Communication-efficient Tensor Parallelism for GPT-2

Eduard Burlacu

October 17, 2025

1 Experiment 1: MiniGPT on Shakespeare Character-level

1.1 Dataset

For these experiments, we used the Shakespeare dataset tokenized at character level. A summary of the dataset and train/val split is provided below:

```
length of dataset in characters: 1,115,394
all the unique characters:
 !$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
vocab size: 65
train has 1,003,854 tokens
val has 111,540 tokens
```

1.2 Training

This experiment was performed on the same small version of GPT-2 that Karpathy called MiniGPT in the codebase.

```
python train.py config/train_shakespeare_char_spps.py --device=mps --compile=False --
    eval_iters=20 --log_interval=1 --block_size=64 --batch_size=12 --n_layer=4 --n_head=4 --
    n_embd=128 --max_iters=2000 --lr_decay_iters=2000 --dropout=0.0
```

The first thing I did was implementing vanilla tensor paralellism(TP) as described in Megatron-LM. We have the suggested approach and nomenclature for:

- all-gather operator

```
class f_op(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x: torch.Tensor, tp_size: int):
        ctx.tp_size = tp_size
        return x.unsqueeze(0).expand(tp_size, *x.shape).contiguous()

    @staticmethod
    def backward(ctx, grad_output):
        ...
        grad_input = grad_output.sum(dim=0)
        return grad_input, None
```

- all-reduce operator

```
class g_op(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x: torch.Tensor):
        ctx.tp_size = x.shape[0]
        return x.sum(dim=0)

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output.unsqueeze(0).expand(ctx.tp_size, *grad_output.shape).
    contiguous()
```

For this experiment, the transformer consists of 4 blocks, so we start by a baseline TP model which requires 8 **AllReduce** operations. In the figure below we can see the validation loss with the new TP blocks and we ensure that the loss curves are similar enough. This sanity-check is critical for the rest of the project.

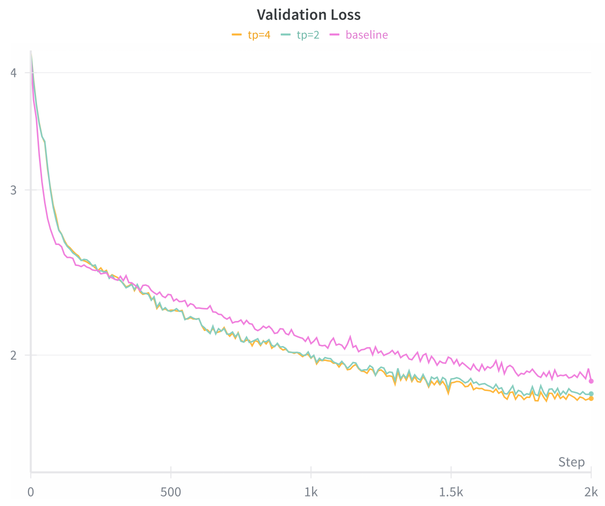

Figure 1: MiniGPT with vanilla TP Parallelism vs the baseline TP=1,2, and 4 workers: serves as a cheap sanity check for the rest of the project

Having found the simplest approach for TP, we **now look at how to reduce this number of communications between workers by slightly changing the transformer architecture to leverage distributed computing**.

One key insight came from the recommended reading SPD: Sync-Point Drop. They show that by carefully redefining the residual connections, we can get rid of 1 `AllReduce` connection. I have followed their approach and implemented the `SPDBlock` layer in the relevant file.

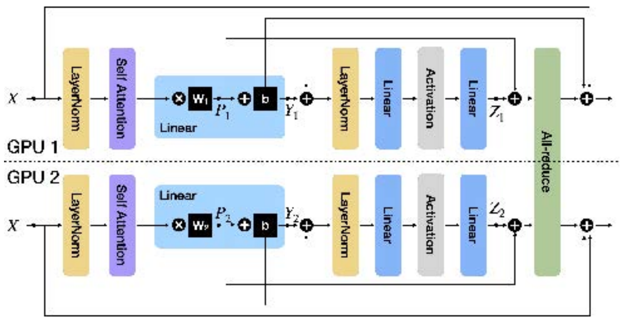

Figure 2: SPD block for TP = 2

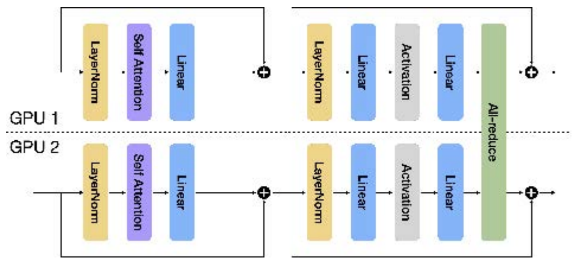

Figure 3: SPD block for TP = 2

SPD: Sync-Point Drop. They show that by carefully redefining the residual connections, we can get rid of 1 `AllReduce` connection. I have followed their approach and implemented the `SPDBlock` layer in the relevant file.

