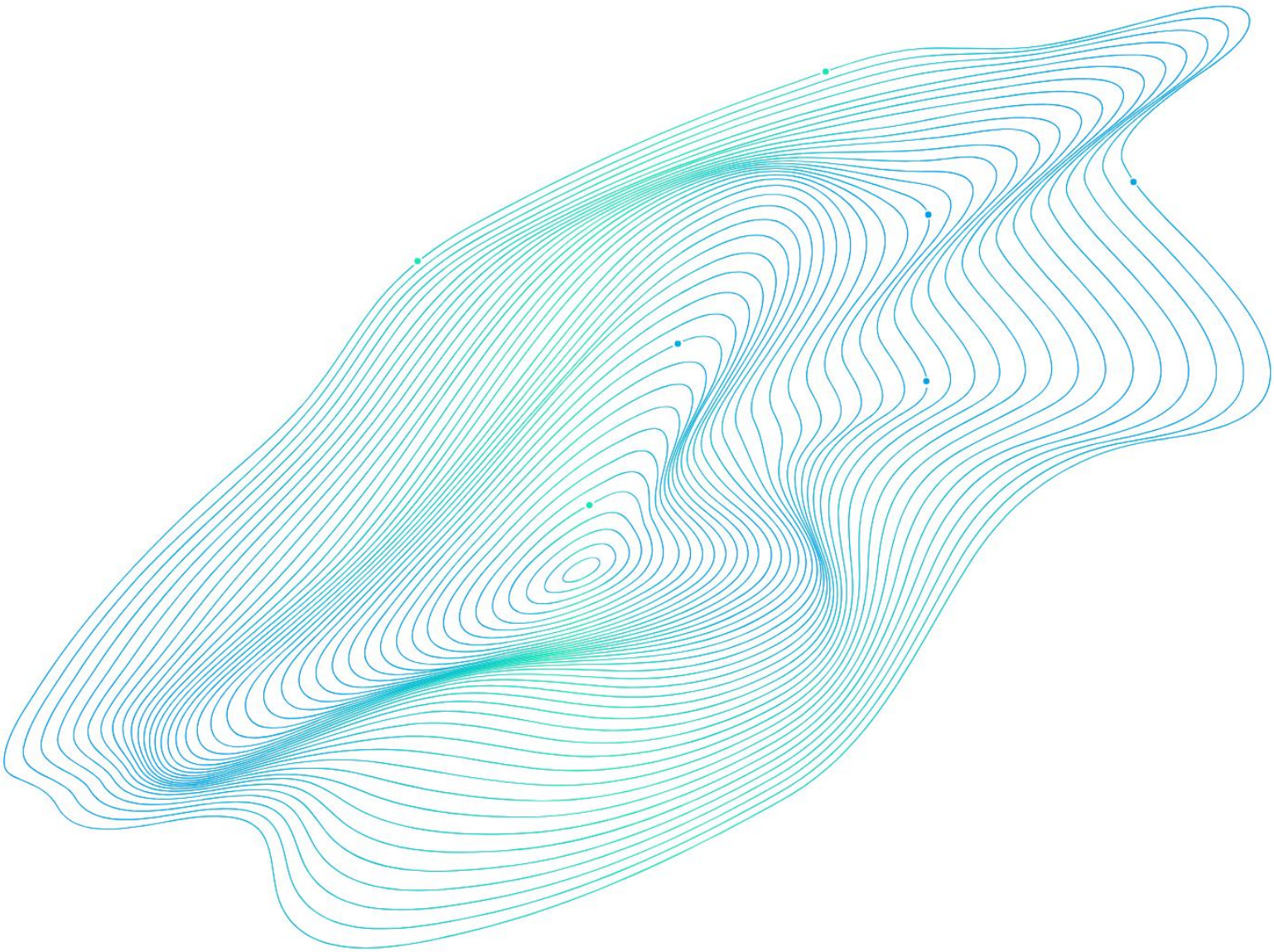
EPICURE

Unlocking European-level HPC Support

HIGH-LEVEL SPECIALISED APPLICATION SUPPORT SERVICE IN HIGH-PERFORMANCE COMPUTING (HPC)



Co-funded by the European Union

EuroHPC Joint Undertaking

This project has received funding from the European High Performance Computing Joint Undertaking under grant agreement No.101139786.

EPICURE

Best Practice Guide on Artificial Intelligence on EuroHPC Systems

Project Title	High-level specialised application support service in High-Performance Computing (HPC)
Project Acronym	EPICURE
Project Number	101139786
Type of Action	DIGITAL JU Simple Grants
Topic	DIGITAL-EUROHPC-JU-2022-APPSUPPORT-01-01
Starting Date of Project	2024-02-01
Ending Date of Project	2028-01-31
Duration of the Project	48 months
Website	epicure-hpc.eu
Document version	1.0
Document publication date	***2026-05-29***

Disclaimer

This project has received funding from the European High Performance Joint Undertaking under grant agreement No. 101139786. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or EuroHPC Join Undertaking. Neither the European Union nor the granting authority can be held responsible for them.

Executive Summary

This document provides practical guidance for users of EPICURE on running and scaling Artificial Intelligence (AI) workloads across EuroHPC supercomputing systems. It outlines the tools, frameworks and deployment approaches available for large-scale AI training. Next, it presents benchmark results on the scalability of different AI workloads on the different EuroHPC machines. In addition, the guide provides practical examples, job scripts and recommendations to help users efficiently deploy and scale AI applications on EuroHPC infrastructure.

Version 1.0 – Incomplete Benchmarks

This document contains the results of benchmarks run across all current EuroHPC systems. Due to a combination of queue congestion, operational downtime and challenges in benchmark deployment, a small subset of benchmark results was not yet available at the time of publication. We intend to update this document with the full set of benchmark results as soon as they are finalized. The following benchmarks are still to be completed:

- **Discoverer**: all benchmarks
- **JUPITER**: all benchmarks
- **Leonardo**: large-scale language benchmarks
- **Vega**: some vision benchmarks, all language benchmarks
- **MeluXina**: language benchmarks

For **LUMI**, **MareNostrum 5**, **Karolina** and **Deucalion** the results are complete. We target the end of June for publication of a complete set of results.

Table of Contents

1. Introduction

This Best Practice Guide on Artificial Intelligence on EuroHPC Systems provides an overview of how users can run and scale their Artificial Intelligence (AI) workloads on any of the currently active EuroHPC supercomputers. It describes the tools and methods available to implement AI workloads at extreme scale.

By leveraging HPC infrastructure for scaling, users can speed up or extend their AI workloads. Using a set of benchmarks, we evaluate the efficiency of such scaling. This efficiency may differ between supercomputers, as it is affected by the system architecture. These results can aid the user in determining the most appropriate systems for a given workload.

As such, this guide includes example job scripts and benchmark outputs collected from multiple EuroHPC machines. These resources are shared on EPICURE's BPG-AI-workloads GitLab repository, enabling users to implement and scale their workloads. Based on these results, we derive practical guidelines for running and scaling AI workloads on EuroHPC systems.

2. Overview of the Benchmarks

We ensured that the selection of benchmarks for this guide was representative of contemporary AI workloads in HPC settings. We selected application fields that are commonly supported within the EPICURE project. Furthermore, we ensured that any frameworks the benchmarks are built on are widely used today, actively maintained, sufficiently mature and supported on all EuroHPC systems. We selected two distinct applications, which together broadly cover the field of AI on HPC.

In the **vision** benchmark, a model based on Convolutional Neural Networks (CNNs) is trained to perform image classification using a dataset of over 1.3 million images spanning 1000 classes. Then, in the **language** benchmark, a Large Language Model (LLM) is pretrained using publicly available data. Both benchmarks are implemented in Python and rely on the ubiquitous PyTorch framework. The language benchmark specifically is implemented with the help of the Megatron library for PyTorch. For scaling, we rely on built-in functionality of PyTorch and Megatron.

The two benchmarks were selected to cover different challenges for AI on HPC. Specifically, the vision benchmark focuses on I/O pressure, while the language benchmark evaluates different approaches and challenges for model distribution and scaling.

Given the scale of these benchmarks, it is entirely infeasible to perform training to completion for each configuration on each system. As such, we only train long enough for the *throughput* (i.e., the training speed) to stabilize, as training usually starts with a slower “warmup” phase. Using this throughput, we determine the *scaling efficiency* for each number of nodes evaluated, where a scaling efficiency of 1 indicates perfect linear scaling. We acknowledge that scaling efficiency mainly deteriorates when model weights are synchronized between GPUs, meaning that it would be possible to artificially improve scaling performance by performing such synchronization less frequently. This would increase the time until the model is trained to satisfaction, but as we cannot realistically measure this, we take special care to ensure that all benchmarks are optimized for *training efficiency* rather than raw *scaling efficiency*.

2.1. Vision

The image classification problem considered in this benchmark is well-known and well-understood. We use the ResNet-50 architecture, an industry-standard architecture for this application[1]. As dataset, we rely on the equally ubiquitous ImageNet-1k dataset, containing over 1.28 million images for training, totalling around 140 GB. The benchmark code can be found in our repository and access to the dataset can be requested on its website.

This model is relatively small, meaning that each GPU can hold hundreds of copies in memory. As such, a *batch* of images (up to 512 in this case) can be processed in parallel. After each such batch, GPUs exchange their gradients such that each can perform the same parameter updates to the model.

[1] Nowadays, transformers are also often used. We chose a CNN-based solution to further distinguish this benchmark from the transformer-based language benchmark.

The relatively small size of the model and large size of input data (i.e., images) result in this training exhibiting significant storage I/O requirements, further exacerbated by reads being randomized throughout the dataset. We mitigate this through an in-RAM dataset.

We perform the first five epochs (i.e., full passes over the dataset) of training in the benchmark. To eliminate any performance impact from warmup or shutdown, we only consider the performance of the fourth epoch. Due to the synchronization at the end of each batch, sporadic slowdown in *any* GPU affects *all* GPUs. As such, we sample the *images-per-second* throughput of each GPU during each batch of images and take its median, filtering out such occasional slowdown. In addition, we measure the runtime of the full epoch, including all slowdown that occurred within it.

2.2. Language

The second benchmark considers pretraining an LLM. We pretrain a model based on the industry-standard GPT-3 architecture, using a subset of the FineWeb dataset (around 700 million of the full 18.5 trillion tokens). The benchmark code is again available in our repository and the dataset can be found on HuggingFace (download script), to be used together with pre-built vocabulary and merge files.

The full training state of the model, consisting of around 175 billion parameters, spans several terabytes, meaning over 100 GPUs are needed for a single copy of the model. Proper distribution is achieved through a combination of different approaches, detailed in Section 4.2. To ensure that this benchmark can also be run on smaller EuroHPC systems, we additionally consider a smaller model, whose 857 million parameters fit into a single GPU.

When scaling to even more GPUs, multiple parallel copies of the model are used. At this point, it takes prohibitively many GPU-hours just for the *throughput* of training to stabilize. As such, a snapshot is taken after processing 23.5 million tokens with a single copy of the model, which can be replicated at larger scales, leading to stabilization of training throughput within minutes.

Once throughput has stabilized, we observe the throughput in TFLOP/s per GPU. The framework reports this once per step, and the value remains consistent between steps.

3. EuroHPC Systems

As part of the EPICURE project, we have access to all currently active EuroHPC supercomputing systems across participating sites. This unique collaboration enables us to run benchmarks and evaluate AI scaling performance on each of these systems, ensuring that the information and examples provided in this guide reflect real, up-to-date usage across the entire EuroHPC landscape.

In this chapter, we present an overview of each EuroHPC system included in our study. We provide an overview of all system partitions, noting that our benchmarks were run solely on the GPU partitions. Furthermore, we summarize how AI jobs relying on Python are recommended to be run, and how we ran our benchmarks specifically. This context will help users understand the capabilities and differences between systems, and how to apply the practical examples shared in this guide to their own jobs.

We also name the network interconnects and topologies used in each system. The simplest topology used is the *fat tree*, in which all nodes are connected in a hierarchical tree-like topology. Links closer to the root are given a higher capacity, as these are used more often. A more complex topology is *Dragonfly*, in which nodes are divided into groups. All nodes within a group are densely connected, along with sparser inter-group connections. Finally, with *Dragonfly+,* the dense intra-group connections are replaced with a scalable *spine-leaf* topology. These topologies are more scalable and, especially with Dragonfly+, cost-efficient for large deployments. The fat-tree topology instead offers simpler and more predictable routing. Given the extremely high-throughput communication bursts that occur in AI workloads, such as with parameter exchanges or all-reduces, a highly performant interconnect is crucial to scaling AI workloads.

3.1. LUMI

LUMI is a pre-exascale EuroHPC supercomputer, supplied by HPE and in production since 2022. It is hosted by CSC in its Kajaani data centre in Finland and offers a sustained performance of 386 petaflops.

Specifications

GPU partition (LUMI-G): 2978 nodes (see Figure 3-1)

- GPU: 4x AMD MI250x “Aldebaran” (128 GiB each)
- CPU: 1x AMD EPYC 7A53 "Trento" (64c, 2.0 GHz[2])
- RAM: 512 GiB
- SSD: none
- Interconnect: 4x HPE Cray Slingshot-11 (200 Gbps each) [Dragonfly]

[2] All reported clock speeds are base clock speeds. Higher speeds may be achieved.

CPU partition (LUMI-C): 2048 nodes

- CPU: 2x AMD EPYC 7763 “Milan” (64c each, 2.45 GHz)
- RAM: 256 GiB [1888 nodes] or 512 GiB [128 nodes] or 1 TiB [32 nodes]
- SSD: none
- Interconnect: 1x HPE Cray Slingshot-11 (200 Gbps) [Dragonfly]

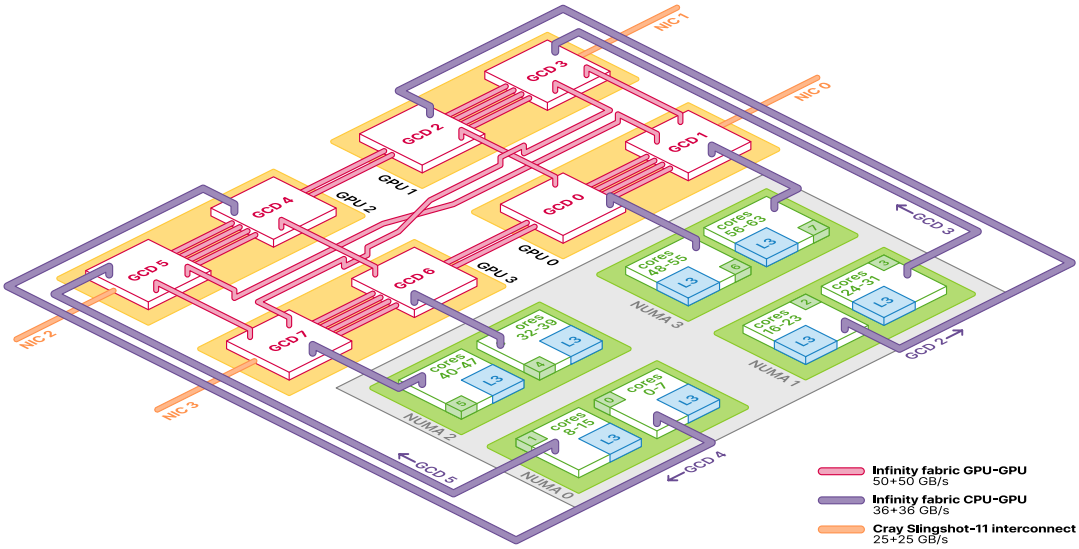


Figure 3-1: Overview of a LUMI-G compute node

Interactive data-analytics partition (LUMI-D): 16 nodes

- GPU: 8x NVIDIA A40 (48 GiB each) [8 nodes] or none [8 nodes]
- CPU: 2x AMD EPYC 7742 “Rome” (64c each, 2.25 GHz)
- RAM: 2 TiB [8 nodes] or 4 TiB [8 nodes]
- SSD: 14 TB [8 nodes] or 25 TB [8 nodes]
- Interconnect: 2x HPE Cray Slingshot-11 (200 Gbps each) [Dragonfly]

Additional info

Python usage

LUMI proposes several methods to run Python plus packages.

- A fully node-local installation on `/tmp`, which is mounted in RAM.
- In case of very few packages: A pre-existing container with Python, or Python from the `cray-python` module, plus packages installed on the file system through a virtual environment.
- A container built from a Conda environment file or pip requirements file using the LUMI container wrapper or cotainr.
- The PyTorch or JAX container from the LUMI Software Library which provide helper scripts to install additional pip packages on demand in a single SquashFS file, avoiding stressing the file system.

