Distributed Data Parallel (DDP) in LLM Training: Implementation and Evaluation

Introduction

Distributed Data Parallel (DDP) is a parallelization technique that distributes the training workload across multiple GPUs and nodes, which (might) enable efficient scaling of model training. DDP replicates the model on each GPU while each GPU processes a different subset of the training data, with gradients synchronized across all replicas. I implement DDP for LLM training and analyze the performance (i.e., how throughput, hardware utilization, and training time scale with different numbers of GPUs). Furthermore, I also combine and benchmark DDP with my *Pretokenization* implementation (feature 1) to see whether combining pretokenization with DDP yields compounding benefits, as we have larger effective batch sizes.

Implementation Details

The key components are:

- Each process joins a common communication group (see `src/utils.py` for implementation)
- Each GPU maintains its own copy of the model (model replication)
- The dataset is split among processes with no overlap (data sharding)
- After the backward pass, gradients are averaged across all processes (gradient synchronization)
- Local metrics are collected and aggregated for global reporting (e.g., global throughput etc.)

Sampling

In order to shard the data, I implemented a custom data sampler. We distinguish between *map-style* datasets (like the padding one) and *iterable dataset* (like the padding-free tokenlist approach). The custom `DistributedSampler` supports deterministic shuffling and ensures each process gets a unique subset of the data. The `create_train_dataloader()` in `distr/data_utils.py` is called in the training setup and detects whether a dataset is map-style or iterable to return the appropriate sampling strategy based on dataset type.

Implementation can be found in `distr/data_utils.py` and `distr/samplers.py` (and also in the Appendix fro the `DistributedSampler`).

Training Loop Modification

I modified the training loop to support distributed training. This includes the wrapping of the model in PyTorch's DDP, implementation of gradient normalization and synchronization and ensured only the master process logs results. Simplified, the modification can be summarized like this:

```
# Wrap model in DDP if distributed training is enabled
if args.distributed:
    # Add a barrier before wrapping
    torch.distributed.barrier()
    model = torch.nn.parallel.DistributedDataParallel(
      model,
      device_ids=[local_rank],
      output_device=local_rank,
      broadcast_buffers=False
    )

# During training:
if args.distributed: # synchronize the loss across all processes
    loss_tensor = torch.tensor([loss.item()], device=device)
    torch.distributed.all_reduce(loss_tensor,
↪   op=torch.distributed.ReduceOp.AVG)
    loss_item = loss_tensor.item()
```

Support for Padding-Free Approaches

To fully integrate DDP with all dataset types, I also updated the iterative datasets to support proper data sharding in distributed environments. For both `IterableParquetDataset` and `IterablePreTokenizedDataset` (see implemenation of pretokenization, feature 1), I implemented rank-aware data partitioning (abstracted):

```
def __iter__(self):
    # Get distributed info
    if torch.distributed.is_available() and
    ↪  torch.distributed.is_initialized():
        self.rank = torch.distributed.get_rank()
        self.world_size = torch.distributed.get_world_size()

    # Calculate the shard size and starting position for this rank
    shard_size = self.real_length // self.world_size
    start_idx = self.rank * shard_size
    end_idx = start_idx + shard_size if self.rank < self.world_size - 1
↪   else self.real_length
```

```
        # Each process gets its own data shard
        self.current_index = start_idx
        self.end_index = end_idx

        return self
```

This is necessary to ensure each process works on a disjoint subset of the data. Unlike map-style datasets, iterable datasets don't support indexing with `__getitem__`, so `DistributedSampler` (which works by returning indices) is not compatible. The streaming nature of these datasets requires the sharding logic to be implemented internally, so each process maintains its own token buffer sourced exclusively from its assigned partition of the data.

DDP Verification

In order to verify our DDP pipeline, I ran DDP with `world_size=1` (e.g., `nodes=1` and `gpus-per-node=1`) with `batch_size=1` and compared it to our two baselines, also with `batch_size=1`. This should yield more or less the same output metrics.

Below is the Validation loss over time shown for all 4 configurations, which show similar learning trajectories. The minor differences in the curves are expected due to non-deterministic initialization – but overall very similar therefore validating our DDP implementation.

