Experimenting with Emerging RISC-V Systems for Decentralised Machine Learning

Gianluca Mittone
gianluca.mittone@unito.it
University of Turin
Turin, Italy

Nicoló Tonci
nicolo.tonci@phd.unipi.it
University of Pisa
Pisa, Italy

Robert Birke
robert.birke@unito.it
University of Turin
Turin, Italy

Iacopo Colonnelli
iacopo.colonnelli@unito.it
University of Turin
Turin, Italy

Doriana Medić
doriana.medic@unito.it
University of Turin
Turin, Italy

Andrea Bartolini
a.bartolini@unibo.it
University of Bologna
Bologna, Italy

Roberto Esposito
roberto.esposito@unito.it
University of Turin
Turin, Italy

Emanuele Parisi
emanuele.parisi@unibo.it
University of Bologna
Bologna, Italy

Francesco Beneventi
francesco.beneventi@unibo.it
University of Bologna
Bologna, Italy

Mirko Polato
mirko.polato@unito.it
University of Turin
Turin, Italy

Massimo Torquati
massimo.torquati@unipi.it
University of Pisa
Pisa, Italy

Luca Benini
luca.benini@unibo.it
University of Bologna
Bologna, Italy

Marco Aldinucci
marco.aldinucci@unito.it
University of Turin
Turin, Italy

ABSTRACT

Decentralised Machine Learning (DML) enables collaborative machine learning without centralised input data. Federated Learning (FL) and Edge Inference are examples of DML. While tools for DML (especially FL) are starting to flourish, many are not flexible and portable enough to experiment with novel processors (e.g., RISC-V), non-fully connected network topologies, and asynchronous collaboration schemes. We overcome these limitations via a domain-specific language allowing us to map DML schemes to an underlying middleware, i.e. the *FastFlow* parallel programming library. We experiment with it by generating different working DML schemes on x86-64 and ARM platforms and an emerging RISC-V one. We characterise the performance and energy efficiency of the presented schemes and systems. As a byproduct, we introduce a RISC-V porting of the PyTorch framework, the first publicly available to our knowledge.

CCS CONCEPTS

• **Hardware** → **Emerging architectures**; • **Computing methodologies** → **Neural networks**; **Distributed algorithms**.

KEYWORDS

Federated Learning, Edge Computing, RISC-V, Energy Consumption, Green Computing

1 INTRODUCTION

Recent years have been characterised by crucial advances in Machine Learning (ML) systems. These advancements have been made possible thanks to the widespread availability of massive and ubiquitous computational resources and copious and distributed data sources. The consequent deployment of ML methods across many industries has generated concerns about data access and movement, such as performance, energy efficiency, security, and privacy [10, 12, 18]. Furthermore, companies consider collected data as competing advantages and are unwilling to share it outside the

organisation. This results in data being dispersed into isolated islands, and ML practitioners are forbidden from collecting, fusing, and ultimately using the data to improve their systems.

Decentralised ML (DML) using Federated Learning (FL) and Edge Inference (EI) tackles the difficulties mentioned above, but practical implementations are not straightforward. The emergence of alternative ISAs to x86-64 and ARM-v8, such as RISC-V, exacerbates the challenges in porting (optimised) libraries and guaranteeing interoperability between heterogeneous systems. Many off-the-shelf frameworks are available, each specifically designed to implement one of a few use-case scenarios. First, while standard FL is based on a master-worker approach, few emerging techniques explore distributed, sparse graphs. Second, to our knowledge, no FL framework still allows the user to specify a personalised, experimental communication graph between the parties taking part in the federation, posing a severe limit to researchers. Third, most current FL frameworks are implemented in Python, and their communication infrastructure is based on the gRPC and protobuf technologies. While this approach is practical, we advocate that it is not the most efficient. Finally, no FL framework has yet been proven on the RISC-V ISA. A modern DML framework should be flexible in defining the system architecture and communication patterns to open up research on alternative FL systems and be able to exploit optimised distributed runtimes for optimal performance on different ISAs.

Given this objective, we propose a top-down methodology to describe and implement flexible and fast DML systems. Our methodology leverages two proven tools: a formal language designed for describing parallel processes, namely RISC-pb^{2}l (RISC Parallel Building Block Library), which we adapt to the DML domain, and a lightweight C++ distributed run-time, namely *FastFlow*, which we port to the emerging RISC-V hardware. The translation between these two components is effortless since any valid DML system description can be directly translated into a *FastFlow* program. We demonstrate the effectiveness and flexibility of our approach by showcasing it across many experiments on three different learning schemes, spanning FL and EI, and different hardware architectures, x86-64, ARM and the emerging RISC-V, characterising their performance and energy efficiency.

In summary, our contributions are as follows:

- a methodology to design flexible, decentralised learning schemes and implement them on a *lightweight distributed runtime* enabling the experimentation and operation of ML at heterogeneous edge, especially aiming at FL and EI;
- proving system design flexibility via three different FL and EI use cases (master-worker, peer-to-peer and tree-based aggregation);
- an evaluation to showcase support of x86-64, ARM and emerging RISC-V ISAs;
- the first publicly available porting of the PyTorch framework and *FastFlow* library for RISC-V.

2 BACKGROUND AND MOTIVATION

2.1 Federated Learning

Federated learning has been proposed by [17] as a way to develop better ML systems without compromising the privacy of final users and the legitimate interests of private companies. Initially deployed by Google for predicting text input on mobile devices, FL has been adopted by many other industries, such as mechanical engineering and health care [15].

FL is a learning paradigm where multiple parties (*clients*) collaborate in solving a learning task using their private data. Importantly, each client's data is not exchanged or transferred to any participant. Clients collaborate by exchanging local models via a central server in its most common configuration. The server (*aggregator*) collects and aggregates the local models to produce a global model. The global model is then sent back to the clients, who use it to update their local models. Then, using their private data, they further update the local model. This process is repeated until the global model converges to a satisfactory solution or another termination condition is met (e.g., a maximum number of rounds).

There are two main FL settings: cross-device and cross-silo, with different challenges. In cross-device FL, the parties can be edge devices (e.g., smart devices and laptops), and they can be numerous (order of thousands or even millions). Parties are considered not reliable and with limited computational power. In the cross-silo FL setting, the involved parties are organisations; the number of parties is limited, usually in the $[2, 100]$ range. Given the nature of the parties, communication and computation are no real bottlenecks.

2.2 Edge Inference

Inference in DNNs [23] is much more lightweight than training. Notice that, roughly speaking, one may expect the single inference step cost to be about 1/3 of the cost needed for performing a single training step of the same model on similarly sized data due to additional loss evaluation and backward pass required in the training phase [1, 16]. Relative performance varies significantly between network architectures (e.g., convolutional networks tend to pay a higher cost during training) and between system architectures (e.g., executing the operations on GPU may change the relative cost of training and inference). The inference is generally computed on a single (possibly SIMD/GPU/TPU accelerated) processing element. At the distributed system level, a distributed inference process can be described with a directed acyclic graph, whereas FL requires a cyclic graph. This makes both cases interesting for experimentation.

