Early Experiences of Noise-Sensitivity Performance Analysis of a Distributed Deep Learning Framework

Elvis Rojas*†, Michael Knobloch‡, Nour Daoud‡, Esteban Meneses*§, Bernd Mohr‡
*Costa Rica Institute of Technology, †National University of Costa Rica, §Costa Rica National High Technology Center
‡Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Germany
erojas@una.ac.cr, m.knobloch@fz-juelich.de, n.daoud@fz-juelich.de, emeneses@cenat.ac.cr, b.mohr@fz-juelich.de

***Abstract*—Deep Learning (DL) applications are used to solve complex problems efficiently. These applications require complex neural network models composed of millions of parameters and huge amounts of data for proper training. This is only possible by parallelizing the necessary computations by so-called distributed deep learning (DDL) frameworks over many GPUs distributed over multiple nodes of a HPC cluster. These frameworks mostly utilize the compute power of the GPUs and use only a small portion of the available compute power of the CPUs in the nodes for I/O and inter-process communication, leaving many CPU cores idle and unused. The more powerful the base CPU in the cluster nodes, the more compute resources are wasted. In this paper, we investigate how much of this unutilized compute resources could be used for executing other applications without lowering the performance of the DDL frameworks. In our experiments, we executed a noise-generation application, which generates a very-high memory, network or I/O load, in parallel with DDL frameworks, and use HPC profiling and tracing techniques to determine whether and how the generated noise is affecting the performance of the DDL frameworks. Early results indicate that it might be possible to utilize the idle cores for jobs of other users without affecting the performance of the DDL applications in a negative way.**

***Index Terms*—Distributed Deep Learning, Performance Analysis, Noisy Environments**

I. Introduction

Today, Deep Learning (DL) applications are used to solve many complex problems efficiently. These applications require complex neural network models composed of millions of parameters and huge amounts of data for proper training. To be able to perform the training in a reasonable time, so-called distributed deep learning (DDL) frameworks are used. They parallelize the training by distributing necessary computations across available GPUs of multiple nodes of HPC cluster. These frameworks mostly utilize the compute power of the GPUs and use only a small portion of the available compute power of the CPUs in the nodes for I/O and inter-process communication, leaving many CPU cores idle and unused. This problem increases with modern powerful many-core CPUs which have up to 128 cores and nodes with multiple CPU sockets. In our experiments with PyTorch/Horovod, we determined that each MPI task, which controlled one GPU, used up to eight threads. So on a high-end HPC system node with four GPUs it would leave half (64-cores/node) or three-quarter (128-cores/node) of the cores unused.

In this paper, we investigate how much of this unutilized compute resources could be used for executing other applications (of potentially other users) without lowering the performance of the DDL frameworks. As most of the compute load of the DDL is utilizing the GPUs attached to the node, running "conventional" MPI/OpenMP HPC applications which only use the CPU cores left idle, is possible. In order to get reliable and repeatable results we used a noise-generation application, which generates a very-high memory, network or I/O load, instead of an example HPC application, and executed it in parallel with DDL frameworks. We used the well-established HPC profiling and tracing tools Score-P, Cube, and Vampir to determine whether and how the generated noise is affecting the performance of the DDL frameworks.

Early results indicate that it might be possible to utilize the idle cores for jobs of other users without affecting the performance of the DDL applications in a negative way. However, more extensive experiments need to be carried out before being able to make this claim.

The contributions of this paper are the following:

- We show how traditional HPC performance instrumentation and measurement tools can be used to analyze DDL frameworks.
- On the basis of a noise-sensitivity performance analysis, we investigated whether utilizing CPU cores left idle by DDL frameworks by executing other HPC applications is possible. First results show no negative performance impact on the execution of the DDL framework Horovod, required proper mapping of task/threads to CPU cores is provided.

After providing some background information on DL and DDL frameworks, HPC performance analysis, and noise generation in Section II, we describe our experimental methodology in more detail in Section III. Section IV describes the results of our preliminary experiments. Concluding remarks and future work are presented in Section V.

II. Background

In this section, we provide some background on (distributed) deep learning, HPC performance analysis, and noise generation frameworks and tools we use in our experiments.

