Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Automatic parallelism strategy generation with minimal memory redundancy*

Yanqi SHI§, Peng LIANG§, Hao ZHENG, Linbo QIAO, Dongsheng LI‡

National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, Changsha 410000, China

E-mail: yqshi@nudt.edu.cn; peng_leung@nudt.edu.cn; zhengh@nudt.edu.cn; linboqiao@nudt.edu.cn; lds1201@163.com

Received Oct. 10, 2023; Revision accepted Oct. 17, 2023; Crosschecked Nov. 14, 2024; Published online Dec. 27, 2024

Abstract: Large-scale deep learning models are trained distributedly due to memory and computing resource limitations. Few existing strategy generation approaches take optimal memory minimization as the objective. To fill in this gap, we propose a novel algorithm that generates optimal parallelism strategies with the constraint of minimal memory redundancy. We propose a novel redundant memory cost model to calculate the memory overhead of each operator in a given parallel strategy. To generate the optimal parallelism strategy, we formulate the parallelism strategy search problem into an integer linear programming problem and use an efficient solver to find minimal-memory intra-operator parallelism strategies. Furthermore, the proposed algorithm has been extended and implemented in a multi-dimensional parallel training framework and is characterized by high throughput and minimal memory redundancy. Experimental results demonstrate that our approach achieves memory savings of up to 67% compared to the latest Megatron-LM strategies; in contrast, the gap between the throughput of our approach and its counterparts is not large.

Key words: Deep learning; Automatic parallelism; Minimal memory redundancy
https://doi.org/10.1631/FITEE.2300684 **CLC number:** TP181

1 Introduction

Large-scale deep learning models have been a tremendously popular area of study in recent years owing to their excellent performance improvements in various fields, such as image recognition, speech recognition, dialogue systems, natural language processing, and recommendation systems (Krizhevsky et al., 2012; Naumov et al., 2019; Brown et al., 2020; Dan et al., 2023), which is a result of increasing model sizes and dataset sizes (Zhuang et al., 2017). For example, PaLM with 540 billion parameters was trained with a corpus of 780 billion tokens representing a wide range of natural language use cases (Chowdhery et al., 2022).

As a result, training these models is computation-intensive and time-consuming, requiring a lot of computing resources. However, resources on a single node are limited and insufficient to train these models (Lan et al., 2018; Guan et al., 2020). To address this challenge, researchers have proposed various parallelism strategies to accelerate the process of training large-scale deep learning models. However, designing parallel strategies is not a straightforward task, and it often requires considerable expertise and trial-and-error exploration to achieve high performance (Mo, 2018; He et al., 2023).

To relieve us from the parallelism design procedure, researchers have proposed auto-parallelism methods (Jia et al., 2018; Cai et al., 2022; Zheng

§ These two authors contributed equally to this work
‡ Corresponding author
* Project supported by the National Natural Science Foundation of China (Nos. 62025208 and 62421002)
ORCID: Yanqi SHI, https://orcid.org/0000-0002-8899-1018; Peng LIANG, https://orcid.org/0000-0002-5590-5179; Dongsheng LI, https://orcid.org/0000-0001-9743-2034

et al., 2022; Liu YL et al., 2023) that can find decent strategies given a specific model and environment. These methods model the communication costs of different parallelism strategies and then use a search algorithm to find the optimal strategy. However, existing studies typically aim to minimize the communication cost for a given model. Under this search objective, the resulting parallelism strategies are often not memory-optimal.

To fill in this gap, we propose an algorithm to search for memory-optimized operator-level parallelism strategies by analyzing the memory overhead of neural networks in distributed training. This enables us to further increase the hidden layer or batch size on hardware resources with limited memory.

Initially, we instantiate a memory analyzer, enabling the computation of memory overhead for any specified strategy. Subsequently, we formalize the problem of strategy search. To accomplish this, we leverage an external solver specifically designed for integer linear programming (ILP) and employ it as a basis for arriving at the decisions, pertaining to optimal multi-dimensional intra-operator parallelism strategies that minimize memory usage.

The algorithm developed in this paper offers several vital contributions. First, it allows the identification of strategies with minimal redundant memory. Second, it involves a simplified expression of the strategy search problem, thus making it more concise and easy to understand. Finally, it allows experiments to be conducted that demonstrate the memory overhead benefits deriving from the present research concept.

2 Background

2.1 Intra- and inter-operator parallelism

Alpa (Zheng et al., 2022) categorizes existing parallelism methods into two orthogonal categories: intra- and inter-operator parallelism. Intra-operator parallelisms are parallelism schemes that partition an operator's involved tensors along some dimensions, assign the resulting partitioned computation to multiple devices, and let them execute different parts of the computation simultaneously. Inter-operator parallelism, including pipeline parallelism (Harlap et al., 2018; Huang et al., 2019; Liu ZM et al., 2023), partitions models into several stages with multiple operators. This paper focuses on generating multi-dimensional and memory-optimal intra-operator parallelism strategies.

2.2 Intra-operator parallelism schemes

Since Hinton trained AlexNet using two graphics processing units in 2012 (Krizhevsky et al., 2012), researchers have proposed many intra-operator parallelism schemes, including data parallelism (DP), model parallelism (MP) (Shazeer et al., 2018; Shoeybi et al., 2019), and zero redundancy data parallelism (ZeRO-DP) (Rajbhandari et al., 2020).