- One of the LUMI AI Factory containers, coming with many popular AI packages pre-installed.

In terms of performance and flexibility, the final two options are recommended. These containers are preconfigured to make optimal use of LUMI's architecture. Other containers may need additional plugins and binds to make optimal use of the Slingshot interconnects and GPUs. The LUMI AI Factory (LAIF) containers are intended to supersede those from the LUMI Software Library, and no new versions for the latter will be provided. However, while these benchmarks were being run, the LAIF containers were still experimental, and as such the older containers were used, as these offered stronger stability. Notably, at the time of writing, the LAIF containers do not offer a performant and convenient mechanism to add Python packages, instead requiring a full container rebuild.

Links

- Main page: https://www.lumi-supercomputer.eu
- Documentation: https://docs.lumi-supercomputer.eu

3.2. Leonardo

Leonardo is a next-generation pre-exascale Tier-0 supercomputer, part of the EuroHPC Joint Undertaking, in production since August 2023, offering a sustained 249 petaflops. It is hosted by CINECA at the Bologna Technopole in Italy and it is developed and supplied by EVIDEN ATOS.

Specifications

Booster Partition: 3456 nodes

- GPU: 4x NVIDIA *custom* A100 "Ampere" (64 GiB HBM2 each)
- CPU: 1x Intel Xeon Platinum 8358 "Icelake" (32c, 2.60 GHz)
- RAM: 512 GiB
- SSD: none
- Interconnect: 2x InfiniBand HDR[3] (200 Gbps each) [Dragonfly+]

Data Centric General Purpose (DCGP) Partition: 1536 nodes

- CPU: 2x Intel Xeon Platinum 8480+ "Sapphire Rapids" (56c, 2.00 GHz each)
- RAM: 512 GiB
- SSD: 4 TB
- Interconnect: 1x InfiniBand HDR100 (100 Gbps) [Dragonfly+]

[3] The InfiniBand HDR (also "HDR 200G") generation is predominantly implemented by the NVIDIA ConnectX-6 hardware (manufactured by Mellanox before acquisition by NVIDIA). The generation supports at most 200 Gbps, and hardware only supporting 100 Gbps may be called "HDR100".

Additional info

Python usage

A guide previously published by EPICURE explains how to run Python AI workloads on Leonardo without using containers. For these benchmarks we do use containers. For the vision benchmark, we used the NeMo container from NGC which we overlay with a virtual environment. For the language benchmark, some customisations to the run script were needed, as Leonardo experiences known issues with NCCL versions between 2.22 and 2.29.1. Addressing this requires setting `NCCL_PXN_DISABLE=1` and using `torchrun` instead of `srun`. As container, we used the NGC PyTorch container overlaid with a Megatron installation, as shown in this setup script.

Links

- Main page: https://leonardo-supercomputer.cineca.eu
- Documentation: https://docs.hpc.cineca.it/index.html

3.3. MareNostrum 5

MareNostrum 5 is a pre-exascale EuroHPC supercomputer hosted at BSC-CNS (Spain), in production since July 2024. It is supplied by Bull SAS, combining Lenovo ThinkSystem SD650 V3 and Eviden BullSequana XH3000 architectures, providing two partitions with different technical characteristics, reaching a sustained 215 petaflops.

Specifications

MareNostrum 5 ACC (Accelerated Partition): 1120 nodes

- GPU: 4x NVIDIA H100 “Hopper” (64 GiB HBM2 each)
- CPU: 2x Intel Xeon Platinum 8460Y+ “Sapphire Rapids” (40c each, 2.00 GHz)
- RAM: 512 GiB
- SSD: 480 GB
- Interconnect: 4x InfiniBand NDR200[4] (200 Gbps each) [Fat tree]

MareNostrum 5 GPP (General Purpose Partition): 6408 nodes

- CPU: 2x Intel Xeon Platinum 8480+ “Sapphire Rapids” (56c each, 2.00 GHz)
- RAM: 256 GiB [6192 nodes] or 1 TiB [216 nodes] or 2 TiB [10 nodes] or 128 GiB HBM [72 nodes]
- SSD: 960 GB
- Interconnect: 0.5x InfiniBand NDR200 (200 Gbps[5]) [Fat tree]

[4] Similarly, InfiniBand NDR is implemented by the NVIDIA ConnectX-7 hardware. It supports up to 400 Gbps, with hardware at a reduced 200 Gbps being referred to as “NDR200”.
[5] Except for the 10 nodes with 2 TiB RAM, each interconnect is shared by 2 nodes, leading to an effective speed of 100 Gbps per node

Additional info

Python usage

Users are recommended to use containers for Python. For these benchmarks, we built custom containers (for vision and language) based on, respectively, PyTorch's container with CUDA and the NVIDIA NGC PyTorch container. Either base container would be appropriate for any similar application.

Links

- Main page: https://www.bsc.es/marenostrum/marenostrum-5
- Documentation: https://www.bsc.es/supportkc/docs/MareNostrum5/intro/

3.4. MeluXina

MeluXina is a petascale supercomputer hosted at LuxProvide in Luxembourg, in production since November 2021. The supercomputer is based on Atos Sequana XH2000 and achieves a sustained 12.8 petaflops.

Specifications

GPU partition: 200 nodes

- GPU: 4x NVIDIA A100 "Ampere" (40 GiB HBM2 each)
- CPU: 2x AMD EPYC 7452 "Rome" (32c each, 2.3 GHz)
- RAM: 512 GiB
- SSD: 1.92 TB
- Interconnect: 2x InfiniBand HDR (200 Gbps each) [Dragonfly+]

CPU partition: 593 nodes

- CPU: 2x AMD EPYC 7H12 "Rome" (64c each, 2.6 GHz),
- RAM: 512 GiB [573 nodes] or 4 TiB [20 nodes]
- SSD: None [573 nodes] or 1.92 TiB [20 nodes]
- Interconnect: 1x [573 nodes] or 2x [20 nodes] InfiniBand HDR (200 Gbps each) [Dragonfly+]

FPGA partition: 20 nodes

- FPGA: 2x BittWare 520N-MX (Intel Stratix 10MX chip, 16 GiB HBM2 each)
- CPU: 2x AMD EPYC 7452 "Rome" (32c each, 2.3 GHz)
- RAM: 512 GiB
- SSD: 1.92 TiB
- Interconnect: 2x InfiniBand HDR (200 Gbps each) [Dragonfly+]

Additional info

Python usage

Several containers from NGC are available as modules on MeluXina, and instructions on building custom containers are provided. For these benchmarks, two custom containers were built (vision).

Links

- Main page: https://www.luxprovide.lu/meluxina/
- Documentation: https://docs.lxp.lu

3.5. Karolina

Karolina is a petascale system hosted by the IT4Innovations National Supercomputing Center (Czech Republic). It is a HPE Apollo (Apollo 200 and Apollo 6500) system, reaching a sustained 9.6 petaflops, in operation since Q2 of 2021.

Specifications

GPU partition: 72 nodes

- GPU: 8x NVIDIA A100 “Ampere” (40 GiB HBM2 each)
- CPU: 2x AMD EPYC 7763 “Milan” (64c each, 2.45 GHz)
- RAM: 1024 GiB
- SSD: None
- Interconnect: 4x InfiniBand HDR (200 Gbps) [Fat tree]

CPU partition: 720 nodes

- CPU: 2x AMD EPYC 7H12 “Rome” (64c each, 2.6 GHz[6])
- RAM: 256 GiB
- SSD: None
- Interconnect: 1x InfiniBand HDR100 (100 Gbps) [Fat tree]

Additional info

Python usage

PyTorch and JAX are available as modules. For maximal performance, users are recommended to provide their own containers based on containers distributed by NVIDIA. For the benchmarks, custom containers (vision, language) based on NVIDIA’s cuDNN RockyLinux8 container were built. However, building a custom container using a leaner base container may be faster and easier, at no performance impact.

[6] Karolina reduces the maximum frequency on the CPUs of the CPU partition by about 20%. Reported clock speeds here are base factory speeds.

Links

- Main page: https://www.it4i.cz/en/infrastructure/karolina
- Documentation: https://docs.it4i.cz/en/docs/clusters/karolina/introduction

3.6. Discoverer

This petascale supercomputer is hosted by Sofia Tech Park (Bulgaria) and has been in production since September 2021, with a sustained 4.5 petaflops. The CPU partition is based on Atos Sequana XH2000, while the GPU partition consists of NVIDIA DGX H200 machines.

Specifications

Discoverer+ GPU partition: 4 nodes

- GPU: 8x NVIDIA H200 “Hopper” (141 GiB each)
- CPU: 2x Intel Xeon 8480c “Sapphire Rapids” (56c each, 2.00 GHz)
- RAM: 1.96 TiB
- SSD: 8x 3.84 TB
- Interconnect: 10x InfiniBand NDR (400 Gbps each) [Dragonfly+]

CPU partition: 1128 nodes

- CPU: 2x AMD EPYC 7H12 “Rome” (64c each, 2.6 GHz)
- RAM: 256 GiB [1110 nodes] or 1TiB [18 nodes]
- SSD: none
- Interconnect: 1x InfiniBand HDR (200 Gbps) [Dragonfly+]

Additional info

Python usage

A PyTorch module is available, although it is not intended for use with GPUs. Custom containers are recommended.

Links

- Main page: https://discoverer.bg
- Documentation: https://docs.discoverer.bg/index.html

3.7. Vega

This system is in production since April 2021 and hosted by IZUM (Slovenia). The petascale supercomputer achieves a sustained 6.9 petaflops and is based on Atos Sequana XH2000.

Specifications

GPU partition: 60 nodes

- GPU: 4x NVIDIA A100 “Ampere” (40 GiB each)
- CPU: 2x AMD EPYC 7H12 “Rome” (64c each, 2.6 GHz)
- RAM: 512 GiB
- SSD: 1.92 TB
- Interconnect: 2x InfiniBand HDR100 (100 Gbps each) [Dragonfly+]

CPU partition: 960 nodes

- CPU: 2x AMD EPYC 7H12 “Rome” (64c each, 2.6 GHz)
- RAM: 256 GiB [768 nodes] or 1 TiB [192 nodes]
- SSD: 1.92 TB
- Interconnect: 1x InfiniBand HDR100 (100 Gbps) [Dragonfly+]

Additional info

Python usage

Users are recommended to use containers, built on top of images from NVIDIA NGC. For these benchmarks, custom containers were built using the NGC PyTorch image.

Links

- Main page: https://www.izum.si/en/hpc-en/
- Documentation: https://doc.vega.izum.si

3.8. Deucalion

Deucalion is a petascale EuroHPC supercomputer, supplied by Fujitsu (currently Fsas Technologies) and in production since June 2024. It reaches 7.5 petaflops sustained and is hosted by FCT at Universidade do Minho in Guimarães, Portugal.

Specifications

GPU partition: 33 nodes

- GPU: 4x Nvidia A100 “Ampere” (40 GiB [17 nodes] or 80 GiB [16 nodes] each)
- CPU: 2x AMD EPYC 7742 “Rome” (64c each, 2.25 GHz)
- RAM: 512 GiB
- SSD: 480 GB
- Interconnect: 2x InfiniBand HDR (200 Gbps each) [Fat tree]

CPU (A64FX) partition: 1632 nodes

- CPU: 1x Fujitsu A64FX (48c, 2.0 GHz)
- RAM: 32 GiB
- SSD: 512 GB

- Interconnect: 1x InfiniBand HDR100 (100 Gbps) [Fat tree]

CPU (x86) partition: 500 nodes

- CPU: 2x AMD EPYC 7742 “Rome” (64c each, 2.25 GHz)
- RAM: 256 GiB
- SSD: 480 GB
- Interconnect: 1x InfiniBand HDR100 (100 Gbps) [Fat tree]

Additional info

Python usage

The Deucalion documentation advises using containers, virtual environments or Conda environments for Python. For our benchmarks, we used a virtual environment (based on requirements files for vision and language). A recent version of Python is available as a module (3.12 at the time of writing), and running on GPUs across multiple nodes requires loading the CUDA, cuDNN and NCCL modules (versions 12.6.0, 9.0.50 and 2.22.3 at the time of writing).

Links

- Main page: https://deucalion.acnca.pt
- Documentation: https://docs.deucalion.macc.fccn.pt

3.9. JUPITER

JUPITER, the “Joint Undertaking Pioneer for Innovative and Transformative Exascale Research", will be the first exascale (sustained 1 exaflop) supercomputer in Europe. The system is provided by a ParTec-Eviden supercomputer consortium and was procured by EuroHPC JU in cooperation with the Jülich Supercomputing Centre (JSC). It is installed in the Forschungszentrum Jülich campus in Germany. The Booster partition consists of NVIDIA GH200 “Grace-Hopper” superchips as shown in Figure 3-2.

Specifications

JUPITER Booster: 5884 nodes

- GPU: 4x NVIDIA H100 “Hopper” (96 GiB HBM3 each)[7]
- CPU: 4x NVIDIA Grace (Arm Neoverse-V2) (72c each, 3.1 GHz)
- RAM: 480 GiB LPDDR5X
- SSD: No
- Interconnect: 4x InfiniBand NDR200 (200 Gbps each) [Dragonfly+]

[7] Each JUPITER Booster node contains 4 NVIDIA GH200 superchips, each consisting of 1 GPU, 1 CPU and 120 GiB RAM.

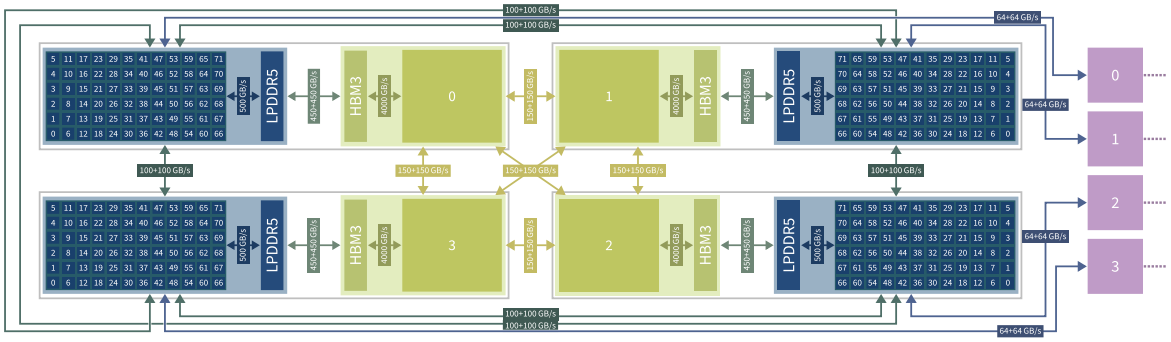


Figure 3-2: Node diagram of the 4× NVIDIA GH200 node design of JUPITER Booster.

Additional info

Python usage

As JUPITER is still in early access, no guidelines on Python usage have been made available yet. For these benchmarks, we used custom containers.

Links

- Main page: https://jupiter.fz-juelich.de/
- Documentation: https://apps.fz-juelich.de/jsc/hps/jupiter/

4. Tools and Systems

In this chapter, we describe the tools (i.e., software) and systems (i.e., algorithms) that were leveraged within the benchmarks of this Best Practice Guide. Many other tools and systems were considered, but ultimately not used. These are described in Appendix A.

4.1. Frameworks

PyTorch

Usage metrics, large-scale surveys, and analysis of projects supported within EPICURE consistently show that PyTorch is the dominant framework for Deep Learning in the current age, as it has been for years. Initially created by Meta and now governed by the PyTorch Foundation, PyTorch is a robust, high-performing and extensive framework for Deep Learning, with a plethora of more specific libraries being built on top of it. It remains under highly active development, is widely supported among EuroHPC systems, optimizing its performance on specific systems is well understood, and a wide array of example code is available. As such, all benchmarks are implemented using PyTorch.

Megatron

While the vision benchmark was implemented in pure PyTorch, the language benchmark requires the addition of the Megatron library, built on top of PyTorch. It offers implementations of state-of-the-art LLM architectures, advanced parallelism strategies (see Section 4.2) and optimizations needed for LLM training at scale. It is arguably the only such library which is both sufficiently battle-tested and under active development. It is developed by NVIDIA, with AMD maintaining a fork ensuring compatibility with their GPUs.