A. Deep Learning

1) Deep neural networks and frameworks: DL applications base their operation on artificial neural networks (ANN). ANNs can be developed to solve specific problems taking

advantage of their ability to learn without requiring a redesign of their structure. In the context of DL, an NN becomes a deep NN when there are at least 3 interconnected layers of ANNs, namely an input layer, an output layer, and at least one layer in between, known as the hidden layer [1]. There is a wide variety of deep NN structures with different amounts of parameters, features and possible applications. One of the most common types of NNs are known as Convolutional NNs (CNNs) [2], which can also vary in structure. In this study we use ResNet, a type of CNN called residual in which shortcuts are used to move between layers [3].

DL frameworks provide support for the creation, implementation and execution of CNNs. They offer a high-level interface to ease the usage of NNs and hide the low-level details. In addition, they contain proprietary and third-party specialized libraries that allow the use of acceleration hardware (GPUs, TPUs) and the implementation of optimized parallelization mechanisms to speed up the execution of DL training. In this paper, the DL framework **PyTorch** [4] was utilized because it provides all the necessary elements for our experiments.

2) Distributed Deep Learning: Today, DL applications are used to solve complex problems efficiently. These applications require complex NN models composed of millions of parameters and huge amounts of data for proper training. This implies long training times that in the past with the limitations of hardware and software were impossible to perform. HPC systems are designed for the parallel execution of applications. These systems have evolved incorporating new accelerator components (GPUs) allowing to speed up DL applications by implementing parallelism in DL training. This leads to the concept of distributed training (DT) or distributed DL in general. In this study, the DL framework **PyTorch** was used, which allows implementing different mechanisms to convert sequential DL training into distributed training. This enables the execution of training with multiple GPUs on one or multiple nodes.

Horovod (HVD) is a library for DT that bases its operation on the data parallel approach. In this approach the CNN model is copied to each of the GPUs requested for training [5]. Figure 1 shows the overall structure of the operation of HVD. First, there is an initialization (hvd.init), then it performs a broadcast that ensures the correct initialization and synchronization of all processes. In addition, a distributed optimizer is implemented to delegate the gradient computation to the original optimizer and average the gradients. HVD bases its operation on the ring-allreduce algorithm to synchronize the gradients [6]. With this algorithm, 2(N-1) communication exchanges are generated between processes for sending and receiving the process data buffer chunks [7].

B. Performance Analysis

Many HPC performance analysis tools exist, each each with its own strengths and weaknesses. A good overview about available vendor-independent tools is provided by Mohr [8]. One example of using HPC tools for the analysis of DDL applications is provided in [9]. In our experiments, we used

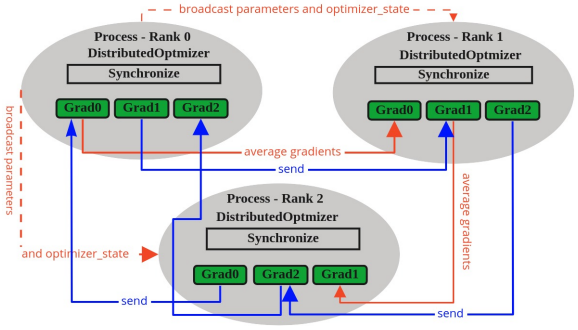

Fig. 1. Distributed training process in HVD.

the instrumentation and measurement framework Score-P, the profile browser CUBE, and the trace visualizer VAMPIR.

1) Score-P: **Score-P** [10] is a highly scalable tool for performance analysis of HPC applications through profiling and event tracing. It provides an instrumentation framework that make it possible to automatically insert measurements probes into HPC applications written in Fortran or C/C++. User functions are instrumented by using compiler options or plugins. Manual instrumentation allows to mark arbitrary phases in the codes, including loops. Additional Score-P measurement runtime libraries collect data related to performance (execution times, visits, communication metrics, hardware counters, etc) during application executions and support a wide variety of instrumentation and measurement options for typical HPC programming paradigms (MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenACC, among others). Profile results are written in the standard CUBE4 format and can be analyzed with Cube [11]. Traces are written in the standard OTF2 format [12] and can be investigated with the help of the trace visualizer Vampir [13].