2.2.1 Data parallelism

DP partitions and distributes the data across devices with a replicated model. Each device computes the gradients using the split data and uses communication mechanisms such as AllReduce or Broadcast to synchronize the gradients or model parameters with other devices, so that after every iteration the models on all workers are the same.

2.2.2 Model parallelism

MP partitions the model parameters across devices and makes devices process the same data. MP produces partial sum or slicing results when the parameter matrix is partitioned in a row-wise or column-wise manner. Row-wise MP (Row-MP) requires synchronization to unify the operator's results on different devices. For column-wise MP (Column-MP), synchronization is performed only in backward propagation.

2.2.3 Zero redundancy data parallelism

ZeRO-DP (Rajbhandari et al., 2020) works by partitioning the model parameters, gradients, and optimizer states into multiple pieces and then distributes these pieces across the nodes in the distributed system in such a way that there is no redundancy in memory usage. ZeRO-DP has three stages: in stage 1 the optimizer states are partitioned, in stage 2 gradients are partitioned additionally, and in stage 3 model parameters are partitioned by introducing another All-Gather operation to guarantee mathematical equivalence. From stage 2, ZeRO-DP replaces the All-Reduce operation in DP with a Reduce-Scatter on gradients and an All-Gather operation on parameters to reduce the memory

redundancy of gradients. Since stage 3 might introduce more communication overhead which lowers the throughput, we apply stage 2 of ZeRO-DP in this work.

2.3 Strategy search algorithm and objective

Researchers have proposed methods to automatically search for parallelism strategies. ToFu (Wang et al., 2019), TensorOpt (Cai et al., 2022), and Alpa (Zheng et al., 2022) generate intra-operator parallelism strategies by minimizing the overall communication cost of a computation graph under the observation that all different strategies of an operator have the same computation cost. In the cases of ToFu and TensorOpt, to produce better results, the dynamic programming algorithm in OptCNN (Jia et al., 2018) is adapted. Alpa formalizes the search problem as an integer programming one and uses a solver to obtain the solution. In this work, we formulate the intra-operator parallelism strategy in a minimization manner, allowing us to use an ILP solver to assist in identifying memory-optimal strategies.

To our knowledge, unlike other automated parallelism methods, which aim to minimize the communication cost, our approach endeavors to generate parallelism strategies with a primary objective of memory minimization and a secondary objective of reducing the communication volume produced by such strategies, while achieving approximately equal performance compared with state-of-the-art (SOTA) methods.

3 Method design

3.1 Overview

Our approach primarily prioritizes the minimization of memory redundancy as an optimization objective. Diminishing memory redundancy enables us to further increase the hidden layer or batch size on hardware resources with limited memory.

To minimize memory redundancy, we propose a method to formalize the strategy search problem into an ILP using an auxiliary computation graph and to solve the problem accurately using a third-party ILP solver.

Furthermore, the constrained communication bandwidth has emerged as a bottleneck in amplifying parallel training efficiency. For DP, the impact of communication bandwidth is manifested mainly in parameter synchronization. In MP, training requires parameters such as gradients frequently. Limited communication bandwidth can result in a longer duration of such data exchanges, slowing down parallel training. Thus, to alleviate communication stress, we treat the communication volume generated during training as a secondary optimization objective.

Algorithm 1 provides an overview framework of the research embodied in the present study. In Section 3.2, we build a memory cost model to calculate the memory overhead of arbitrary intra-operator strategies. We generate candidate strategies for each operator in the computation graph and initialize an auxiliary computation graph. Each node in the auxiliary computation graph represents a node in the original graph with a specific candidate strategy. Details about the auxiliary computation graph can be

Algorithm 1 Parallelism strategy generation with minimal memory redundancy

Require: computation graph $G=(V,E)$ and device graph $D=(V_{\mathrm{D}},E_{\mathrm{D}})$
Ensure: parallelism strategy P
$G_{\mathrm{A}}=(V_{\mathrm{A}},E_{\mathrm{A}}),V_{\mathrm{A}}\leftarrow\varnothing,E_{\mathrm{A}}\leftarrow\varnothing$
// Generate an auxiliary graph
for $(u,w)\in E$ **do**
  $U_{\mathrm{A}}=\mathrm{GenerateCandidateStrategies}(u,D)$
  $W_{\mathrm{A}}=\mathrm{GenerateCandidateStrategies}(w,D)$
  $V_{\mathrm{A}}=U_{\mathrm{A}}\cup W_{\mathrm{A}}\cup V_{\mathrm{A}}$
  for $u_{\mathrm{A}}\in U_{\mathrm{A}}$ **do**
    for $w_{\mathrm{A}}\in W_{\mathrm{A}}$ **do**
      $E_{\mathrm{A}}=E_{\mathrm{A}}\cup\{(u_{\mathrm{A}},w_{\mathrm{A}})\}$
    end for
  end for
end for
$M\leftarrow\varnothing,C\leftarrow\varnothing$
// Compute redundant memory cost and communication
// volume
for $(u_{\mathrm{A}},w_{\mathrm{A}})\in E_{\mathrm{A}}$ **do**
  $m_{(u_{\mathrm{A}},w_{\mathrm{A}})}=\mathrm{ComputeMemory}(u_{\mathrm{A}},w_{\mathrm{A}})$
  $M=M\cup\{m_{(u_{\mathrm{A}},w_{\mathrm{A}})}\}$
  $c_{(u_{\mathrm{A}},w_{\mathrm{A}})}=\mathrm{ComputeResharding}(u_{\mathrm{A}},w_{\mathrm{A}})$
  $C=C\cup\{c_{(u_{\mathrm{A}},w_{\mathrm{A}})}\}$
end for
for $v_{\mathrm{A}}\in V_{\mathrm{A}}$ **do**
  $c_{u_{\mathrm{A}}}=\mathrm{ComputeParallel}(u_{\mathrm{A}},w_{\mathrm{A}})$
  $C=C\cup\{c_{v_{\mathrm{A}}}\}$
end for
// Search for the optimal strategy
$P=\mathrm{PathSearch}(C,M,G_{\mathrm{A}})$