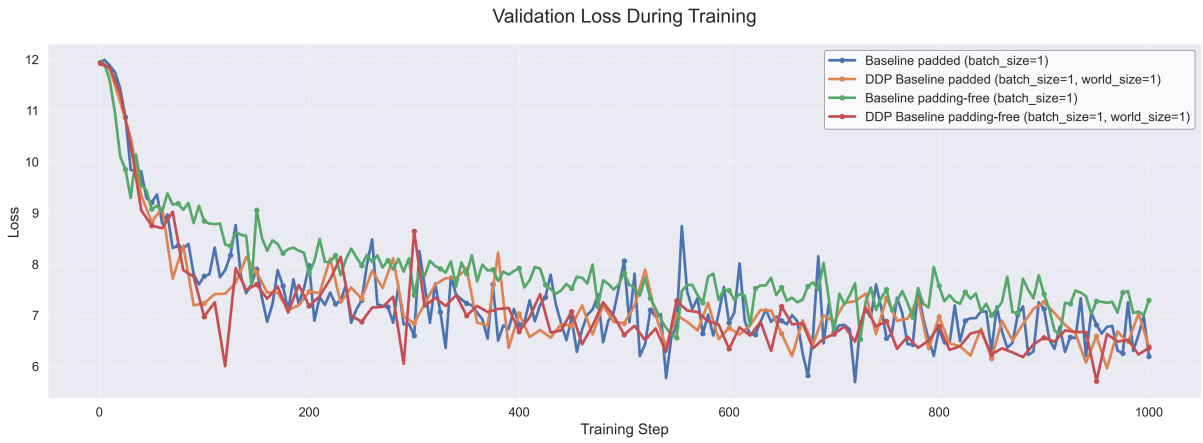

Figure 1: Verification: Loss over time

Configuration	Tokens/sec	MFU %	TFLOPs
Baseline padded (batch_size=1)	6544.40	31.97	316.23

Configuration	Tokens/sec	MFU %	TFLOPs
DDP Baseline padded (batch_size=1, world_size=1)	5990.44	31.22	308.75
Baseline padding-free (batch_size=1)	6902.55	35.97	355.77
DDP Baseline padding-free (batch_size=1, world_size=1)	5874.02	30.61	302.75

The DDP versions show a 8-15% performance overhead compared to non-DDP versions. This is expected and might be due to additional gradient synchronization operations and NCCL communication overhead.

Additionaly, I created a script `src/validate_ddp.py` that performs a more rigorous verification by testing gradient synchronization across multiple nodes. This script feeds different inputs to each GPU across 2 nodes (8 GPUs total) and verifies that all processes produce identical gradients after the backward pass.

Results and Analysis

Evaluation Setup: I used a smaller model than the one given in Assignment 2, namely `dim=2048`, `n_layers=16`, `n_heads=16` and `n_kv_heads=4`. This allows us to test (local) batch sizes up to 8. Furthermore, I also evaluate the pre-tokenization strategy I implemented in Feature 1. In the following, I kept the `batch_size=8` fixed and also the number of GPUs per node and tested with different world sizes (1, 2, 3 and 4 nodes, corresponding to world sizes 4, 8, 12 and 16 respectively). I experimented with all four dataset approaches: padded, pretokenized padded, padding-free, and pretokenized token-list.

We use a local batch size of 8 throughout the experiment. However, it is important to know that the gloabl batch sizes corresponds to World size $\times$ Local batch size. Therefore, we can effectively test the effect of pretokenization on much larger batch sizes compared to the setting in the report of feature 1, and see the effect more clearly.