2.3 Limits of Mainstream FL Frameworks

Different FL frameworks already exist on the market [7]. However, they are relatively homogeneous, mainly targeting DNNs and focusing on the goodness of the learned model rather than the performance of the distributed infrastructure and communications involved in the federated process.

Table 1 compares the FL frameworks found mature by [7]. Most of these target the cross-silo scenario, with only a few capable of handling the intricacies of large-scale, unreliable cross-device training, such as FederatedScope [24] and Flower [8]. Similarly, all reported FL frameworks support a simulation mode, allowing experimenting and debugging a Federated system locally; however, it is not trivial that they also support a real-world-oriented distributed mode, like TFF [5], SecureBoost [11], and OpenFL [20]. From the abovementioned perspective, the most limited FL framework is LEAF [9]: this software is explicitly designed to be used only for

Table 1: Brief overview of some of the mature FL frameworks available on the market, based on [7].

Framework	Target	Scenario	Comm. protocol	Implementation	Programmable comm. graph
TFF [5]	Cross-silo	Simulation/Real	gRPC/proto	Python	✗
PySyft [25]	Cross-silo/Cross-device	Simulation/Real	Websockets	Python	✗
SecureBoost [11]	Cross-silo	Simulation/Real	gRPC/proto	Python	✗
FederatedScope [24]	Cross-silo/Cross-device	Simulation/Real	gRPC/proto	Python	✗
LEAF [9]	Cross-silo	Simulation	-	Python	✗
FedML [14]	Cross-silo	Simulation/Real	gRPC/proto MPI, MQTT	Python	✗
OpenFL [20]	Cross-silo	Simulation/Real	gRPC/proto	Python	✗
Flower [8]	Cross-silo/Cross-device	Simulation/Real	gRPC/proto	Python	✗
Our proposal	Cross-silo	Simulation/Real	**TCP/Cereal MPI/Cereal**	**C/C++**	✓

benchmarking purposes. From a communication infrastructure perspective, almost all frameworks rely on the gRPC/proto combo. While such an implementation is effective and well-supported, some implementations try to offer alternatives: this is the case of PySyft [25], which offers a communication interface based on web sockets, and FedML [14], which supports a wide range of communication backends, including MPI and MQTT for, respectively, more high-performance and energy-efficient communication.

All listed FL frameworks share two characteristics, though: i) they are all implemented in Python, thus not taking performances seriously into account; and ii) all of them offer just one (or a small subset of) federation scheme; in other words, this software is highly specialised in handling just one communication graph and does not allow the user to specify and experiment with their personalised, complex, non-standard federation schema. These implementation choices constrain the possibility offered to researchers, who must roam over a vast range of FL frameworks to study how different federation schemes perform at the system and model levels. Furthermore, since all FL frameworks have similar design ideas and implementation, there is no real alternative besides Python for FL practitioners. This fact can be problematic, especially when working with experimental or embedded hardware architectures which are power and computation constrained, such as edge nodes using emerging RISC-V CPUs. In contrast, our proposal leverages C/C++ for highly optimised code execution and a domain-specific language to specify the most suitable federation scheme. We verified experimentally the overhead introduced by using PyTorch's Python interface in place of the C++ LibTorch interface on the RISC-V platform. We chose RISC-V to stress the difference on a platform with low computational power and lack of optimised libraries. Using the MNIST benchmark from the official PyTorch examples, we trained the same CNN, comparing the runtimes of the two APIs. The results averaged across 5 runs show that the C++ API (314.5s) is about 30% faster than the Python API (442.8s).

3 A TOP-DOWN APPROACH TO FL

We propose a top-down approach to implement a FL (or EI) system in the following. We show that modifying the common FL abstraction stack for more flexible and powerful frameworks is possible. This approach led us to deal with high-level issues, such as modelling of the distributed computation, and low-level issues, such as software porting to the RISC-V platform. The process starts with the definition of the system through a formal language for describing distributed processes, namely an adapted version of the RISC-pb^{2}l [2] language. This definition then translates into an implementation by mapping it to the building blocks of a computation middleware, namely the *FastFlow* parallel library [4]. This workflow is straightforward and will be practically demonstrated in Section 4.

3.1 Describing the FL System

The first step in creating our FL framework is to design a high-level modelling language. Such a language should be sufficiently abstract to allow to model a wide range of distributed computations while still being sufficiently specific to be easily applicable in different contexts. With this aim in mind, we use the RISC-pb^{2}l formal language [2] for parallel processes and adapt it to describe distributed FL workloads. RISC-pb^{2}l defines so-called Building Blocks (BBs), recurrent data-flow compositions of concurrent activities working in a streaming fashion, which are the primary abstraction layer for building parallel patterns and, more generally, streaming topologies [2, 22]. Furthermore, RISC-pb^{2}l has an already available high-performance software implementation, that is *FastFlow*, which is also compatible with the RISC-V ISA. These facts were crucial for choosing RISC-pb^{2}l as language abstraction for this research work.

We make RISC-pb^{2}l location-aware to suit the FL systems characteristics better since, in this scenario, each node owns different data. More specifically, we introduce a *distribute* building block which computes distributively a function on a set of nodes specified via a superscript. Table 2 reports the syntax and description of all RISC-pb^{2}l building blocks used in this paper. For the sake of space, we refer to [2] for the complete set and detailed composition grammar. The advantage of such a formalism is the possibility to

Table 2: Brief description of used RISC-pb^{2}l building block.

Syntax	Semantics
$((f))$	**Seq wrapper** Wraps sequential code into a RISC-pb^{2}l "function".
$(\|f\|)$	**Par wrapper** Wraps any parallel code into a RISC-pb^{2}l "function".
$[\|\Delta\|]^N$	**Distribute** Computes $\|N\|$ Δ distributively on the node set N producing $\|N\|$ outputs.
$\Delta_1 \bullet \cdots \bullet \Delta_n$	**Pipe** Uses n different programs as stages to process the input data items and to obtain output data items.
$(_g \triangleright)$	**Reduce** Computes a output item using an l level ($l \geq 1$) k-ary tree. Each node in the tree computes a k-ary function g.
$(_f \triangleleft)$	**Spread** Computes n output items using an l level ($l \geq 1$) k-ary tree. Each node in the tree computes a k-ary function f.
$\triangleleft_{D-Pol}$	**1-to-N** Sends data received on the input channel to one or more output channels. $D - Pol \in [Unicast(p), Broadcast, Scatter]$.
$\triangleright_{G-Pol}$	**N-to-1** Sends data from the n input channels on the single output channel. $G - Pol \in [Gather, Gatherall, Reduce]$.
$\overleftarrow{(\Delta)}_{cond}$	**Feedback** Routes output data y back to the input channel according to $Cond(x)$.

design and discuss the distributed system before its implementation in a clear and structured way. Furthermore, this formalism allows discussion of the computational properties of a design and also to apply optimisations before the implementation phase.

