大規模言語モデルにおける分散並列学習

スパコンコロキウム（第10回）
オンライン
2024/01/30

東京工業大学
学術国際情報センター
横田 理央

自己紹介

学歴・職歴

2009年	慶應義塾大学大学院理工学研究科　博士(工学)　取得
2009年—2010年	ブリストル大学数学科　ポスドク研究員
2010年—2011年	ボストン大学機械工学科　ポスドク研究員
2011年—2015年	アブドゥラ国王科学技術大学　常勤研究員
2015年—2022年	東京工業大学　学術国際情報センター　准教授
2023年—	東京工業大学　学術国際情報センター　教授

研究経歴

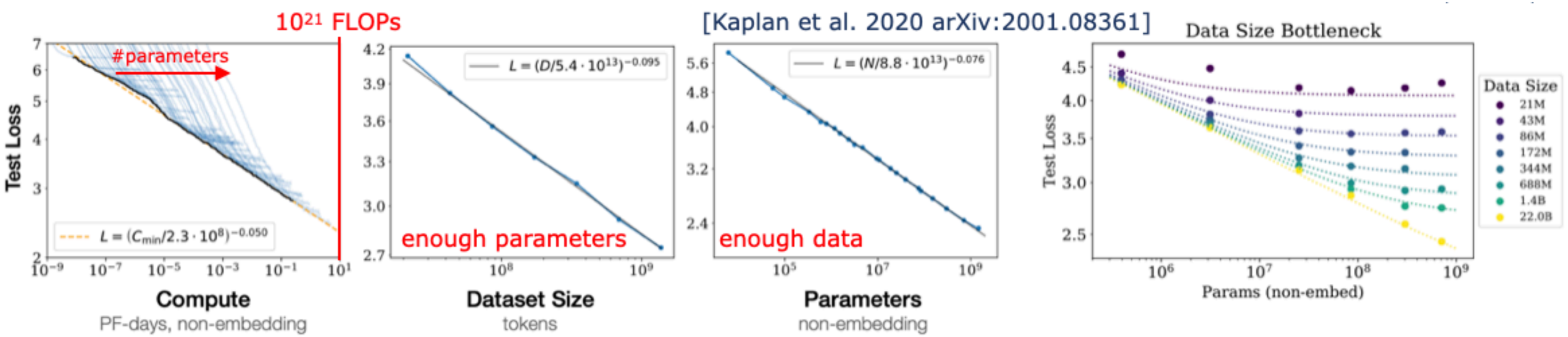
2005年ー2010年	渦法による乱流解析、高速多重極展開法(FMM)、GPU
2010年ー2015年	分子シミュレーション、FMM、GPU、H行列
2015年ー	深層学習、FMM、GPU、H行列

日本語 LLM ベンチマーク

	run name	Overall	llm-jp-eval	QA	NLI	FA	RC	MR	EL	MC	MT-bench	coding	extraction
25	gpt-4-0613	0.7622	0.6463	0.415	0.768	0.2199	0.891	0.97	0.3004	0.96	8.781	7.8	9.65
31	gpt-4-1106-preview	0.7479	0.6295	0.4002	0.74	0.2388	0.8206	0.96	0.317	0.93	8.663	7.5	9.5
22	gpt-3.5-turbo	0.6701	0.5161	0.2723	0.56	0.1886	0.8406	0.67	0.2913	0.79	8.241	7.9	9
2	anthropic.claude-v2:1	0.6682	0.5188	0.2882	0.676	0.129	0.6575	0.84	0.201	0.84	8.175	8.6	8.7
1	anthropic.claude-v1	0.6387	0.4911	0.3326	0.694	0.1613	0.4951	0.75	0.2244	0.78	7.863	7.1	7.85
3	anthropic.claude-v2	0.6345	0.4834	0.2888	0.642	0.1454	0.4977	0.82	0.2	0.79	7.856	7.45	8.4
4	gemini-pro	0.5851	0.4415	0.421	0.68	0.1745	0.7095	0	0.1856	0.92	7.287	5.55	8.85
12	stabilityai/StableBeluga2	0.5284	0.4111	0.2123	0.654	0.1158	0.8839	0.72	0.2915	0	6.456	4.3	7.1
16	mistralai/Mixtral-8x7B-Instruct-v0.1	0.5006	0.2774	0.1155	0.672	0.0793	0.7563	0.13	0.189	0	7.238	6.9	8.8
15	tokyotech-llm/Swallow-70b-instruct-hf	0.4712	0.5036	0.4803	0.642	0.1797	0.8506	0.7	0.0927	0.58	4.387	2.4	6.95
19	stabilityai/japanese-stablelm-instruct-beta-70b	0.3732	0.2432	0.2975	0.062	0.0558	0.7055	0.55	0.0117	0.02	5.031	3	5.65
29	tokyotech-llm/Swallow-13b-instruct-hf	0.373	0.3716	0.3946	0.454	0.1623	0.7811	0.28	0.049	0.48	3.744	1.2	5.45
21	tokyotech-llm/Swallow-7b-instruct-hf	0.3689	0.3734	0.3947	0.454	0.1591	0.7871	0.27	0.049	0.5	3.644	1.25	5.6
10	rinna/nekomata-14b-instruction	0.3644	0.4375	0.3402	0.494	0.1651	0.8663	0.42	0.0067	0.77	2.912	2.5	2.45
34	stabilityai/StableBeluga-13B	0.3626	0.2965	0.1893	0.572	0.06	0.8114	0.44	0.0029	0	4.288	2.85	7.4
26	stabilityai/StableBeluga-7B	0.3284	0.2567	0.09	0.5	0.052	0.655	0.5	0	0	4	2	5
6	elyza/ELYZA-japanese-Llama-2-13b-instruct	0.3278	0.1506	0.1741	0.128	0.0668	0.5352	0.15	0	0	5.05	2.9	5.3
11	meta-llama/Llama-2-70b-chat-hf	0.3004	0.0783	0.0471	0.152	0.0117	0.3373	0	0	0	5.225	4.05	6.75
20	llm-jp/llm-jp-13b-instruct-lora-jaster-v1.0	0.2947	0.4687	0.5239	0.928	0.0059	0.9229	0.01	0	0.89	1.206	1	1.3
30	llm-jp/llm-jp-13b-instruct-full-jaster-v1.0	0.2927	0.4698	0.5371	0.93	0.0016	0.91	0	0	0.91	1.156	1	1.6

