International Journal of Parallel Programming (2023) 51:109–127
https://doi.org/10.1007/s10766-022-00741-6

SMSG: Profiling-Free Parallelism Modeling for Distributed Training of DNN

Haoran Wang[1,2] · **Thibaut Tachon**[1] · **Chong Li**[1] · **Sophie Robert**[2] · **Sébastien Limet**[2]

Received: 10 September 2022 / Accepted: 21 November 2022 / Published online: 12 December 2022

Abstract
The increasing size of deep neural networks (DNNs) raises a high demand for distributed training. An expert could find good hybrid parallelism strategies, but designing suitable strategies is time and labor-consuming. Therefore, automating parallelism strategy generation is crucial and desirable for DNN designers. Some automatic searching approaches have recently been studied to free the experts from the heavy parallel strategy conception. However, these approaches all rely on a numerical cost model, which requires heavy profiling results that lack portability. These profiling-based approaches cannot lighten the strategy generation work due to the non-reusable profiling value. Our intuition is that there is no need to estimate the actual execution time of the distributed training but to compare the relative cost of different strategies. We propose SMSG (Symbolic Modeling for Strategy Generation), which analyses the cost based on the communication and computation semantics. With SMSG, the parallel cost analyses are decoupled from hardware characteristics. SMSG defines cost functions for each kind of operator to quantitatively evaluate the amount of data for computation and communication, which eliminates the heavy profiling tasks. Besides, SMSG introduces how to apply functional transformation by using the Third Homomorphism theorem to control the high searching complexity. Our experiments show that SMSG can find good hybrid parallelism strategies to generate an efficient training performance similar to the state of the art. Moreover, SMSG covers a wide variety of DNN models with good scalability. SMSG provides good portability when changing training configurations that a profiling-based approach cannot.

Keywords Distributed training · Deep network networks · Parallel modeling · Symbolic cost model · Functional transformation · Performance analysis

✉ Chong Li
ch.l@huawei.com

Extended author information available on the last page of the article

1 Introduction

The size of DNN models has been scaling up dramatically in recent years. Many gigantic models achieved remarkable accuracy in domains like natural language processing (NLP) [1], computer vision [2], recommendation systems [3], etc. The execution of such a gigantic model requires tremendous computational and memory resources [1]. To be able to train gigantic models in a distributed way, different *parallelism*, like *data parallelism* [4], *operator parallelism* [5] and *pipeline parallelism* [6], have been proposed. Different parallelisms were initially designed for different basic model structures. Mixing parallelisms [7] could provide a more general approach for modern complex structures and a more efficient solution to train a DNN model than applying single parallelism.

Efficient mixing parallelism with *hybrid strategies* could be designed by a human expert [7]. However, it is time and labor-consuming to find a well-behavior strategy for each given DNN model. Experts usually spend months to decide an efficient parallel strategy for a new DNN model. Therefore, the DNN model researchers crucially desired to generate efficient strategy generation systematically.

Several strategy searching approaches have been proposed, like [8–13], and could provide effective hybrid strategies for their targeted DNN models. However, these approaches exhibit poor generality for new-coming DNN models because they are all based on a profiling-based cost model. The number of operators in the modern DNN model is more than 1000 [1], and the number of possible partition dimensions increases exponentially w.r.t the number of devices [13]. Therefore, the profiling work takes a lot of time [8]. Besides, when the shapes or the substructures of a DNN change or deal with a different hardware environment, the profiling tasks must be prepared again. All these time-costly tasks result in the poor generality of these approaches.

To avoid the limitations of the profiling-based approaches, we propose SMSG (Symbolic Modeling for Strategy Generation), a profiling-free approach to expand the generality of automatic strategy searching in this paper. Our main contributions are the followings. (i) Based on the computation and communication semantics, we build a symbolic cost model to quantitatively evaluate the relative cost instead of predicting the execution time of different parallelism strategies. This model separates the cost into two parts: the hardware parameters and the computation and communication data quantity. We decouple the hardware characteristics from the parallel execution details with this model. Therefore, we only need to profile the hardware parameters rather than the execution time of each operator under uncountable configurations.