2) Python Bindings for Score-P: Score-P is supporting instrumentation and measurement of applications written in C/C++ and Fortran. However, the DL framework used in this study for experimentation is PyTorch, whose high-level implementation is implemented in Python. To facilitate the instrumentation and measurement of Python components, we utilize a publicly-available extension to Score-P [14] for instrumenting Python code for profiling and tracing.

3) Visualization tools: **Cube** [11] provides the data format for the profile performance analysis reports and graphical user interface (GUI) for report examination. The Cube data model describes the performance behavior of an application in a three-dimensional space consisting of performance metric, call tree and system location. The GUI visualizes this in three coupled tree-browsers. An example screenshot of the Cube GUI is shown in Figure 4.

Vampir [13] is a trace visualization tool with which parallel program executions can be analyzed through multiple graphical representations like state diagrams, activity charts, time-line displays, and statistics (see Figure 5). In addition, it implements a powerful zooming feature that allows executions to be analyzed at any level of detail.

C. Noise Generation

1) NOIGENA: The noise-generator NOIGENA developed as part of the ExtraNoise project [15] allows to produce a consistent and repeatable noise effect for configurable amount of time on node level (intra-node) caused by shared resources contention (memory, network and IO). For generating a specific noise type, it integrates three open-source benchmarks:

- **Stream** for memory noise. STREAM[1] is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. It was developed by John D. McCalpin, Ph.D.
- **FZJLinkTest** for network noise . The Linktest[2] program is a parallel ping-pong test between all possible MPI connections of a machine. It is developed and maintained by Jülich Supercomputing Centre.
- **IOR** for IO noise . IOR[3] is a parallel IO benchmark that can be used to test the performance of parallel storage systems using various interfaces and access patterns. It is developed by Lawrence Livermore National Laboratory.

The noise-generation application can be configured based on a YAML file. It has two parts: The first part is allows configuring the memory, network and IO benchmarks. The second part specifies the exact noise patterns, which should be produced by the generator. A noise pattern either describes the specific sequence of the noise modes and their desired run time, or the desired run time of random noise with a specified distribution between the different noise types.

III. Methodology

A. System configuration

The experiments carried out used the JUWELS supercomputing system located at the Jülich Supercomputing Centre (JSC). This system is divided into two hardware partitions called Cluster and Booster. For this paper the Booster hardware partition was used. This partition is made up of 936 nodes with a performance of 73 petaFLOPS. Each node has two 24-core AMD EPYC Rome CPUs and four NVIDIA A100 GPUs. Table I shows the software tools and versions used in the experiments.

B. System architecture

Figure 2 shows the internal structure of a JUWELS Booster node. Experiments were performed on this architecture, so it is relevant to understand how the components are interconnected and interact to perform a more efficient assignment of tasks.

The important noteworthy aspects are:

- The two 24-core AMD EPYC CPU (shown as red blocks in the figure) each are divided up into four NUMA domains with six cores each. Each NUMA domain has separate network channels. The configuration originates from the production of the CPU chip, which is not produced as one monolithic die, but rather consists of four individual dies.
- The GPUs are not connected to a CPU (socket) in general but to specific NUMA domains in unexpected ways (namely GPU0 to NUMA domain 3, GPU1 to NUMA domain 1, GPU2 to NUMA domain 7, and GPU3 to NUMA domain 5)
- Each of the sockets (composed of 4 NUMA domains) is directly connected to a memory controller.

All of this has to be taking into account when mapping CPU tasks and threads to specific cores.

TABLE I
Software tools and versions

Software tool	Name	Version
Profile and trace tool	Score-P	7.1
Noise generator	NOIGENA	0.9
Visualization tools	Cube Vampir	4.7 10.0.0
Deep learning framework	PyTorch	1.11
DT mechanism	Horovod	0.24.3
Network model	ResNet	50
Dataset	CIFAR	100

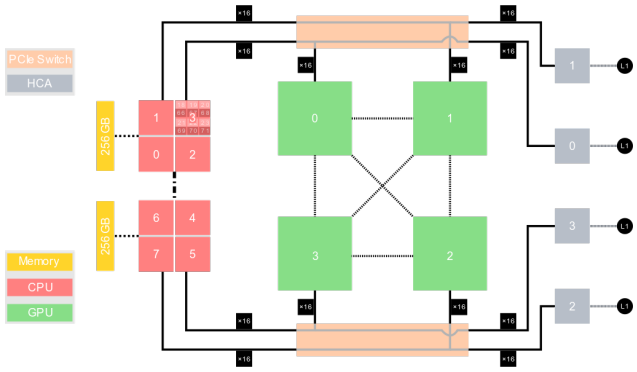

Fig. 2. Juwels-Booster node architecture[4]