https://wandb.ai/wandb-japan/llm-leaderboard/reports/Nejumi-Leaderboard-Neo--Vmlldzo2MTkyMTU0

Transformerのスケール則 (Scaling Law)



・パラメータ数が十分な場合、データ量に対してTest Lossはべき乗則に従って減少する

・データ量が十分な場合、バラメータ数に対してTest Lossはべき乗則に従って減少する

・必要な計算資源 [FLOPs] = パラメータ数 x テータ量 [tokens] x 5.7

・どのくらいの計算資源(予算)を投入すればどのくらいの性能のものができるかが予想できる

・データ量はパラメータ数に比例させる必要がある （データ量：パラメータ数=20:1）

・GPT-3は データ量：パラメータ数が2:1 になっており圧倒的にデータが不足している

・最近は推論コストを抑えるためにパラメータ数に対してデータ量を増やす傾向が顕著に

Model	Size (# Parameters)	Training Tokens
LaMDA (Thoppilan et al., 2022)	137 Billion	168 Billion
GPT-3 (Brown et al., 2020)	175 Billion	300 Billion
Jurassic (Lieber et al., 2021)	178 Billion	300 Billion
Gopher (Rae et al., 2021)	280 Billion	300 Billion
MT-NLG 530B (Smith et al., 2022)	530 Billion	270 Billion
Chinchilla	70 Billion	1.4 Trillion

[Hoffmann et al. 2022 arXiv:2203.15556]

GPT-4(3x10^{24} FLOPs)をスパコンで学習するには？

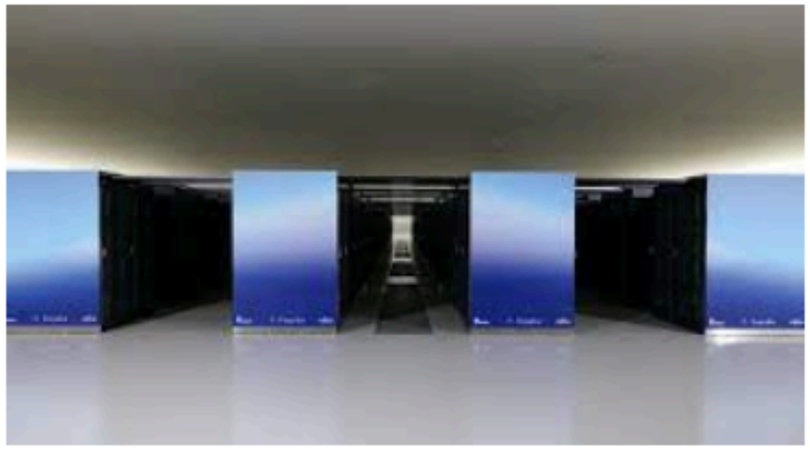
OpenAI (A100x25,000) 90日?