(ii) Inspired by the Homomorphism theory, we derive the strategy generation from the symbolic cost model and reduce the exponential complexity caused by the redistribution cost between operators. Therefore, the searching time is restricted to an acceptable range, contributing to our approach's generality.

To validate the generality of SMSG, we conducted the following experiments. We choose expert-designed strategies for targeted DNN models as the best performance baseline, and we choose TensorOpt [8], the method of operator-level strategy searching, as the automatic searching approach baseline. First, we compare the strategy quality of ResNet [2] on different hardware architectures. With careful profiling, SMSG and TensorOpt can find the strategy with similar end-to-end performance as the expert-designed strategy. However, the experiments also show that profiling-based approaches like TensorOpt find strategies with sub-optimal performance without carefully profiling results. Moreover, we tested SMSG on varieties of DNNs including ResNet50/101/152, Wide & Deep [3], Bert [14], GPT-3 [1], and T5 [15]. The similar performance of the expert-designed strategy validates the generality of SMSG.

2 Modeling the Cost of Distributed Training

2.1 Distributed Strategies Searching

DNN training aims to find proper parameters to predict results from new inputs. A training iteratively executes *Forward Propagation* (FPG) and *Backward Propagation* (BPG). An FPG computes a batch of inputs using current parameters to predict results; a BPG starts from the derivative of the last operator back to the derivative of the first operator, computes the gradients, and updates the parameters according to the Loss. The input data of a DNN model is always processed in a batch, i.e. dozens or hundreds of data items are executed simultaneously in an iteration step.

A DNN model could be represented as a *computational graph*, which is a directed acyclic graph (DAG), where vertices are the operators and edges connected to the operators signify the dataflow direction in the graph. *Operators* are DNN-level mathematical computations such as Matrix Multiplication (MatMul), Convolution (Conv), Element-wise operator (Ele-W, e.g. Add/Mul), etc. The data of DNN training is *tensor* structured as a multi-dimension array. The tensor dimension, which organizes data (e.g. an image, a sentence, or a vector of features) in a batch, is called *batch dimension*.

DNN training can be parallelized differently. In the paper, we take the following most used parallelisms as examples:

- *Data parallelism* partitions a batch into several mini-batches onto multi-devices while each device owns the same entire DNN models. Data parallelism is efficient for the DNNs with a small parameter size because there is no extra communication cost except the parameter updating at the end of each *step* (training iteration). The cost of parameter updating corresponds to the q_p in Sect. 2.3. However, gigantic state-of-the-art models can have trillions of parameters [1] where data parallelism suffers from high parameter updating costs.
- *Operator parallelism* splits the other dimension except for the batch dimension of the operator, which causes extra intra-communication, which corresponds to the q_c. The number of possible dimensions of operator parallelism for a single operator is enormous, so the analysis of the intra-communication cost is compli-

cated. Besides, different dimensions partitioned for connected operator generates data redistribution, corresponding to the q_r.

Figure 1 shows a computational example composed by two MatMul. The first MatMul is partitioned along its batch dimension according to Data Parallelism. In contrast, the second one is partitioned along the vertical dimension of its second input tensor, which is one kind of model parallelism. It can be conducted from this graph that the output of the first MatMul is the same tensor as the first input of the second MatMul. Data redistribution is generated because of the mismatch of the data partitioning.

Deep Learning (DL) frameworks [16–18] can accelerate the development of DNN models by systematically generating execution code from a computational graph. Advanced frameworks can even automatically insert communication for distributed training from parallelism strategies. A parallelism strategy denotes along which dimensions the tensors of an operator are partitioned, as shown in Fig. 3.

Deciding on a suitable hybrid parallelism strategy is difficult for the following reasons: (i) Different types of operators may prefer different parallelism strategies. For example, it is not suitable to partition the kernel tensor of a Conv op because the shape of the kernel is usually $3 * 3$ or $5 * 5$. A common practice for Conv op is to partition it along its *batch dimension* (data parallelism) or *channel dimension* (one of the possible operator parallelism strategies) [2]. (ii) Besides, the shape of the operators also affects the strategy choices [8]. A MatMul op is more suitable for data parallelism when its parameter tensor is small and should be configured as operator parallelism when its parameter tensor is very large. (iii) The operators are not executed separately. They are all connected in the computational graph via the edges [19]. The output tensor of an operator is also the input tensor of its successive operator. If the parallel strategies differ for these two operators, the tensor needs to be redistributed in the cluster, which generates additional cost and should also be considered.