C. Design of Experiments

The experiments carried out consist of planned executions of DDL training, which are instrumented for subsequent performance analysis. Additionally, these experiments are executed together with a noise generator software that is used to simulate special conditions that can affect performance (network, memory or I/O noise) in a consistent way. The most important elements that are part of the experiments are detailed below:

Deep Learning: The experiments were executed using the DL framework PyTorch together with Horovod to perform the distributed training. ResNet50 is used as CNN and CIFAR100 [16] as dataset. Each one of the trainings was executed for 5 epochs, which allows obtaining adequate execution times to be analyzed. It was necessary to instrument the PyTorch code to allow Score-P to collect the profiling and tracing information during the execution of the training runs.

[1]https://www.cs.virginia.edu/stream/

[2]https://www.fz-juelich.de/en/ias/jsc/services/user-support/jsc-software-tools/linktest

[3]https://sourceforge.net/projects/ior-sio/files/IOR%20latest/IOR-2.10.3/

[4]https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

To keep measurement overhead minimal, we didn't instrument the whole python code but used manual and selective instrumentation [17] to get detailed information on the relevant parts of the application.

Noise generator: A noise generator software called NOIGENA is used. For this initial study, only memory noise was used, as it generates noise with the highest impact.

Hardware resource planning: Resource allocation is done based on the architecture shown in Figure 2. The PyTorch-Horovod application makes use of all four GPUs of the node. In addition, 6 CPU cores per MPI task (4 x 6 = 24 cores) are used for Horovod initialization and synchronization tasks. The remaining 24 cores are assigned to the noise generator. The allocation of resources was done manually and two scenarios were implemented:

- **Exclusive** NUMA domains: In this scenario the cores used by PyTorch/Horovod are assigned to the same NUMA domains where the GPUs are connected. The noise generator is mapped to the remaining NUMA domains without GPU connections. For example, based on Figure 2, the GPUs are connected to NUMA domains 1, 3, 5, and 7, so the cores used by the noise generator are mapped to the cores of NUMA domains 0, 2, 6, and 4.
- **Shared** NUMA domains: In this scenario the cores used by PyTorch/Horovod and the noise generator are evenly spread over all NUMA domains, i.e. each application gets assigned 3 of the 6 cores in each NUMA domain.

We manually assign/map the tasks and threads of the DDL training and noise generation to specific CPU cores for controlled and repeatable experiments using extra options to the parallel execution command (`srun`) of the job scheduler (`slurm`). Figure 3 shows the basic slurm job script template we used.

```
#SBATCH --nodes=1
#SBATCH --gres=gpu:4
#SBATCH --partition=booster

srun -n 4 --exact --cpu-bind=<spec1> \
     horovod-script &
srun -n 24 --exact --cpu-bind=<spec2> \
     --gpus=0 noigena &
wait
```

Fig. 3. Simplified Template slurm job script for simultaneous execution.

Each of the experiments carried out requires the simultaneous execution of the training and the noise generator on distinct subsets of cores of the compute node (specified via the `--exact` option and the `<spec>` parameter of the `--cpu-bind` option). By running the commands in the background (via "&" at the end of the command) and then waiting for them, enables the asynchronous and thereby simultaneous execution of the two commands. In addition, we added a simple time synchronization at the beginning of the Horovod main code and NOIGENA, to ensure that the jobs submitted to run in parallel start executing at the same time. Finally, the CPU-only command (here: the noise generator) has to declare that it is not using any GPUs (via `--gpus=0`), otherwise it would be blocked until the GPUs are available, and therefore would not run simultaneously.

Analysis and visualization: After the execution of the experiments, an analysis of the profiling and tracing data generated by Score-P is performed. For this analysis, the Cube and Vampir applications are used.

TABLE II
PERFORMANCE OF DL TRAINING (EXECUTION TIME IN SECONDS) IN SINGLE NODE SHARED AND EXCLUSIVE NUMA ENVIRONMENTS.