4.2. Distribution

Both weak scaling (“doing more in the same amount of time”) and strong scaling (“doing the same amount of work in less time”) are useful in an AI context. With weak scaling, hyperparameter sweeps can be made more extensive. Strong scaling makes training massive LLMs feasible, as this takes months even on large clusters. Regardless of the type, different approaches to the distribution needed can be taken.

Data Parallelism

The simplest type of parallelism is *data parallelism*, shown in Figure 4-1. In this case, multiple copies of the model, on different devices[8], each run on different datapoints. In

[8] Within the context of EuroHPC supercomputers, these *devices* across which models are distributed will usually be GPUs

a sense, this is a multi-device extension of the *minibatch* concept, in which multiple datapoints are processed in parallel on *one* device. Once all devices have completed their minibatch, the computed gradients are exchanged through an **`all-reduce`** operation, after which each device can perform the same update to the model.

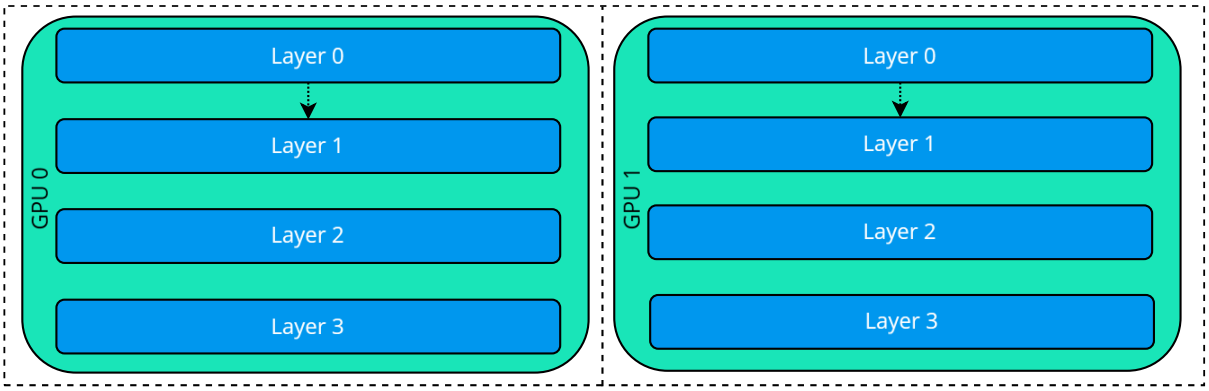


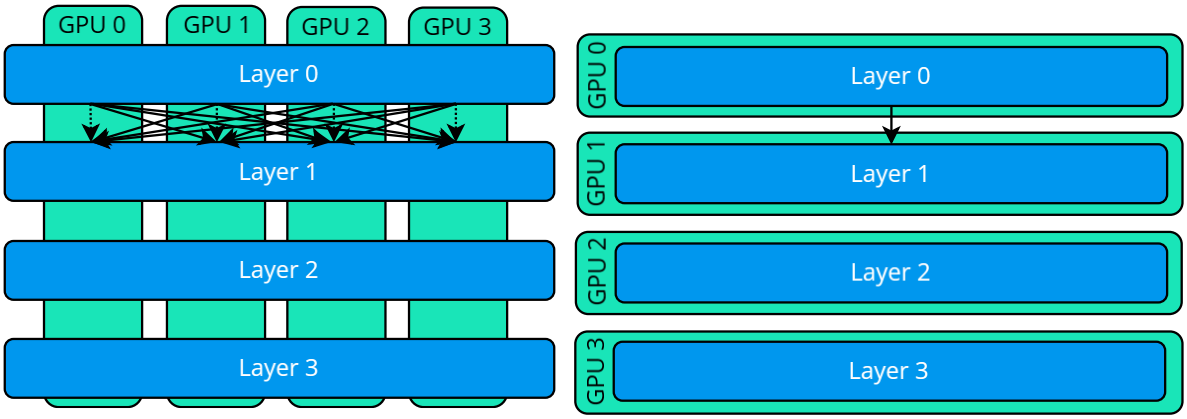
Figure 4-1: With data parallelism, each device contains a full copy of the model, denoted by dashed boxes. The different GPUs only need to communicate at the end of the batch to perform weight updates.

Overheads introduced by this approach are limited to waiting times between the first and last device finishing minibatch processing, and the **`all-reduce`** operation. PyTorch offers this type of parallelism out-of-the-box with its Distributed Data Parallel (DDP) module.

In terms of scaling efficiency, this is the most performant type of parallelism. As long as the full model fits into the memory of a single device, there is no incentive to incorporate other types of parallelism. As such, the vision benchmark and the small language benchmark rely solely on data parallelism.

Model Parallelism

When the model no longer fits in a single device, *model parallelism* becomes necessary to train the model at all. The two main types of model parallelism are **tensor parallelism** and **pipeline parallelism**.



(a) Tensor parallelism (b) Pipeline parallelism

Figure 4-2: With model parallelism, the model is divided into separate parts and divided across the GPUs. Solid lines denote activations communicated between devices. Dotted lines are activations staying within one device. At the cost of increased communications, all GPUs can immediately start calculations with tensor parallelism, in contrast with pipeline parallelism, where later GPUs need to wait for the result of earlier GPUs before starting calculations.

With tensor parallelism, shown in Figure 4-2(a), *each* layer of the model is divided across devices, such that each device contains a part of each layer. This significantly increases communication overhead, as intermediate outputs must frequently be exchanged between devices.

In contrast, pipeline parallelism, shown in Figure 4-2(b), assigns layers to one device each, meaning that each device contains only some of the layers, but holds those layers in full. The communication overhead for this type of parallelism is less severe, as output generated by a device only needs to be sent to the one device running the next layer. However, this approach inevitably leads to *pipeline bubbles*, meaning devices will occasionally idle while waiting for input from a previous device.

For extremely large models, these two types of parallelism are often combined, meaning that *each layer* is divided across a *subset* of devices. As communication within devices (over NVLink or Infinity Fabric) in the same node is generally significantly faster than between nodes (over InfiniBand or Slingshot), training is usually configured such that *tensor* parallelism occurs *within* a node, and *pipeline* parallelism occurs *between* nodes in an HPC context.

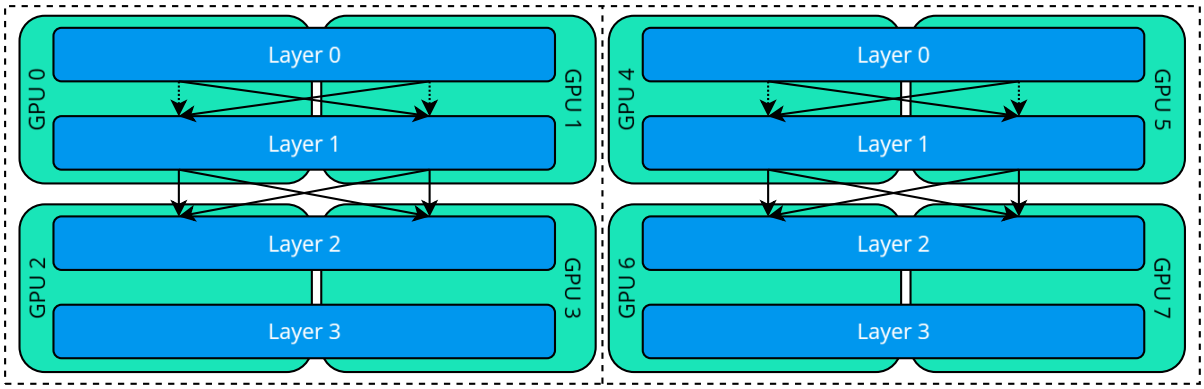


Figure 4-3: 3D parallelism combines data, tensor and pipeline parallelism. Each layer of each copy of the model is distributed across a subset of GPUs. Ideally, GPUs in charge of the same layers (e.g., GPUs 0 and 1) are part of the same node.

On large systems, these can be further combined with data parallelism, with the combination often being called **3D parallelism**, as shown in Figure 4-3. This was used for the full-size language benchmark.

Activation Partitioning

Another approach to parallelism is *activation partitioning*, aimed at reducing the memory usage of calculating layer activations.

Its simplest form is **sequence parallelism**, an extension of tensor parallelism in which intermediate activations of an input sequence (i.e., string of tokens) are distributed across devices. These activations are then communicated as they are required by operations spanning the full sequence, increasing communication overhead but reducing memory footprint.

A more recent and drastic approach is **context parallelism**, in which a sequence itself is partitioned across devices for the entire model. Unlike sequence parallelism, this partitioning also applies throughout most *computations* on the sequence, enabling training on massively long sequences without unrealistically high per-device memory requirements.

As this benchmark uses a moderate sequence length, context parallelism is not considered. We do enable sequence parallelism, as the AMD-specific branch of Megatron enforces having sequence parallelism enabled when tensor parallelism is. Although this requirement may have been added inadvertently, removing the enforcement to be able to disable sequence parallelism noticeably reduced throughput per GPU.

Distributed Optimizers

Orthogonal to the above types of parallelism, one can also distribute the model's optimizer. With a *distributed optimizer*, each device contains only one *shard* of the optimizer state, rather than having it replicated on all devices. This reduces memory requirements at the cost of modest additional communication. By overlapping this communication with computations, the net impact on training efficiency is minimal. We enabled the distributed optimizer for the language benchmark, although we did not observe any impact on performance.

4.3. Storage

While the language benchmark requires massively more *device* storage (i.e., VRAM) than the vision benchmark to store model weights, the scales tip in the opposite direction when considering *dataset* storage.

The language dataset is archived into a few files by default, and can simply be stored in a Lustre-based shared file system, as the read intensity is modest enough to not have the shared file system be a bottleneck. In fact, using a shared file system is *necessary*, as Megatron uses a global file-based cache to coordinate token selection between devices.

In contrast, the vision benchmark performs significantly fewer computations per sample, as the model is so much smaller. This leads to a massively higher read intensity for this benchmark. By default, ImageNet is distributed as one file per image, and Lustre is well known to suffer from poor I/O performance in this scenario. Even when the entire dataset is archived into a single file (we used Lightning Memory-Mapped Database (LMDB)), reading images in a random order is crucial for training these models, and such small random-access reads are again fundamentally incompatible with Lustre. As the order needs to be re-randomized every epoch, “pre-shuffling” data is also not feasible.

The impact of this is massive, with training slowing down to a crawl, even when the file is properly striped across OSTs. While solutions exist where each node holds a shard of the dataset in-memory and then randomly exchanges samples with other nodes after each epoch[9], we opted to simply store a local copy of the entire dataset on each node. While the initial copy operation adds some minutes of startup time to each benchmark, it speeds up training by several orders of magnitude.

[9] Nguyen, Truong Thao, et al. "Why globally re-shuffle? Revisiting data shuffling in large scale deep learning." *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022.

Some EuroHPC systems offer node-local SSD storage, but because all nodes offer hundreds of GiBs of RAM, which are largely unused during the benchmark, we instead opt to store these copies in local memory using `tmpfs` or `ramfs`. However, some EuroHPC clusters limit how much RAM may be used for this purpose, in which case we resort to the TinyImageNet-200 dataset, derived from ImageNet-1k. This dataset uses only 20% of the original classes, retaining only 40% of the images within them, and downscaling them severely, reducing the storage requirements by over 99.75%. To mimic training behaviour of the full dataset with the reduced one, each image is read 13 times per epoch instead of just once. Images are scaled to a fixed resolution regardless of dataset before being fed to the model, and pre-loaded in dedicated threads.

4.4. Floating-Point Precision

By default, training historically used *single-precision* 32-bit floating point numbers wherever a real number is needed. Some operations, such as convolutions, can be calculated significantly faster when instead using *half-precision* 16-bit floating point numbers. These also require less memory, enabling larger models or batch sizes. Other operations, such as reductions, require full single-precision values to ensure numerical stability.

Using single precision where necessary, and half precision (or even quarter precision) where advantageous is called *mixed precision*. PyTorch contains an Automated Mixed Precision (`AMP`) module, which selects the most appropriate data type for each operation according to predefined rules, and handles proper scaling between the two. With Megatron, this behaviour is built directly into the core stack.

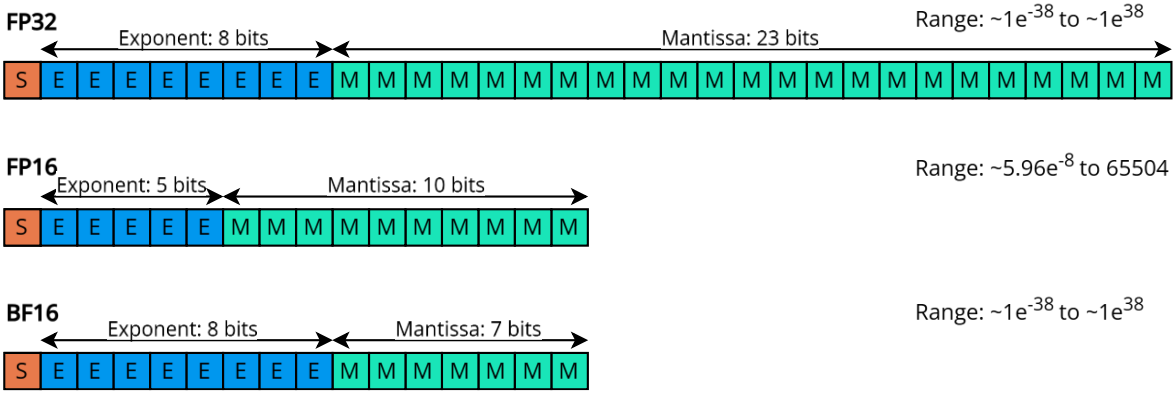


Figure 4-4: Structure of different floating-point representations. S is sign, and mantissa is also known as significand

Half precision can be implemented using either the `FP16` or `BF16` format, as shown in Figure 4-4. The `FP16` format is the older of the two, offering a smaller range of representable values than `FP32`, the default single-precision format. This reduced range can cause instability in neural network training. The `BF16` format preserves the exponent range of `FP32` while sacrificing precision instead. Neural network training is known to not be very sensitive to this loss of precision. Furthermore, conversion with `FP32` is simpler when using `BF16`, as their equal range means no scaling needs to occur.

In this Best Practice Guide, we evaluate the vision benchmark both with single precision everywhere, and with **AMP** enabled. With the language benchmark, we only consider mixed precision, as disabling it significantly impacts the resource requirements of the models under consideration. While **BF16** has become the industry standard, especially for LLM training, we use **FP16**, as this avoided some incompatibility issues with some other algorithms and hardware. The raw training speedup of the two formats is known to be comparable, and a detailed comparison between the two formats is considered out of scope of this document.

5. Benchmark Results

5.1. Scripts

The implementations of both the vision and language benchmarks were adapted from official examples provided by the frameworks. We specifically selected implementations designed for multi-GPU and multi-node execution of well-known models. This increases confidence in the stability of the implementations.

Vision

The vision experiment is based on the ImageNet example from PyTorch's repository of examples. The following noteworthy changes were made to the script:

- `AMP` was added as an option
- Multi-GPU was reworked with the script no longer being responsible for spawning additional instances, facilitating usage of Slurm
- Several tweaks to the model and training, aimed at better performance with thousands of GPUs, as proposed by Goyal *et al.*[10], were added
- A custom PyTorch `DataLoader` for LMDB datasets was added
- Synchronization points for more accurate runtime measuring were added
- Output was buffered to avoid slowdown due to the shared filesystem
- Unused code branches were removed

Language

The language benchmark is taken directly from the Megatron codebase. “Megatron” is made up of two parts: the **Megatron Core** library consists of building blocks for large-scale distributed LLMs, and **Megatron-LM** contains reference implementations and training scripts. The GPT implementation was taken from Megatron-LM and not modified. The parameter configuration was adapted from the example for GPT-3 pretraining. The following was modified:

- Added the small single-GPU GPT-style model
- Added the option to start from checkpoint
- Added weak/strong scaling depending on world size (i.e., number of GPUs)
- Added numerous flags empirically found to improve performance

Orchestration

Each data point in each benchmark corresponds to a single Slurm job, started through a straightforward Slurm script. An overarching script queues the jobs for the minimal

[10] Goyal, Priya, et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour." *arXiv preprint arXiv:1706.02677* (2017).

number of nodes (16 or 32 nodes for the full-size LLM, whichever provides 128 GPUs, and 1 node for all others) up to the highest number of nodes a user would realistically run on each system, doubling the count at each step. Each node is used in exclusive mode. For the language benchmark, Slurm's `--dependency` flag was used to ensure only one benchmark runs at any time, avoiding any potential issues with multiple instances accessing the same dataset on the shared filesystem simultaneously.