Frontier (MI250Xx37,888) 75日

LUMI (MI250Xx20,480) 140日

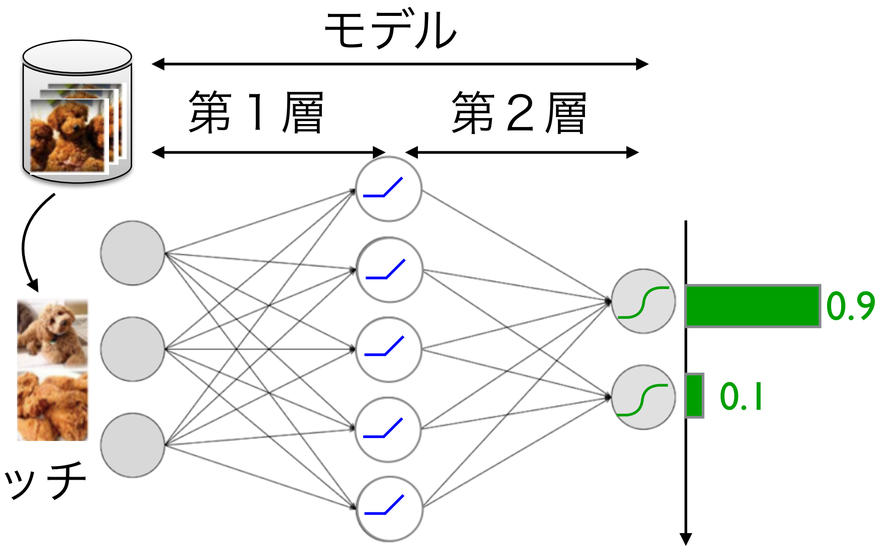
Aurora (PVCx63,744) 30日?

Fugaku (A64FXx158,976) 2100日



ABCI (A100x960+V100x4352) 770日

深層学習の用語説明



$$x_0 \quad x_1 = f(z_0) \quad p = g(z_1) \quad \mathcal{L} = \sum^{batch} -\log p$$

$$z_0 = W_0 x_0 \quad z_1 = W_1 x_1$$

Forward propagation

Backward propagation

$$\frac{\partial x_1}{\partial z_0} * \frac{\partial z_1}{\partial x_1} * \frac{\partial p}{\partial z_1} * \frac{\partial \mathcal{L}}{\partial p} = \frac{\partial \mathcal{L}}{\partial W}$$

$$\frac{\partial z_0}{\partial W_0} \quad \frac{\partial z_1}{\partial W_1}$$

微分の連鎖則 (Chain rule)

x_0：第１層の入力（モデルの入力）
W_0：第１層の重み
z_0：第１層の出力
$f()$：活性化関数
x_1：第２層の入力（活性）
W_1：第２層の重み
z_1：第２層の出力
$g()$：Softmax関数
p：モデルの出力
$\mathcal{L}$：損失関数
$\frac{\partial \mathcal{L}}{\partial W}$：勾配
η：学習率
m, v：Optimizer state

SGD（確率的勾配降下法）

$$W^{t+1} = W^t - \eta \frac{\partial \mathcal{L}}{\partial W^t}$$

Adam

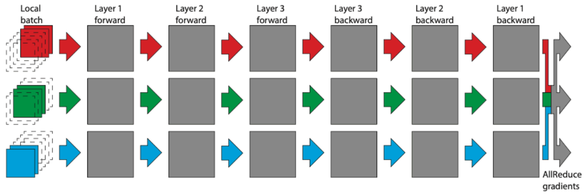
$$m^{t+1} = \beta_1 m^t - (1-\beta_1)\frac{\partial \mathcal{L}}{\partial W^t}$$

$$v^{t+1} = \beta_2 v^t - (1-\beta_2)\left(\frac{\partial \mathcal{L}}{\partial W^t}\right)^2$$

$$W^{t+1} = W^t - \eta \frac{m^{t+1}}{\sqrt{v^{t+1}+\epsilon}} b^{t+1}$$

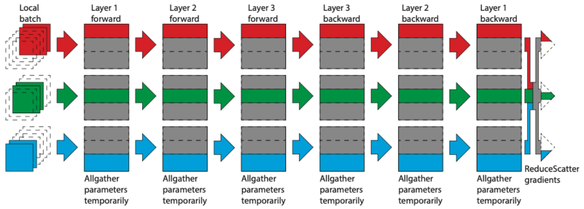
分散並列学習

データ並列 (DP)



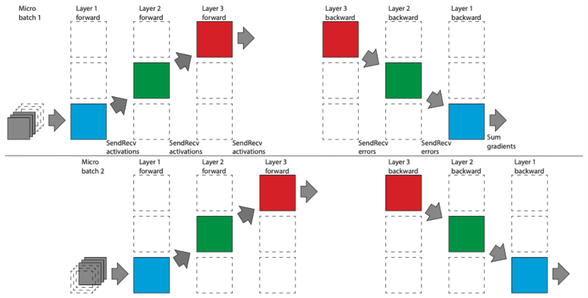
データ：分散
モデル：冗長
通信内容：勾配
通信形式：AllReduce
通信頻度：ステップ毎
長所：実装が簡単
短所：ラージバッチ問題
　　メモリ消費量

ZeRO (FSDP)