found in Section 3.3. Then, we apply the cost model to the auxiliary graph, assigning redundant memory cost to each node to form a complete graph. In Section 3.4, we illustrate the process of calculating communication costs from two perspectives by analyzing the auxiliary computation graph and subsequently incorporating them into the optimization objective. Finally, we formalize the strategy search problem as an ILP problem in Section 3.5. The strategy search algorithm inputs the complete graph to find minimal-memory strategies.

3.2 Redundant memory cost model

To accurately evaluate the memory cost arising from the use of different parallelism strategies, we first build a memory cost model to calculate the memory overhead of arbitrary intra-operator strategies. The memory cost model takes an operator and its candidate parallelism strategy as inputs and returns the corresponding memory cost.

The key to building a memory cost model is understanding the redundancy of different strategies. DP, MP, and ZeRO-DP stage 2 bring memory redundancy to the training cluster. Taking matrix multiplication (MatMul) as an example, its forward computation is shown as Eq. (1), and its backward computation is shown as Eqs. (2) and (3).

$$Y = XW, \tag{1}$$

$$\nabla W = X^{\mathrm{T}}\nabla Y, \tag{2}$$

$$\nabla X = \nabla Y W^{\mathrm{T}}, \tag{3}$$

where X, W, and Y represent the input tensor, weight matrix, and output tensor, respectively. ∇X, ∇W, and ∇Y represent the gradients of X, W, and Y, respectively.

As Fig. 1 shows, although DP partitions X and Y along the batch size axis, it duplicates the model weights across devices within the ZeRO-DP (stage 2) group. Suppose that the parallelism degree is N. Then the redundancy it brings will be $(n-1)\mathrm{size}(W)$, where $\mathrm{size}(W)$ means the number of elements in W. Similarly, the redundancies of Row-MP and Column-MP are $(n-1)\mathrm{size}(Y)$ and $(n-1)\mathrm{size}(X)$, respectively. Suppose that the DP, Row-MP, and Column-MP degrees are d, r, and c, respectively. Then, the memory redundancy that a MatMul operator brings would be

$$\begin{aligned} M =& \frac{1}{drc}(d(d-1)\mathrm{size}(W) \\ &+ r(r-1)\mathrm{size}(Y) + c(c-1)\mathrm{size}(X)). \end{aligned} \tag{4}$$

With these formulations, we construct memory cost models for each operator. Using this redundant memory cost model, each memory redundancy caused by different strategies can be calculated as the weight of edges on the auxiliary graph.

3.3 Path search problem modeling and candidate strategy generation

The computation graph is a graphical structure used to describe and organize the computational operations of neural network models. The nodes in the computation graph represent operations, while the edges represent dependencies between different operations.

The nodes in the computation graph are amenable to the use of various potential parallelism strategies that can be employed for each operation. The collection of all possible parallelism strategies that a node can adopt is referred to as a candidate strategy. We expand the original computation graph to generate a new auxiliary computation graph based on these candidate strategies.

Each node in the auxiliary computation graph represents a parallelism strategy adopted for the computational operations corresponding to the original computation graph. Similar to the original computation graph, the auxiliary computation graph is a directed graph where edges represent the data flow. The weight of an edge refers to the memory overhead incurred by choosing a particular parallelism strategy. The value of the weight is determined through the redundant memory cost model.

Dissimilar to the case with the original calculation graph, the data flowing through the nodes are fixed. Different paths can be chosen in the auxiliary calculation graph. By generating an auxiliary calculation graph and calculating memory cost, we model the strategy search problem as a path search problem.

Fig. 2 shows an example: we first expand the original computation graph $G = (V, E)$ to form an auxiliary computation graph $G_{\mathrm{A}} = (V_{\mathrm{A}}, E_{\mathrm{A}})$. For node $A \in V$ in Fig. 2a, since it has K_A different strategies, there will be K_A auxiliary nodes in V_{A},