Table 1 provides links to all Slurm job scripts and overarching scripts for all benchmarks we ran. In addition, we provide the PyTorch script for the vision benchmark, and the "train" script which sets the Megatron parameters for the language benchmark. As we did not alter the Megatron GPT-3 Python script, we do not link it per-system. Then, the "full-size" column indicates if the full-size benchmark was deployed on said machine. For vision this means the full ImageNet-1k dataset instead of TinyImageNet-200, and for language it means the 175B parameter model and not the 857M one. When marked as full-size (✓), the scaled-down version was run in addition if deemed useful.

Table 1: All scripts to schedule and run the benchmarks on each machine. Missing entries to be added in an update.

Machine	Benchmark	Full-size	Scripts
LUMI	Vision	✓	Run, Slurm, PyTorch
	Language	✓	Run, Slurm, Train
Leonardo	Vision	✗	Run, Slurm, PyTorch
	Language		Run, Slurm, Train
MareNostrum 5	Vision	✓	Run, Slurm, PyTorch
	Language	✓	Run, Slurm, Train
MeluXina	Vision	✓	Run, Slurm, PyTorch
	Language		
Karolina	Vision	✓	Run, Slurm, PyTorch
	Language	✗	Run, Slurm, Train
Discoverer	Vision		
	Language		
Vega	Vision	✓	Run, Slurm, PyTorch
	Language		
Deucalion	Vision	✓	Run, Slurm, PyTorch
	Language	✗	Run, Slurm, Train
JUPITER	Vision		
	Language		

5.2. Vision Results

Overview

For the vision benchmark, we scale up the number of nodes from 1 to some realistic upper limit for the given machine, doubling the node size at every step. For this, we note that 1 node contains 4 GPUs with Leonardo, MareNostrum 5, MeluXina, Vega, Deucalion and JUPITER, but 8 GPUs with LUMI[11], Karolina and Discoverer.

As is common with distributed CNN training, we keep the *per-GPU* batch size fixed throughout the whole scaling benchmark. Each GPU keeps many copies of the model in memory, then processes a whole batch of images in parallel. Then, all copies of the model are given the same weight updates, before a new batch is processed. Picking the largest batch size that comfortably sits within VRAM is seen as the best practice. These values were empirically found to be 128 for 40 GiB, 256 for 64 GiB and 512 for 80 GiB (see hardware specifications in Chapter 3). Taking a smaller batch size wastes GPU capacity. Taking a larger one would require running multiple *sub-batches* in sequence, which frameworks do not usually support. In essence, this just means skipping some of the weight updates, and is known to slow down convergence.

As the *global* batch size increases along with the number of nodes, this benchmark evaluates *weak scaling*[12]. Weak scaling of ResNet-50 up to hundreds, or even a few thousands of GPUs is known to work well, given a few tweaks to the training process, which are implemented in our benchmark. A small selection of the benchmarks was run to completion to validate that *per-epoch* training progress was similar regardless of node count. We then only run the first few epochs of the full benchmark set on all machines. From this, we derive the **scaling efficiency** given the number of nodes. In general, we define scaling efficiency of n nodes as:

$$e(n) = \frac{t(n)}{t(1)}$$

Where $t(n)$ is the runtime for n nodes. We select three different ways of measuring $t(n)$, each offering a different insight:

- $t_{ep}(n)$: the runtime of the full epoch under consideration. Gives the actual full runtime in the benchmark.
- $t_{tr}(n)$: the runtime of the training part (i.e., excluding validation) of the epoch under consideration.
- $\tilde{t}_b(n)$: the *median* time to process one training batch (including weight updates). This filters out the performance reduction from occasional GPU slowdown, and shows potential performance of the main parallelisable part of training.

By definition, scaling based on median batch processing time will show better performance than based on full runtime. In addition to the slowdown from occasional outliers, the first type of scaling also excludes any warmup overheads at the start of

[11] Technically, a LUMI node contains 4 AMD MI250x GPUs with 128 GiB VRAM each. Each is divided into two Graphics Compute Dies (GCDs). As software such as PyTorch treats each GCD as a GPU (with 64 GiB VRAM), we consider LUMI nodes as having 8 GPUs.

[12] Note that *strong* and *weak scaling* in a Deep Learning context specifically is often used with a slightly different meaning from usual. Here, they refer to the amount of work *per batch* instead of *in total*.

each epoch. Overheads like filling the `DataLoader` cache are non-parallelisable, in the sense that they need to occur on each GPU and will take roughly the same amount of time on each. Overall, scaling based on median batch processing time degrades mainly from the communication overhead of synchronising weights between GPUs, while scaling based on full runtime additionally degrades from occasional slowdown (e.g., due to congestion in the network) as well as non-parallelisable fixed per-epoch warmups. As the number of nodes increases, slowdown is expected to occur more frequently due to increased communications. Furthermore, as the wall-clock time of the parallel part decreases, further parallelisation achieves diminishing returns, as per Amdahl's law. As such, poor per-epoch scaling at very high node counts is inevitable.

All statistics are logged by the primary worker in distributed training. Experiments are repeated without and with `AMP`, and may be run for either the full ImageNet dataset or the reduced TinyImageNet dataset. In the latter case, the dataset is iterated over multiple times per epoch, to ensure that epoch length is equal between the two, making results as comparable as possible.

Results and Analysis

We analyse the results for each machine separately. We start with LUMI, drawing several general conclusions from it, before continuing to all other machines. As such, readers only interested in a specific machine are advised to also read through at least the LUMI results.

LUMI

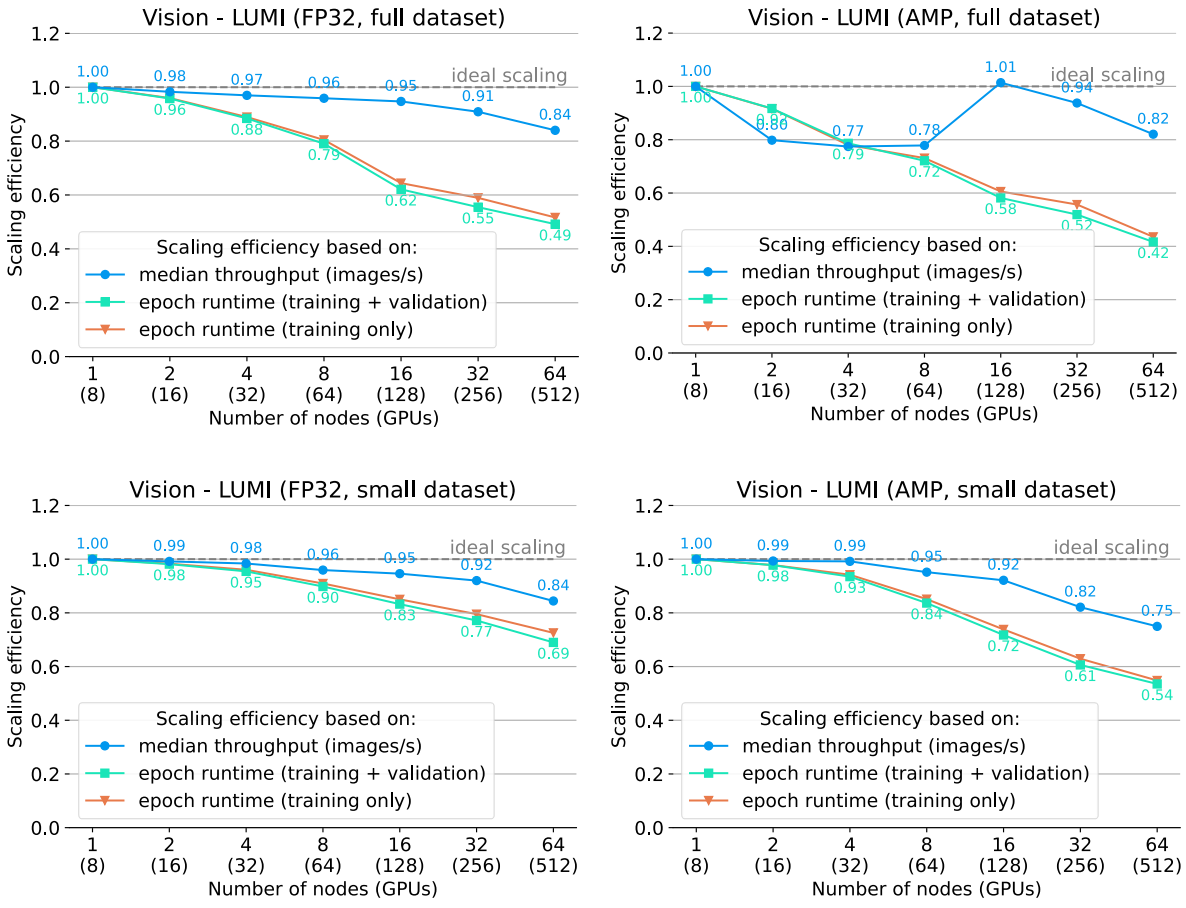


Figure 5-1: Vision scaling on LUMI

Per-epoch vs. per-batch scaling

Figure 5-1 visualizes the results on LUMI, both with **`FP32`** only and with **`AMP`**, with the full and small datasets. Overall, training epoch runtime scales slightly better than full epoch runtime, meaning that the efficiency reduction of the validation step is minor, but consistently present. This makes sense, as the validation step itself is short compared to training, but has some fixed overheads.

Very clearly, the *median* per-batch scaling efficiency is significantly higher than when considering the full training time of the epoch. This has several causes. For one, the *first* batch of *every* epoch takes longer due to warmups. The **`DataLoader`** is only initialized when the first inputs are needed, meaning the GPUs need to wait for data loading and preprocessing in this first batch. Later batches are preloaded while previous batches are still being processed. The **`persistent_workers`** flag, ensuring that **`DataLoaders`** are not fully reinitialized from scratch every epoch, was enabled, and alleviated this warmup somewhat, but not entirely. A second, and usually more significant source of occasional slowdown is that whenever one batch is delayed on one GPU, all of them must wait until it is finished. In a sense, the effective speed of *each* GPU becomes that of the slowest one. Extensive evaluation of profiler traces showed that slowdown on single GPUs or network links to it indeed caused this, but root causes of the slowdown could not be evaluated.

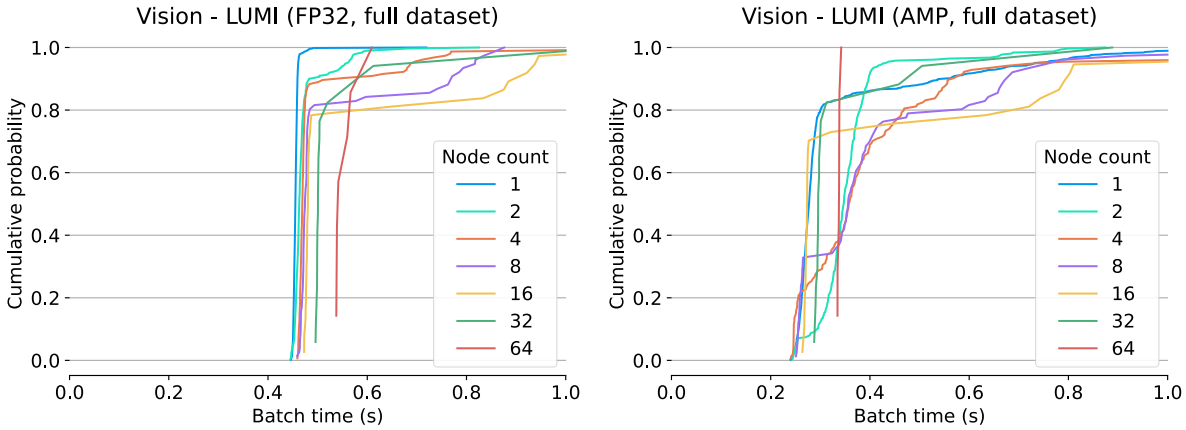


Figure 5-2: Cumulative Distribution Function (CDF) of per-batch runtimes on LUMI

Distribution of batch runtimes

Figure 5-2 shows Cumulative Distribution Functions (CDFs) of these per-batch runtimes (for the full dataset). From these plots, one can read the percentage of batches that were completed within any given time. To eliminate warmup overheads as well as exceptionally fast processing of final, non-full batches, the first and last batch of each epoch are filtered out.

Most lines initially rise quickly, followed by a sudden *elbow point* and then a large spread of highest per-batch runtimes (i.e., those impacted by occasional slowdown). In general, higher node sizes have an earlier elbow point. In these examples, no clear elbow point occurs in the 64-node case. We note that, in this case, an epoch consists of only 8 global batches (taking around half a second each). Coincidentally, no slowdown occurred during these 4 seconds. In other cases (not visualized here) there does appear to be a clear elbow point with 64 nodes. As the relative impact of a

slowdown is larger with shorter epochs, this should not be taken as an indication that scaling improves with higher node counts.

Importantly, all elbow points with **FP32** occur past the 50th percentile, meaning occasional slowdown is filtered out of the median runtimes. Some odd behaviour does occur, like the 16-node benchmark being slower than larger benchmarks around the 80th percentile. We could not identify a root cause, but note that it was consistent across repeats at different times, and therefore not likely caused by a congested network, or malfunctioning nodes. With **AMP**, the elbow point occurs before the 50th percentile for 2-8 nodes, explaining the apparent valley in per-batch efficiency there with the full dataset. We are again unsure as to why this occurs, but note that it did not happen with the downsized dataset.

Full vs. downsized dataset

Comparing the full dataset with the downsized one, using Figure 5-1, we observe that the per-batch efficiency is nearly identical between the two with **FP32**. The per-epoch efficiencies are significantly worse with the full dataset, which could indicate that slowdown may be caused when preprocessing images (as the full dataset contains larger ones, which require downscaling). This holds for the **AMP** case, when ignoring the valley in per-batch efficiency.

FP32 vs AMP

Finally, when directly comparing **FP32** and **AMP**, the latter appears to scale worse than the former. Here, it is important to note that we are evaluating *relative* efficiency. As Figure 5-3 shows, **AMP** is considerably faster than **FP32** in all cases when looking at median images per second (derived from the batch runtimes). This means that any GPU slowdown, which we expect to be independent of the precision used, has a higher impact on **AMP** in relative terms. Furthermore, using **AMP** shifts the *compute-to-communication balance*. **AMP** speeds up compute significantly, but has limited impact on communication speed, which effectively reduces the scaling efficiency. The absolute throughputs however make clear that this does *not* make **AMP** the inferior choice when running with many nodes.

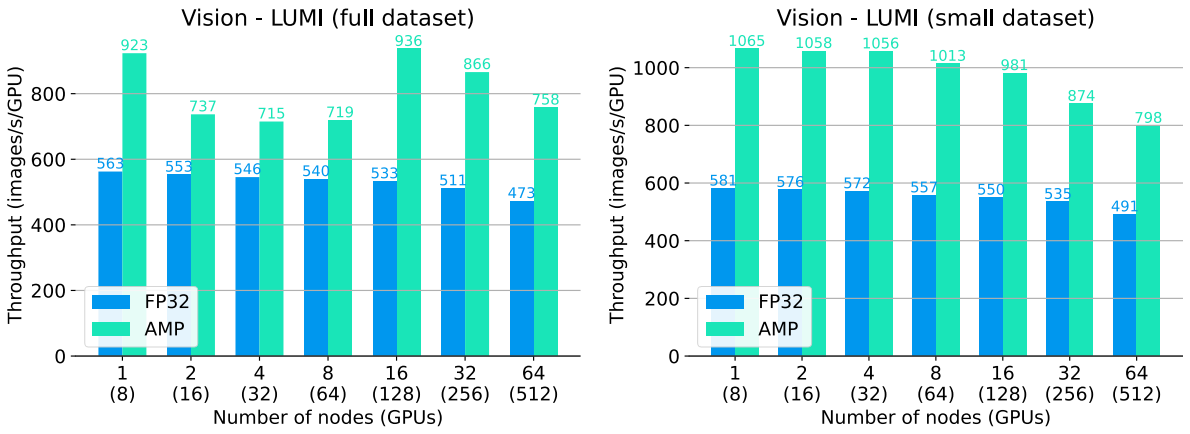


Figure 5-3: Median per-GPU throughputs on LUMI

Leonardo

Figure 5-4 shows the scaling efficiency on Leonardo, using the small dataset. With **AMP**, the efficiency gradually decreases, with especially 32 and 64 nodes having

seriously reduced performance. With **FP32**, the per-batch scaling remains near-perfect even at 64 nodes, indicating highly efficient node-to-node communications on Leonardo, as also indicated by Figure 5-5.