3.2 Implementing the FL System

We need to map the system description to runnable code to implement the FL (or EI) system. Here we use the C++ header-only *FastFlow* library [4], which was developed alongside RISC-pb^{2}l. The *FastFlow* building blocks match in a one-to-one fashion the RISC-pb^{2}l building blocks, thus allowing for a straightforward implementation of a system given its formal specification. Following the principles of the structured parallel programming methodology, a parallel application (or one of its components) is conceived by selecting and adequately assembling a small set of well-defined BBs modelling data and control flows. These can be combined and nested in different ways, forming acyclic or cyclic concurrency graphs. The original *FastFlow* implements nodes as concurrent entities and edges as communication channels carrying data pointers. Recently, the *FastFlow* run-time system has been extended to deploy *FastFlow* programs in distributed-memory environments [21]. By introducing a small number of edits, the programmer may port shared-memory parallel *FastFlow* applications to a hybrid implementation (shared-memory plus distributed-memory) in which parts of the concurrency graph will be executed in parallel on different machines according to the well-known SPMD model. Such refactoring involves introducing distributed groups (dgroups) which identify

shared-memory streaming graphs

distributed-memory streaming graph with K+1 groups (G0,…, Gk)
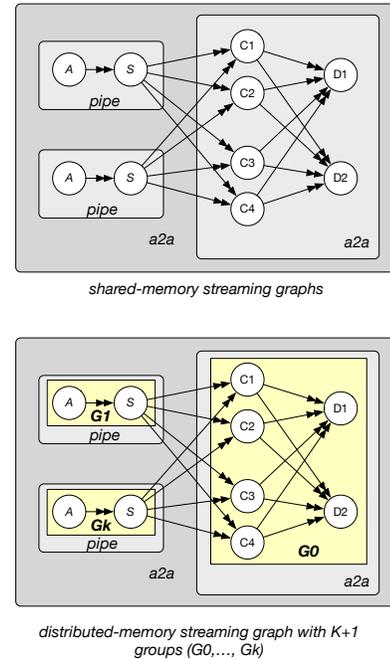
Figure 1: Example of a *FastFlow* application: The communication topology is described as a composition of building blocks in a data-flow graph, partitioned into distributed groups.

logical partitions of the BBs composing the application streaming graph according to a small set of graph-splitting rules. An example of a *FastFlow* application partitioned in distributed groups is given in Figure 1. Currently, inter-*dgroup* (i.e., inter-process) communications leverage raw TCP/IP or MPI, whereas intra-*dgroup* communications use highly efficient lock-free shared-memory communication channels [3].

The translation from RISC-pb^{2}l into *FastFlow*'s BB is straightforward since BBs can be configured differently and composed together to obtain a one-to-one mapping. In this sense, the *FastFlow* BB result to more expressive than their symbolical counterparts and can consequently be modelled to achieve efficient code implementations. The expressiveness and flexibility of *FastFlow*'s BB are offered in shared-memory and distributed-memory systems almost transparently. The translation has been done manually, but we envision that a small compiler would be capable of handling such a job.

4 USE CASES

4.1 Federated Learning

We adopt federated averaging (FedAvg) [17] as aggregation strategy for our use cases since it is one of the most popular FL algorithms. FedAvg organizes model training into rounds. At the start of each round, the aggregator distributes the weights of the aggregated (global) model to (a random subset of) participants; then, each client trains the model over one or more epochs using its local data before sending back the updated weights. Finally, the aggregator averages the received model weights to produce the updated global model
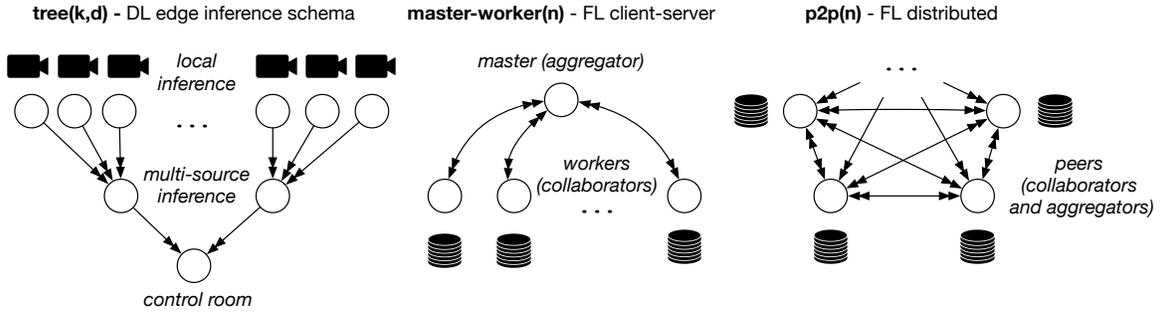
Figure 2: The three considered use cases: tree-based edge inference, master-worker, and peer-to-peer federated training.

for the next round. To showcase the flexibility of our approach, we consider two system topologies for training (see Fig. 2): a tree rooted at the aggregator mimicking the classic master-worker approach and a mesh made of peers, which combines the functionality of an aggregator and a worker, to avoid any central point of failure.

Using the RISC-pb^{2}l formalism, the master-worker FL process can be described as

$$((\text{init})) \bullet \overbrace{\left([|(|\text{test}|) \bullet (|\text{train}|)|]^W \bullet (_{\text{FedAvg}} \triangleright) \bullet \triangleleft_{\text{Bcast}}\right)}_r$$

where $((\text{init}))$ is the function initialising the communication graph and creating the starting models, W is the set of workers, and r is the condition checking if the prefixed number of rounds has been reached. On the other hand, the peer-to-peer scenario can be formalised through RISC-pb^{2}l as

$$[|((\text{init}))|]^P \bullet \overbrace{\left([|(|\text{test}|) \bullet (|\text{train}|) \bullet \triangleleft_{\text{Bcast}} \bullet (_{\text{FedAvg}} \triangleright)|]^P\right)}_r$$

As can be seen, the two formalisations are similar, which is not casual; it is possible to prove logically that the two formulas yield the same outputs if given the same inputs, modulo a different number of computations and communications. This fact can be easily seen by comparing the last two terms of both feedback blocks; with a bit of rewriting, it is possible to state that $(_{\text{FedAvg}} \triangleright) \bullet \triangleleft_{\text{Bcast}} \equiv [| \triangleleft_{\text{Ucast}_A} |]^W \bullet (_{\text{FedAvg}} \triangleright)$ and $[| \triangleleft_{\text{Bcast}} \bullet (_{\text{FedAvg}} \triangleright)|]^P \equiv [| \triangleleft_{\text{Bcast}} |]^P \bullet [| \triangleright_{\text{FedAvg}} |]^P$. With this new formulation, it is easy to see that, even if they are equivalent output-wise, these two computations exploit different amounts of communications: $([| \triangleleft_{\text{Ucast}_A} |]^W$ vs. $[| \triangleleft_{\text{Bcast}} |]^P)$ and computations $((_{\text{FedAvg}} \triangleright)$ vs. $[| \triangleright_{\text{FedAvg}} |]^P)$. This result implies that the two computations are mathematically equivalent output-wise: this means that the two communication topologies will produce the same final ML model, with the same learning performances, assuming the same hyper-parametrisation and modulo the differences in communication involved in the process. This simple analysis exemplifies the potential of using a formal tool such as RISC-pb^{2}l for modelling and discussing distributed systems.