Due to these difficulties, it is time and labor-consuming to design efficient parallelism strategies according to researchers' expertise and experiments. Moreover, expert-designed hybrid parallelism usually returns bad performances when migrated to new DNN models. Therefore, systematic strategy searching has become a crucial research topic.

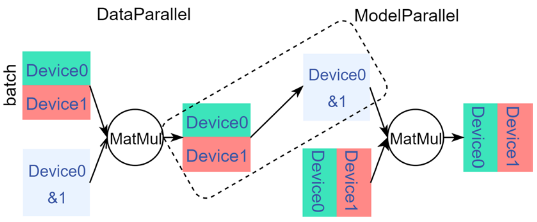


Fig. 1 Simplest computational graph with DP/MP examples

2.2 Profiling-Based Modeling

Many parallelism strategy generators have been proposed recently: OptCNN [11], ToFu [13], TensorOpt [8], etc. OptCNN supports automatic operator-level (data parallelism and operator parallelism) parallel strategy generation for each layer (a group of operators to perform one object) in a neural network with a numerical cost model and a dynamic programming algorithm. ToFu and TensorOpt extend OptCNN to generate strategies for each operator. The results of these works show the feasibility and the potential of automatic parallelism strategy generation.

The principal idea of distributed cost modeling is to describe the time consumption of distributed training. Related works [8, 12] try to build a general cost model to predict the actual execution time. They compare the predicted time of different parallelism strategies of the given DNN on a specific hardware platform and choose the strategy with the lowest predicted time. Let us take OptCNN [11] as an example since other solutions are extensions of it.

A computational graph $G = (V, E)$ is defined by the vertices set V such that $v_i \in V$ is an operator and the edges set E such that e_{ij} represents the dataflow direction between v_i and v_j. The hybrid parallelism strategy $\mathcal{S}$ of the Graph G is defined as the set of operator-level strategies. Let Op an operator, if n denotes its number of input tensors, $S_{Op} = \{s_i, 1 \leq i \leq n\}$ where $s_i = [x_{d1}, x_{d2}, \ldots]$ is a set of integer to define how to partition each tensor of d_i dimension.

The predicted global execution time T is defined as follows:

$$T(G, \mathcal{S}, D) = \sum_{v_i \in V} \left(t_e(v_i, S_{v_i}, D) + t_p(v_i, S_{v_i}, D) \right) + \sum_{e_{ij} \in E} t_r(e_{ij}, S_{v_i}, S_{v_j}, D) \tag{1}$$

- $t_e(v_i, S_{v_i}, D)$ is the time to execute the operator v_i under its strategy. It includes the time for local computation and the communication time caused by the partition strategy. t_e is an average time measured from several executions of the operator v_i profiled with its strategy under hardware environment D.
- t_p denotes the parameter updating time at the end of each iteration (after backward propagation). Parameter updating is usually implemented by all-reduce, requiring the same profiling method as t_e.
- t_r, the redistribution cost between two connected operators v_i, v_j, is usually estimated by the multiplication of the data size and the known communication bandwidth.

This modeling methodology describes the total cost generated from the distributed training. Experiments show that optimal hybrid strategies are found in this way. However, OptCNN can only search layer-level strategies; Tofu/TensorOpt searches operator strategies only for small DNNs. The main limitation of these modeling methods is that they require inevitable preparation work to profile the operator under different configurations. New types of DNNs come out today, which also carries out new types of operator and partition dimensions. This modeling methodology becomes unrealistic for the following reasons: (i) A computational graph may contain thousands of operators. Even the operators could be classed into dozens of types, but profiling works are required when the shape of the operator changes. (ii) For one operator, all the dimensions of its tensors are splittable, thus causing a polynomial search space, making the profiling work heavy. (iii) Profiling results of an operator are heavily dependent on the hardware. Re-profiling is needed when the type, number of the accelerator, or cluster connections change.