Even when considering per-epoch scaling efficiency, it remains high, with 90% at 16 nodes and 77% at 64 nodes with **FP32**. This decrease at higher node counts is inevitable with weak scaling, as epochs become very fast, while certain fixed overheads (such as warmups) cannot be eliminated. When looking at the raw throughputs in Figure 5-6, it is remarkable that **FP32** outperforms **AMP** at 64 nodes. This may be due to the **AMP** algorithm misbehaving or due to a very busy network while running the **AMP** benchmarks. Due to severe congestion on the machine, we were unable to run repeat benchmarks to confirm this.

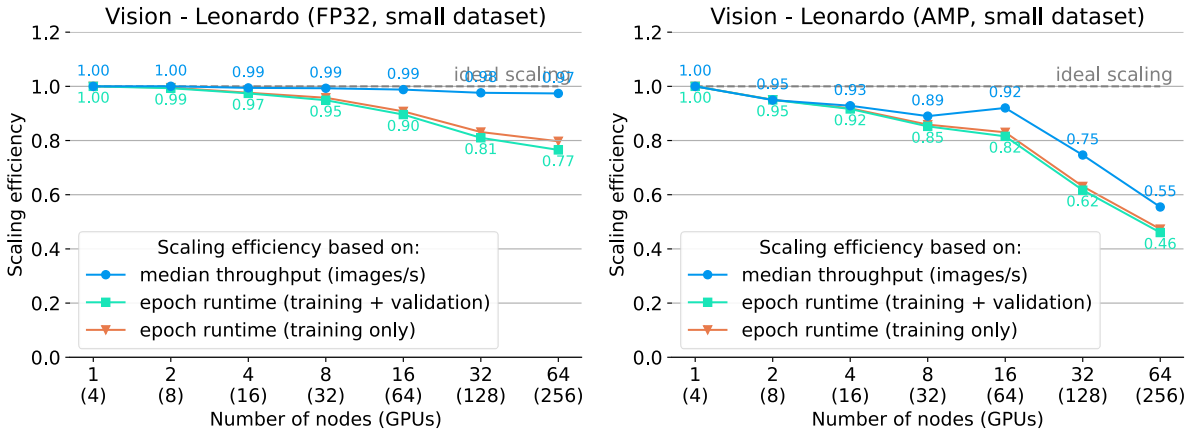


Figure 5-4: Vision scaling on Leonardo

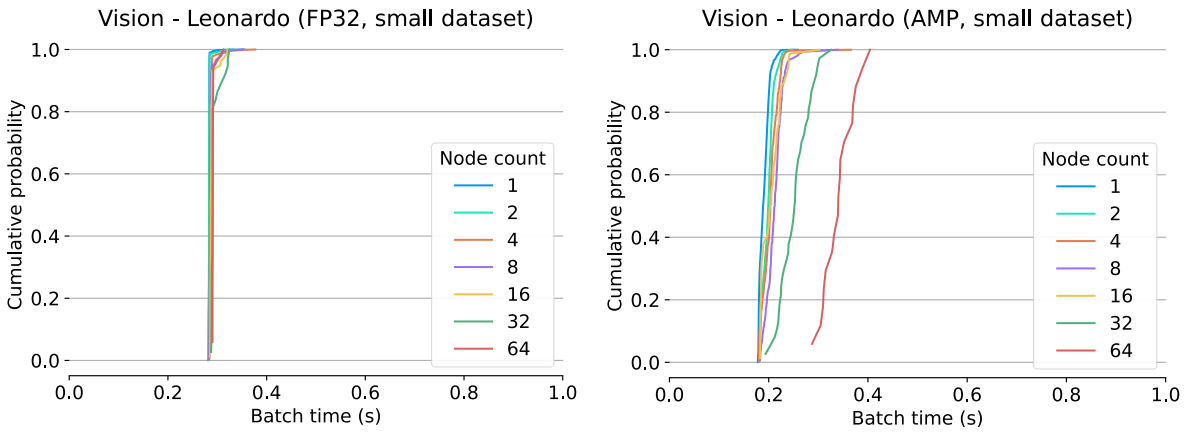


Figure 5-5: Cumulative Distribution Function (CDF) of per-batch runtimes on Leonardo

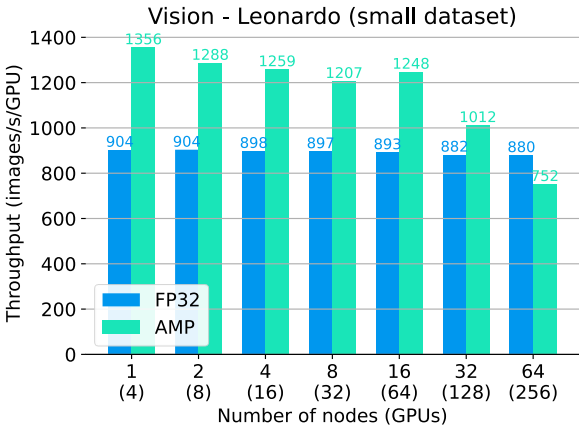


Figure 5-6: Median throughput on Leonardo

MareNostrum 5

MareNostrum 5 scales very well in terms of per-batch efficiency both with **`FP32`** and **`AMP`**, never dropping below 89% in Figure 5-7. Nearly no slowdown is seen from the per-batch CDFs in Figure 5-8. Notably, MareNostrum 5 is the only pre-exascale EuroHPC system with a fat-tree network topology rather than Dragonfly(+). It is possible that the fat tree's more straightforward routing is responsible for the absence of slowdown, though this is challenging to confirm.

At high node counts, per-epoch efficiency reduces rapidly. However, note that, as shown in Figure 5-9, the per-GPU throughput of MareNostrum 5's H100 "Hopper" GPUs is considerably higher than for the other pre-exascale systems. As non-parallelisable overheads such as **`DataLoader`** warmup do not run on GPUs, this higher throughput inevitably leads to apparent lower scaling as per Amdahl's law. This is not a cause for concern and does not indicate scaling issues on MareNostrum 5.

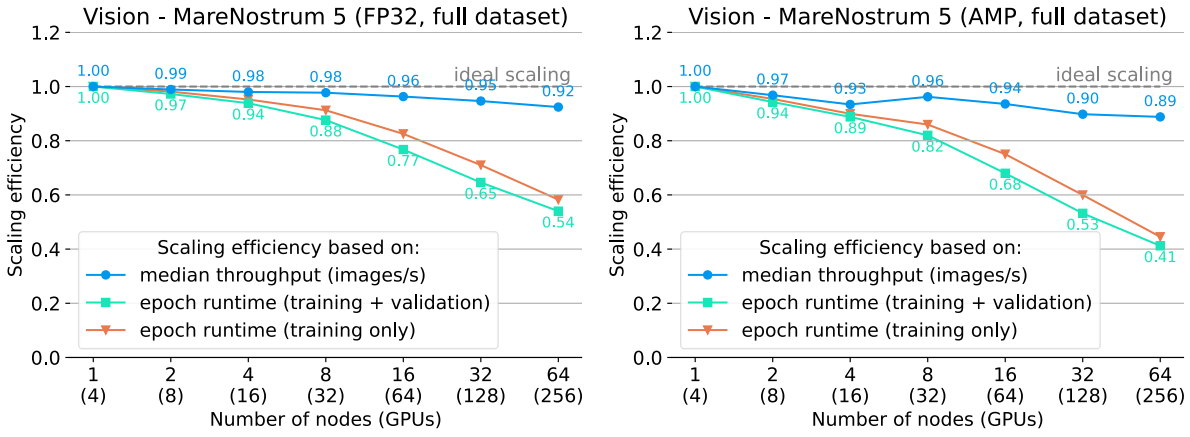


Figure 5-7: Vision scaling on MareNostrum 5

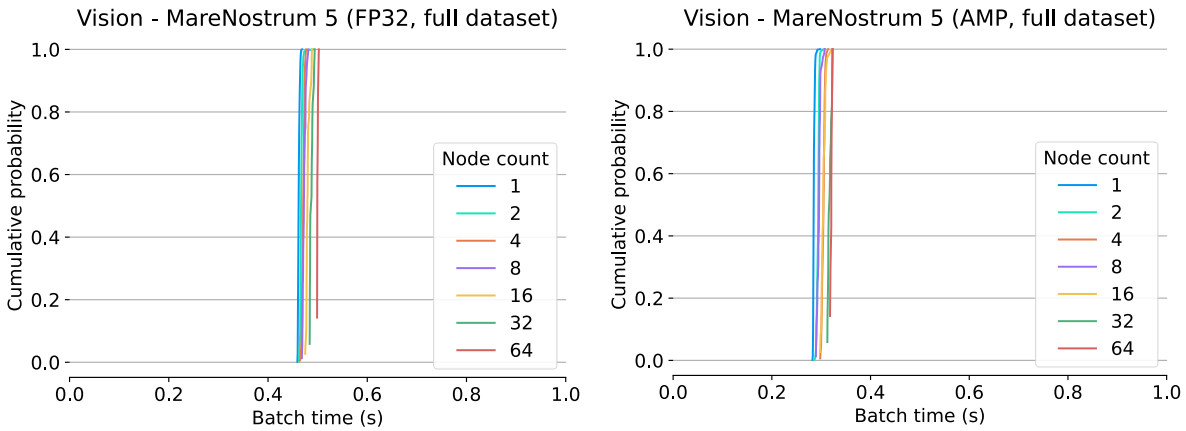


Figure 5-8: CDF of per-batch runtimes on MareNostrum 5

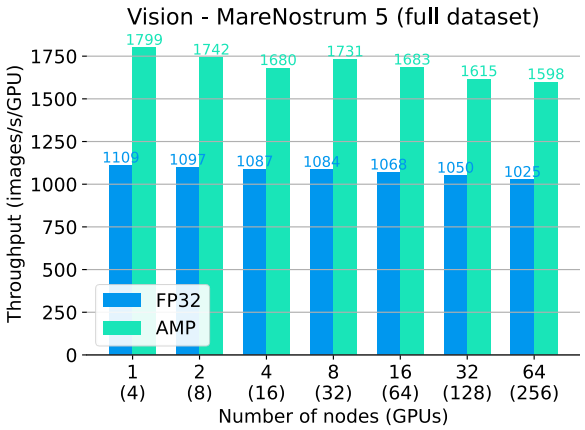


Figure 5-9: Median throughput on MareNostrum 5

MeluXina

As was the case with LUMI, MeluXina shows clear slowdown starting from around the 80th percentile with **FP32**, as shown in Figure 5-11, likely again indicating a busy network. Again similar to LUMI, slowdowns are more prevalent with some **AMP** runs, but *not* the largest ones, leading to the apparent valley in per-batch scaling in Figure 5-10. This may indicate that this inconsistent behaviour is related to the **AMP** algorithm.

Overall, the impact on per-epoch scaling is as would be expected based on previous systems' results. Finally, Figure 5-12 shows **AMP** comfortably outperforming **FP32** for all node counts.

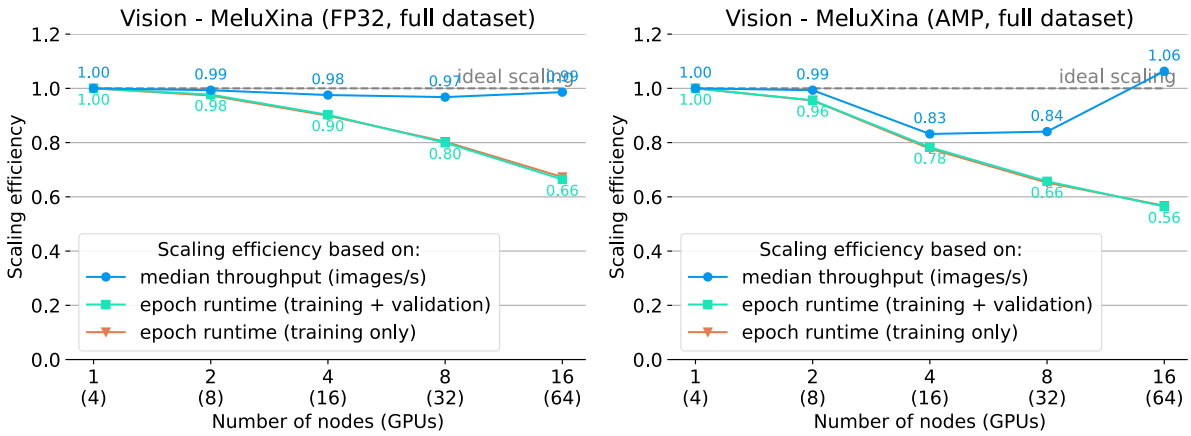


Figure 5-10: Vision scaling on MeluXina

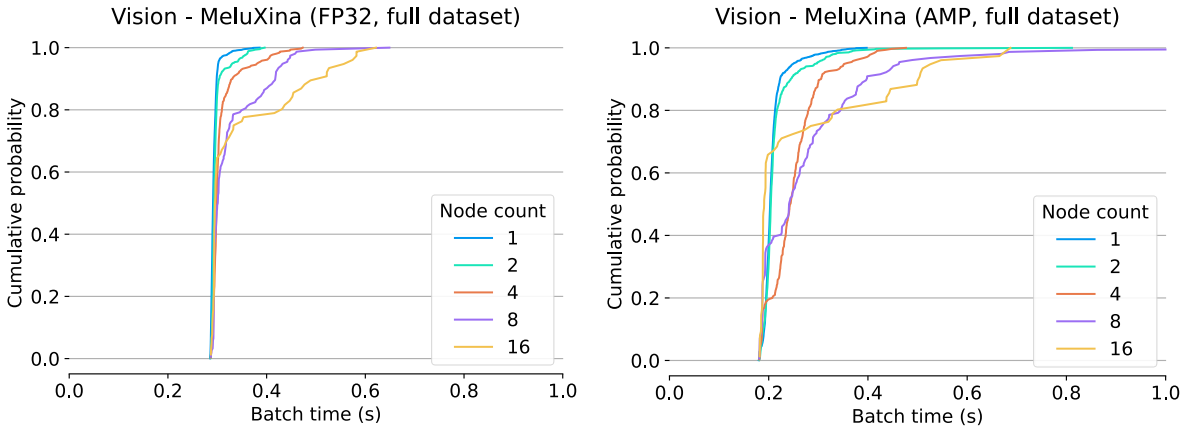


Figure 5-11: CDF of per-batch runtimes on MeluXina

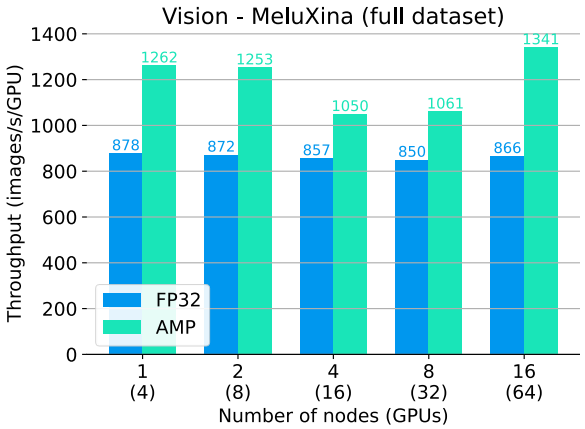


Figure 5-12: Median throughput on MeluXina

Karolina

On Karolina, the median per-batch runtime in Figure 5-13 degrades fairly rapidly compared to other systems on which 64 nodes were used. However, Figure 5-14 shows that, while batch times are higher for each subsequent step, the batch times *within* one run are highly consistent, with elbow points mostly only occurring at the very top of the plot. This indicates that, while the benchmark places severe stress on the system's network, the network performs consistently given some workload. This

makes sense as the largest run uses over 90% of all GPU nodes of Karolina. In that case, when all nodes in the benchmark communicate, this approaches the upper limit of what the network was designed to support. However, barely any other jobs, which could cause intermittent network congestion, can run at the same time.

Finally, Figure 5-15 shows that **AMP** always outperforms **FP32**. The gap begins to close at higher node counts, but this is as expected, as communication overheads begin to dominate the runtime.

