IOV

can be enabled to provide direct hardware access. This capability must be enabled on the VSI profile during its initial creation. Once the VSI is provisioned with an SR-IOV capable profile, the process for attaching an SR-IOV network interface is similar to that of a standard vNIC.

Navigate to the **Cluster network attachments** tab of the VSI (Figure 5).

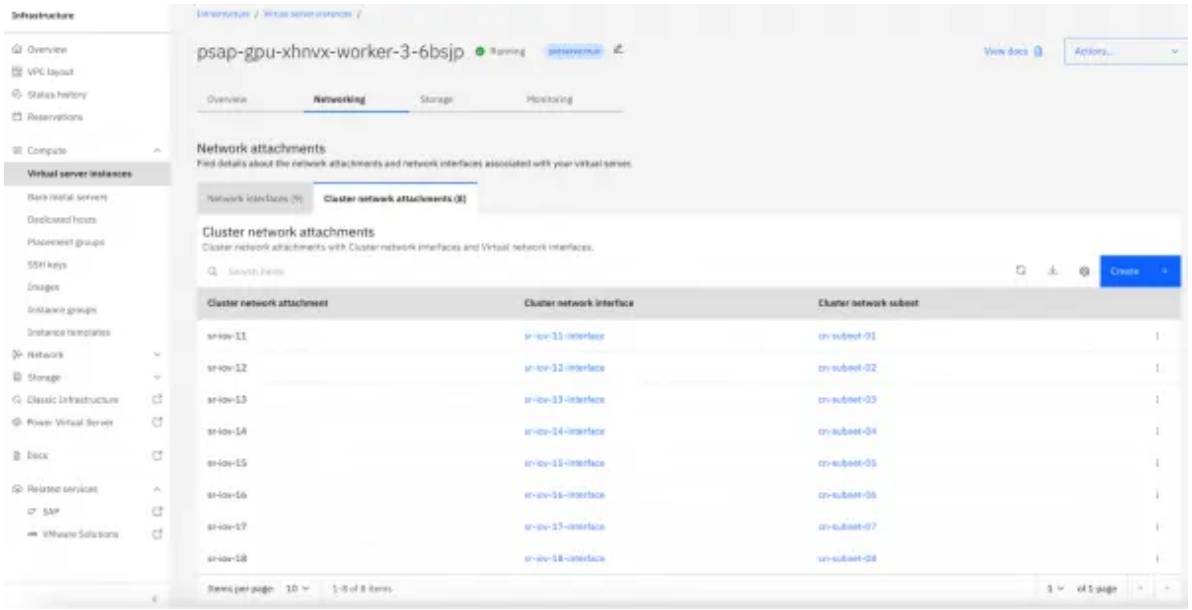

Figure 5: Viewing details about the network attachments and interfaces associated with the server.

Within the interface creation menu, an additional option or tab will be available for SR-IOV. From here, you can create the SR-IOV interface, which allocates a VF from the physical NIC directly to the virtual machine. This bypasses the hypervisor's virtual switch, significantly reducing network latency and CPU overhead, and exposes the interface to the guest operating system as a physical device ready for high-throughput communication.

Entrypoint script

The `run_ilab.sh` script dynamically configured the networking environment and launched the training process. The following sections highlight its key logic.

RDMA configuration

This block activates only when the `WITH_RDMA` variable is set. It detects the available SR-IOV NICs with RDMA capabilities and sets the necessary `NCCL` environment variables to ensure the PyTorch distributed back end uses the high-performance RDMA path for internode communication.

```
if [[ "${WITH_RDMA:-}" ]]; then
  export NCCL_TOPO_FILE=/mnt/storage/topo.xml
  num_rdma=$(ls /sys/class/infiniband/ | wc -l)
  IFS=',' read -ra ADDR <<< "$NCCL_SOCKET_IFNAME"
  length=${#ADDR[@]}
  NCCL_IB_HCA=''
  for idx in $(seq $((num_rdma-1)) -1 $((num_rdma-length))); 
    if [ -z "$NCCL_IB_HCA" ]; then
      NCCL_IB_HCA="mlx5_$idx"
    else
      NCCL_IB_HCA="$NCCL_IB_HCA,mlx5_$idx"
    fi
  done
  export NCCL_IB_HCA="$NCCL_IB_HCA"
  export NCCL_IB_DISABLE=0
  export NCCL_IB_GID_INDEX=3
  export NCCL_DEBUG=info
fi
```