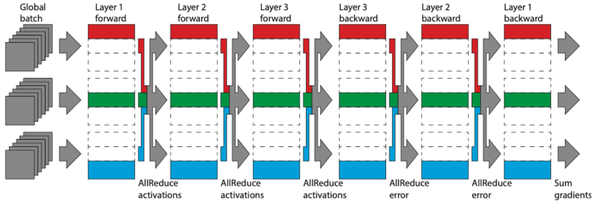
データ：分散
モデル：一時的に分散
通信内容：勾配＋重み
通信形式：ReduceScatter
　　＋AllGather
通信頻度：層毎
長所：実装が簡単
　　省メモリ
短所：ラージバッチ問題

パイプライン並列 (PP)



データ：冗長
モデル：分散
通信内容：活性
通信形式：SendRecv
通信頻度：層毎
長所：省メモリ
　　演算量低減
短所：パイプラインバブル

テンソル並列 (TP)



データ：冗長
モデル：分散
通信内容：活性
通信形式：AllReduce
通信頻度：層毎
長所：省メモリ
　　演算量低減
短所：通信オーバーヘッド
　　オーバーラップ不可
　　実装が複雑

データ並列

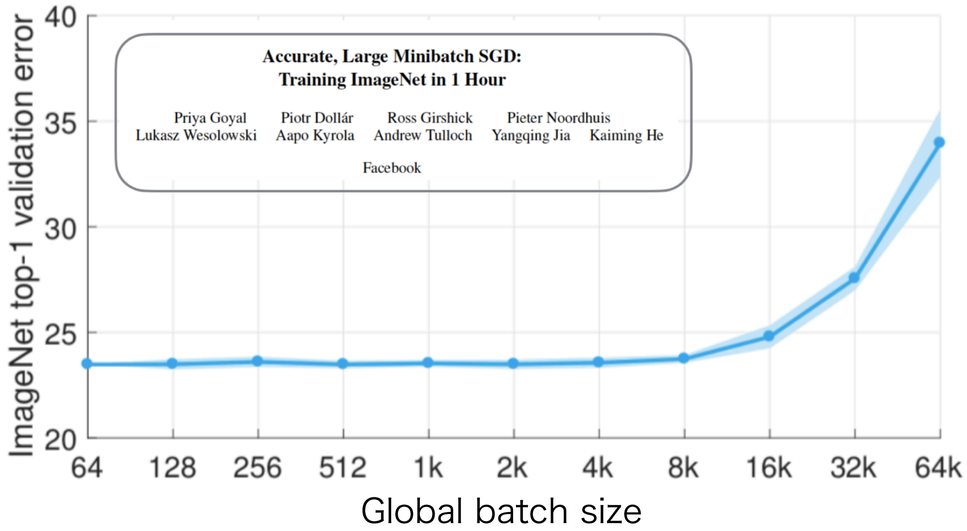
Local batch：各プロセスのバッチ → 小さいとFLOP/sが出ない → メモリ上限まで増大