	Exclusive NUMA Domains		**Shared NUMA Domains**	
Repetition	No Noise	With Noise	No Noise	With Noise
1	135.36	144.24	145.80	284.34
2	138.81	138.94	147.63	190.37
3	136.11	138.31	147.91	242.95
4	134.32	134.83	146.40	245.49
5	136.01	138.12	145.06	263.57
6	135.17	137.04	144.38	194.85
7	137.26	135.39	144.55	257.87
8	134.33	138.29	143.31	283.81
9	135.05	141.70	146.14	293.36
10	135.31	135.16	144.99	306.13
MIN	134.22	134.83	143.31	190.37
Average time	135.77	138.20	145.62	256.27
Standard Dev.	1.377	2.97	1.45	39.26

TABLE III
SUMMARY OF THE PERFORMANCE OF DL TRAINING (EXECUTION TIME IN SECONDS) IN TWO NODE SHARED AND EXCLUSIVE NUMA ENVIRONMENTS.

	Exclusive NUMA Domain		**Shared NUMA Domain**	
Repetition	No Noise	With Noise	No Noise	With Noise
MIN	81.13	77.92	83.44	213.86
Average time	81.81	81.05	85.58	231.21
Standard Dev.	1.54	1.45	1.36	23.50

IV. EXPERIMENTAL RESULTS

A. Noise-Sensitivity Analysis of DDL

For our main experiment, we used one node of the JSC JUWELS Booster partition and we setup four different configurations. We executed the PyTorch/Horovod DDL with and without noise, i.e. executing NOIGENA in memory-noise mode in parallel on the remaining cores or leaving these idle. We repeated these experiments twice with different mappings of the task and threads to CPU cores, namely with exclusive or shared NUMA domains as explained in the last section. Finally, for each configuration, we executed it 10 times to be able to observe the usual run-time fluctuations on a heavily loaded cluster like the JSC JUWELS production system. Table II summarizes the results from this experiments. The values in the table are the total execution times of the Horovod main code in seconds. From the table, we can make multiple observations:

- As expected, the execution time of the shared NUMA domains mapping is higher compared to the exclusive one. In shared mode, half of the Horovod threads end up

on the NUMA domains with only indirect access to the GPUs, slowing down execution.
- Also as expected, when executed with the shared NUMA domains mapping, Horovod is very sensitive to NOISE, as in that case PyTorch/Horovod and NOIGENA share Level-2 and Level-3 caches. The impact can be quite high: on average, the execution is 75% slower, and in one case we measured a doubling of the execution time.
- However, when executed with the exclusive NUMA domains mapping, where PyTorch/Horovod and NOIGENA do not share any caches, there is no measurable difference in execution time of the DDL training. Please note that NOIGENA is representing the worst case here, as in memory noise mode, it only stresses the memory. A typical HPC MPI/OpenMP application also has computation, communication and I/O phases, so the "noise" impact on the DDL training would be lower.
- We repeated the experiments on two nodes (using eight GPUs). The results (see Table III) are confirming the observations from the one node experiments.
- The results indicate that it might be possible to run (non-GPU) HPC applications simultaneously with DDL trainings on the same node utilizing cores left idle by the DDL framework, **iff** the processes and threads of both applications are carefully mapped to distinct subsets of cores taking the node architecture into account.

B. Performance Analysis of DDL

In this section, we want to demonstrate that traditional HPC tools can be used to analyze the performance of DDL frameworks like PyTorch/Horovod. With the help of the Score-P Python extension [17], we instrumented the main Horovod phases. With the core Score-P, the MPI communication and synchronization, POSIX thread management, and CUDA kernel executions were captured and measured.

Figure 4 shows the Cube result display of a profile measurement of the fastest plain PyTorch/Horovod execution on one node of JUWELS with exclusive NUMA domains mapping (without noise). The left pane of Cube shows the available (measured) metrics, the middle pane the program call tree, and the right pane the system tree. In the left pane, we see that 65.63% of the time, the programs spends computing, 21.29% in MPI functions, and 12.27% in total in CUDA (General and Memory Management, Synchronization, Kernel Launches). That means, that in this experiment, about a third of the total execution time in used for managing inter-process communication and GPU interaction. Besides measuring execution time, number of calls (visits), MPI statistics, Score-P also measured some additional GPU metrics via CUPTI (the last seven metrics in the left pane).