Copy snippet

vNIC IP address remapping

This section runs when secondary vNICs or SR-IOV NICs are used (specified by the `NCCL_SOCKET_IFNAME` variable). It reads a predefined mapping file to dynamically reconfigure each pod's secondary network interfaces, assigning them the correct IP addresses and adding the necessary routes to enable communication across different subnets.

```
if [[ "${NCCL_SOCKET_IFNAME:-}" ]]; then
    MAPPING="$(cat /mnt/nic-mapping/nodename_ip_mapping.yaml)
    for ifname in $(echo $NCCL_SOCKET_IFNAME | tr , " "); do
```

```
        current_ip=$(ip route | grep "$ifname " | cut -d" " -
        correct_ip=$(echo "$MAPPING" | grep "$NODE_HOSTNAME"

        ip addr del "$current_ip/24" dev "$ifname"
        ip addr add "$correct_ip/24" dev "$ifname"

        while read remote_mapping; do
            remote_ip=$(echo "$remote_mapping" | cut -d: -f4)
            ip route add $remote_ip/32 via "$correct_ip" metr
        done <<< $(echo "$MAPPING" | grep -v "$NODE_HOSTNAME"
    done
fi
```

Copy snippet

IP address mapping file (nodename_ip_mapping.yaml)

The IP remapping logic relies on a pre-generated YAML file that is mounted into each pod. This file serves as the single source of truth for the network topology, mapping each node's hostname to its assigned secondary network interfaces and their correct IP addresses. The content of the file is stored inside a ConfigMap in the cluster:

```
apiVersion: v1
data:
  nodename_ip_mapping.yaml: |
    mini-scale-cjzbq-worker-2-4k44l:ens13:net1:10.249.65.9
    mini-scale-cjzbq-worker-2-4k44l:ens14:net2:10.249.66.9
    mini-scale-cjzbq-worker-2-4k44l:ens15:net3:10.249.67.9
    mini-scale-cjzbq-worker-2-4k44l:ens16:net4:10.249.68.9
    mini-scale-cjzbq-worker-2-4k44l:ens17:net5:10.249.69.9
...
kind: ConfigMap
metadata:
  name: ilab-nic-mapping
  namespace: fine-tuning-testing
```

Copy snippet

torchrun launch command

This is the final execution step. The script launches the distributed training job using `torchrun`. It dynamically builds the command-line arguments by parsing a `config.json` file, which contains all the hyperparameters for the training run, such as `--nnodes`, `--nproc_per_node`, and `--max_batch_len`.

```
config_json=$(jq . "$CONFIG_JSON_PATH")
if ! torchrun \
    --node_rank "${RANK}" \
    --rdzv_endpoint "${MASTER_ADDR}:${MASTER_PORT}" \
    $(echo "$config_json" | jq -r '. | to_entries | .[] | ("-
then
    ret=1
    echo "TORCHRUN FAILED :/ (retcode=$ret)"
fi
```

Copy snippet

Hyperparameter configuration (config.json)

The torchrun command is configured via a `config.json` file mounted into the pod. This allows for flexible and repeatable test execution. Below is an example configuration used in one of the test runs:

```
{
  "nnodes": 8,
  "nproc_per_node": 2,
  "module": "instructlab.training.main_ds",
  "model_name_or_path": "/mnt/storage/model/granite-3.1-8b-st
  "data_path": "/mnt/storage/dataset/data.jsonl",
  "output_dir": "/mnt/output/model",
  "num_epochs": 1,
  "effective_batch_size": 3840,
  "learning_rate": 0.0001,
  "num_warmup_steps": 800,
  "save_samples": 0,
  "log_level": "INFO",
```

```
  "max_batch_len": 35000,
  "seed": 42,
  "distributed_training_framework": "fsdp",
  "cpu_offload_optimizer": true,
  "cpu_offload_params": true,
  "fsdp_sharding_strategy": "HYBRID_SHARD"
}
```

Copy snippet

Network attachment definitions

To make secondary networks available to the training pods, OpenShift uses `NetworkAttachmentDefinition` (NAD) custom resources. These NADs define how the Container Network Interface (CNI) plug-ins should attach additional network interfaces to a pod. Different NADs were created for the vNIC and SR-IOV tests.

vNIC (host-device) attachment definition

For the standard vNIC tests, a `host-device` NAD was used. This configuration instructs the Multus CNI to take an existing network device on the host node (for example, `ens13`) and move it directly into the pod's network namespace, providing direct access to the physical network. The `whereabouts` IP Address Management (IPAM) plug-in was used to assign a static IP address to this interface from a predefined range.