The result can be summarized in the following table:

Configuration	World Size	Global Tokens/sec	Global MFU %	Global TFLOPs
Padded (local_batch_size=8)	4	161853.91	37.89	1499.00
Padded (local_batch_size=8)	8	308941.12	36.16	2861.24
Padded (local_batch_size=8)	12	457370.81	35.69	4235.91

Configuration	World Size	Global Tokens/sec	Global MFU %	Global TFLOPs
Padded (local_batch_size=8)	16	600019.40	35.12	5557.04
Pretokenized padded (local_batch_size=8)	4	164892.61	38.60	1527.14
Pretokenized padded (local_batch_size=8)	8	313336.67	36.68	2901.95
Pretokenized padded (local_batch_size=8)	12	468530.85	36.56	4339.27
Pretokenized padded (local_batch_size=8)	16	625349.22	36.60	5791.63
Padding-free (local_batch_size=8)	4	151486.14	35.46	1402.98
Padding-free (local_batch_size=8)	8	287911.61	33.70	2666.47
Padding-free (local_batch_size=8)	12	426904.93	33.31	3953.75
Padding-free (local_batch_size=8)	16	565362.05	33.09	5236.06
Pretokenized token-list (local_batch_size=8)	4	161442.42	37.80	1495.19
Pretokenized token-list (local_batch_size=8)	8	310187.37	36.31	2872.78
Pretokenized token-list (local_batch_size=8)	12	463466.41	36.17	4292.36
Pretokenized token-list (local_batch_size=8)	16	610935.46	35.76	5658.14

Throughput Scaling

The throughput scaling results show how effectively each approach scales as we increase the number of GPUs:

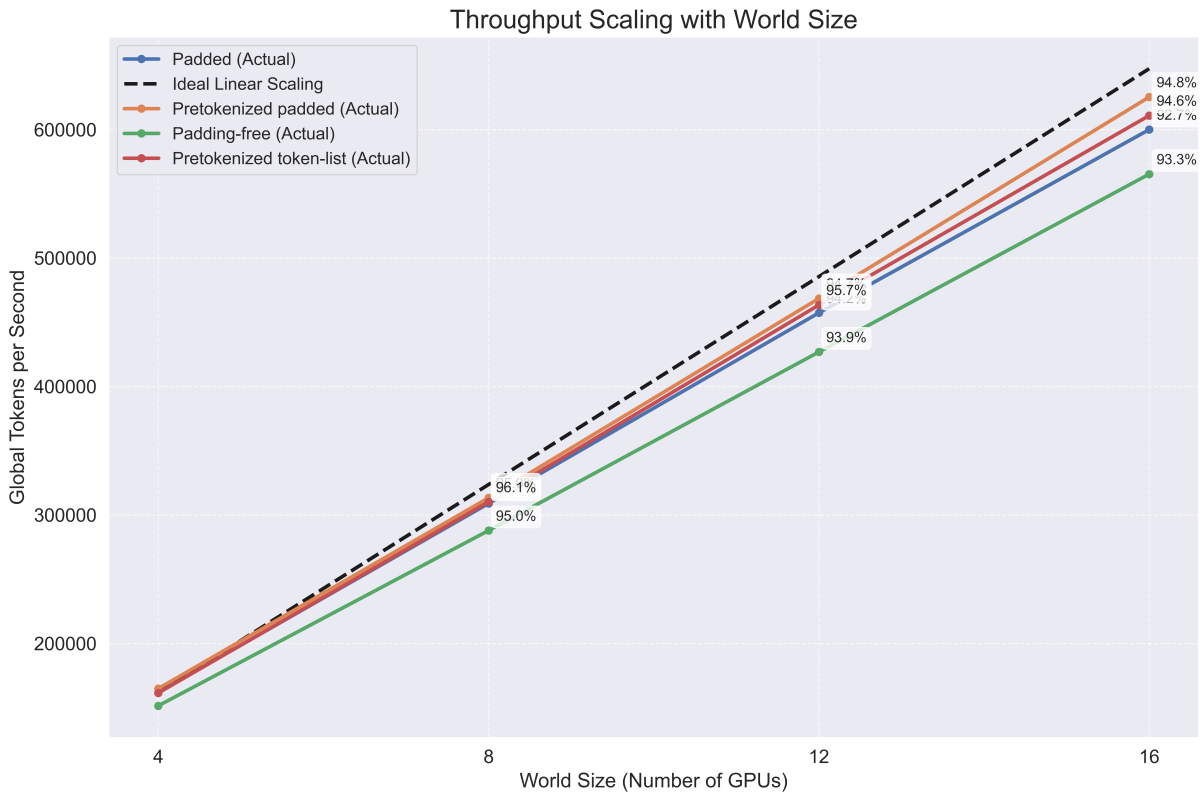

Figure 2: Throughput Scaling with World Size

All approaches achieve remarkable scaling, but we see that the pretokenized approaches outperform their on-the-fly tokenization equivalent. This shows that pretokenization helps maintain better scaling efficiency as we add more GPUs.

