

1 Overview

This document and the code provided alongside it allow you to reproduce the main results of the paper “*Interval-Asynchrony: Delimited Intervals of Localised Asynchrony for Fast Parallel SGD*” as presented in Euro-Par 2025. Specifically, it will generate plots corresponding to figures 4 & 5 in the paper, which compare the accuracy and training time of our method to the main two baselines – fully asynchronous with and without adaptive parallelism. Please refer to the paper for more details about our algorithm and the baselines.

1.1 Required software and libraries

The following software is all you need to ensure manually. The included script will install a few necessary Python dependencies automatically, contained within a virtual environment.

- Linux. We used Linux 6.13.1, but there’s no reason that any recent version wouldn’t work.
- C++ compiler toolchain with support for C++20. We used GNU g++ 14.2.1.
- CMake. The specific version shouldn’t matter as long as it’s recent, but we used CMake 3.31.7.
- Python 3 (for rendering plots). We used Python 3.13.2.

2 Quick Evaluation (recommended)

Once you are sure you have all of the software mentioned in Section 1, **you may execute `do_everything.sh`**. By default it assumes that your machine has at least 160 cores, since 128 were used in the paper for running SGD, and an additional 32 were used for asynchronous accuracy evaluation; see Section 4.2 if you wish to reduce the number of accuracy evaluation cores. Modify the `MS` variable in `run_experiments.sh` if you want to change the range of numbers of threads to evaluate.

This script downloads an additional required C++ library (Eigen), downloads two datasets, compiles the program, runs a number of experiments, and then produces three graphs as PDF files. Running the experiments is unsurprisingly the longest part of this process. Expect it to take a good few hours. You could run `tail -f experiments.log` if you want to watch the progress of the experiments in real time.

This process is entirely self-contained and should work “out of the box”, but in case it doesn’t you may wish to try the manual steps presented in the next section to determine where the problem lies.

3 Step-by-Step Manual Instructions

3.1 Running experiments

1. Run `build.sh` to compile the program. It will produce a folder named `build` containing an executable called `mininn`. **If it fails here**, then most likely your C++ toolchain doesn’t support C++20 features, or you don’t have CMake installed.
2. Run `download_data.sh` to download the CIFAR-10 and CIFAR-100 datasets, which are used for the evaluation. The datasets are unpacked into a new folder called `data`. **If it fails here**, you either don’t have an internet connection, or the server which hosts the CIFAR datasets is down.
3. Run `run_experiments.sh &>> experiments.log` to run all the experiments necessary to reproduce the plots in the paper, optionally including the redirection to pipe output to a log file. **Note:** this script assumes that your system has at least 160 cores, as this many were used in the paper (128 for SGD execution, and 32 for asynchronous accuracy evaluation; see Section 4.2). If you would like to test a different range of parallelism, you could change the `MS` variable in the script. If you would like to change how threads are pinned to CPUs, refer to Section 4.1. **If it fails here**, look at `experiments.log` and consider the following:

- Is the error something to do with CPU IDs and/or threads? Perhaps your machine has fewer threads than specified in the *MS* variable in the script. You can change it to test a smaller range of thread counts.
- Is the error something to do with files not being found? Make sure you're running this script from the artifact root directory. Make sure that you successfully downloaded the data in step 2. You should have a directory called *data* in the directory from which you ran `run_all.sh`. Inside this should be directories named *cifar-10* and *cifar-100*.
- Is the error something to do with being out of memory? The more threads that are used, the higher the memory usage. If you don't have enough, try reducing the range in the *MS* variable in the script.
- Something else? If you still can't get it to run, feel free to email the first author at the email mentioned at the top of this document.

3.2 Producing plots

We use Python 3 with *matplotlib*, *pandas*, and *numpy* to generate plots. Simply `run mkplots.sh` from the artifact root directory. It will do all of the following:

1. Create and activate a Python virtual environment if it detects that you're not already in one.
2. Install the aforementioned Python libraries from *requirements.txt*.
3. Run scripts from the *plot* directory to generate the graphs. They should be in the form of three PDFs in the same directory that you ran the script from.

4 Advanced

4.1 Configuring thread pinning

On our system, and likely on many others, it makes sense to assign threads to CPUs using a very simple scheme: thread number *X* gets pinned to CPU logical ID *X*. Threads are numbered starting from 0.

If, however, you wish to customise this (for example if your CPUs are numbered in some way that it's better to use every *other* CPU, e.g. to skip hyperthreading cores), then read on. Otherwise, you can ignore this section. If you don't know what this means, or don't know what thread pinning scheme would be better, then also ignore this.

To change the CPU allocations, you need to edit the file `code/include/minidnn/Component/Worker.hpp`. On line 55 you'll find a call to `set_cpu(id)`. Here, *id* is the thread ID. You may add any logic you wish in order to customise the pinning order. There are some commented out examples that we used during testing that you could use as inspiration.

Once you make this change, remember to re-run `build.sh`.

4.2 Asynchronous accuracy evaluation (read if <160 cores)

Since evaluating the accuracy over the entire testing data set takes a significant amount of time (many seconds), we decided to spawn a number of worker threads separate to the SGD execution to take queued model snapshots, evaluate them, and save the measured accuracy to a list. The code as provided will spawn 32 of these threads, pinned from CPU IDs 128 to 159 inclusive. It's important that these worker threads don't use the same CPUs as used for the SGD execution, since the performance would take a hit.

If you have fewer than 160 cores, you will need to change this behaviour. To do so, simply change the values for `NUM_ACCURACY_WORKERS` and `FIRST_ACCURACY_WORKER` at lines 18 and 20 in the file `code/include/minidnn/modular_components.hpp`, and then re-build the program. `NUM_ACCURACY_WORKERS` determines how many threads are spawned, and they are pinned at incremental CPU IDs starting at `FIRST_ACCURACY_WORKER`.

As an example, if your machine only has 64 cores, you may wish to spawn only four accuracy workers, starting at CPU 61. You would then also have to change the *MS* variable in the `run_experiments.sh` bash script so that you only evaluate numbers of threads up to 60, e.g. `MS='16 32 48 60'`.