Global batch：全プロセスのバッチ → プロセス数に比例

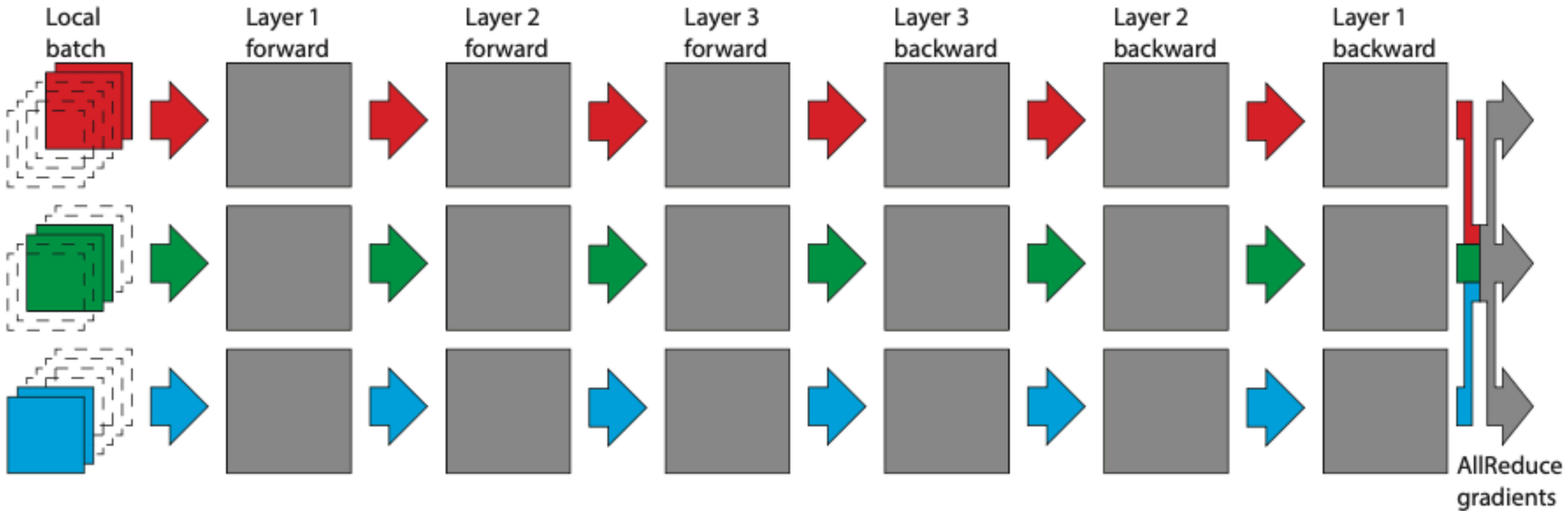
Global batch sizeが増大すると汎化性能が低下 →

解決策

- 途中からbatch sizeを上げる
- 特殊な最適化手法を使う (LARS,LAMB)
- 特殊な正則化項を加える (勾配分散)



Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour
https://arxiv.org/abs/1706.02677



ZeRO

データ並列におけるモデルの冗長性を緩和 → 消費メモリはプロセス数に反比例

ZeRO-1：Optimizer stateを分散

ZeRO-2：Optimizer state＋勾配を分散

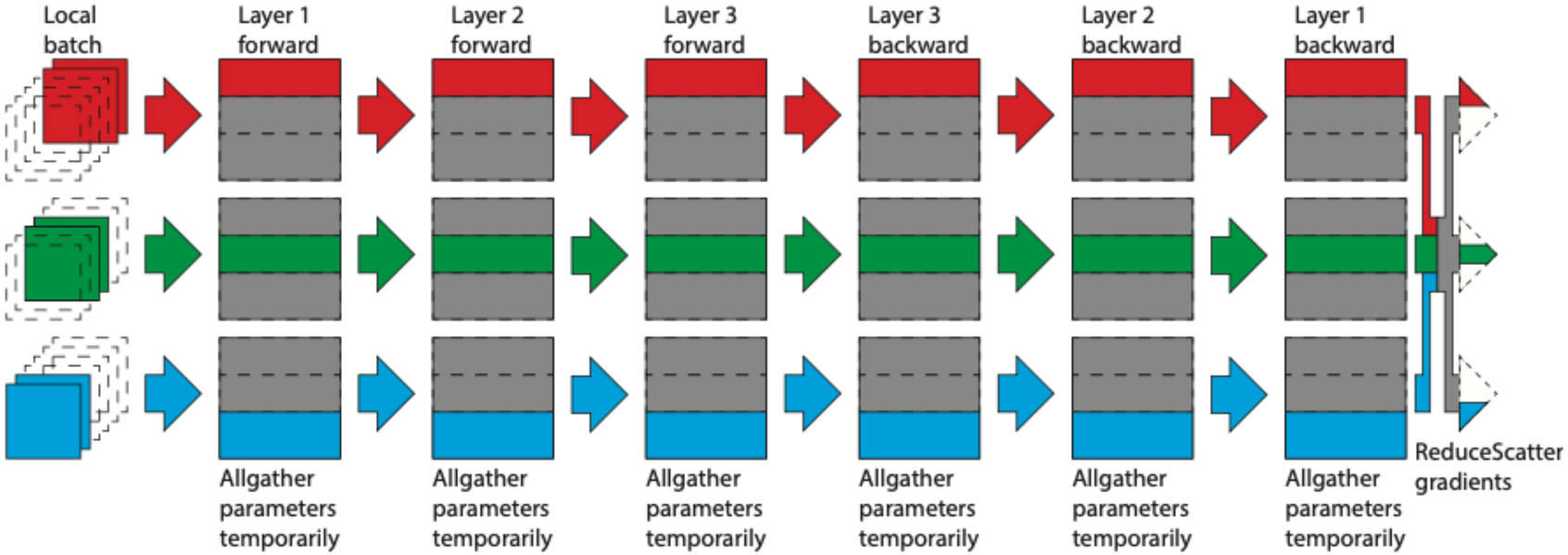
ZeRO-3：Optimizer state＋勾配＋重みを分散（重みをAllGather）

AllGatherして逐次計算をするMPIコードのようなもの → 演算量はデータ並列と同じ

バッチサイズが増大する問題もデータ並列と同じ

データ並列の通信量：AllReduce = ReduceScatter + AllGather

ZeRO-3の通信量：ReduceScatter + AllGather x 2 (データ並列の1.5倍)



パイプライン並列

Local batchをさらに細かくmicro batchに分けてパイプライン処理する

隣の層を担当するプロセスとの近接通信(SendRecv)

自分が担当する層のみを計算する → メモリも演算もプロセス数に反比例

パイプラインバブルを低減するために様々な手法が提案されている

- GPipe：micro batch

  [https://arxiv.org/abs/1811.06965]

- PipeDream：非同期

  [https://arxiv.org/abs/1806.03377]