The next insight came after the question: "but what if we drop both and do an all reduce after more blocks and not just one?". This leads to the implementation of the `ParallelBlock` which only does an All-Reduce in the last-layer of a sequence of such modules. This enabled us with an approach to reduce the communications under 1 comm/layer, as we were hinted with the questions 9?6? given that the transformer has 12 layers.

In the next figure we show that we can successfully interchange these layers with no significant loss of efficiency. Starting from the baseline and the model with 8 communications, I named the runs based on the sequence type, for example "tstt" means that we have `TPBlock` in the 1st, 3rd, and 4th positions, resulting in $3 \times 2 + 1 = 7$ communications. "sppp" means that we have 1 `SPDBlock` in the first position, followed by 3 `ParallelBlock` which only communicate once, so in total 2 communications. I found out that fir this task we can succesfully use "sppp" which only has 2 communications, reducing it from 8.

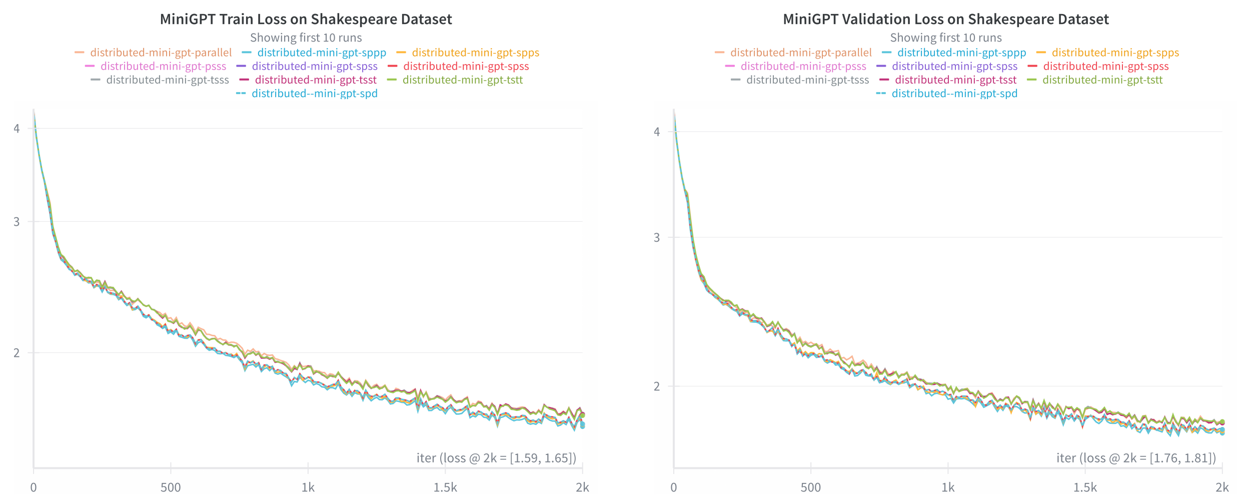

Figure 4: *MiniGPT Training* ***Left:*** *Train Loss Curve;* ***Right:*** *Validation Loss Curve*

2 GPT2

In this part we scale up the model to the GPT-2 architecture with 12 blocks and 123.69M parameters.

2.1 Dataset

We follow the same processing as done by Andrej Karpathy in the reference codebase nanoGPT. There was a problem which took some time to solve, the datasets newer versions were not compatible with the dataset generation script. The correct requirement for the project to work is `datasets==3.6.0`(or at least a version that works).

2.2 Training

My 3 blocks enable us to have a lot of flexibility in the number of communications, see the `NanoGPT/config` folder for examples.

Due to limited compute and especially time, I did a binary-search through the options listed in the assignment and I found out that we can:

- Train GPT-2 with SPD-only blocks, so we can achieve 12 communications i.e. 1 communication per layer

- I also tried running `3x[sppp]` i.e. a sequence of 1 SPD block followed by a pipeline of 3 Parallel blocks. Repeating this pattern 3 times, we get 12 layers which need $3 \times (1 + 1) = 6$ communications, which is an amazing $4\times$ improvement of 0.5 communications/block.