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: network-port-01
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "network-port-01",
  "device": "ens13",
  "type": "host-device",
  "ipam": {
```

```
    "type": "whereabouts",
    "range": "10.241.129.0/24"
    }
}'
```

Copy snippet

SR-IOV with RDMA attachment definition

For the high-performance tests, the `host-device` CNI plug-in was also used, but with a specific custom flag to enable RDMA capabilities. This NAD takes a specific SR-IOV network device on the host node (for example, `enp223s0`) and passes it into the pod. The key difference is the `isRdma: true` flag, which signals to the underlying network fabric to enable RDMA (kernel-bypass) mode for this interface, providing the lowest possible latency for internode communication.

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: subnet-rdma-port-07
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "subnet-rdma-port-07",
  "device": "enp223s0",
  "type": "host-device",
  "ipam": {
    "type": "whereabouts",
    "range": "10.241.129.0/24"
    }
  "isRdma": true
}'
```

Copy snippet

Performance results and analysis

Our tests clearly demonstrate that network configuration is the single

most critical factor for achieving high GPU utilization and scalability in a distributed training environment.

L40S cluster results

The baseline performance was established with a single node, achieving a throughput of 2.04 samples per second. When scaling to multiple nodes, a clear distinction in performance emerged based on the network configuration used for internode communication. Using the default pod network, scaling to 2, 4, and 8 nodes yielded throughputs of 2.91, 3.73, and 5.46 samples per second, respectively. In contrast, using dedicated vNICs resulted in significantly better performance, with throughputs of 3.98, 7.16, and 12.68 samples per second for the same node counts. For each multi-node vNIC test, the number of secondary network interfaces used was equal to the number of nodes participating in the training job.

The performance gap widens as the node count increases, highlighting the pod network's limitations for high-bandwidth communication. The overhead from the virtual overlay network, which encapsulates all pod-to-pod traffic, introduces latency and consumes CPU cycles, creating a bottleneck that prevents the GPUs from being fully utilized. The dedicated vNICs provide a more direct, lower-latency path to the physical network, which is critical for the frequent gradient synchronization required by distributed training, thus enabling more efficient scaling.

The performance comparison is shown in Figure 6.

Figure 6: 2xL40S distributed training performance: throughput comparison by node count and network type.

H100 cluster results

With a single 8xH100 node, the training achieved a baseline throughput of **21.34 samples per second**. When scaling to two nodes, the results dramatically illustrated the impact of the network bottleneck. Both the default pod network (3.46 samples per second) and standard vNICs (12.28 samples per second) failed to scale, delivering significantly lower throughput than the single-node baseline.

This negative scaling occurs because the high latency and low bandwidth of these networks created a communication bottleneck so severe that the time spent waiting for gradient synchronization between nodes exceeded the computational gains from the additional GPUs. The FSDP training protocol attempts to hide communication overhead by overlapping computation with communication. For slower GPUs like the L40S, it was possible to hide the communication overhead of using vNICs, but for faster GPUs like the H100 it was not possible to hide the communication overhead of vNICs, which is why SR-IOV interfaces were necessary for better performance.

A profound performance increase was only achieved by using high-performance networking. Switching to SR-IOV interfaces yielded a throughput of **40.36 samples per second**, nearly doubling the single-node performance and confirming that direct hardware access is essential to prevent network bottlenecks with top-tier GPUs.

Enabling RDMA provided a final, marginal improvement to **40.58 samples per second**, suggesting that for this model size, the primary bottleneck was hypervisor overhead, which SR-IOV solves, rather than CPU involvement in data transfer, which RDMA addresses. For the 2-node

tests, each configuration used a single secondary network interface; further tests showed that increasing the number of interfaces to 2, 4, or 8 did not yield a significant change in throughput, indicating that a single high-bandwidth NIC was sufficient for this workload.

See Figure 7 for the 8xH100 distributed training performance comparison.