Hardware Utilization (MFU)

The Global Model FLOPS Utilization (MFU) metrics reveal how efficiently we use the available hardware:

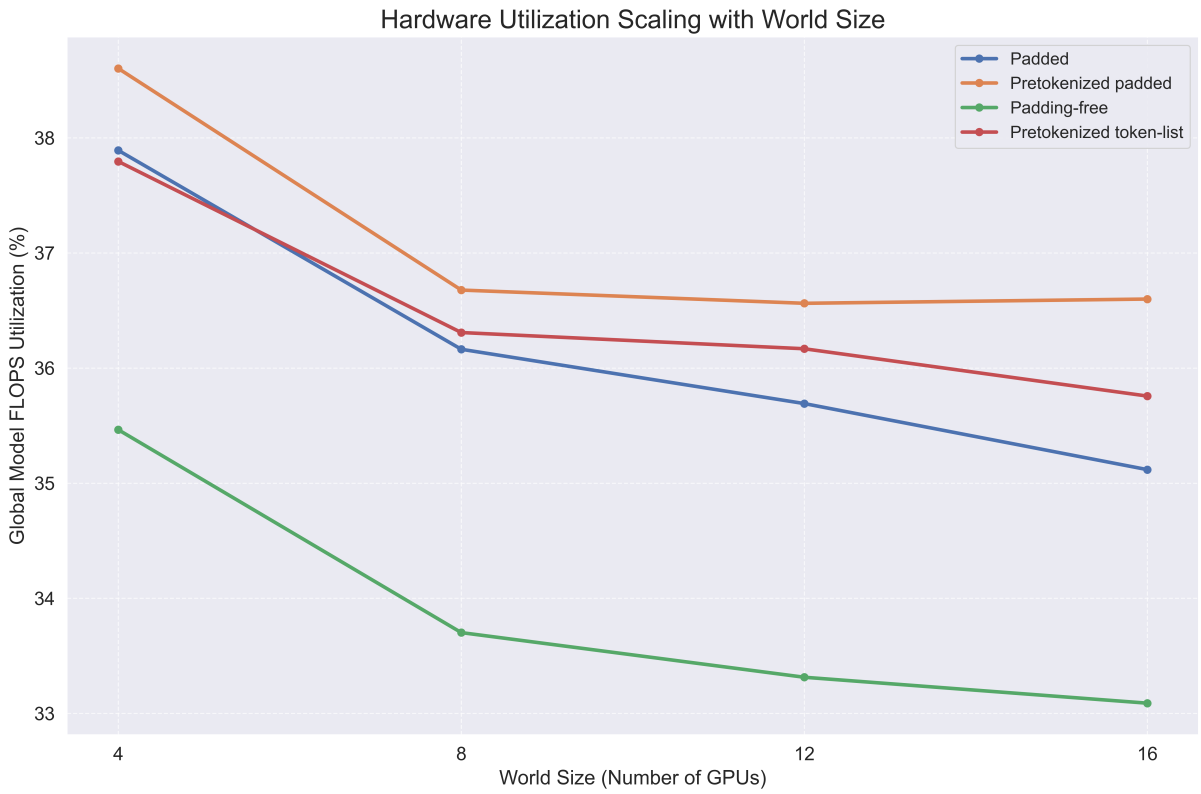

Figure 3: Hardware Utilization Scaling with World Size

MFU decreases with increasing world size for all approaches. This might be due to growing communication overhead of DDP (GPUs spend more time idle waiting for `all_reduce`).

Also here, we can see a widening gap between pretokenized and standard approaches at larger scales indicating that pretokenization becomes increasingly beneficial when scaling to larger global batch sizes.