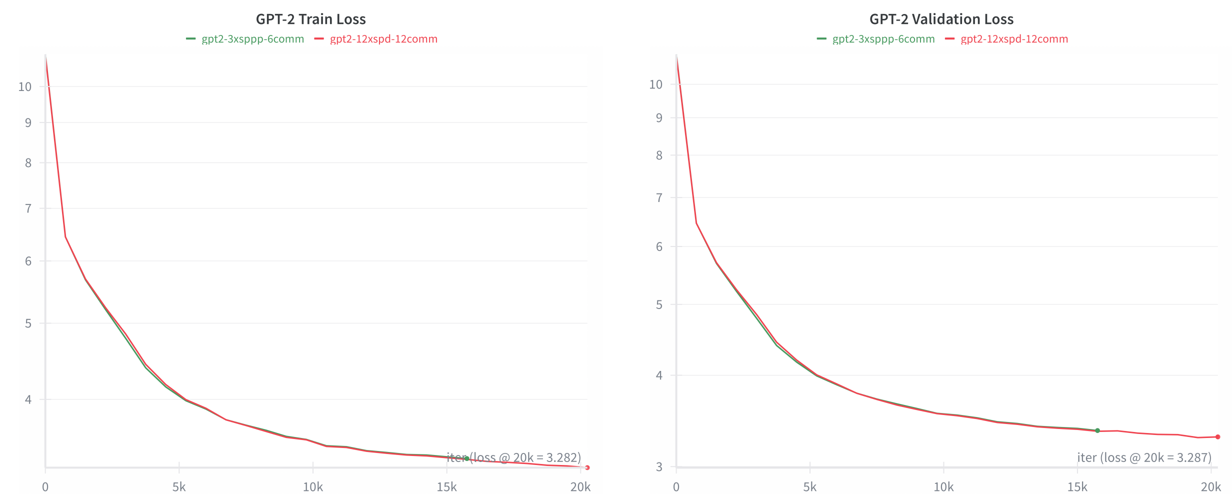

Figure 5: *GPT-2 Training* ***Left:*** *Train Loss Curve;* ***Right:*** *Validation Loss Curve*

Unfortunately at this point I ran out of time. I assume the limit is a bit below, so somewhere between 1 and 6 communications. The lowest configuration possible with my approach is only parallel blocks which has 1 communication at the end, but I have not experimented to say clearly if the convergence will be significantly affected by this aggresive lack of communication. Moreover, our analysis did not also investigate the effect on the terminal/ convergence loss due to reduced expressivity, which is likely to be affected and is of practical significance(especially when training models at this scale). At least for an insight, in the MiniGPT experiment the full-parallel model was able to match the performance of the rest of the models, but the models were still not converging at that point and the quality of the generation was still quite bad, which does not allow me to draw a clear conclusion regarding the convergence loss. See some examples here:

```
ENCEN:
But Mut fore full, a this everinted, it wand, so his tor you deans
Suchoud the oward for I brovem the me.
Hefen main for yie sing my for frow's of our cated;
The and this afeef, he the long my kine,
Sin the futy same lord a joren themer deal'd in
His as our oun aie of eare, thou with hern.

Sirhanous:
He dother and you; Gie an bot as as thear
For in thou nereie, be fanding thing orre came,
But your life an uight sweere of they not; for his
thou veat fince to life soufer of his proopsele
```

3 Experiments Conclusions

We see that both GPT-2 models with 12, respectively 6 communication points follow the same loss curve. This is relieving, and at 20k steps we get really close to the expected value of 3.28 recorded by Andrej Karpathy. Therefore, these methods could be promising for edge ML where the communication cost is prohibitively large and often rely on Federated Learning for fully independent training. However, this is impossible for LLMs that do not fit on regular edge devices.

4 Limitation and Future Work

- For these experiments I focused solely on the Tensor Parallelism of **transformer blocks**. Other key components of GPT-2 are the embeddings and the positional encodings. In Megatron-LM, this is analyzed and corresponds to a similar chunking of the embedding matrix similarly to our approach for TP in linear layers. However, this becomes insignificant as we scale-up transformers to Ts of parameters, so I have decided to focus the analysis on the sole effect of tranformer architecture parallelism
- It would be interesting to extend this analyis with a full distributed implementation with `torch.distributed` to see the practical time improvements of this method for some real delays of communcation.