Due to the heavy preparation work, these profiling-based modeling methods lack portability and generality. As a result, a more general method for new DNNs with different environments is demanded.

2.3 Profiling-Free Modeling

The profiling task is inevitable for these approaches because, from an AI expert's point of view, the operator is the basic unit of performance modeling that cannot be split. As a result, $t_e(v_i, S_{v_i}, D), t_p(v_i, S_{v_i}, D)$ are unsplittable and the values could only be estimated through profiling.

Our insight is to model the cost through a parallel computing model like the bridging model of Valiant [20]. We model the computation and communication costs caused by parallel strategy instead of the execution time of operators and edges. With such a modeling methodology, the parallel execution analysis can be decoupled with the hardware environment, which is critical in avoiding the heavy profiling task. Besides, we do not need to compare the real predicted time to evaluate the performance of hybrid parallelism strategies $\mathcal{S}$. The relative value could compare the performance.

Based on what has been presented above, we propose the following symbolic cost model as a metric to compare the performance of two strategies $\mathcal{S}$ on a computational graph G:

$$\begin{aligned} C(G,\mathcal{S}) = &\sum_{v_i \in V} \left(w \times q_x(v_i, S_{v_i}) + g \times (q_c(v_i, S_{v_i}) + q_p(v_i, S_{v_i}))\right) \\ &+ \sum_{e_{ij} \in E} g \times q_r(e_{ij}, S_{v_i}, S_{v_j}) \end{aligned} \tag{2}$$

In this model, the variables could be classified into two categories as follows:

- *Profiled Hardware Parameters: w*, g denote the accelerator's real-time computation and communication capacities. The calibrated FLOPS and bandwidth usually cannot be fully used, and these two values are obtained through profiling the hardware environment.
- *Data Quantity Function Without Profiling:* $q_x(v_i, S_{v_i}), q_c(v_i, S_{v_i})$ and $q_p(v_i, S_{v_i})$ are respectively the computation quantity and intra-communication caused by parallelism and parameter updating communication for an operator v_i under the strategy S_{v_i}. $q_r(e, S_{v_i}, S_{v_j})$ denotes the quantity of data redistribution caused by conflicting strategy of two connected operators.

The hardware features and parallel costs are separated with this symbolic modeling method. The computation and communication capacities w, g can be estimated by profiling the hardware. The quantities q_x, q_c, q_p, and q_r can be symbolically analyzed without profiling.

Various new DNNs have come out in recent years, but they are all based on 20+ computational operators (e.g., MatMul, Conv, Etc.) and 100+ element-wise operators. These operators can be classed into 20+ types. We analyze the semantics of these 20+ operator types and build the symbolic cost model for each operator type under different strategies. As for the possibilities of partition dimension, even though there are many dimensions for each operator, only a few are practical for parallel training after semantic studying. For example, Conv operator has seven possible partition dimensions including *batch, input_channel, input_height, input_weight, output_channel, kernel_height, kernel_weight*, but only *batch* and two *channel*s are practical because the other dimensions generates super large communication cost. For partitioning kernel dimension, the communication is hard to be implemented because the shape is usually small, like 3×3. As a result, we define the cost model for the 20+ types of operators with limited possible partition dimensions, which could be generally applied to any DNN.

Our symbolic model can help find the optimal strategy because it is essentially an abstraction of the profiling-based model. In fact, the two models describe the same parallel cost in different scope, $t_e(v_i, S_{v_i}, D)$ in profiling-based model is actually the summation of $w \times q_x(v_i, S_{v_i}) + g \times q_c(v_i, S_{v_i})$ in our symbolic model. $t_p(v_i, S_{v_i}, D)$ equals to $g \times q_c(v_i, S_{v_i})$ and $t_r(e, S_{v_i}, S_{v_j}, D) = g \times q_c(e, S_{v_i}, S_{v_j})$. Thanks to symbolic modeling, heavy profiling works are avoided. We can quantitatively evaluate the cost of computation, intra-communication, and redistribution with the defined symbolic cost model. We only profile the communication and computation capacity of the hardware. However, because we do not use the real profiled value, the redistribution cost of the symbolic cost invokes high search complexity.