Training Time

Comparing absolute training times for DDP is not insightful, this is because we do *more* work for increasing world size (remember: `global batch size = local batch size × world size`). For the plot of comparing absolute training time I refer to the Appendix.

We therefore plot "time to reach target accuracy" (e.g., time to reach $loss = 5.5$):

Figure 4: Time to reach target loss with annotated efficiency

The efficiency percentages show how much of the theoretical linear speedup we're achieving. We see that all approaches show generally significant speedup when increasing world size - the time to reach the target loss decreases substantially. However, I cannot explain why for some configuration (especially for `world_size=12`) we have slightly higher times again (maybe due to topology effects).

Loss Convergence

The validation loss curves reveal important insights about how world size affects the convergence:

Figure 5: Loss over time for the different configurations we tested

Looking at these curves, we observe that configurations with larger world sizes (12, 16) consistently achieve lower loss values at the same training step compared to smaller world sizes (4, 8). This confirms that the effectively larger batch sizes created through DDP (`global batch size = local batch size × world size`) lead to more efficient optimization *per step*.

Conclusion

Our implementation and evaluation of Distributed Data Parallel (DDP) training for LLMs yielded several key findings:

- DDP enables effective scaling of training across multiple nodes and GPUs, as we achieved throughput scaling efficiencies of 92-96% of ideal linear scaling.
- Our time-to-target-loss analysis reveals the fundamental benefit of DDP - reducing training time to reach a target performance level. Furthermore, the loss curves demonstrate that the configurations with larger world sizes (12, 16) achieve lower loss values at the same training step compared to smaller world sizes (4, 8).
- Pretokenization provides greater benefits in distributed settings than in single-GPU training. As we scale to larger world sizes, the gap in hardware utilization (MFU) between pretokenized and standard approaches widens, which highlights the compounding benefits of combining these optimizations.
- When comparing to our findings in *Feature 1 (Pretokenization)*, we see that the negligible benefits of pretokenization for small batch sizes become more pronounced with the effectively

larger batch sizes created through DDP (`global batch size = local batch size × world size`). This confirms our hypothesis that pretokenization benefits increase with larger effective batch sizes.

The complete implementation, including all source code, benchmark scripts, and analysis tools used, is available at: https://github.com/RamonKaspar/Large-Scale-AI-Engineering-Project.

Appendix

Implementation `DistributedSampler`

```python
class DistributedSampler(Sampler):

    def __init__(self, dataset: Dataset, num_replicas: Optional[int] =
    ↪  None, rank: Optional[int] = None, shuffle: bool = True, seed: int =
    ↪  0, drop_last: bool = False) -> None:
        if num_replicas is None:
            num_replicas = torch.distributed.get_world_size()
        if rank is None:
            rank = torch.distributed.get_rank()
        self.dataset = dataset
        self.num_replicas = num_replicas
        self.rank = rank
        self.epoch = 0
        self.drop_last = drop_last

        # If the dataset length is known
        if hasattr(dataset, "__len__"):
            self.num_samples = len(dataset) // self.num_replicas
            if not self.drop_last and len(dataset) % self.num_replicas !=
            ↪  0:
                # Extra samples to distribute among ranks
                self.num_samples += 1
            # Determine the global offset for each process
            self.total_size = self.num_samples * self.num_replicas
            # Record key parameters used
            self.shuffle = shuffle
            self.seed = seed
        else:
```

```
            # For iterable datasets where length is unknown, we just track
            ↪  epoch for seed changes
            self.shuffle = shuffle
            self.seed = seed

    def __iter__(self) -> Iterator:
        if hasattr(self.dataset, "__len__"):
            # Map-style datasets with known length
            if self.shuffle:
                # deterministically shuffle based on epoch and seed
                g = torch.Generator()
                g.manual_seed(self.seed + self.epoch)
                indices = torch.randperm(len(self.dataset),
↪   generator=g).tolist()
            else:
                indices = list(range(len(self.dataset)))

            # Ensure all workers process the same number of samples
            if not self.drop_last:
                # Add extra samples to make it evenly divisible
                padding_size = self.total_size - len(indices)
                if padding_size > 0:
                    indices += indices[:padding_size]
            else:
                # Remove tail of data to make it evenly divisible
                indices = indices[:self.total_size]

            # Subsample
            assert len(indices) == self.total_size
            indices = indices[self.rank:self.total_size:self.num_replicas]
            assert len(indices) == self.num_samples
            return iter(indices)
        else:
            # The actual sharding is done by the IterableDataset itself
            return iter(range(self.num_replicas))

    def __len__(self) -> int:
        if hasattr(self.dataset, "__len__"):
            return self.num_samples
        else:
            # For iterable datasets, we cannot know the length in advance
            return 0
```

```
    def set_epoch(self, epoch: int) -> None:
        self.epoch = epoch
```