- Interleaved 1F1B：層数をプロセス数x2に分割

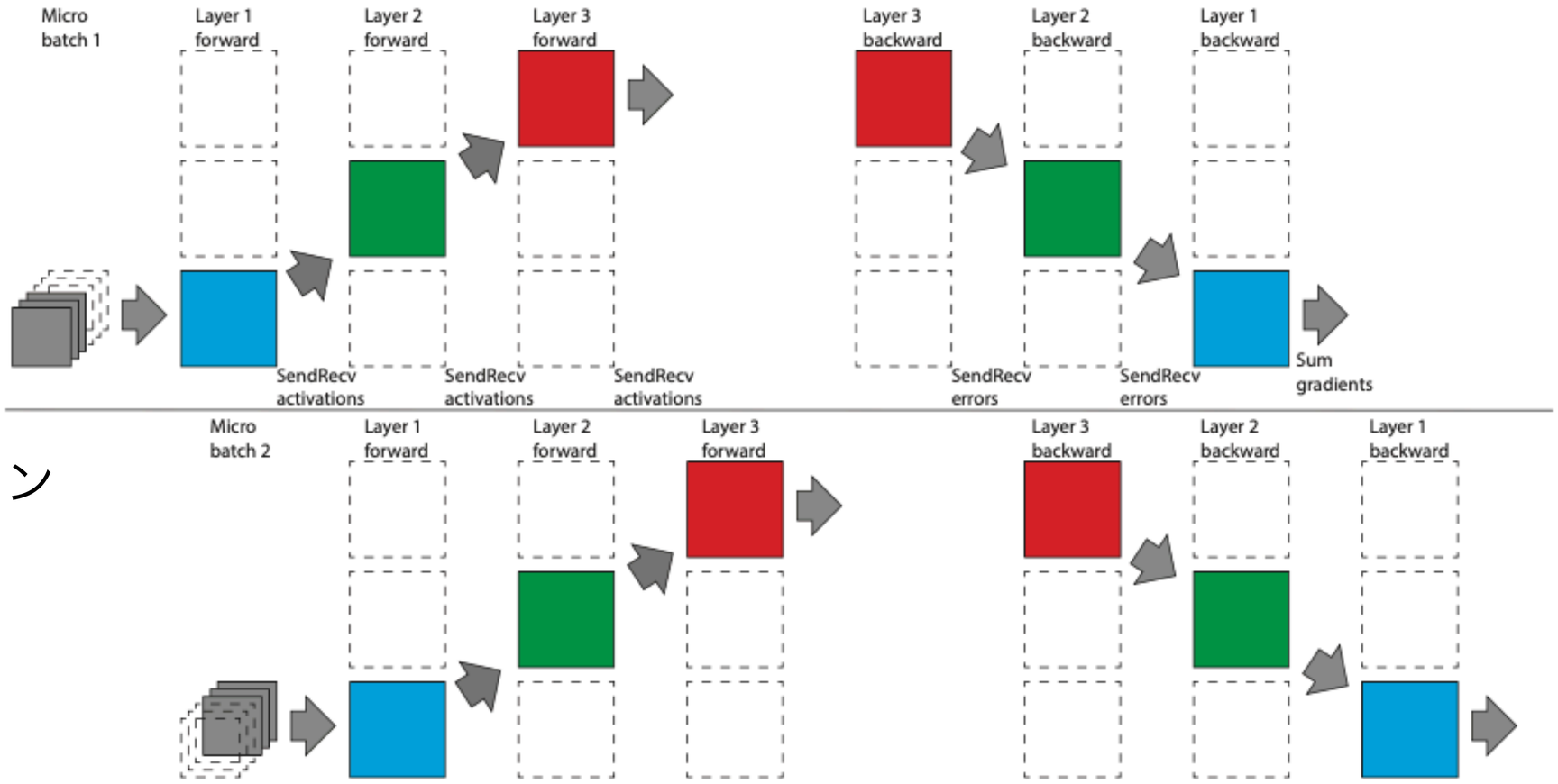
  [https://arxiv.org/abs/2104.04473]

- Chimera：双方向パイプライン

  [https://arxiv.org/abs/2107.06925]

- ZeroBubble：Backwardのw微分とx微分を分ける

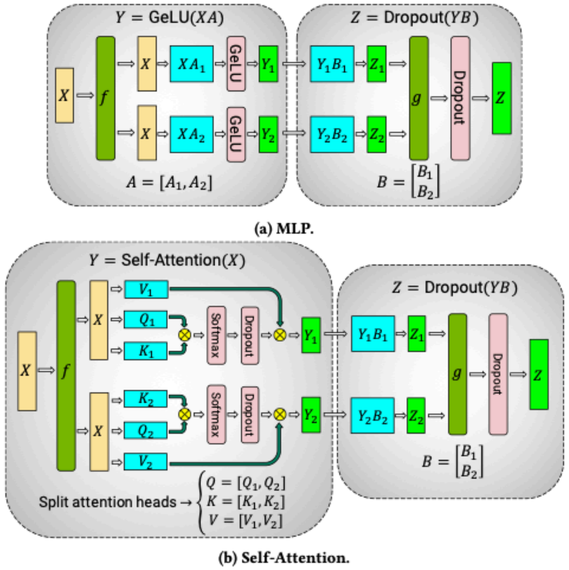
  [https://arxiv.org/abs/2401.10241]