Section 3 proposes a method to reduce the strategy generation complexity. Although this complexity reduction comes at the cost of decreasing the quality of the strategy found, we present a scheme to mitigate the decrease in quality. This scheme can be seen as a heuristic-based greedy approach in which we prove that the heuristic-based vertex reordering preserves the graph cost thanks to its definition of homomorphism.

3 Functional Transformation and Reduction

The cost of a vertex depends on its own strategy (computation amount q_x, intra-communication q_c and parameter updating q_p) but also of the strategy of its neighbors (redistribution q_r). Choosing a strategy for a vertex will influence its neighbors that will influence their neighbours until the whole graph recursively. Finding the optimal distribution strategy for real-life deep neural networks is impossible in a reasonable amount of time.

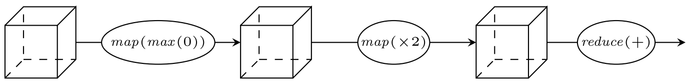


Fig. 2 Minimal neural network example

Example 1 We illustrate the complexity of the problem through the following minimal example. Consider a toy neural network represented on Fig. 2 of 3 operators on 3-dimensional tensors that we aim to distribute over four devices. Operators may be, for example, a RELU followed by an element-wise twofold increase followed by the sum of all elements. Possible distributions of a 3D tensor over four devices are illustrated on Fig. 3, and the number is 6. As there are three operators, the total number of possibilities for this tiny graph is $6^3 = 216$.

We plan to cut the complexity of this problem by making decisions based on local contexts and not questioning them afterward, which is a greedy method. To mitigate the difference between our cost and the optimal one, we treat vertices by order of decreasing importance. This way, the most critical vertices will have a strategy that benefits them the most. Although the less important ones may not benefit from the best strategy, the global impact on performances will be smaller. In order to justify this reordering, we formulate our algorithm as a homomorphism that presents the benefit of being computable in any order.

To do so, we will first introduce how the cost may be computed from a homomorphism of a vertex list in Sect. 3.3. However, the data-flow representation of the computation is not a list but a directed acyclic graph (DAG). Morihata [21] showed that the homomorphism theory (especially its third theorem) might be extended to trees. The only difference between a tree and a DAG lies in the number of parents (outputs) that may be more than one in a DAG which leads to several possible paths from one vertex to another. To remove the existence of these different paths, we propose to consider the spanning tree of the DAG that would select only one of those paths for each case.

3.1 Notations

S is the set of all strategies of all vertices in G. We note s_i the strategy of vertex v_i in S. The cost of the computational graph was given by Eq. 3.

$$\begin{aligned} cost_{op}(v_i, s_i) &= w \times q_x(v_i, s_i) + g \times \left(q_c(v_i, s_i) + q_p(v_i, s_i)\right) \\ cost_{rdst}(v_i, v_j, s_i, s_j) &= g \times q_r(e_{ij}, s_i, s_j) \end{aligned} \tag{3}$$

This way, Eq. 3 can be rewritten

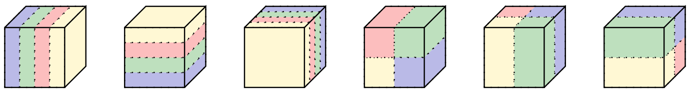


Fig. 3 Possible distribution of a 3D tensor over 4 devices

$$C(G,S) = \sum_{v_i \in V} cost_{op}(v_i, s_i) + \sum_{(v_i,v_j)\in E} cost_{rdst}(v_i, v_j, s_i, s_j) \tag{4}$$