When we select a metric in the left pane (like Computation in the Figure), we can see how the value of this metric is distributed over the source code. In the example, we can see that 12.25% of the computation time are spent in `torch.autograd:backward`, 4.87% in `torch.nn.modules.module:_call_impl`, and 3.62% in `torch.optim.optimizer:wrapper`. By selecting these three call tree nodes, we can see that the execution time is spread equally over the four MPI processes (in the right pane). Besides the main Horovod phases, the middle pane also list all kernel executions (not all shown in the picture).

When we repeat this analysis for the slowest plain PyTorch/Horovod execution on one node of JUWELS with shared NUMA domains mapping (without noise), we find that now only 62.70% is spent computing, while MPI overhead increases by 4.73%. On the code side, the optimizer step (`torch.optim.optimizer:wrapper`) takes here a 24.65% share of the computation time, compared to the 3.62% in the other case. Further, more detailed analysis is required to better understand the impact of noise to the DLL training.

Figure 5 shows the Vampir display of a trace measurement of the same plain PyTorch/Horovod execution on one node of JUWELS (without noise). In the left pane, a (small) portion of the overall timeline of the execution is shown, at a point where the the Master threads of each MPI task is executing one instance of the `torch.autograd:backward` function (shown as purple line).

The next 4 lines below them are showing kernel executions on CUDA streams. It is clearly visible that only the first 2 streams are used, and that the GPUs are not fully utilized. The line with the red bars shows each of the CPU threads executing MPI communications, with MPI rank 0, 1, and 3 mostly spending their time waiting in `MPI_Allreduce` for MPI rank 2. Most of the remaining CPU threads are idle with the exception of one CPU thread per rank which is launching asynchronously CUDA kernels.

Besides showing the timeline, Vampir also provides various displays for statistics. The values shown there depend on the execution time window of the main time line, allowing the user to calculate statistics for any portion of the execution. In the Figure, the right side shows a Function Execution Time Summary Statistic.

V. Concluding Remarks

In this paper, we presented results from early experiments, which show that it is possible to use cores left idle by DDL frameworks on HPC GPU clusters by executing other applications in parallel without a negative impact on the performance. A careful mapping of task and threads to distinct subset of cores is required to allow a simultaneous execution of both applications without interfering each other.

While these results are promising, more experiments are needed for confirmation. In the future, we want to extend our experiments by running them on a much larger scale (multi-node), which would also allow us to use more realistic use cases (with a much higher memory footprint) for the DDL training. Multi-node cases will us also allow to experiment with additional generated noise sources like network or I/O noise. Finally, we also want to experiment with other DL/DDL frameworks like TensorFlow/Horovod and PyTorch/DDP.