Given the two above descriptions, it is straightforward to translate them into *FastFlow* programs (serialisation is performed using the Cereal [13] library). Both topologies are based on the all-to-all BB (ff_a2a), which efficiently models together the reduce and broadcast operators required by both workflows and rounds are modelled as cycles (i.e., feedback channels) created by activating

the wrap_around feature. The ff_a2a BB defines two sets of nodes connected according to the shuffle communication pattern; this means that each node in the first set is connected to all nodes in the second set. If the first set contains the aggregator and the right set all worker nodes, then we have the master-worker topology. On the other hand, for creating a mesh topology, peers are split into a modified aggregator, which additionally trains and includes a model trained on local data, and a distributor, which sends the local model to the other peers' aggregators; then, all aggregators are assigned to the left set and all distributors to the right set. Aggregator and distributor of the same peer are tied together by assigning them to the same distributed group. Leveraging the message routing options, we enforce that each aggregator sends its aggregated model only to its distributor; in contrast, each distributor forwards the aggregated model to all other aggregators on the feedback channel.

Model training is based on the PyTorch library via the C++ API. All nodes are designed for modularity and implemented as derived ff_node C++ classes. Multi-input multi-output nodes, i.e., aggregator node in tree topology and peer nodes in a mesh topology, are implemented exploiting the combiner BB (ff_comb) by combining a multi-input adapter forwarding messages and a multi-output node containing the logic. Models are specified as the generic PyTorch torch::nn::Module class, allowing the framework to work with any PyTorch model. For future extensibility, the aggregation methodology is specified as a separate policy class, e.g., FedAvg for federated averaging. Similarly, workers allow specifying the training strategy, i.e., optimizer, to use as (automatic type inferred) template argument.

The two FL use cases exploit a Multilayer Perceptron (MLP) made up of three fully connected layers trained to recognize digits from the MNIST dataset[1]; as hyper-parameters, we used cross-entropy as loss function and SGD as the optimizer, with a learning rate 0.01 and momentum 0.5. Concerning the MNIST dataset, we split the training set into equally sized random subsets assigned to each worker; this has been done to simulate a genuine federation in which each client possesses only a subset of the whole data distribution.

---

[1]http://yann.lecun.com/exdb/mnist/

4.2 Edge Inference

The EI is showcased by using a tree topology for modelling a control-room use case, which aims to solve the underlying problem of raising alerts for man-on-the-ground events. Leaf nodes containing multiple cameras feed into local pre-trained standard YOLOv5 networks pre-processed video images by resizing them to the correct resolution (e.g., 640x640), adding a border, and ensuring that the image has the correct tensor shape. After applying the YOLOv5 model, the aggregation nodes post-process the result to extract the bounding boxes having a classification score larger than a given threshold (detecting man-on-the-ground events) and output aggregating results along the tree till the control room located at the root (see Fig. 2). In our example, we modelled a three-level tree through RISC-pb^{2}l as

$$((\text{init})) \bullet \overleftarrow{\left([|\text{infer}|]^L \bullet (\mathcal{F} \triangleright) \bullet [|\text{combine}|]^C \bullet (\mathcal{F} \triangleright) \bullet ((\text{alert}))^R \right)}_\infty$$

where L, C, R are, respectively, the sets of leaf, control and root nodes, and $\mathcal{F}$ is the function routing to the father node. The subsequent *FastFlow* implementation resulted in nesting two levels of all-to-all BB.

To work with PyTorch-based networks in a C++-only environment, we exported the network provided by the YOLOv5 maintainers into a TorchScript archive. TorchScript archives have the advantage of serialising the Python code describing the network and the necessary weights. The Python code is represented in the archive using a subset of Python itself (TorchScript is the name of this subset); weights are serialised in pickle format. When a TorchScript archive is deserialised, the Python is just-in-time compiled, and the binary is dynamically linked into the running program.

5 EXPERIMENTATION

Each use case has been tested on three different hardware architectures hosted on the two research systems described below.

5.1 Computational infrastructures

5.1.1 Monte Cimone (RISC-V). The Monte Cimone [6] system is the first physical prototype and test-bed of a complete RISC-V (RV64) compute cluster, integrating not only all the key hardware elements besides processors, namely main memory, non-volatile storage, and interconnect, but also a complete software environment for HPC, as well as a full-featured system monitoring infrastructure. Monte Cimone comprises eight computing nodes running Linux Ubuntu 21.04 and is enclosed in four computing blades. Each computing node is based on the U740 SoC from SiFive and integrates four U74 RV64GCB application cores, running up to 1.2 GHz and 16GB of DDR4, 1 TB node-local NVME storage, and PCIe expansion cards. Each Monte Cimone computing node integrates separated shunt resistors in series with each of the SiFive U740 power rails as well as for the on-board memory banks, which can be leveraged to attain fine-grained power monitoring of power rails, including the core complex, IOs, PLLs, DDR subsystem and PCIe one. The power rails current and voltage are monitored by a PXIe-4309 board from National Instruments. The PXIe-4309 module features 8 ADC devices and supports a maximum data acquisition rate of 2 MSamples per second. Collected current and voltage traces are then post-processed to obtain an average power consumption for each application run.

5.1.2 EPI-TO (ARM-v8+x86-64+RISC-V). The EPI-TO system is a modular cluster designed to experiment with the technologies under development in the European Processor Initiative (EPI). It includes an ARM-v8 module (4 nodes), an x86-64 module (4 nodes), and a RISC-V module (2 nodes). The x86-64 module comprises 4 Supermicro servers, each including 2 Intel Xeon Gold 6230 CPU (20-cores@2.10GHz) sockets, and 1536GB RAM. The ARM-v8 module comprises 4 ARM-dev kits, each including one socket Ampere-Altra Q80-30 (80-core@3GHz), 512GB RAM, 2 x NVidia BF-2 DPU, and 2 x NVidia A100 GPU. The Intel and Ampere servers are connected via an Infiniband HDR and a 1Gb/s Eth networks. The GPUs are not used in the present experiments. All nodes run Linux Ubuntu 20.04 and share a high-performance BeeGFS file system. All the servers are set to use the "performance governor" mode. The RISC-V module is composed of 2 servers identical to Monte Cimone servers.