Remark that if $s_j \notin S$ (because v_j has not been seen yet) then

$cost_{rdst}(v_i, v_j, s_i, s_j) = 0$. Assume function $strt(v)$ gives the set of possible strategies for a given vertex v. We note $S_{i<|V|}$ the strategies of the first i vertices visited. The strategy generation consists of taking for each vertex the strategy that minimizes its cost. It may be expressed recursively as

$$\begin{aligned} S_0 &= \emptyset \\ S_{i<|V|} &= search(v_i, s_i, G) \\ cost_v(v_i, s_i, S, G) &= cost_{op}(v_i, s_i) + \textstyle\sum_{(v_i,v_j)\in E} cost_{rdst}(v_i, v_j, s_i, s_j) \\ search(v_i, S, G) &= S \cup \{s_i\} \text{ such that } s_i \in strt(v_i) \text{ minimizes } cost_v(v_i, s_i, G, S) \end{aligned} \tag{5}$$

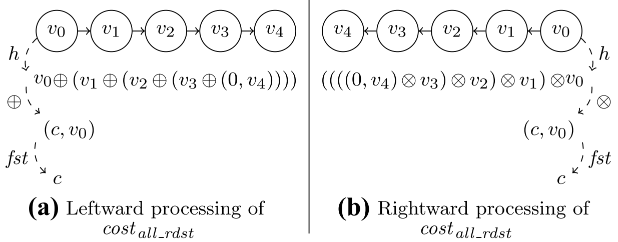
The remaining of this section will express Eq. 5 as a homomorphism. As a preliminary, we introduce the following notations taken from [22]. $hom(\oplus, f, a)(l)$ is a homomorphism that maps function f on each element of l before reducing with the binary operator $\oplus$ whose first application will be with initialization element a. For example

$$hom(\oplus, f, a)([x, y, z]) = a \oplus f(x) \oplus f(y) \oplus f(z)$$

We note $\alpha list$ the polymorphic type: list of elements of any type α. Functions will be noted in Curry notation. For example, $f : A \rightarrow B \rightarrow C$ is the function f that, when applied to an argument of type A will produce a function of type $B \rightarrow C$ that when applied to an argument of type B will produce a value of type C. We use $+\!\!+ : \alpha\ list \rightarrow \alpha\ list \rightarrow \alpha\ list$ as the concatenation operator. The function composition is noted $(f \circ g)(x) = f(g(x))$. To access elements of a pair, we use function first $fst(x, y) = x$ and function second $snd(x, y) = y$.

3.2 Redistribution Cost as a Homomorphism

We define the leftward operation $cost_{all\_rdst}$ to compute all redistribution costs of a linear graph (list). A leftward operation is a recursive operation for which the



v_0
v_1
v_2
v_3
v_4
h
$v_0 \oplus (v_1 \oplus (v_2 \oplus (v_3 \oplus (0, v_4))))$
$\oplus$
(c, v_0)
fst
c
(a) Leftward processing of $cost_{all\_rdst}$
v_4
v_3
v_2
v_1
v_0
h
$((((0, v_4) \otimes v_3) \otimes v_2) \otimes v_1) \otimes v_0$
$\otimes$
(c, v_0)
fst
c
(b) Rightward processing of $cost_{all\_rdst}$

Fig. 4 Leftward and rightward processing of $cost_{all\_rdst}$ over a vertex list

elements are added to the left (input) side (see Fig. 4a). To be a leftward operation, h and $\oplus$ need to be defined such that Eq. 6 holds.

$$\begin{array}{lll} h([v] +\!\!+ x) & = & v \oplus h(x) \\ \text{with} & & \\ \quad h & : & \alpha\ list \to \beta\ list \\ \quad \oplus & : & \alpha \to \beta \to \beta \end{array} \tag{6}$$

We define the operation $cost_{all\_rdst}$ leftward, that respects Eq. 6 below in Eq. 7.

$$\begin{array}{ll} cost_{all\_rdst} & = fst \circ h \\ h([v] +\!\!+ x) & = v \oplus h(x) \\ h\big([v_0]\big) & = (0, v_0) \\ v_i \oplus (c, v_j) & = (c + cost_{rdst}(v_i, v_j, s_i, s_j), v_i) \end{array} \tag{7}$$