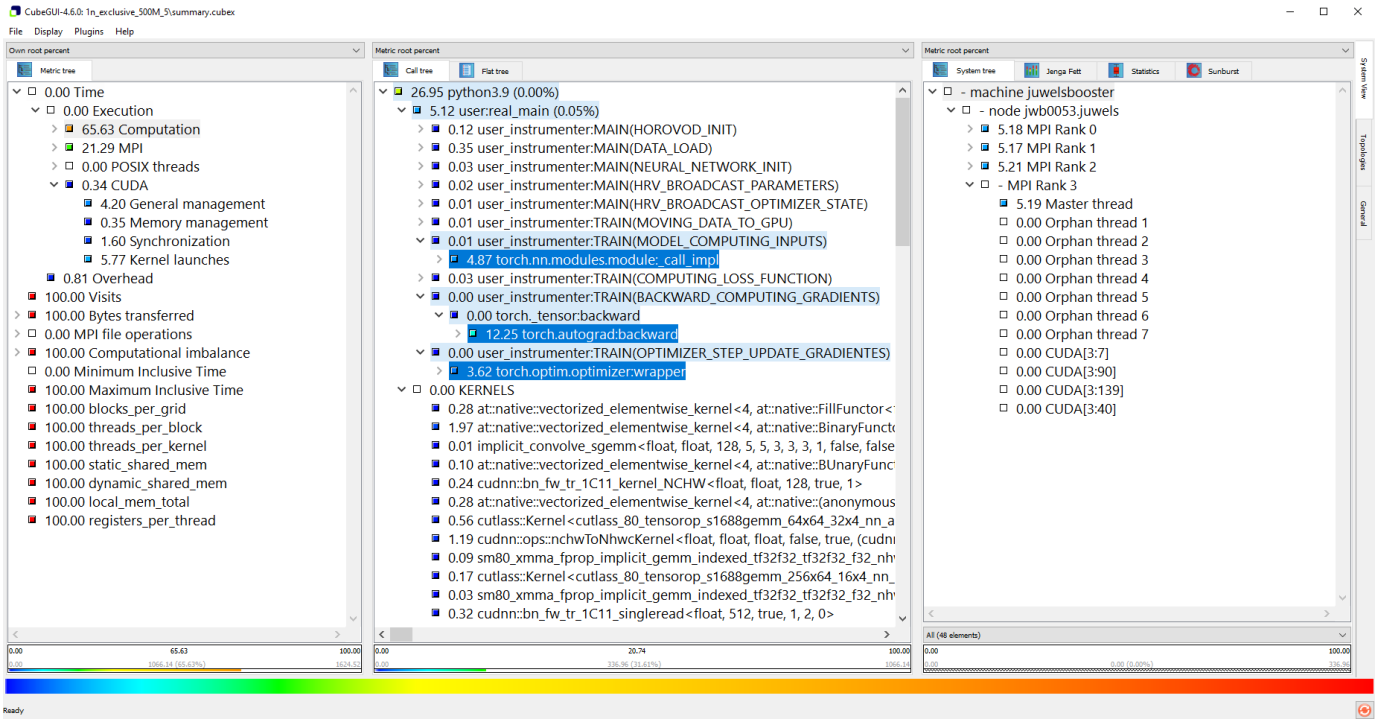

Fig. 4. Cube result display of plain PyTorch/Horovod execution on one node of JUWELS with exclusive NUMA domains mapping (without noise). Left pane shows available metrics, middle pane the program call tree, right pane the system tree. Numbers are shown in "Metric Percentage" mode.