Absolute Training Time

We see that the (absolute) training time increases with increasing world size. This is because each step takes longer because of gradient synchronization overhead, however, we also process more data per step (because `global batch size = local batch size × world size`).

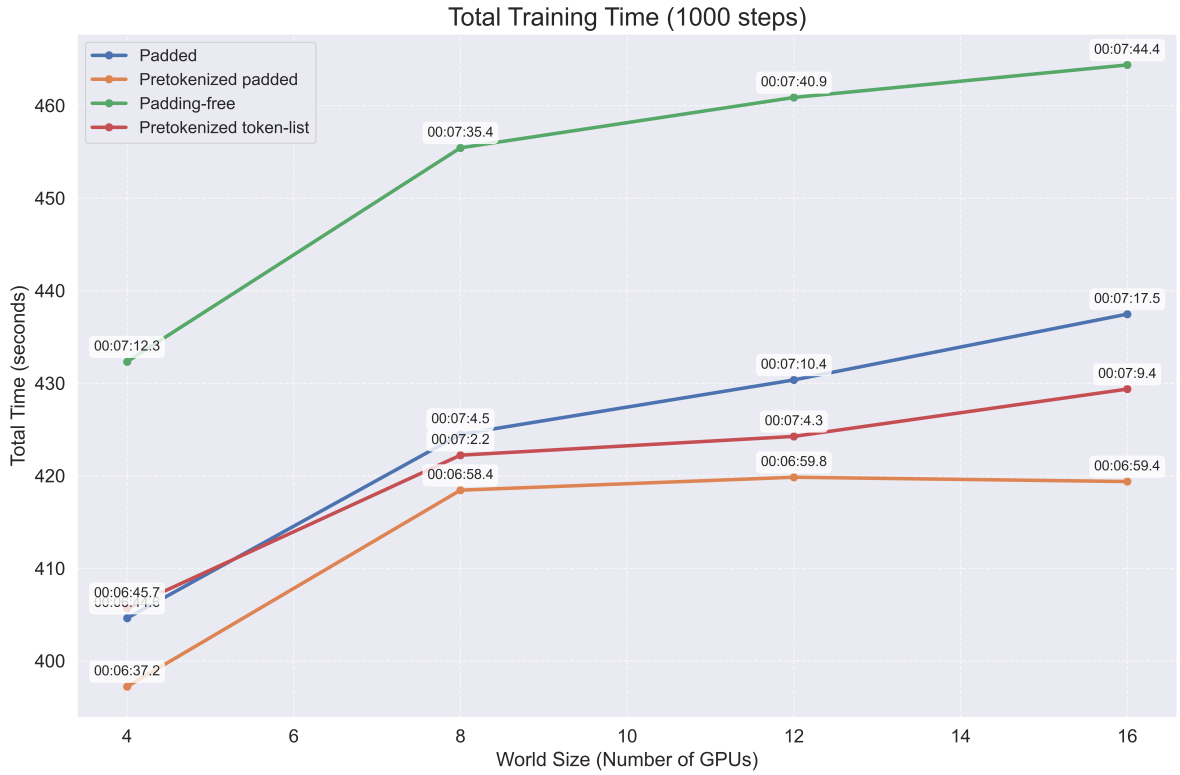

Figure 6: Total Training Time (1000 steps)