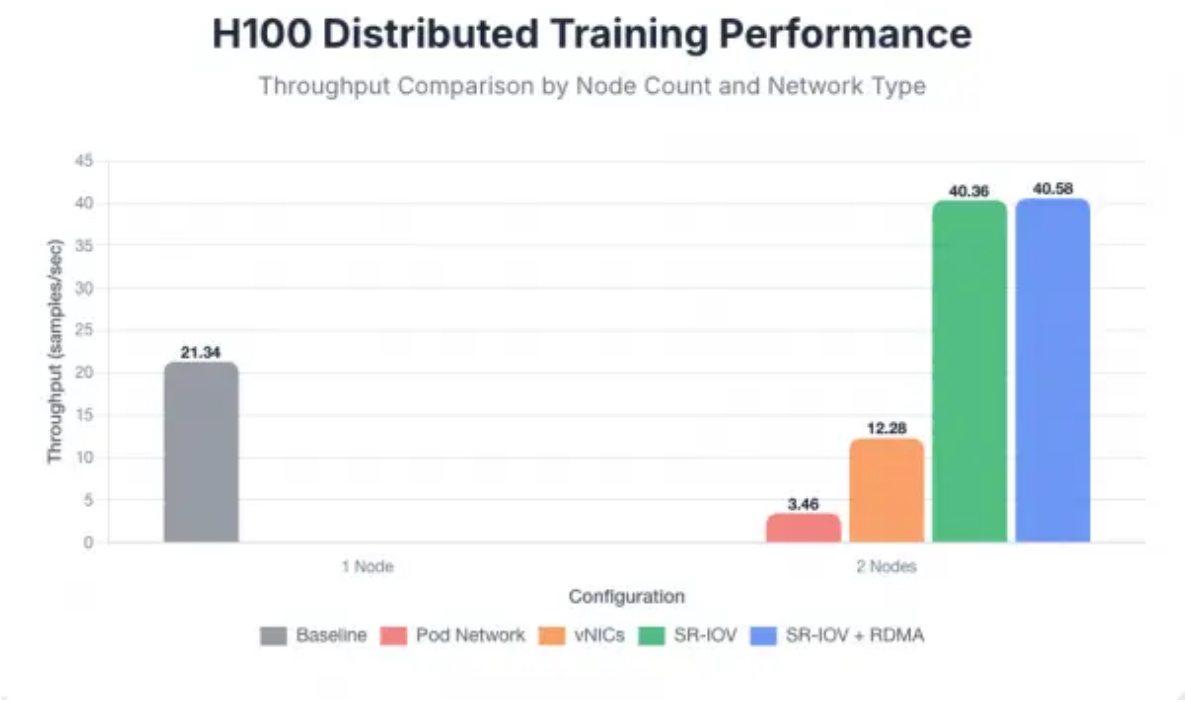

Figure 7: 8xH100 distributed training performance: throughput comparison by node count and network type.

Conclusions and strategic recommendations

The results from this comprehensive testing yield several key findings for optimizing distributed AI training on OpenShift.

First and foremost, the default OpenShift pod network is insufficient for high-performance distributed workloads, creating a significant bottleneck that worsens at scale. Utilizing secondary networks is essential.

For mid-tier GPU clusters like the L40S, standard vNICs provide a substantial and adequate performance uplift. However, for high-end clusters using H100 GPUs, the network becomes the primary limiting factor, and a high-throughput solution like SR-IOV is required to prevent GPU starvation and achieve optimal throughput.

GPU starvation and achieve optimal throughput.

Finally, while the benefits of RDMA were marginal for the 8B model tested, the data indicates its value will become critical when training larger models with more intensive internode communication requirements.

Recommendations

Based on the test results, the following strategic recommendations are proposed:

- **Avoid the pod network for multi-node training.** The default OpenShift SDN is not suitable for the high-bandwidth, low-latency communication required by distributed training. It should be avoided for any multi-node workloads.
- **Match network technology to GPU tier.** For clusters with mid-range GPUs (for example, L40S), standard vNICs offer a significant performance improvement and represent a cost-effective solution for achieving good scalability.
- **Mandate SR-IOV for high-end GPUs.** For clusters equipped with top-tier GPUs (for example, H100), SR-IOV is essential. The performance gains are substantial and necessary to prevent the network from becoming the primary bottleneck, ensuring that the GPU investment is fully utilized.
- **Plan for RDMA with larger models.** While not critical for the 8B model tested, RDMA should be considered a standard requirement when planning infrastructure for training much larger models, as its CPU-bypass capabilities will become crucial for preventing network bottlenecks.

Related Posts

Improve GPU utilization with Kueue in OpenShift AI

Optimize GPU utilization with Kueue and KEDA

GPU enablement on MicroShift