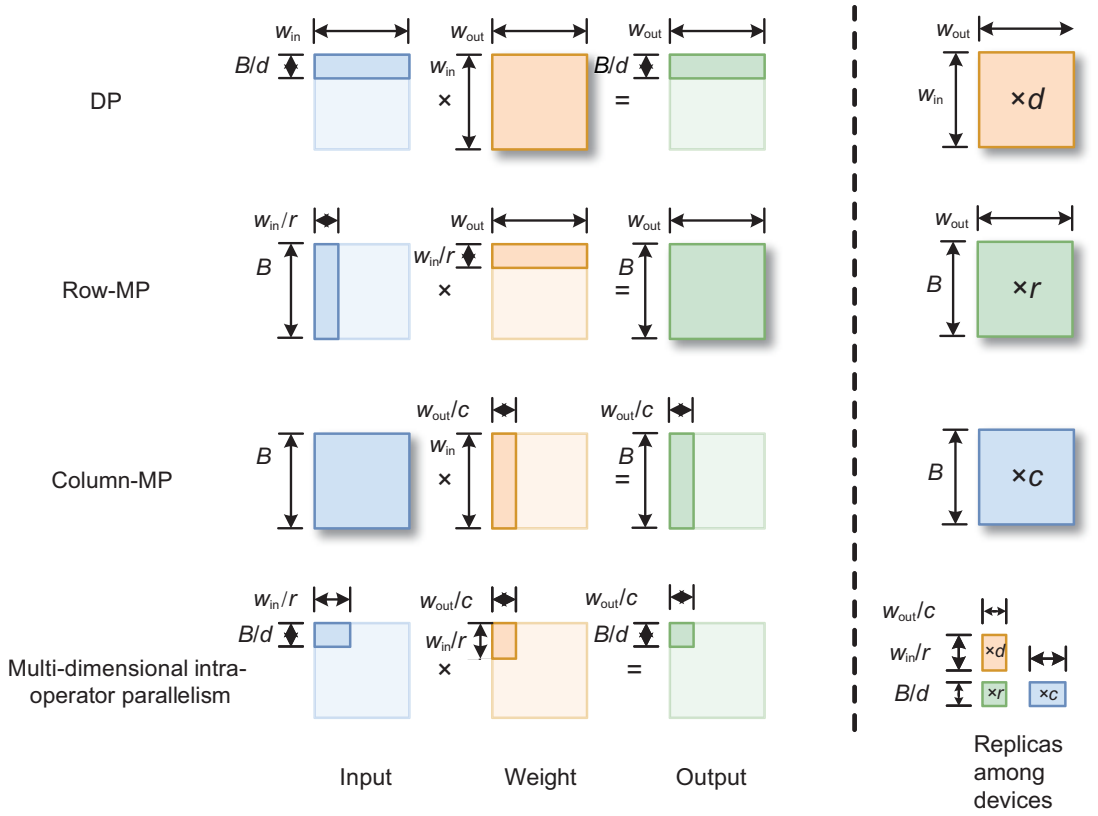

Fig. 1 Illustration of redundancy memory in intra-operator parallelism

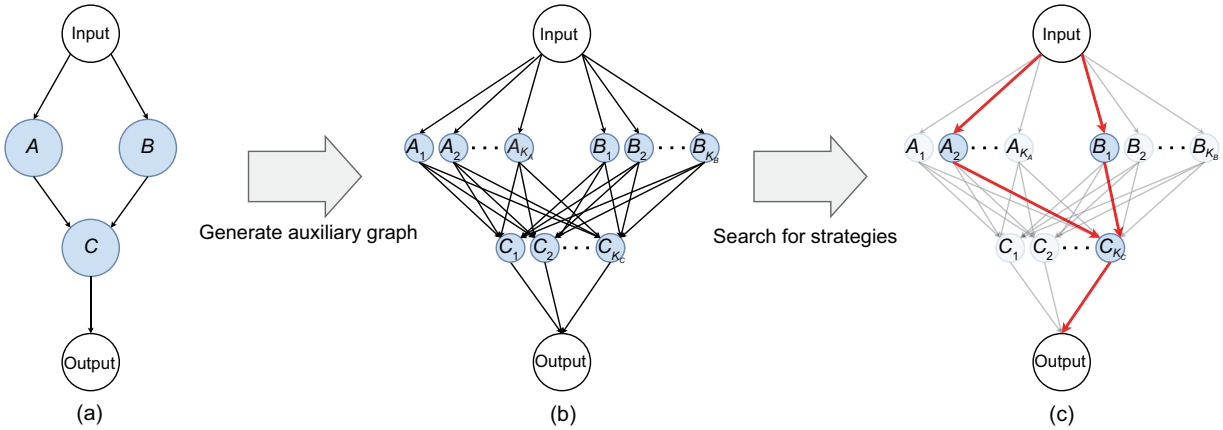

Fig. 2 Illustration of an auxiliary computation graph: (a) original computation graph; (b) auxiliary computation graph; (c) strategy decision example (References to color refer to the online version of this figure)

as shown in Fig. 2b. Auxiliary node $A_i \in V_{\mathrm{A}}$ represents the i^{th} strategy of node $A \in V$. As seen in Fig. 2c, the strategy search problem becomes a path search problem in the auxiliary computation graph. The red lines in Fig. 2c indicate that we choose the second, first, and K_C^{th} strategies for nodes A, B, and $C \in V$, respectively.

3.4 Communication overhead

Considering that communication overhead has become the bottleneck hindering improvement in the performance of parallel training in deep learning, to maximize the throughput of training while generating parallel strategies with minimal memory redundancy, we take the computation of communication

volume produced by parallelism strategies as a secondary objective during strategy search.

In executing operator-level parallelism, the principal genesis of training communication overhead stems from the enactment of the parallel operator in conjunction with the resharding procedure. Contemporary automatic parallelism methodologies typically compute the aforementioned two categories of communication volume individually, and the approach employed to measure communication volume is comparatively well-established.