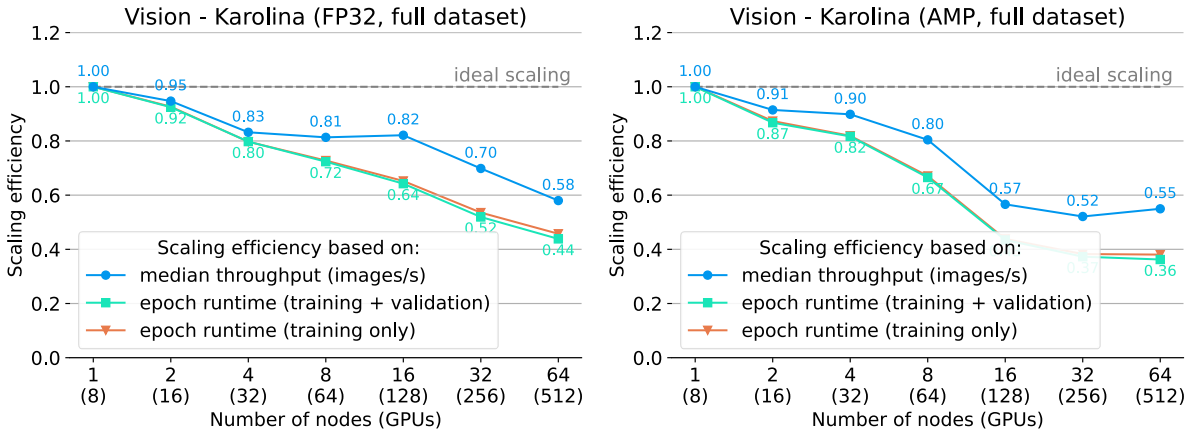


Figure 5-13: Vision scaling on Karolina

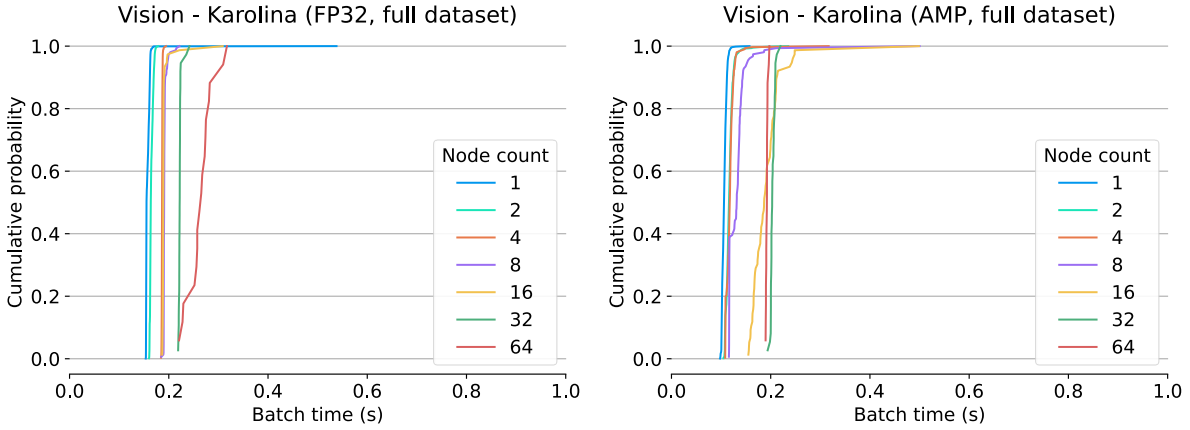


Figure 5-14: CDF of per-batch runtimes on Karolina

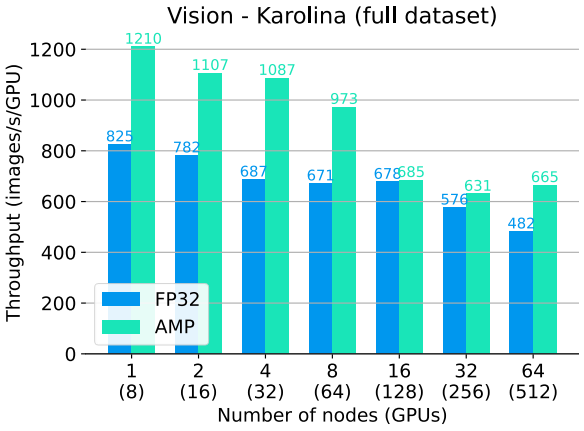


Figure 5-15: Median throughput on Karolina

Vega

The results for Vega are only partially available at the time of publication. The full set, along with an analysis, will be added in a future version of this document.

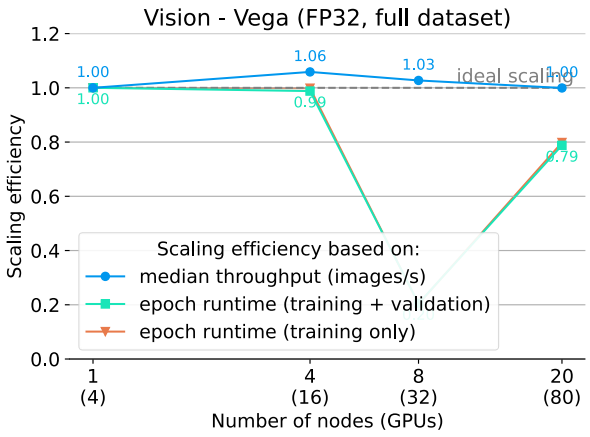


Figure 5-16: Vision scaling on Vega

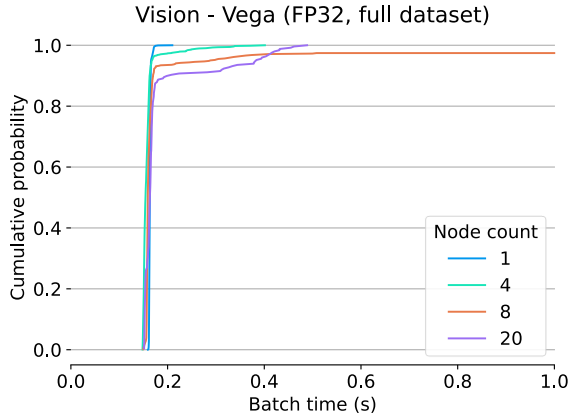


Figure 5-17: CDF of per-batch runtimes on Vega

Deucalion

On Deucalion, scaling is near-perfect with 2 nodes, but degrades rapidly with 4 nodes, as per Figure 5-18. As shown in Figure 5-19, there is an elbow point around the 80th percentile with a very long tail with **AMP**, and a consistently poor batch time with **FP32** at 4 nodes.

The system was under very high usage during the benchmarks. We hypothesize that, for the 2-node benchmarks, adjacent node pairs near the leaves of the fat tree were selected, meaning that the network links between those were not used by many other jobs, leading to high scaling efficiency there. With 4 nodes, we suspect that nodes more scattered across the fat tree were selected, meaning that highly congested links near the tree's root were being used.

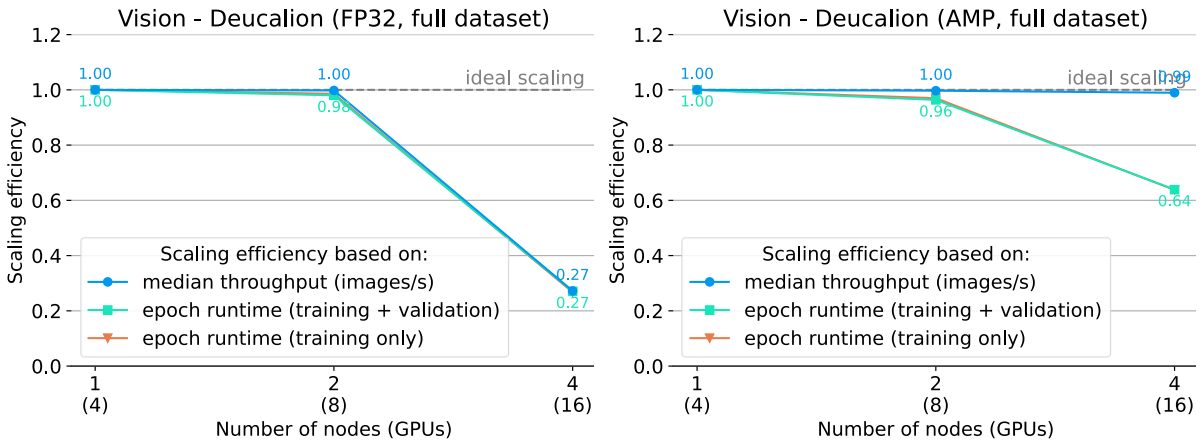


Figure 5-18: Vision scaling on Deucalion

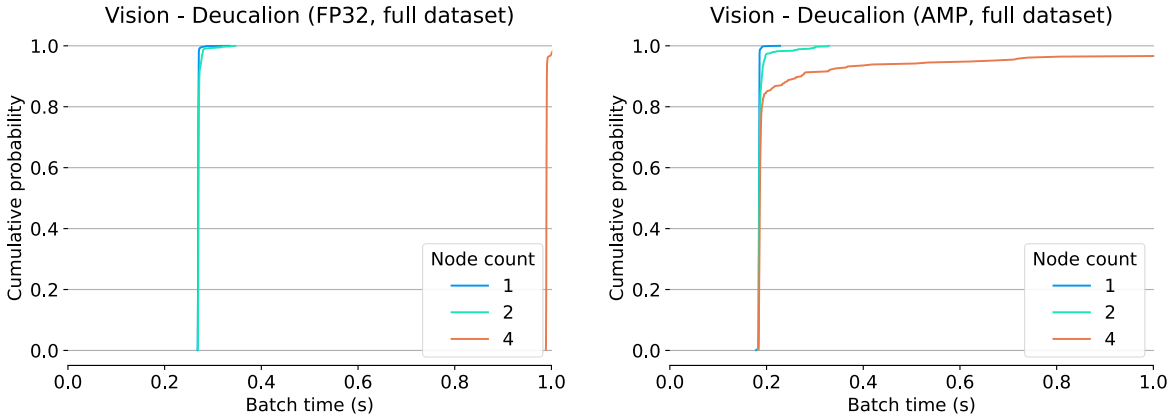


Figure 5-19: CDF of per-batch runtimes on Deucalion

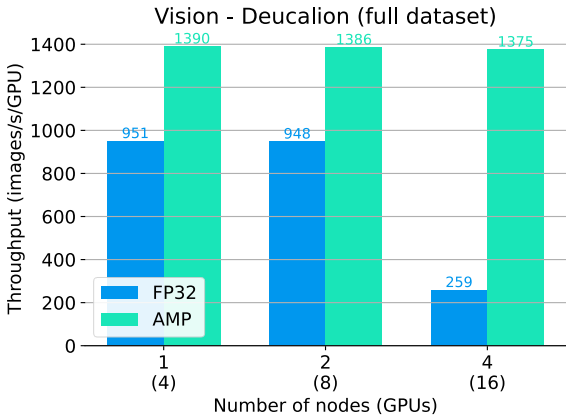


Figure 5-20: Median throughput on Deucalion

5.3. Language Results

Overview

The language benchmark starts at the number of nodes which results in 128 GPUs, the minimum needed to hold one copy of the model. It then scales up to the largest

number of nodes we could realistically reserve on each machine, again doubling each step.

The benchmark works with a *microbatch* size of 1, meaning different samples are not processed in parallel on the same GPU. These microbatches are still combined into regular batches, after which the model weights are updated. In this case, we consider both strong and weak scaling. With strong scaling, the batch size is kept at 1536, the default number of GPT-3. With weak scaling, the number of *gradient accumulation steps* (i.e., the number of microbatches *one* copy of the model processes before updating weights) is consistent at 192. The global batch size then doubles every time the node count doubles, and batch sizes are equal for weak and strong scaling at 8 times the minimum number of nodes. Scaling benchmarks often select these configurations such that the batch sizes are equal at the *largest* configuration. As we work with different ceilings for different systems, we chose 1024 GPUs as a middle ground.

To measure the scaling efficiency, we rely on the throughput in TFLOPs/GPU as reported once per batch by Megatron. The scaling efficiency then becomes

$$e(n) = \frac{tput(n)}{tput(n_{min})}$$

Where $tput(n)$ is the per-GPU throughput in TFLOPS and n_{min} is the smallest possible number of nodes. Note that the efficiency is undefined for $n < n_{min}$. Each benchmark is run long enough for $tput(n)$ to stabilize.

As not all machines offer the hundreds of GPUs needed to properly run this benchmark, we also consider a significantly scaled-down version of the LLM, around 200 times smaller than the full model, making it fit on a single GPU. For this configuration, we take batch size 512 (strong scaling) or 16 gradient accumulation steps (weak scaling). In this case, $n_{min} = 1$.

Results and Analysis

As with the previous benchmark, we again analyse the results for each machine separately. The first analysis, on LUMI, is again recommended to all readers, as it contains generally applicable insights.

LUMI

Figure 5-21 shows the weak and strong scaling efficiency of both the full and small model on LUMI. The full model shows a steady reduction in efficiency with increasing node counts. With weak scaling, the frequency of weight updates (per sample per GPU) remains constant while scaling. This requires communications between all GPUs, and is expected to be the main cause of reduced scaling efficiency. With strong scaling, weight updates are initially very infrequent, but their frequency (per sample per GPU) doubles with each doubling of node count.

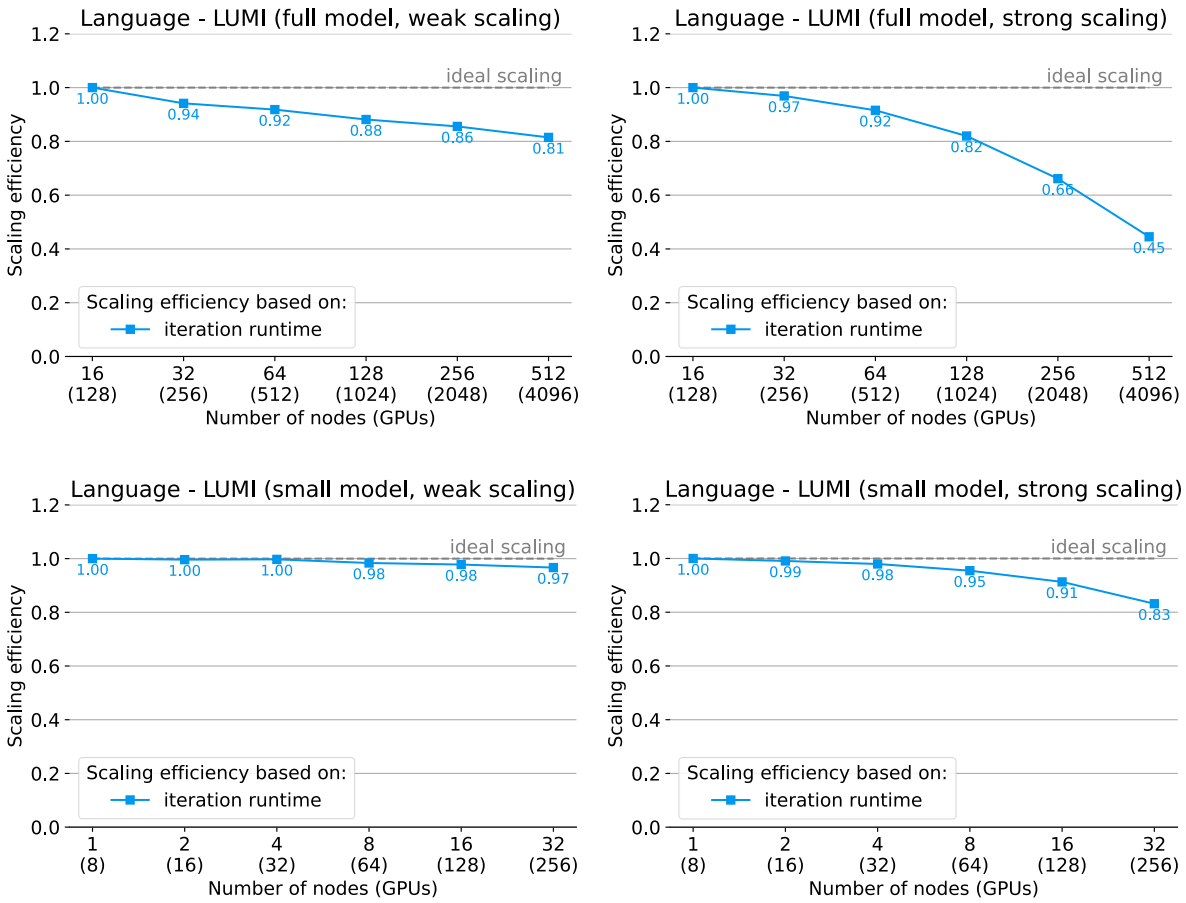


Figure 5-21: Language scaling on LUMI

When considering the full model with weak scaling, the efficiency remains above 80% even with 32 times the baseline number of nodes. With strong scaling, the degradation of efficiency speeds up with higher node counts, as more and more weight updates are scheduled, eventually reaching only 45%. Initially, efficiency is better than with weak scaling: even though more weight updates are scheduled when node counts increase, the *absolute* number of updates is still very low then, meaning the impact on overall efficiency is more limited.