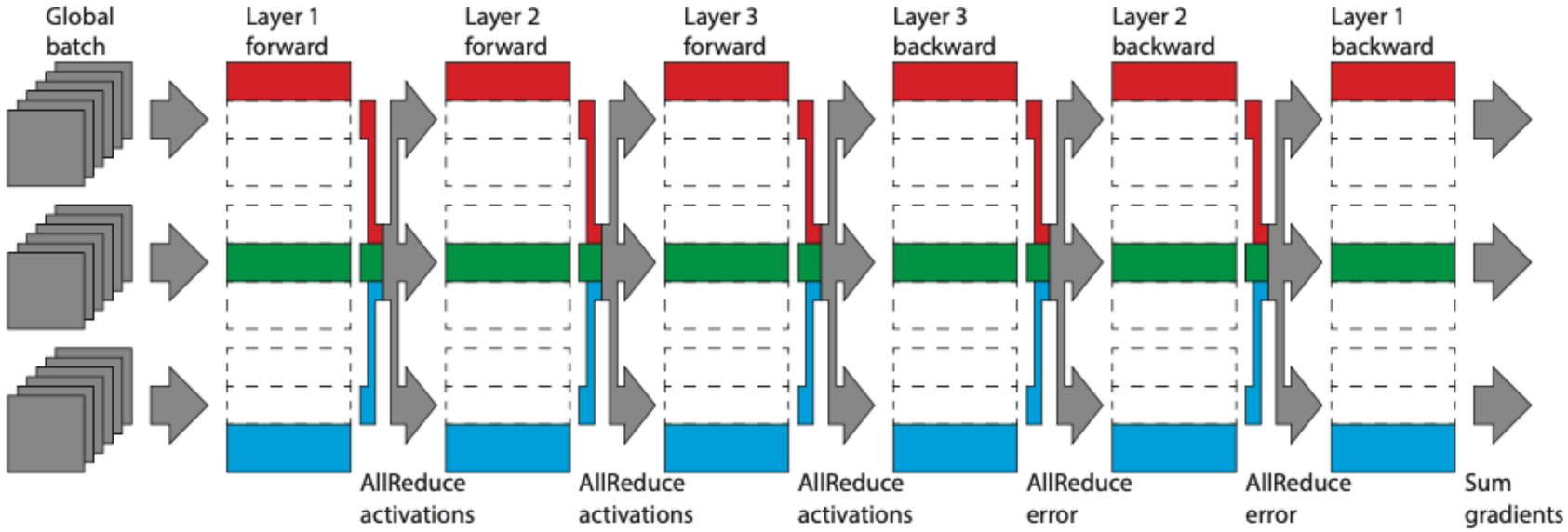
テンソル並列

層内の行列積の分散並列処理 → メモリも演算もプロセス数に反比例
→ 実装がモデルアーキテクチャに依存するため自動並列化が困難 →

次の層の計算をするには必ず活性のAllReduceが必要
AllReduceをするには前の層の計算が完了していることが必要
→つまり、計算と通信をオーバーラップする余地が全くない

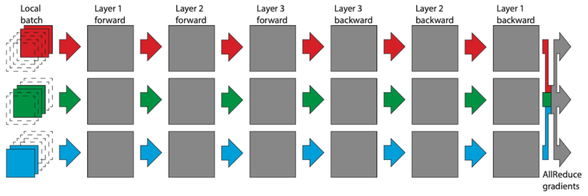


Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM
https://arxiv.org/abs/2104.04473



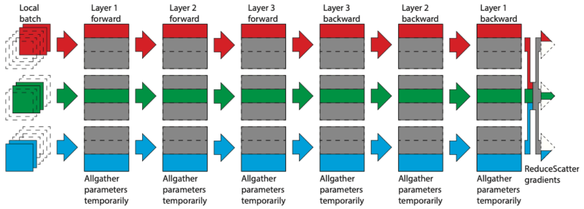
分散並列学習

データ並列 (DP)



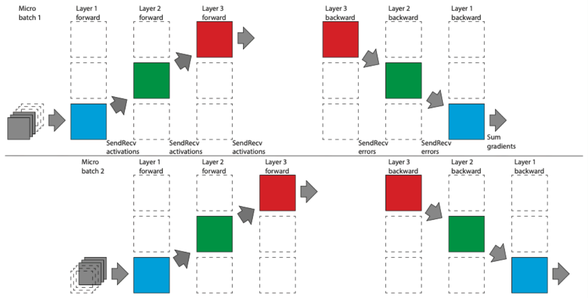
データ：分散
モデル：冗長
通信内容：勾配
通信形式：AllReduce
通信頻度：ステップ毎
長所：実装が簡単
短所：ラージバッチ問題
　　メモリ消費量