5.1.3 Achieving fairness of comparison. Many steps have been taken to ensure a fair comparison between the different architectures, since they have different degrees of computational power and maturity: x86-64 and ARM-v8 platforms are server-grade machines, while RISC-V is still a young and experimental embedded-like platform (even if some more HPC-oriented prototypes are being actively developed, such as the Ventana processors). First of all, the number of cores available to each machine has been taken into account; since the least powerful platform from this point of view is SiFive RISC-V (4 cores per node), we capped all the processes in the Intel and Ampere experiments consequently, so that precisely 4 cores would be assigned to each of them: this is done through the `taskset` command.

The maximum number of computational nodes available on the Monte Cimone computational system was 8, so we calibrated our experiments on a federation of a maximum of 8 workers. Since the number of Intel and Ampere servers is limited to 4, we place two nodes per server when running 8 node configurations. In such cases, we take care to place the node threads in different areas of the processors and near the memory banks to limit the interference between them as much as possible.

Due to the invasive PyTorch threading policy, only using the `taskset` command is insufficient in this scenario. Even if the process is restricted to 4 cores, PyTorch still creates a thread pool of as many threads as available cores. This behaviour can lead to many threads that, swapping between each other, can spoil the performance of PyTorch and subsequently ruin the fairness of the comparison. This problem has been resolved by setting `OMP_NUM_THREADS=4`, thus limiting the OpenMP threads created by PyTorch to 4, making it behave on Ampere exactly like on the SiFive platform. The `MKL_NUM_THREADS=4` has also been set to prevent the MKL library from creating more threads than the assigned cores to obtain the same behaviour on Intel.

Another precaution to exploit the few cores available at maximum is to use the TCP backend of FastFlow instead of the MPI one. The motivation behind this is the computational behaviour of the OpenMPI blocking `receive`. When issued, it actuates a busy waiting policy, directly occupying a core at 100% of its capability, which would skew the energy results. Moreover, in the context of

Table 3: Experimental setting description. FLOPs estimated using PyTorch profiler. In the case of YOLOv5n, only inference (forward pass) is performed. Data are reported when meaningful.

Experiment	Aggreg. data	worker/peer data	Model param.	Forward FLOPs	Backward FLOPs
MLP - master-worker	10K images	7.5K images	52.6K	105.1K (1 image)	109.8K (1 image)
MLP - peer to peer	—	7.5K train + 10K test images	52.6K	105.1K (1 image)	109.8K (1 image)
YOLO v5n - EI tree	None	148 frames	1.87M	2.68G (1 frame)	—

SiFive RISC-V, this means wasting 1/4 of the available computational power. We thus resorted to the TCP backend.

Lastly, a shared workload for the different experiments has to be set. Table 3 summarises the details of the data used and estimated model complexity. Due to the massive difference in the computational performance of the various machines, the choice has been made with particular attention to the SiFive RISC-V. We have chosen to train the MLP on MNIST for 100 epochs, subdivided into 20 federated rounds composed of 5 epochs each. This choice allows the learning curve to stabilise. At the same time, it allows the assessment of relevant measurements (e.g., communication and computation costs). We also experimented with a larger ML workload: training a ResNet18 network on the CIFAR10 dataset. Unfortunately, such a configuration required 24 hours on the RISC-V processor to complete a single training epoch, which is a time span incompatible with our experimental setting.

A brief video of 148 frames containing people moving has been chosen for the YOLOv5 experiments. We report the mean experimental results across five different runs in Table 4. Note that these choices, especially the choice of using only a subset of the available cores, do not hinder x86-64 and ARM platforms in favour of RISC-V. In fact, due to the chosen workload, the total computation time increases with the number of involved cores. This is due to the reduced benefits that are gained in parallelizing the training of a small model and the additional costs required by thread handling and synchronization (in particular under the Python threading model).

5.2 PyTorch on RISC-V

Before this work, the PyTorch framework [19] could not be compiled for the `riscv` and `riscv64` CPU architectures. In particular, at least three internal dependencies had not yet been ported to the RISC-V ISA: the Chromium Breakpad library[2], the SLEEF Vectorized Math Library[3], and the PyTorch CPU INFOrmation library (cpuinfo)[4]. This work introduces the RISC-V porting of the first two libraries as a byproduct. The experiments described in this section also served as functional tests to assess the maturity of the porting. In particular, the lack of cpuinfo porting affects some low-level multithreading functions (e.g., `torch::set_num_threads`), and the lack of vector registries in the SiFive Freedom U740 chip affects the SLEEF performance, with Fused Multiply-Add (FMA) as the only optimized instruction for tensor math.

Despite such limitations, the current PyTorch porting is mature enough for research and development. On the other hand, efforts to provide a full-featured RISC-V implementation are ongoing. The

Breakpad porting has already been merged into the official codebase and is publicly available. A porting of cpuinfo and a RISC-V vector implementation of the SLEEF library are in the plan for the EuroHPC EUPilot project.

5.3 Results analysis

The results of the experiments are reported extensively in Table 4. As can be seen, SiFive is an order of magnitude slower than the other systems, being almost always between 25-35 times slower than Intel and Ampere. This is to be expected due to the young stage of development of the platform, the absence of optimisaed libraries for deep learning-specific computations, and the lack of vectorial accelerators. We believe the lacking of vectorial units to be particularly detrimental to the SiFive processor. The code running on the other two platforms is compiled with libraries explicitly optimised for exploiting vectorial units (the oneMKL library for Intel and the Arm Performance Library for Ampere). On the other hand, the gap in performance between Intel and Ampere is negligible: Ampere is almost as fast as Intel in the computation while consuming an order of magnitude less power. The difference in power consumption is discussed more in detail in Section 5.4.

From a scaling perspective, all experiments behaved as expected: the recorded execution times are congruous with the weak scalability law since they remain constant or slightly increase with the number of processes; this confirms the goodness of the software and the capabilities and flexibility of the *FastFlow* runtime.

The heterogeneous experiments are another strong point in favour of the *FastFlow* capabilities: thanks to the Cereal serialisation back-end, it is possible to create a federation in which different workers are hosted on different computational systems. This feature is not trivial since moving data from one system to another usually implies conversion issues. We succeeded in running a distributed master-worker training across all the computational platforms exploited in this study (Intel, Ampere and SiFive), highlighting the flexibility and compatibility features of the proposed software stack. This fact is crucial since the federation of different entities cannot come with the requirement that all entities employ the same underlying computational infrastructure.

While not explicitly relevant to this discussion, it is worth reporting that as a sanity check we measured the classification accuracy of the tested systems. All the proposed FL architectures achieve the expected learning performances. Specifically, the MLP model achieved more than 95% of accuracy in all configurations, reaching up to 97% in most runs (YOLOv5 performances are not relevant here, since we used pre-trained models).