With the small model, the same patterns hold, but are less severe. Performance efficiency remains at 97% (weak) or 83% (strong) even when increasing node size 32-fold. This was expected, as the overall number of nodes involved in communications is 16x lower, reducing that overhead in absolute terms.

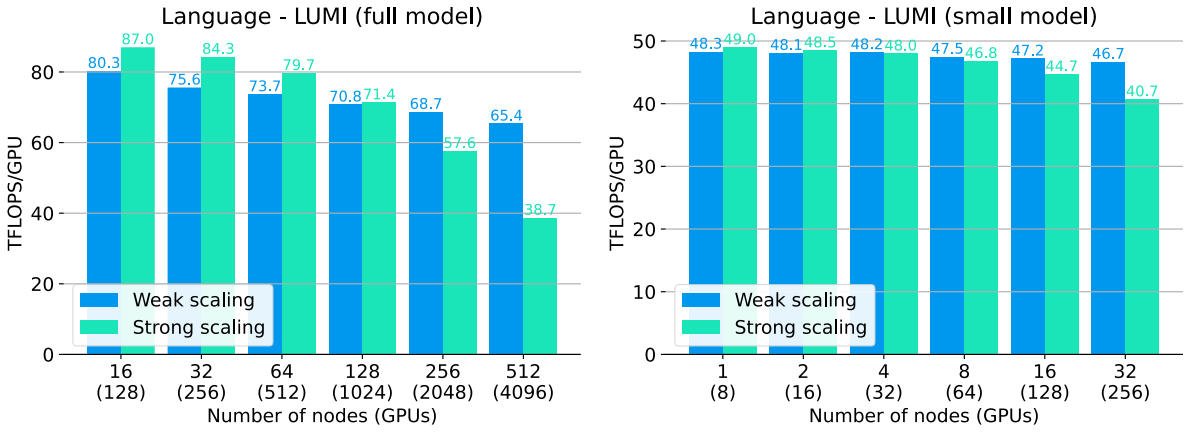


Figure 5-22: Language throughput per GPU on LUMI

Next, Figure 5-22 shows the absolute per-GPU throughputs in TFLOPS, as reported by Megatron. Notably, the full model achieves significantly higher TFLOPS than the small one, as it saturates the GPUs better. Furthermore, as expected, the absolute throughput is higher with strong scaling for low node counts, and higher with weak scaling for high ones. The only difference between the two versions of the benchmark is the global batch size, which is fixed with strong scaling, and scales with node count for weak scaling. The two scaling types have the same global batch size at 1024 GPUs for the full model, or 32 GPUs for the small one. Any difference between weak and strong scaling at those points is due to variance. Given the resource requirements of these benchmarks, running them repeatedly was infeasible.

Leonardo

The results for Leonardo are only partially available at the time of publication. The full set, along with an analysis, will be added in a future version of this document.

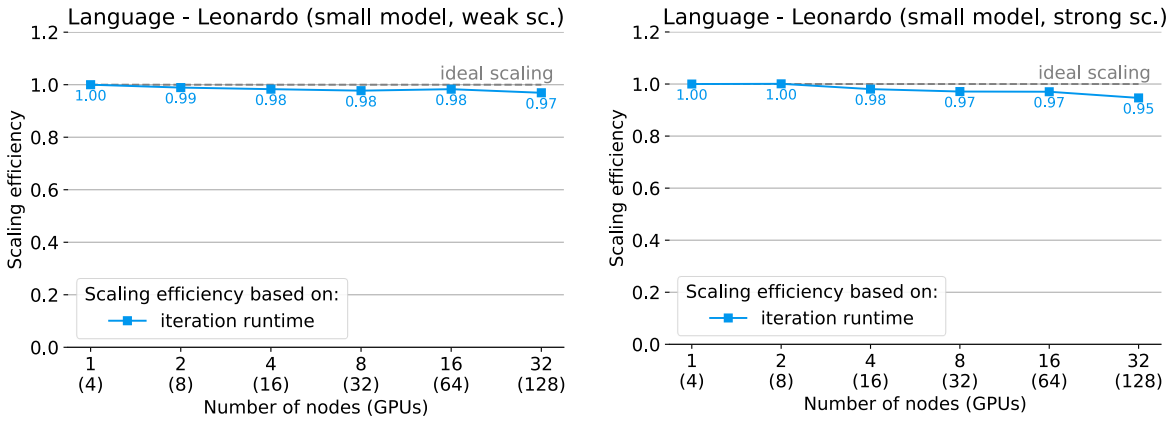


Figure 5-23: Language scaling on Leonardo

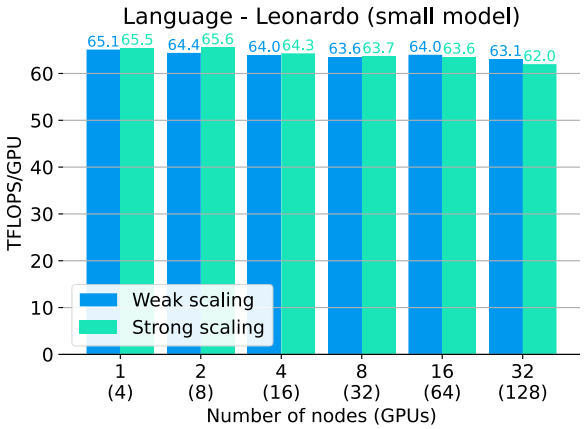


Figure 5-24: Language throughput per GPU on Leonardo

MareNostrum 5

With MareNostrum 5, we were able to run with up to 512 GPUs for the full model, as shown in Figure 5-25. The general trend here is similar to the previous results, with the largest run achieving 75-80% scaling efficiency depending on the scaling type. With the smaller model, scaling efficiency is again above 90% even when scaling 32-fold.

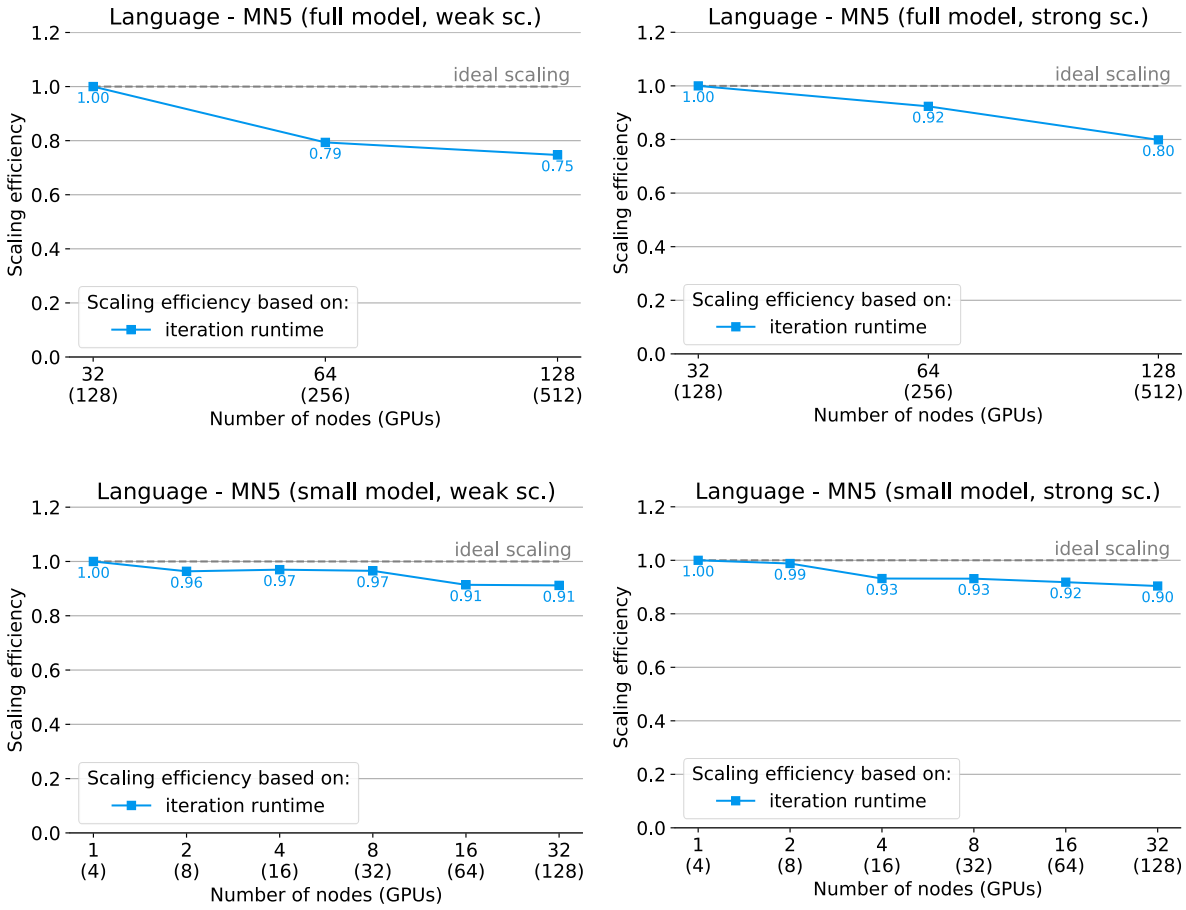


Figure 5-25: Language scaling on MareNostrum 5

The raw throughput achieved on MareNostrum 5 is very high, with Figure 5-26 showing up to 384 TFLOPS per GPU. The performance impact from switching to the smaller model is more pronounced here, losing up to 77% of raw performance. This indicates that more raw performance could possibly be gained with the small model by increasing the microbatch size above 1, although tuning of this hyperparameter was outside of the scope of this document.

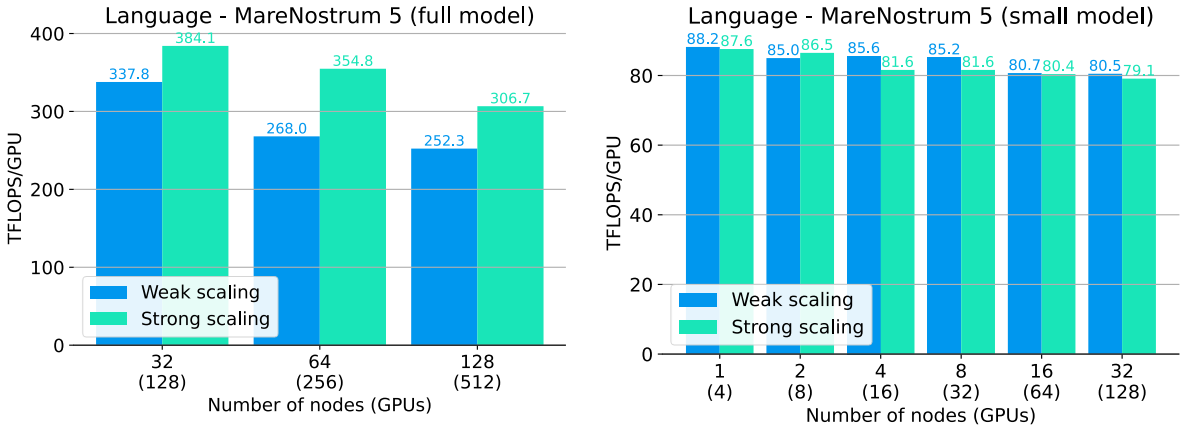


Figure 5-26: Language throughput per GPU on MareNostrum 5

Karolina

On Karolina, we ran the small model on up to 256 GPUs in Figure 5-27. As we observed with the vision results, the scaling efficiency degrades steadily, likely because we are using a large part of the cluster and therefore stressing the network significantly.

More than with previous results, it is remarkable here that raw throughput with 1 node is nearly equal for weak and strong scaling, seen in Figure 5-28. Ignoring the 1-node result, the trend where lower node counts perform better with strong scaling and higher ones do with weak scaling does hold. This result makes sense, as all communications in the 1-node case occur over the intra-node NVLink interconnect, bypassing the main network, and speeding up the communications enough to minimize their performance impact.

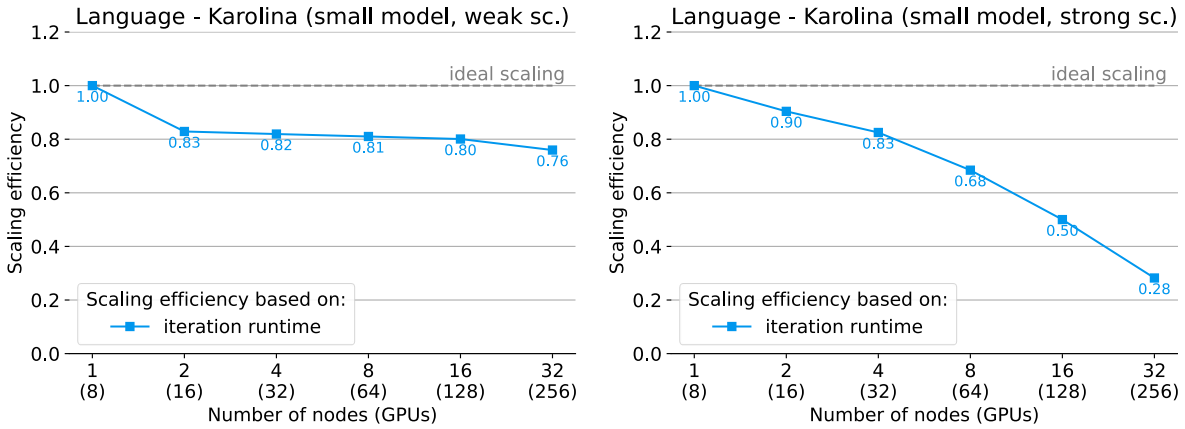


Figure 5-27: Language scaling on Karolina

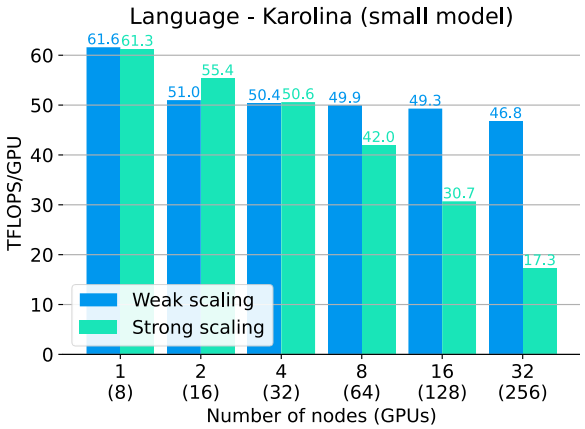


Figure 5-28: Language throughput on Karolina

Deucalion

With Deucalion, we were only able to run the language benchmark with up to 4 nodes. In this small setup, however, scaling efficiency was very high, scaling virtually perfectly to 4 nodes in Figure 5-29. Furthermore, the raw performance difference between weak and strong scaling was negligible as per Figure 5-30, indicating that the network was consistently very fast and reliable in these benchmarks.