The communication overhead brought about by the execution of parallel strategies can correspond to each selected node on the auxiliary graph. The specific value of the communication volume can be calculated based on the shape of the tensor, the partitioning method, and the type of operator execution. It is denoted as C_v; here v is a node in the auxiliary graph.

Since our method uses a combination of different parallelism strategies, it may result in a mismatch between the tensor distribution of the output of the previous operator and the distribution required by the input of the next operator. The extra communication volume brought about by resharding is denoted as $C(i,j)$, where (i,j) is an edge in the auxiliary graph.

3.5 Strategy search algorithm

Using the methods in Sections 3.2 and 3.3, the computational operations during model training, candidate parallelism strategies, and the memory overhead incurred with the use of different strategies have been represented by an auxiliary computation graph with costs. To generate parallelism strategies more efficiently, we formalize the path search problem as an ILP problem:

$$\min \quad \sum_{(i,j)\in E_{\mathrm{A}}} B_{ij}M_{ij} + \alpha C(V_{\mathrm{A}}, E_{\mathrm{A}}) \tag{5}$$

$$\text{s.t.} \quad \sum_{v_{\mathrm{A}} \in V_{\mathrm{A}}} X_{v_{\mathrm{A}}} = 1, \tag{6}$$

$$\sum_{(i,v_{\mathrm{A}})\in E_{\mathrm{A}}} B_{iv_{\mathrm{A}}} = X_{v_{\mathrm{A}}} \cdot \mathrm{in\_degree}(v), \tag{7}$$

$$\sum_{(v_{\mathrm{A}},k)\in E_{\mathrm{A}}} B_{v_{\mathrm{A}}k} = X_{v_{\mathrm{A}}} \cdot \mathrm{out\_degree}(v), \tag{8}$$

$$B_{ij}, X_{v_{\mathrm{A}}} \in \{0,1\},\ \forall (i,j) \in E_{\mathrm{A}},\ \forall v_{\mathrm{A}} \in V_{\mathrm{A}},$$

where $X_{v_{\mathrm{A}}}$ and B_{ij} are to-be-solved Boolean values indicating the selection of vertex $v_{\mathrm{A}} \in V_{\mathrm{A}}$ and edge $(i,j) \in E_{\mathrm{A}}$. M_{ij} is the memory overhead of edge $(i,j) \in E_{\mathrm{A}}$. $C(V_{\mathrm{A}}, E_{\mathrm{A}})$ is the total communication volume; it may be calculated as

$$C(V_{\mathrm{A}}, E_{\mathrm{A}}) = \sum_{v_{\mathrm{A}}\in V_{\mathrm{A}}} X_{v_{\mathrm{A}}} C_{v_{\mathrm{A}}} + \sum_{(i,j)\in E_{\mathrm{A}}} B_{ij}C_{ij}.$$

α is a hyperparameter that controls the proportion of communication volume to the objective. The larger the value of α, the higher the proportion of communication attributed to the optimization objective. Constraint (6) informs the solver that we select only one strategy for all $v \in V$. Constraints (7) and (8) limit any $v_{\mathrm{A}} \in V_{\mathrm{A}}$ to have the same indegree and outdegree as their original vertex $v \in V$. To avoid selecting multiple strategies for $v \in V$, we set the indegree and outdegree of $v_{\mathrm{A}} \in V_{\mathrm{A}}$ to zero if it is not selected.

Formalizing the problem into ILP formulation endows the generated parallelism strategies with the capability to fully adhere to the current hardware constraints, thereby ensuring the non-occurrence of issues like memory overflow or communication conflicts. Moreover, unlike the case with other solution methods, solving the ILP problem guarantees to obtain the optimal solution, which helps us find the parallelism strategy with minimal memory overhead given the memory constraints.

Additionally, existing high-performance ILP solvers can efficiently accelerate the process of solving integer programming problems, potentially saving time compared to using alternative methods for strategy search.

4 Experiments

Our experimental trials were conducted using the PyTorch (version 1.11) framework on a server configuration consisting of eight A100 graphic processing units (GPUs), two AMD EPYC 7282 16-core processors, a memory capacity of 512 GB, and the Ubuntu 20.04 operating system. CUDA (version 11.1) was employed, and the Cardinal Optimizer (COPT) was used as an external integer programming solver.

Our approach primarily uses a combination of DP and MP. These parallelism strategies guarantee the accuracy of gradient calculations and

parameter modifications. Consequently, parallel training via our strategic framework is mathematically congruent with conventional training methodologies. In congruence with preceding research on automated parallelism, we abstained from contrasting model efficiency, directing our focus solely towards memory expenditure and throughput.

4.1 Megatron-LM, Alpa, and Colossal-Auto

We employed the parallelism strategy provided by Megatron-LM (Shoeybi et al., 2019; Narayanan et al., 2021) as our baseline approach. Megatron-LM is a SOTA system engineered for training Transformer-based (Vaswani et al., 2017) language models on GPUs. It combines data, pipeline, and operator parallelism for enhanced performance. The effectiveness of these techniques can be controlled using three integer parameters that dictate the degree of parallelism allocated to each technique. The best configuration is determined by conducting a grid search and following the guidelines outlined in Zheng et al. (2022) and Liu YL et al. (2023). This approach ensures sufficient utilization of the parallelism techniques, resulting in superior training outcomes for homogeneous Transformer-based language models.