Finally, to give a perspective to the presented performance, we have reproduced one of the proposed scenarios (the master-worker

---

[2]https://chromium.googlesource.com/breakpad
[3]https://sleef.org
[4]https://github.com/pytorch/cpuinfo

structure with 4 workers) with OpenFL, one of the mature FL frameworks from related work. To further highlight our effort to the RISC-V developers community, we made an effort to port OpenFL to this computational platform. This porting required recompiling in ad-hoc ways the software dependencies (ninja, openblas, grpc/grpcio, and crytography).

As reported in table 4a, our implementation on the SiFive can complete the training of 100 epochs over the MNIST dataset in 673.70 seconds on average (11 minutes and 7 seconds). Conversely, OpenFL, with the same model, hyper-parameters, data pre-processing and distribution, achieved an average running time of 2,486 seconds(41 minutes and 26.42 seconds). Additionally, we repeated the same experiment on the x86-64 platform: even in this case we assessed the efficiency of our implementation (23.56 seconds) with respect to OpenFL (59.15 seconds). The difference between the two execution times is stunning, and its motivations have already been discussed throughout the entire paper; this is just another example of how a more high-performance-orientated FL framework would benefit the overall FL research environment.

5.4 Power consumption analysis

Looking at Table 5, we can highlight that the three tested systems belong to different computing classes. Indeed the table reports idle CPU and system power consumption as well as the Thermal Design Power (TDP) per node and system type. The Ampere and Intel CPUs are server processors with a TDP of over 100 Watts, while the SiFive CPU is an embedded class processor with TDP below 5 Watts. Table 4 reports the dynamic energy (Δ) as well as total energy per worker/peer/leaf when training the MNIST model (a,b) and inference of the YOLOv5 model (c). The Intel and Ampere CPUs have comparable performance, but the Ampere CPU has a lower power consumption of almost an order of magnitude. This fact makes the Ampere CPU the most performing one and the most energy-efficient one in this study. When compared to the Intel CPU, the Ampere one attains an average reduction for delta and total energy of 8.3x (4.3x) for the MNIST master-worker case, 5.7x (3.7x) for the MNIST peer-to-peer case and 4.7x (2.6x) for the YOLOv5 tree-based case. The SiFive CPU, when compared to the Ampere one, attains an average delta (total) energy consumption which is only 1.9x (5.0x) higher for the MNIST master-worker case, 2.4x (6.0x) for the MNIST peer-to-peer case and 2.7x (5.4x) for the YOLOv5 tree-based case.

While this is an unexpected result, given the absence of SIMD/Vector extension for the SiFive compute nodes, the significantly longer compute times drive up the consumed energy. We expect improved results once new silicon implements the RISC-V ISA vector extensions. Figure 3 reports an extract of three minutes of the CPU power consumption (in the y-axis) for a single SiFive compute node for the three experimental settings. We can notice that the MNIST master-worker (MNIST M-W in the figure) achieves a significantly lower power consumption than the MNIST peer-to-peer (MNIST P2P in the figure) while having a lower time-to-solution. This fact can be explained by the additional computation due to each peer computing its own global model and the increased traffic to handle peer-to-peer communication. In both traces, data transfers are visible as lower power consumption segments, each corresponding to
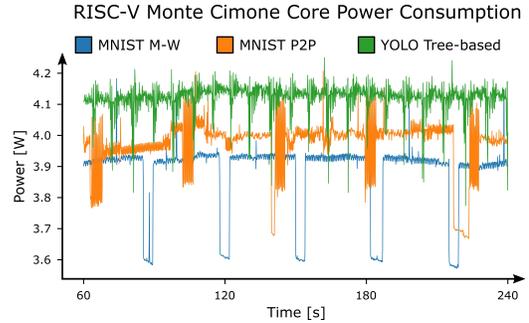

Figure 3: Three minutes of power consumption traces on the Monte Cimone RISC-V system.

a federated round. The YOLOv5 tree-based experiment has a higher computational intensity in line with the higher model complexity, which translates into a higher peak power consumption.

When comparing the different configurations, we can notice an overall increase in the time-to-solution while increasing the number of workers/peers/leaves, which directly translates into an increase in the energy consumption of the models. The scalability is different for the peer-to-peer and master-worker communication topologies, where the master-worker time-to-solution overhead saturates with the 4 workers case; in contrast, the peer-to-peer time-to-solution overhead continues to scale with the number of peers. This behaviour is expected due to the exponential cost in the communication scalability in the latter.

In Table 5, we computed the energy efficiency (Joules per Floating point operation) of the different processors by extracting the floating-point operations for each training epoch and the energy consumed by a single worker in the master-worker scenarios. The reported values are calculated following the equation: $E_{\text{FLOP}} = P \cdot t_{\text{epoch}} / [N_{\text{images}} \cdot (N_{\text{Forward}} + N_{\text{Backward}})]$ where P is the mean consumed power, t_{epoch} the time to train one epoch, N_{images} the number of images, and N_{Forward}, N_{Backward} the profiled FLOPS for forward and backward pass. Taking into account only the energy effectively consumed by the calculation itself (Δ energy), both Ampere and SiFive result to be more energy efficient than the Intel CPU, respectively 7x and 3.7x; this is good, especially for the SiFive platform, given the system's novelty. On the other hand, if the whole system consumption is taken into account, then the results are dramatically different: in this case, Ampere and Intel result more energy efficient than SiFive, by 5x and 1.2x respectively; this is due to the way higher execution time required by the SiFive platform to complete the same amount of computation done by the other two processors.

6 CONCLUSIONS

While FL is becoming a popular research topic for all the aspects related to model quality and robustness against model inversion, the research on enabling methodologies and architectures for developing new FL systems is lagging. Most implementations deeply encode the cooperation protocol and the message serialisation in the framework, making experimentation with new topologies, protocols, and

Table 4: Computational performance of proposed FL schema. Each result is averaged over 5 runs. The Intel-Ampere experiments have been executed heterogeneously, with half processes allocated to the Intel cluster and the other half to the Ampere cluster.

(a) MNIST master-worker training results: These performance metrics have been taken on a set of 20 federation rounds made up of 5 training epochs each (total 100 epochs); each client was assigned 1/8 of the entire dataset.

	master + 2 workers		master + 4 workers		master + 7 workers	
	time (s)	energy/worker (J): Δ (tot)	time (s)	energy/worker (J): Δ (tot)	Time (s)	energy/worker (J): Δ (tot)
x86-64 (Intel)	23.84	973 (1992)	**23.56**	1011 (2069)	**24.38**	1049 (2146)
ARM-v8 (Ampere)	**23.33**	**133 (483)**	25.66	**146 (531)**	25.86	**148 (535)**
RISC-V (SiFive)	674.47	269 (2562)	673.70	269 (2560)	687.03	274 (2610)
Intel-Ampere	29.50	—	29.55	—	33.34	—

(b) MNIST peer-to-peer training results: These performance metrics have been taken on a set of 20 federation rounds made up of 5 training epochs each (total 100 epochs); each client was assigned 1/8 of the entire dataset.