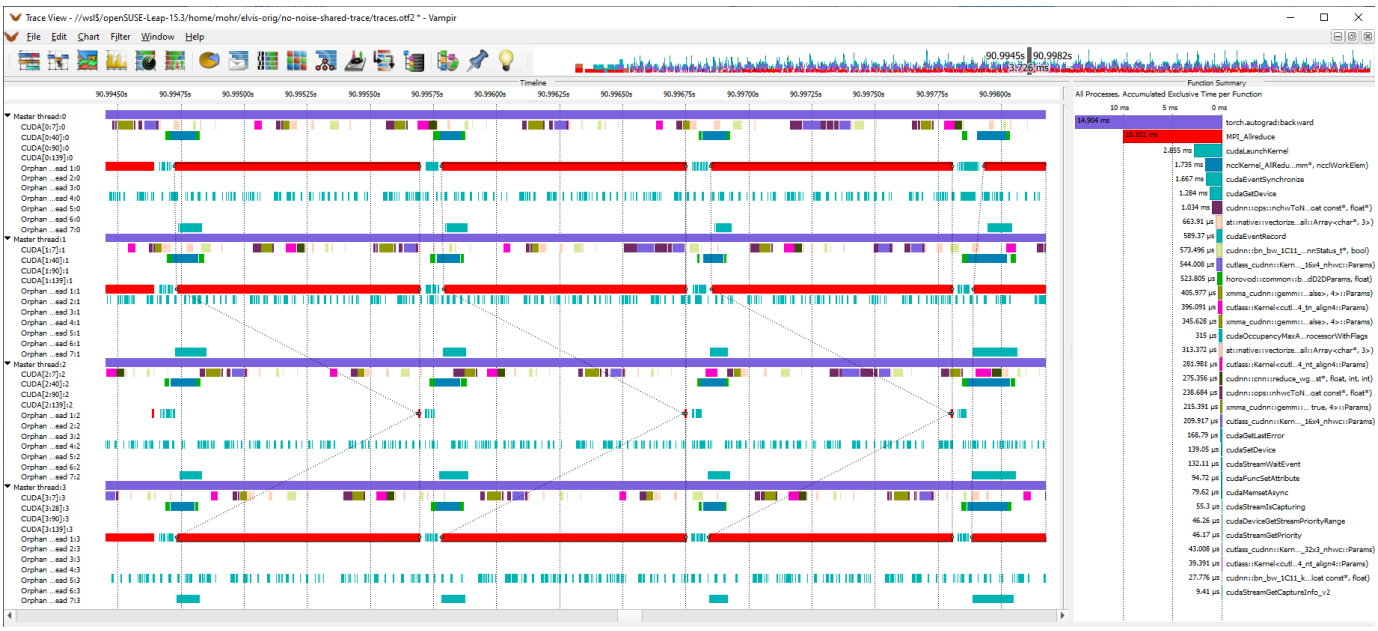

Fig. 5. Vampir display of plain PyTorch/Horovod execution on one node of JUWELS (without noise). On the left, a (small) portion of the overall timeline of the execution is shown, on the right, the function execution time statistic.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS [18] at Jülich Supercomputing Centre.

We also want to thank JSC support staff Max Holicki for the very helpful explanations of the Juwels Booster compute node structure and the details of task/thread mapping onto CPU cores.

REFERENCES

[1] A. Farkas, G. Kertsz, and R. Lovas, "Parallel and distributed training of deep neural networks: A brief overview," in *2020 IEEE 24th International Conference on Intelligent Engineering Systems (INES)*, 2020, pp. 165–170.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 8490, may 2017. [Online]. Available: https://doi.org/10.1145/3065386

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[4] A. Paszke and et. al., *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[5] E. Rojas, F. Quirós-Corella, T. Jones, and E. Meneses, "Large-scale distributed deep learning: A study ofmechanisms andtrade-offs withpytorch," in *High Performance Computing*, I. Gitler, C. J. Barrios Hernández, and E. Meneses, Eds. Cham: Springer International Publishing, 2022, pp. 177–192.

[6] A. Gibiansky. (2017, feb) Bringing hpc techniques to deep learning. [Online]. Available: https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/

[7] A. Sergeev and M. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," 02 2018.

[8] B. Mohr, "Scalable parallel performance measurement and analysis tools - state-of-the-art and future challenges," *Supercomputing Frontiers and Innovations*, vol. 1, no. 2, p. 108123, Sep. 2014. [Online]. Available: https://superfri.org/index.php/superfri/article/view/18

[9] A. L. Veroneze Solórzano and L. Mello Schnorr, "Understanding distributed deep learning performance by correlating hpc and machine learning measurements," in *High Performance Computing*, A.-L. Varbanescu, A. Bhatele, P. Luszczek, and B. Marc, Eds. Cham: Springer International Publishing, 2022, pp. 275–292.

[10] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[11] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, "Cube v4: From performance report explorer to performance analysis tool," *Procedia Computer Science*, vol. 51, pp. 1343–1352, Jun. 2015.

[12] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open trace format 2: The next generation of scalable trace formats and support libraries," in *Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 2012, pp. 481–490.

[13] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis toolset," in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 139–155.

[14] A. Gocht, R. Schöne, and J. Frenzel, "Advanced python performance monitoring with score-p," in *Tools for High Performance Computing 2018/2019*. Springer, 2021, pp. 261–270.

[15] D. A. Nikitenko, F. Wolf, B. Mohr, T. Hoefler, K. S. Stefanov, V. V. Voevodin, A. S. Antonov, and A. Calotoiu, "Influence of Noisy Environments on Behavior of HPC Applications," *Lobachevskii journal of mathematics*, vol. 42, no. 7, pp. 1560 – 1570, 2021. [Online]. Available: https://juser.fz-juelich.de/record/894423

[16] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[17] A. Gocht-Zech, A. Grund, and R. Schone, "Controlling the runtime overhead of python monitoring with selective instrumentation," in *2021 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2021, pp. 17–25. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ProTools54808.2021.00008

[18] JSC, "JUWELS Cluster and Booster: Exascale Pathfinder with Modular Supercomputing Architecture at JSC," *Journal of large-scale research facilities*, vol. 7, no. A138, 2021.