We also provided a comparison with another SOTA method, Alpa. The basic situation of Alpa has been described above. Alpa also uses ILP solvers to search for the intra-operator parallelism strategy; the difference, however, is that Alpa's optimization object is to reduce communication as much as possible.

Finally, we compared our method with Colossal-Auto (Liu YL et al., 2023) using the Colossal-AI (Li et al., 2023) framework, a novel deep learning framework implemented using Python.

4.2 Evaluation

Our methodology preserves the integrity of the synchronous gradient descent algorithm, ensuring that the convergence of the model remains unaffected. As a result, our evaluation primarily focuses on quantifying the training memory overhead.

The experiments were conducted using a single-layer Transformer model (Vaswani et al., 2017), specifically an encoder layer of the Transformer, at four different scales: BERT-Large (Devlin et al., 2019) and 1.7B, 3.6B, and 7.5B BERT-like models similar to the model used in Megatron-LM's scaling experiment (Narayanan et al., 2021). The specific hyperparameters used for each scale are presented in Table 1.

Table 1 Parameters of the model in the experiments

Model name	Batch size	Hidden layer size	Sequence size	Head number
BERT-Large	2	1024	512	16
1.7B	16	2304	2048	24
3.6B	16	3072	2048	24
7.5B	16	4096	2048	32

These hyperparameters were chosen to evaluate the performance and effectiveness of our method with varying model sizes. α was set to 1.

By examining the results obtained from these experiments, insights can be gained into the scalability and capability of the system when training Transformer-based language models at different scales.

In our experiments the memory overhead during the runtime of single-layer Transformer models at different batch, hidden layer, and sequence sizes (BERT-Large, 1.7B, 3.6B, and 7.5B) was evaluated; the parameters are shown in Table 1. To assess this metric, we used the built-in CUDA interface routines to record the memory overhead during runtime.

Using a dedicated ILP solver, the parallel strategy generation in the environments of two, four, and eight GPUs can be completed in 0.02, 0.05, and 0.23 s, respectively.

Table 2, Fig. 3, and Fig. 4 illustrate our comparative analysis of the parallelism strategy generated by our approach versus Megatron-LM's, Alpa's, and Colossal-Auto's on two, four, and eight devices. Megatron-LM's parallelism strategy served as the baseline for our evaluation.

The results showed that our strategy consistently reduced memory overhead compared to other methods, while the throughput during model training was similar to those of the three other methods.

Specifically, when conducting parallel training experiments on two devices, the memory overhead remained lower than that of the Megatron-LM strategy, with memory overhead in the worst- and best-case scenario being 0.86 times and 0.54 times that of the Megatron-LM strategy, respectively. When models were trained on four devices, the memory

Table 2 The memory overhead of our method during runtime compared to those of Megatron-LM, Alpa, and Colossal-Auto

Model	Device number	Memory overhead (MB)				Relative*		
		MLM	Alpa	C-Auto	Ours	MLM	Alpa	C-Auto
BERT-Large	2	216.06	148.81	331.63	185.05	0.86	1.24	0.56
1.7B	2	25 948.15	24 976.17	15 708.42	18 825.93	0.73	0.75	1.20
3.6B	2	39 620.04	37 473.76	16 136.24	27 634.30	0.70	0.74	1.71
7.5B	2	41 473.96	38 345.58	42 265.51	22 369.74	0.54	0.58	0.53
BERT-Large	4	219.22	158.90	331.64	148.70	0.68	0.94	0.45
1.7B	4	31 043.46	27 606.46	15 708.42	10 575.90	0.34	0.38	0.67
3.6B	4	41 419.42	35 070.54	16 136.24	13 568.87	0.33	0.39	0.84
7.5B	4	43 822.80	35 633.63	41 862.86	18 219.26	0.42	0.51	0.44
BERT-Large	8	199.81	177.01	335.73	128.05	0.64	0.72	0.38
1.7B	8	30 067.89	47 090.08	21 731.21	21 450.06	0.71	0.46	0.99
3.6B	8	40 104.50	57 417.39	24 169.72	26 477.73	0.66	0.46	1.10
7.5B	8	42 044.41	54 803.73	41 877.61	26 593.83	0.63	0.49	0.64

* Ratio of memory overhead of our method to that of other methods. MLM: Megatron-LM; C-Auto: Colossal-Auto