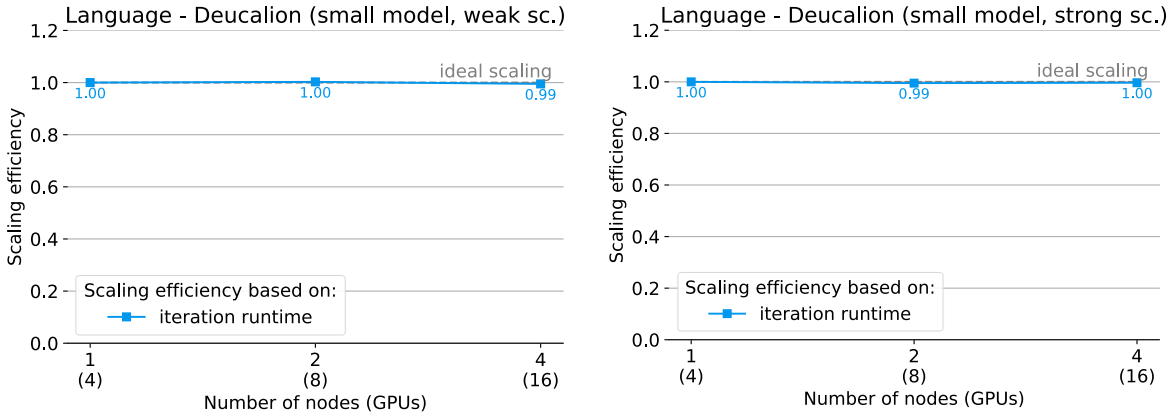


Figure 5-29: Language scaling on Deucalion

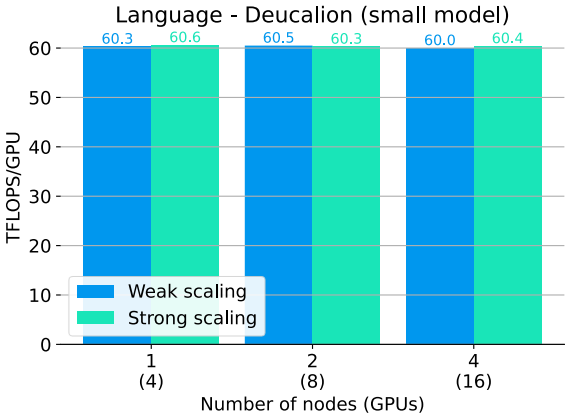


Figure 5-30: Language throughput on Deucalion

5.4. Summary

General trends in scaling efficiency occur throughout all EuroHPC systems on which we ran our benchmarks. As the number of nodes increases, the rate at which scaling efficiency degrades also rises. These *diminishing returns* with scaling are well-known and well-understood, and can be explained through Amdahl's law. The point at which adding more resources is no longer worth the cost depends on the specific system, with network speed and congestion expected to be the main culprit in accelerating these diminishing returns. As this depends on many factors outside of our control, like node selection and network usage of other simultaneous jobs, this is difficult to evaluate thoroughly, especially given the high resource requirements of these benchmarks and high load of many systems. As such, users are advised to perform their own scalability benchmarking for their specific applications and systems.

Still, some systems showed consistently stable network performance. For example, MareNostrum 5 has a consistently fast network, which may be due to its fat-tree topology. Network throughput for a given benchmark size was very stable on Karolina too, although scaling up the number of nodes did significantly reduce this throughput, likely due to the high stress the benchmark places on its network. On other systems, the network performance is somewhat more erratic, possibly due to complicated Dragonfly-style routing, somewhat lowering the scaling limits of applications. Still, at-scale AI is very possible on these machines.

Furthermore, results show that scaling efficiency values alone should not be considered in a vacuum. Faster GPU hardware or implementations can lead to apparently lower scaling efficiency, with the parallelisable part of the workload taking up a smaller fraction of the total runtime. As such, scaling efficiency values cannot be directly compared across systems (due to hardware differences) or even with differently performing implementations on the same machine. Absolute performance metrics (per-GPU throughput metrics, like images/s or TFLOPS) must be consulted in this case. For example, MareNostrum 5 does not appear to outperform other systems in scaling efficiency despite its consistently very fast network, likely due to its GPUs being faster than those of many other systems. Also, `AMP` shows lower scaling efficiency than `FP32` because it is faster. Still, some odd performance behaviour occurs with `AMP`, meaning users should properly benchmark it when enabling it.

6. Conclusions

Chapter 2 describes the benchmarks being used to give an overview of AI scaling capabilities on the different machines. Chapter 3 presents the available EuroHPC machines, including specifications and guidelines for Python usage. Chapter 4 discusses tools and systems which enable running AI workloads at massive scale. Chapter 5 contains the results of running the benchmarks on all EuroHPC machines, providing insights into their performance.

From the available results, users cannot directly select an optimal machine and scale for their work. Instead, similar benchmarking, tuned for their specific application, should be performed, and special care should be taken to enable high performance at scale, both in terms of scaling efficiency and training efficiency. We conclude this Best Practice Guide with some general guidelines:

- Even if `module`-based Python packages are available, using containers is strongly recommended on Lustre-based systems, both for their higher performance, and control over package versions.
- Special care must be taken to pick base containers that perform well on the specific hardware of the given HPC.
- Benchmarking must be performed to find a desirable trade-off between training speed and training cost. For machines under high load, reducing scale to reduce queueing time may be desirable
- An informed selection of model distribution approaches must be made given system and model specifics.
- Apart from pure scaling efficiency, the impact of scaling on training efficiency and optimal hyperparameter selection must be investigated.
- Benchmark performance may fluctuate due to system load or node selection, meaning benchmarks should be repeated if possible.

Appendices

A. Unused Tools and Systems

Apart from those we eventually selected, many other tools and systems were also considered to be evaluated within this Best Practice Guide. Not being selected does not necessarily imply we recommend not using it. For transparency, all tools and systems we decided not to use are discussed in this appendix. We briefly describe it, then provide our motivation for omitting it. Motivations include the tool or system not being sufficiently mature, its development having stalled, being out of scope of this document, complicating accurate benchmarking, reducing configurability of the benchmark, or functionality already being included in the benchmark.

A.1. Frameworks

TensorFlow

TensorFlow was the dominant framework in Deep Learning research in the mid-to-late 2010s, before being overtaken by PyTorch. In industry, TensorFlow remains slightly more popular than PyTorch. It is, however, mostly used because of its stronger support for deploying and serving models. TensorFlow offers extensive support for deploying to browsers, mobile devices and other resource-constrained devices. Its serving system offers an extensive REST API, versioning and A/B testing.

In the context of this BPG however, publicly available tooling for LLM pretraining at scale with TensorFlow is severely lacking compared to PyTorch. A TensorFlow-based benchmark for vision was considered, as an implementation of ResNet-50 is offered in the Keras library. Given the limited use of TensorFlow in EPICURE projects, and Keras' recent move from being a frontend specifically for TensorFlow to being backend-agnostic (see Section A.2), we eventually decided against it.

JAX

JAX is a more recent Deep Learning framework, focusing on an interface reminiscent of NumPy, and performance speedups through Just-In-Time (JIT) compilation. It offers Deep Learning support through its Flax and Optax libraries, and supports running on CPU, GPU and TPU. Furthermore, it was designed with scalability and distribution in mind.

It is regularly used for LLM pretraining at massive scale, although this mostly stems from its TPU support. As such, most code available online is intended specifically for use on TPUs. Given that JAX is still experimental and rapidly evolving, we decided not to incorporate it into our benchmarks for now. Furthermore, its JIT compilation makes benchmarking its scaling performance in a repeatable manner challenging.

PyTorch Compile

Similar to JAX, PyTorch also introduced JIT compilation through PyTorch Compile in 2023. In addition to the challenges related to benchmarking JIT code, Compile is not compatible with all PyTorch functionality. Most notably, distributed models are only

supported when using data parallelism. As such, Compile was not considered in the benchmarks.

A.2. Higher-Level Deep Learning Libraries

Keras3

Keras was initially a library on top of TensorFlow, providing Deep Learning functionality. In late 2023, Keras 3 was released, rewriting the entire codebase, and supporting TensorFlow alongside PyTorch and JAX as backends. While being able to toggle between these backends at the flick of a switch may sound appealing for these benchmarks, Keras 3's support for *distributed* Deep Learning is lacklustre as of writing.

Data parallelism is supported for all backends, but only through each backend's own distribution modules. As such, the backends are no longer easily swappable.

For model parallelism, Keras 3 does offer an API, but it is only compatible with the JAX backend. In addition, Keras 3 offers a *sharding* API, but its use is only documented for the single-node use-case. As such, the main appeal of using Keras 3, flick-of-a-switch backend swapping, does not apply to the distributed multi-node HPC use case.

ONNX

ONNX is another framework often associated with interoperability between backends. One of its main features is that a model could be trained using one backend, then exported to another for deployment. ONNX does not facilitate writing code which allows for training with different backends. As such, ONNX is outside of the scope of this document.

PyTorch Lightning

Lightning is a high-level layer atop PyTorch which aims to simplify and more clearly structure code, eliminating much of the common boilerplate. Notably, it does not offer functionality which could not be achieved through pure PyTorch, but just simplifies the implementation. As such, Lightning is not expected to impact runtime performance of a benchmark.

Furthermore, the benchmark code considered is already rather straightforward, and taken from well-tested examples. Introducing Lightning could also complicate the introduction of certain tweaks that were needed for either at-scale training or compatibility with specific hardware.

NeMo

NVIDIA NeMo is an end-to-end framework for training, fine-tuning and deploying generative AI models, such as LLMs. It is built on top of PyTorch and offers implementations of many LLM architectures, alongside a wide range of components for data preparation, checkpointing and inference. It is primarily intended for use at scale on HPC systems and clusters.

For pretraining at scale, NeMo supports many advanced parallelism techniques. For this functionality, NeMo relies on Megatron. As such, for the purpose of our benchmarks, the added value of using the NeMo framework instead of interfacing directly with Megatron would be limited. The extra layer could abstract away the fine-grained control over training behaviour, which we need for proper benchmarking. Furthermore, as NeMo is built by NVIDIA, it is tightly coupled with CUDA, meaning that a NeMo-based benchmark would not be compatible with AMD-based machines. While we did not use NeMo, we note that it might be an appropriate tool for users working on NVIDIA-based supercomputers.

Foundation Model Stack

The Foundation Model Stack (FMS) is a relatively unknown framework by IBM for LLM training and inference. For training specifically, it relies on PyTorch, and implements Fully Sharded Data Parallelism (see Section A.3). However, it only implements the LLaMA model, and we were unable to get the framework running on AMD systems. Documentation is also very limited. As such, we did not use this framework.

OLMo

OLMo is a family of open-source language models. As it open sources all its training scripts (including checkpoints), it could also be considered a higher-level framework. For distribution, it mostly relies on PyTorch's Fully Sharded Data Parallelism implementation (see Section A.3). However, it does not appear to offer any functionality relevant to this BPG that cannot also be achieved through the more commonly used and more modifiable Megatron.

A.3. Parallelism Approaches

Fully Sharded Data Parallelism

Fully Sharded Data Parallelism (FSDP) is a more far-reaching version of the distributed optimizer used in the language benchmark. With a distributed optimizer, the state of the optimizer is distributed across devices, saving memory, and exchanged when needed for computation. With FSDP, the same is also done with the gradients and parameters of the model. Whenever a device needs gradients or parameters stored elsewhere for a calculation, they are collected, used and then again forgotten, reducing the memory footprint. FSDP can be used as the only type of parallelism, but is also compatible with all other types of parallelism described in this document.

PyTorch supports FSDP out-of-the-box. In 2023, PyTorch released a new implementation of the FSDP concept with the FSDP2 module. The original implementation has since been marked as deprecated. Both implementations are integrated into Megatron, but the integration of FSDP2 is still considered experimental by the developers. In addition, PyTorch FSDP developers note that the combination of FSDP with tensor and/or pipeline parallelism is, while theoretically valid, non-trivial to

implement properly. Furthermore, the LLM we consider in our benchmarks is both too large to be trained with only FSDP, but small enough to fit into a moderate number of devices with only model parallelism and no FSDP. Finally, we observed some stability issues when combining all types of parallelism. As such, FSDP is not considered in these benchmarks.

Expert Parallelism

In the Mixture of Experts (MoE) technique, certain layers of an LLM model are subdivided into sets of smaller layers, named *experts*. As the name implies, each expert is controlled to learn a specific “type” of information. Whenever data reaches this layer, a trained *router* will select a subset of experts, expected to be most eligible to process that data. Compared to an LLM without the technique, an LLM with MoE can reach the same performance with a lower number of parameters (or higher performance with the same number).

Whether or not an LLM of a specific number of parameters uses MoE is not expected to impact its scaling efficiency. However, its usage does enable another type of parallelism. In *Expert Parallelism*, different experts are placed on different devices. Compared to an approach where these layers are not distributed in an *MoE-aware* fashion, this reduces communication overhead, as devices running experts which were not selected are excluded from communication. This does however complicate scaling efficiency analysis. Experts on certain devices being selected more often than those on others may lead to idle time on less popular devices, reducing scaling efficiency. Importantly, the extent to which this occurs could differ significantly between different instantiations of training. Complicated load balancing mechanisms to counteract this exist, but implementing this was considered outside the scope of these benchmarks, meaning no expert parallelism (or MoE layers) was enabled.

A.4. Distribution and Scaling Tools

DeepSpeed

DeepSpeed is a PyTorch library commonly associated with Deep Learning at scale. Most notably, DeepSpeed is the reference implementation of ZeRO, a family of memory optimization techniques, which eliminate state redundancy between GPUs in distributed learning.

ZeRO provides three different stages, with each next stage offering more memory reduction at the cost of having a higher impact on model implementation and requiring more communication. With ZeRO-1, only the optimizer states are distributed across GPUs. In fact, Megatron's distributed optimizer, which was enabled in the language benchmark, is an implementation of ZeRO-1.

With ZeRO-2 and ZeRO-3, the gradients and the model parameters are additionally distributed across GPUs. As such, ZeRO-3 is conceptually very similar to FSDP. These latter stages are mostly intended for massive models, where the memory reduction is necessary to enable training at all.

To facilitate integration with Megatron, the DeepSpeed team maintains a fork of Megatron with DeepSpeed incorporated into it. However, it has not been synchronized

with upstream Megatron since mid-2023 and has not received a single commit since mid-2025. This would severely complicate synchronization with the (relatively up-to-date) AMD-specific fork of Megatron. Furthermore, as ZeRO-1 is already implemented within Megatron, we do not rely on any DeepSpeed implementations in these benchmarks.

Horovod

Horovod is another library commonly associated with at-scale Deep Learning. It implements data parallelism for both PyTorch and TensorFlow through its `ring-allreduce` algorithm. In the late 2010s, Horovod was essential for scaling training. However, PyTorch and TensorFlow have since introduced their own built-in modules for data parallelism, which perform similarly to Horovod. The library has not seen a new release since mid-2023, and not a single commit in five months as of writing. As such, we consider Horovod obsolete, and do not consider it in our experiments.

HF Accelerate

HF Accelerate, part of the Hugging Face ecosystem, is an abstraction layer for distributed training. Behind the scenes, it relies on lower-level libraries such as DeepSpeed or PyTorch's built-in distribution. It eliminates most boilerplate code, and is mainly aimed at users who are not yet familiar with the underlying systems. It does not implement its own algorithms, making it conceptually similar to PyTorch Lightning, but specifically for distribution. As such, our reasons for not incorporating it into the benchmarks are similar. In addition, Accelerate requires its own launcher, complicating integration with Slurm.

TorchTitan

TorchTitan is the PyTorch's team framework specifically for training LLMs at scale. It integrates tightly with PyTorch, and offers many advanced features such as 3D parallelism, FSDP, PyTorch Compile support and float8 support. AMD maintains its own fork of the framework. While promising for LLM training, the current state of TorchTitan is still too experimental to be considered for scaling performance benchmarking. Furthermore, it requires a very recent build of PyTorch, impeding deployment on some HPC systems. Once matured, this could however be a prime candidate for benchmarks like the language benchmark presented in this document.

A.5. Storage

In the vision benchmark, LMDB is used to wrap all images into a single file, and the file is placed in node-local storage, as the required random access is fundamentally incompatible with the Lustre file system. At the start of each run, it takes a few minutes to copy the LMDB files to node-local storage. This does not impact benchmark results, but does waste GPU-hours.

While the copy from shared storage to node-local was inevitable, we did consider formats other than LMDB. The commonly used HDF5 format was eliminated as it does not support datapoints with inconsistent dimensions well (images may have different

resolutions in ImageNet-1k). Another option is to store the images in a single archive (e.g., `.tar` or `.tar.gz`) and then extracting this to local storage, where having many files is acceptable. This, however, significantly increased startup times of each run, even without compression.

A viable approach is to use a SquashFS file containing all images, which could then be mounted from a node-local storage. Experimentation showed no performance differences compared to LMDB. We note that this is to be expected: PyTorch's `DataLoader` *preloads* input using the CPU, already performing preprocessing before the data is needed on the GPU. Analysis showed that a properly configured `DataLoader` can load data faster than it is consumed in the vision benchmark. At this point, any performance improvement in data preloading has no practical effect, as the CPUs just idle for longer. Similarly, slower preloading does not degrade overall performance as long as it remains faster than GPU-side processing of the data. Benchmarking of *only* the `DataLoader` also showed no noticeable difference between LMDB and SquashFS.