	2 peers		4 peers		8 peers	
	time (s)	energy/peer (J): Δ (tot)	time (s)	energy/peer (J): Δ (tot)	time (s)	energy/peer (J): Δ (tot)
x86-64 (Intel)	**23.15**	2082 (4261)	**24.05**	2162 (4422)	**24.95**	2210 (4522)
ARM-v8 (Ampere)	24.39	**169 (535)**	24.90	**173 (546)**	26.65	**185 (585)**
RISC-V (SiFive)	819.35	409 (3195)	815.55	407 (3180)	933.62	466 (3641)
Intel-Ampere	45.20	—	39.13	—	50.88	—

(c) YOLO tree-based inference results: These performance metrics have been obtained by assigning each leaf a video with 148 frames.

	root + 2 leaves		root + 4 leaves		root + 7 leaves	
	time (s)	energy/leaf (J): Δ (tot)	time (s)	energy/leaf (J): Δ (tot)	time (s)	energy/leaf (J): Δ (tot)
x86-64 (Intel)	**19.76**	1520 (2389)	**19.38**	1491 (2343)	**19.01**	1462 (2298)
ARM-v8 (Ampere)	37.16	**291 (848)**	39.88	**312 (910)**	43.15	**338 (985)**
RISC-V (SiFive)	1201.51	841 (4926)	1205.77	844 (4943)	1212.77	848 (4972)
Intel-Ampere	35.65	—	35.65	—	36.10	—

Table 5: Comparison of the different systems based on CPU power consumption. These results have been calculated as the mean across the three different configurations of the master-worker scenario.

	Δ energy/FLOP (CPU only)	energy /FLOP	avg CPU power (idle)	TDP /socket
x86-64 (Intel)	6.3 nJ	12.8 nJ	44 W	125 W
ARM-v8 (Ampere)	0.9 nJ	3.2 nJ	15 W	250 W
RISC-V (SiFive)	1.7 nJ	15.9 nJ	3.4 W	5 W

FL schema too complex. This work addresses this problem, proposing a lightweight methodology to experiment with FL at the edge, obtained by combining RISC-pb^{2}l and *FastFlow*. We experimented with three different edge ML architectures that can be compiled as shared-memory (for simulation) or distributed-memory (for deployment) versions with the same code compatible and interoperable with ARM-v8, RISC-V and x86-64 platforms. Notably, as a byproduct, we developed the first working binary of PyTorch for the RISC-V system. The experiments' results depict the three processors' maturity levels. The SiFive multi-core (4 x U74 RV64GCB) is the most performing RISC-V we found on the market, and it is still far away from mainstream processors, at least on ML workloads. Many novel

RISC-V implementations, including RISC-V accelerators (e.g., for vectors), are expected to reduce the gap, provided that properly optimised middleware will be available for the whole computing spectrum, including distributed computing and ML. This work aims to contribute to the emerging RISC-V ecosystem directly, benchmarking the performance of a current implementation of this ISA in perspective of future high-performance oriented developments, and also porting widely used software to this environment.

ACKNOWLEDGMENTS

REFERENCES

[1] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2016. Fathom: Reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Providence, RI, USA, 1–10. https://doi.org/10.1109/IISWC.2016.7581275

[2] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2013. Design patterns percolating to parallel programming framework implementation. *International Journal of Parallel Programming* 42, 6 (26 September 2013), 1012–1031. https://doi.org/10.1007/s10766-013-0273-6

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. 2012. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In *Euro-Par 2012 Parallel Processing*, Vol. 7484. Springer Berlin Heidelberg, Berlin, Heidelberg, 662–673. https://doi.org/10.1007/978-3-642-32820-6_65

[4] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. *Fastflow: High-Level and Efficient Streaming on Multicore*. Wiley-Blackwell, 261–280.

[5] The TensorFlow Federated Authors. 2019. *TensorFlow Federated*. Google. https://github.com/tensorflow/federated

[6] Andrea Bartolini, Federico Ficarelli, Emanuele Parisi, Francesco Beneventi, Francesco Barchi, Daniele Gregori, Fabrizio Magugliani, Marco Cicala, Cosimo Gianfreda, Daniele Cesarini, et al. 2022. Monte Cimone: Paving the Road for the First Generation of RISC-V High-Performance Computers. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. IEEE, Belfast, United Kingdom, 1–6. https://doi.org/10.1109/SOCC56010.2022.9908096

[7] Enrique Tomás Martínez Beltrán, Mario Quiles Pérez, Pedro Miguel Sánchez Sánchez, Sergio López Bernal, Gérôme Bovet, Manuel Gil Pérez, Gregorio Martínez Pérez, and Alberto Huertas Celdrán. 2022. Decentralized Federated Learning: Fundamentals, State-of-the-art, Frameworks, Trends, and Challenges. *arXiv preprint arXiv:2211.08413* (15 November 2022). https://doi.org/10.48550/ARXIV.2211.08413

[8] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, Pedro PB de Gusmão, and Nicholas D Lane. 2020. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390* (28 July 2020). https://doi.org/10.48550/ARXIV.2007.14390

[9] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečný, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2018. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097* (3 December 2018). https://doi.org/10.48550/ARXIV.1812.01097

[10] Jihong Chen and Jiabin Sun. 2021. Understanding the Chinese Data Security Law. *International Cybersecurity Law Review* 2, 2 (12 October 2021), 209–221. https://doi.org/10.1365/s43439-021-00038-3

[11] Kewei Cheng, Tao Fan, Yilun Jin, Yang Liu, Tianjian Chen, Dimitrios Papadopoulos, and Qiang Yang. 2021. Secureboost: A lossless federated learning framework. *IEEE Intelligent Systems* 36, 6 (25 May 2021), 87–98. https://doi.org/10.1109/MIS.2021.3082561

[12] Bryce Goodman and Seth Flaxman. 2017. European Union Regulations on Algorithmic Decision-Making and a "Right to Explanation". *AI Magazine* 38, 3 (2 Oct 2017), 50–57. https://doi.org/10.1609/aimag.v38i3.2741

[13] Shane W. Grant and Randolph Voorhies. 2013. *Cereal a C++11 library for serialization*. USCiLab. https://github.com/USCiLab/cereal

[14] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, et al. 2020. Fedml: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518* (27 July 2020). https://doi.org/10.48550/ARXIV.2007.13518

[15] Li Li, Yuxi Fan, Mike Tse, and Kuo-Yi Lin. 2020. A review of applications in federated learning. *Computers & Industrial Engineering* 149 (November 2020), 106854. https://doi.org/10.1016/j.cie.2020.106854

[16] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (28 June 2020). https://doi.org/10.48550/ARXIV.2006.15704

[17] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics AISTATS (Proceedings of Machine Learning Research, Vol. 54)*, Aarti Singh and Xiaojin (Jerry) Zhu (Eds.). PMLR, Fort Lauderdale, FL, USA, 1273–1282.