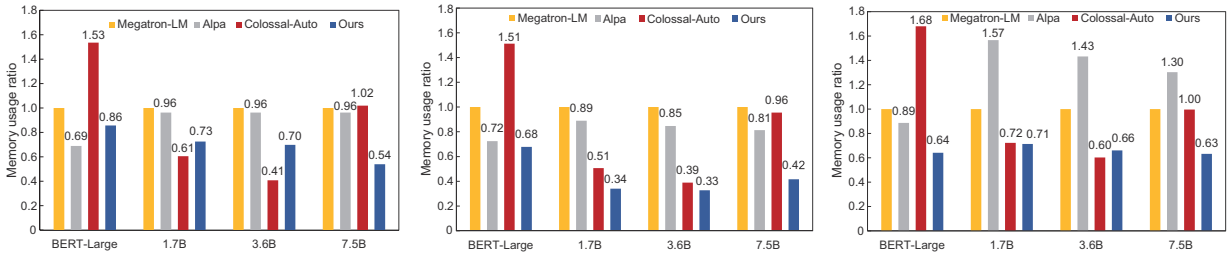

Fig. 3 Memory overhead comparison among two (a), four (b), and eight (c) devices

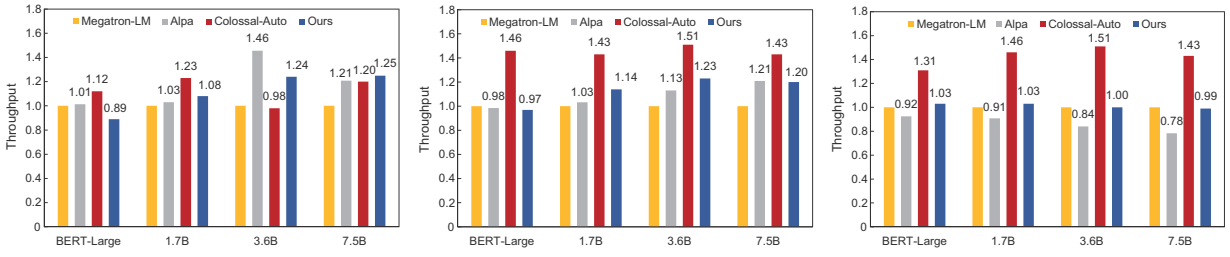

Fig. 4 Throughput comparison among two (a), four (b), and eight (c) devices

overhead was consistently reduced compared to the original strategy of Megatron-LM, with memory overhead in the worst- and best-case scenario being 0.68 times and 0.33 times that of the Megatron-LM strategy, respectively. When models were trained on eight device, the memory overhead of our method was about 66% that of Megatron-LM.

Compared with Alpa's strategy, our method generally had a smaller memory overhead when training models of different sizes. When models were trained on two devices, the memory overhead of our method can be as low as 0.58 times that of Alpa. When training a 1.7B model with four devices, the memory overhead of our method can be

as low as 0.38 times that of Alpa. When models were trained on eight devices, the memory overhead of our method can be as low as 0.46 times that of Alpa. Only when BERT-Large was trained on two devices, was memory overhead in the training 1.24 times that of Alpa. We speculated that the additional memory overhead was brought about by deep learning frameworks when requesting memory from the system. This was more evident when the model size was not large.

Presumably, owing to the utilization of the Colossal-AI framework, the Colossal-Auto approach manifested a certain performance advantage compared to alternative methods conducted on our self-implemented code. However, its peak memory consumption exhibited some inconsistency across distinct training scenarios. Nonetheless, amidst such circumstances, our method showcased a steady advantage in reducing memory overhead.

Furthermore, we observed that the reduction in redundant memory led to a decrease in the amount of data that needed to be copied between devices. Hence, in applying a parallel strategy that minimized memory redundancy, there was a certain reduction in communication overhead. This is one of the reasons why our method is comparable to SOTA approaches in terms of throughput.

As our approach generates parallel strategies with less memory redundancy and communication volume, the load balance of memory and computing resources is more reasonable. Compared to the parallelism strategies by Megatron-LM and Alpa, our approach significantly reduced memory overhead while achieving similar performance. Moreover, the performance advantage became increasingly pronounced as the model size increased, and maximum memory savings were achieved when four GPUs were used in parallel.

5 Conclusions

To conclude, we present an algorithm that can generate minimal memory redundancy parallelism strategies. The algorithm was designed based on minimal memory redundancy, aiming to address the challenges of training huge models on a single computing node. The effectiveness and efficiency of the proposed algorithm have been verified through extensive experiments and analysis.

The key findings and contributions of this research can be summarized as follows:

1. A novel algorithm was proposed that optimizes parallelism strategies, reducing memory redundancy and improving memory efficiency in large-scale deep learning models by leveraging the extraction of the model's computational graph and employing auxiliary graphs.

2. By conducting experiments on neural networks with various sizes within the Transformer class, the proposed parallel strategy, which aims to minimize memory redundancy as detailed in the paper, achieved up to 67% reduction in memory overhead. In contrast, the gap between the throughput of the proposed strategy and those of SOTA methods is not large.

There are areas, however, that require further investigation and improvement:

1. While the method's effectiveness has been demonstrated on small-scale machines, the absence of experimental results on large-scale devices limits this study.

2. Given that the choice of traditional memory optimization methodologies in strategy formulation can significantly enhance the utilization of available memory resources, insufficient integration with memory optimization techniques is another limitation characterizing the research methodology adopted in the present study.

To summarize, this research advances automatic parallelism by introducing a novel algorithm that demonstrates superior performance. It explores new possibilities and provides insights for researchers and practitioners training large models on resource-constrained devices. The present research establishes a robust foundation for future investigations and encourages further exploration into the parallelization of training deep learning models.

Contributors

Yanqi SHI designed the research. Yanqi SHI and Peng LIANG processed the data. Yanqi SHI drafted the paper. Hao ZHENG, Linbo QIAO, and Dongsheng LI helped organize the paper. Yanqi SHI and Peng LIANG revised and finalized the paper.

Conflict of interest

Dongsheng LI is a corresponding expert of *Frontiers of Information Technology & Electronic Engineering*, and he

was not involved with the peer review process of this paper. All the authors declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

References

Brown TB, Mann B, Ryder N, et al., 2020. Language models are few-shot learners. Proc 34th Int Conf on Neural Information Processing Systems, Article 159.