ZeRO (FSDP)



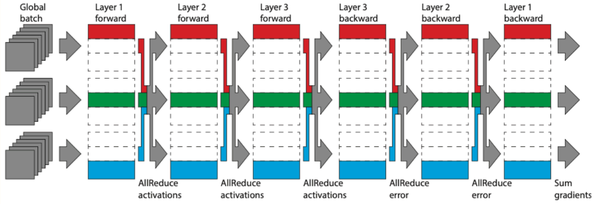
データ：分散
モデル：一時的に分散
通信内容：勾配＋重み
通信形式：ReduceScatter
　　＋AllGather
通信頻度：層毎
長所：実装が簡単
　　省メモリ
短所：ラージバッチ問題

パイプライン並列 (PP)



データ：冗長
モデル：分散
通信内容：活性
通信形式：SendRecv
通信頻度：層毎
長所：省メモリ
短所：パイプラインバブル

テンソル並列 (TP)



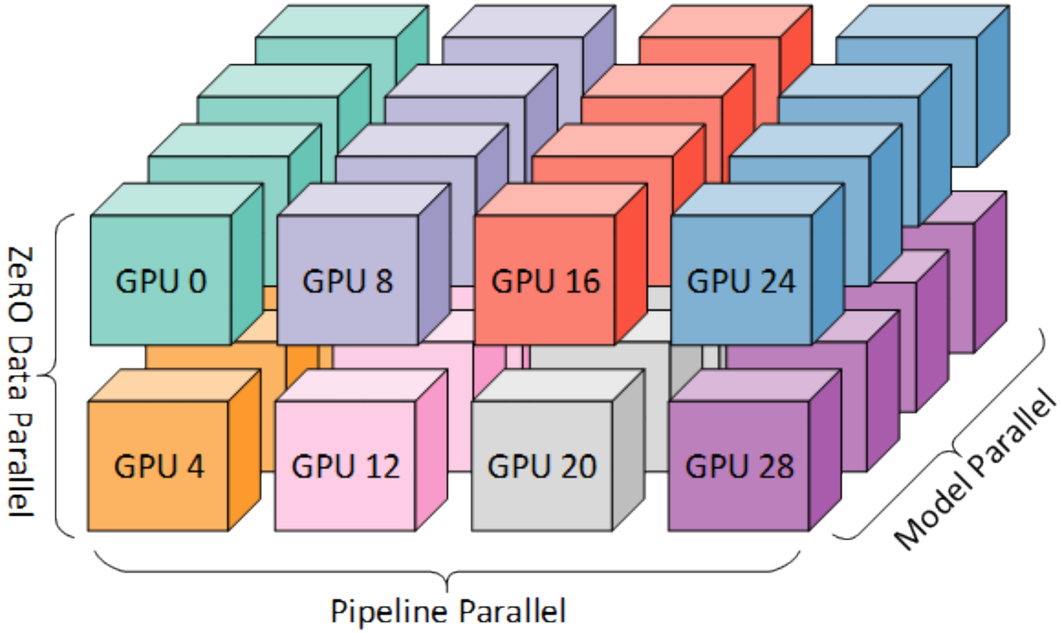
データ：冗長
モデル：分散
通信内容：活性
通信形式：AllReduce
通信頻度：層毎
長所：省メモリ
短所：通信オーバーヘッド
　　オーバーラップ不可
　　実装が複雑

3-D Parallelism

プロセスを３次元分割し、
データ並列とZeROの軸、
パイプライン並列の軸、
テンソル並列の軸
の3軸でそれぞれの分散並列化を行う

それぞれのAllReduce等は
サブコミュニケータ上で行われる

よくある設定としては
ノード内のGPU間でテンソル並列
ラック内のノード間でパイプライン並列
ラック間でデータ並列及びZeRO
→ ラックが増えるとラージバッチ問題が起きる



まとめ

大規模言語モデルの事前学習は、最大級のスパコンの全系を使っても何ヶ月もかかる

全系までスケーリングさせるためには、４種類の並列化手法を組み合わせる必要がある

データ並列は最も簡単に実装できるが、メモリ消費量が多く、大きなモデルを扱えない

ZeROはメモリ消費量は低減できるが、バッチサイズの増大に伴う汎化性能の問題は解決できない

パイプライン並列はバッチサイズの増大は起きないが、パイプラインバブルが問題になる

テンソル並列は上記の問題は全て解決できるが、通信が隠蔽できない

よって、４種類の並列化手法を組み合わせて、それぞれを上限までスケーリングさせることが
今後の分散並列学習において重要な課題となっている