[18] Stuart L Pardau. 2018. The California consumer privacy act: towards a European-style privacy regime in the United States. *J. Tech. L. & Pol'y* 23, 1 (2018), 68–114.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.), Vol. 32. Vancouver Convention Center, Vancouver CANADA, 8024–8035.

[20] G. Anthony Reina, Alexey Gruzdev, Patrick Foley, Olga Perepelkina, Mansi Sharma, Igor Davidyuk, Ilya Trushkin, Maksim Radionov, Aleksandr Mokrov, Dmitry Agapov, Jason Martin, Brandon Edwards, Micah J. Sheller, Sarthak Pati, Prakash Narayana Moorthy, Shih-han Wang, Prashant Shah, and Spyridon Bakas. 2021. OpenFL: An open-source framework for Federated Learning. *arXiv preprint arXiv:2105.06413* (13 May 2021). https://doi.org/10.48550/arXiv.2105.06413

[21] Nicolò Tonci, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. 2023. Distributed-memory FastFlow Building Blocks. *International of Parallel Programming* 51 (2 December 2023), 1–21. https://doi.org/10.1007/s10766-022-00750-5

[22] Massimo Torquati. 2019. *Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns*. Ph. D. Dissertation. University of Pisa.

[23] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, Washington, DC, USA, 331–344. https://doi.org/10.1109/HPCA.2019.00048

[24] Yuexiang Xie, Zhen Wang, Daoyuan Chen, Dawei Gao, Liuyi Yao, Weirui Kuang, Yaliang Li, Bolin Ding, and Jingren Zhou. 2022. Federatedscope: A comprehensive and flexible federated learning platform via message passing. *arXiv preprint arXiv:2204.05011* (11 April 2022). https://doi.org/10.48550/ARXIV.2204.05011

[25] Alexander Ziller, Andrew Trask, Antonio Lopardo, Benjamin Szymkow, Bobby Wagner, Emma Bluemke, Jean-Mickael Nounahon, Jonathan Passerat-Palmbach, Kritika Prakash, Nick Rose, et al. 2021. Pysyft: A library for easy federated learning. *Federated Learning Systems: Towards Next-Generation AI* 965 (12 June 2021), 111–139. https://doi.org/10.1007/978-3-030-70604-3_5

A ARTIFACT

A.1 Abstract

Fast Federated Learning (FFL) is a C/C++-based Federated Learning framework built on top of the parallel programming FastFlow framework. It exploits the Cereal library to efficiently serialise the updates sent over the network and the libtorch library to bypass the need for Python code fully. It has been successfully tested on x86_64, ARM and RISC-V platforms. FFL has scripts for automatically installing the framework and reproducing all the experiments reported in the original paper.

A.2 Artifact check-list (meta-information)

Obligatory. Use just a few informal keywords in all fields applicable to your artifacts , and remove the rest. This information is needed to find appropriate reviewers and gradually unify artifact meta information in Digital Libraries.

- **Algorithm:** Federated Averaging (FedAvg)
- **Compilation:** C++17 compatible compiler
- **Binary:** to be compiled from source
- **Model:** Multi-Layer Perceptron (MLP), YOLO v5n
- **Data set:** MNIST, Ranger Roll (video)
- **Run-time environment:** Linux, MacOS, FastFlow, Cereal, OpenCV
- **Hardware:** x86_64, ARM, RISC-V, power consumption counters
- **Execution:** sole user
- **Metrics:** time, accuracy
- **Output:** console: mean execution time, logs: time, accuracy
- **Experiments:** README, scripts
- **How much disk space required (approximately)?:** 874 MB
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 20 minutes on Intel, 25 minutes and ARM and 800 minutes on RISC-V
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** GPL-3.0
- **Data licenses (if publicly available)?:** GPL-3.0, CC BY
- **Archived (provide DOI)?:** 10.5281/zenodo.7807974

A.3 Description

A.3.1 How to access. FFL can be obtained by simply cloning the official GitHub repository: https://github.com/alpha-unito/FastFederatedLearning.git. The approximate disk occupation after the setup and compilation is approximately 874 MB.

A.3.2 Hardware dependencies. No specific hardware dependency is needed to run FFL. However, to reproduce the experimental results proposed in the paper, it is necessary to have access to 4 Supermicro servers including 2 Intel Xeon Gold 6230 CPU (20- cores@2.10GHz) sockets with 1536GB RAM, 4 ARM-dev kits including one socket Ampere-Altra Q80-30 (80-core@3GHz) with 512GB RAM, and 8 U740 SoC from SiFive running up to 1.2 GHz and 16GB. Intel and Ampere CPUs nodes should be interconnected with an Infiniband network, while the SiFive ones with a 1Gb/s Ethernet. Also, hardware power monitoring counter should be available for the Intel and ARM platforms, while an external monitoring board, such as the PXIe-4309 National Instruments, is required to record the power consumption on RISC-V.

A.3.3 Software dependencies. Starting from a fresh Ubuntu 22.04 installation, the first step to reproduce the reported results reported in the paper is to use apt to update the available package information and install (tested versions are reported in brackets): build-essential (12.9), cmake (3.22.1), libopencv-dev (4.5.4), and unzip (6.0-26) packages. A C/C++-17 compatible compiler is required for compilation. Furthermore, the FastFlow (DistributedFF branch), Cereal (1.3.2), and libtorch (2.0.0) libraries are required (installed via helper script see §A.4). Optionally, the Powermon utility is required to measure system power consumption.

A.3.4 Data sets. The MNIST dataset and a short video (Ranger Roll) are needed for the experiments. The setup script automatically downloads these files: MNIST is retrieved from the owner's website, while the Ranger Roll video is downloaded from our servers.

A.3.5 Models. A simple, three-layer MLP is used for part of the experiments, while the others exploit a YOLO v5n neural network. Our artifact provides both models.

A.4 Installation

To set up the whole system and to compile the example, it is sufficient to run: source setup.sh. This script will automatically download all the required libraries, update the environment variables, build the dff_run utility, launch CMake and build all the available examples.

A.5 Experiment workflow

All what is needed to run the full set of experiments is: bash reproduce.sh. This script will take care of running in a replicated manner (5 times) all available examples (3) in all the available configurations (3), for a total of 533=45 runs. To run the experiments across multiple servers the json config files must be modified (see §A.7).

A.6 Evaluation and expected results

The mean execution time for each combination will be reported on the output, and logs will be saved for each experiment in the respective folder. If the computational platforms are the same as the one reported in the paper, each process is allocated to a different computational node, and each process is assigned a different set of cores (in case of multiple processes on a single node), then the obtained results should be congruent with the one reported in the paper.

A.7 Experiment customization

To configure the experiments, the .json files describing the distributed configuration of the computations have to be personalised. All the instructions for doing that are available on the README.md file. Additionally, inside the reproduce script, the MAX_ITER variable can be set to change the replica factor of the experiments.

A.8 Notes

The reviewers can find additional information on the FFL software in the official README.md file.