Cai ZK, Yan X, Ma KH, et al., 2022. TensorOpt: exploring the tradeoffs in distributed DNN training with auto-parallelism. *IEEE Trans Parall Distrib Syst*, 33(8):1967-1981. https://doi.org/10.1109/TPDS.2021.3132413

Chowdhery A, Narang S, Devlin J, et al., 2022. PaLM: scaling language modeling with pathways. https://arxiv.org/abs/2204.02311

Dan YH, Lei ZK, Gu YY, et al., 2023. EduChat: a large-scale language model-based chatbot system for intelligent education. https://arxiv.org/abs/2308.02773

Devlin J, Chang MW, Lee K, et al., 2019. BERT: pre-training of deep bidirectional Transformers for language understanding. Proc Conf of the 9th American Chapter of the Association for Computational Linguistics: Human Language Technologies, p.4171-4186. https://doi.org/10.18653/v1/N19-1423

Guan L, Sun T, Qiao LB, et al., 2020. An efficient parallel and distributed solution to nonconvex penalized linear SVMs. *Front Inform Technol Electron Eng*, 21(4):587-603. https://doi.org/10.1631/FITEE.1800566

Harlap A, Narayanan D, Phanishayee A, et al., 2018. PipeDream: fast and efficient pipeline parallel DNN training. https://arxiv.org/abs/1806.03377

He XB, Chen X, Guo H, et al., 2023. Scalability and efficiency challenges for the exascale supercomputing system: practice of a parallel supporting environment on the Sunway exascale prototype system. *Front Inform Technol Electron Eng*, 24(1):41-58. https://doi.org/10.1631/FITEE.2200412

Huang YP, Cheng YL, Bapna A, et al., 2019. GPipe: efficient training of giant neural networks using pipeline parallelism. Proc 33rd Int Conf on Neural Information Processing Systems, Article 10.

Jia ZH, Lin SN, Qi CR, et al., 2018. Exploring hidden dimensions in accelerating convolutional neural networks. Proc 35th Int Conf on Machine Learning, p.2274-2283.

Krizhevsky A, Sutskever I, Hinton GE, 2012. ImageNet classification with deep convolutional neural networks. Proc 26th Annual Conf on Neural Information Processing Systems, p.1106-1114.

Lan Q, Qiao LB, Wang YJ, 2018. Stochastic extra-gradient based alternating direction methods for graph-guided regularized minimization. *Front Inform Technol Electron Eng*, 19(6):755-762. https://doi.org/10.1631/FITEE.1601771

Li SG, Liu HX, Bian ZD, et al., 2023. Colossal-AI: a unified deep learning system for large-scale parallel training. Proc 52nd Int Conf on Parallel Processing, p.766-775. https://doi.org/10.1145/3605573.3605613

Liu YL, Li SG, Fang JR, et al., 2023. Colossal-Auto: unified automation of parallelization and activation checkpoint for large-scale models. https://arxiv.org/abs/2302.02599

Liu ZM, Cheng SG, Zhou HT, et al., 2023. Hanayo: harnessing wave-like pipeline parallelism for enhanced large model training efficiency. Proc Int Conf for High Performance Computing, Networking, Article 56. https://doi.org/10.1145/3581784.3607073

Mo ZY, 2018. Extreme-scale parallel computing: bottlenecks and strategies. *Front Inform Technol Electron Eng*, 19(10):1251-1260. https://doi.org/10.1631/FITEE.1800421

Narayanan D, Shoeybi M, Casper J, et al., 2021. Efficient large-scale language model training on GPU clusters using Megatron-LM. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 58. https://doi.org/10.1145/3458817.3476209

Naumov M, Mudigere D, Shi HJM, et al., 2019. Deep learning recommendation model for personalization and recommendation systems. https://arxiv.org/abs/1906.00091

Rajbhandari S, Rasley J, Ruwase O, et al., 2020. ZeRO: memory optimizations toward training trillion parameter models. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 20.

Shazeer N, Cheng YL, Parmar N, et al., 2018. Mesh-TensorFlow: deep learning for supercomputers. Proc 32nd Int Conf on Neural Information Processing Systems, p.10435-10444.

Shoeybi M, Patwary M, Puri R, et al., 2019. Megatron-LM: training multi-billion parameter language models using model parallelism. https://arxiv.org/abs/1909.08053

Vaswani A, Shazeer N, Parmar N, et al., 2017. Attention is all you need. Proc 31st Int Conf on Neural Information Processing Systems, p.6000-6010.

Wang MJ, Huang CC, Li JY, 2019. Supporting very large models using automatic dataflow graph partitioning. Proc 14th EuroSys Conf, Article 26. https://doi.org/10.1145/3302424.3303953

Zheng LM, Li ZH, Zhang H, et al., 2022. Alpa: automating inter- and intra-operator parallelism for distributed deep learning. Proc 16th USENIX Symp on Operating Systems Design and Implementation, p.559-578.

Zhuang YT, Wu F, Chen C, et al., 2017. Challenges and opportunities: from big data to knowledge in AI 2.0. *Front Inform Technol Electron Eng*, 18(1):3-14. https://doi.org/10.1631/FITEE.1601883