Suppose that S, for which $s_i, s_j \in S$, is a global constant fixed for the whole cost computation. As edges direction do not influence the redistribution cost, function $cost_{all\_rdst}$ is also computable rightward (in the output direction, as illustrated in Fig. 4b). A rightward operation must define h and $\otimes$ such that Eq. 8 holds.

$$\begin{array}{lll} h(x +\!\!+ [v]) & = & h(x) \otimes v \\ \text{with} & & \\ \quad h & : & \alpha\ list \to \beta\ list \\ \quad \otimes & : & \beta \to \alpha \to \beta \end{array} \tag{8}$$

We define the operation $cost_{all\_rdst}$ rightward, that respects Eq. 8 below in Eq. 9.

$$\begin{array}{ll} cost_{all\_rdst} & = fst \circ h \\ h(x +\!\!+ [v]) & = h(x) \otimes v \\ h\big([v_0]\big) & = (0, v_0) \\ (c, v_j) \otimes v_i & = (c + cost_{rdst}(v_i, v_j, s_i, s_j), v_i) \end{array} \tag{9}$$

Thus, by the third homomorphism theorem [22], $cost_{all\_rdst}$ is a homomorphism because it is defined both leftward (Eq. 7) and rightward (Eq. 9).

3.3 Cost and Strategy Generation as a Homomorphism

The intra-communication cost is defined directly as the homomorphism

$$cost_{all\_op} = hom(+,\ cost_{op},\ 0)$$

Hence, the whole cost of the linear graph ($cost_l$) may be defined as the addition of the two homomorphism, where l represents the linear graph

$$cost_l(l) = cost_{all\_op}(l) + cost_{all\_rdst}(l)$$

The two may also be computed together as

$$\begin{aligned} cost_l &= fst \circ h \\ h([v] +\!\!+ x) &= v \oplus h(x) \\ h\big([v_0]\big) &= (cost_{op}(v_0, s_0), v) \\ v_i \oplus (c, v_j) &= (c + cost_{rdst}(v_i, v_j, s_i, s_j) + cost_{op}(v_i, s_i), v_i) \end{aligned} \tag{10}$$

and symmetrically for the rightward notation.

$$\begin{aligned} cost_l &= fst \circ h \\ h(x +\!\!+ [v]) &= h(x) \otimes v \\ h\big([v_0]\big) &= (cost_{op}(v_0, s_0), v) \\ (c, v_j) \otimes v_i &= (c + cost_{rdst}(v_i, v_j, s_i, s_j) + cost_{op}(v_i, s_i), v_i) \end{aligned} \tag{11}$$

In this case too, the third homomorphism theorem tells us that $cost_l$ is a homomorphism because it is defined both leftward (Eq. 10) and rightward (Eq. 11). Now that we have shown how the cost may be formulated as a homomorphism, the strategy generation may be, in turn, written leftward

$$\begin{aligned} cost_l &= fst \circ h \\ h([v] +\!\!+ x) &= v \oplus h(x) \\ h\big([v_0]\big) &= \big(search(v_0, \emptyset, [\,]), [v_0]\big) \\ v_i \oplus (S, L) &= \big(search(v_i, S, L), v_i +\!\!+ L\big) \end{aligned} \tag{12}$$

and rightward

$$\begin{aligned} cost_l &= fst \circ h \\ h(x +\!\!+ [v]) &= h(x) \otimes v \\ h\big([v_0]\big) &= \big(search(v_0, \emptyset, [\,]), [v_0]\big) \\ (S, L) \otimes v_i &= \big(search(v_i, S, L), v_i +\!\!+ L\big) \end{aligned} \tag{13}$$

The strategy generation being a homomorphism allows us to state that the vertex list (linear graph) may be processed in any order. This homomorphism may be extended to trees thanks to the work of Morihata [21] thereby enabling us to consider a much more complex graph topology than a mere vertex list.

3.4 Tree Homomorphism

For the sake of understanding, this section will consider homomorphisms of binary trees defined in Eq. 14. This definition may be read as: a tree is either a node with a value V, a left-hand-side tree and a right-hand-side tree, or a leaf.

$$\textbf{data}\ T_{bin} = Node\ (V \times T_{bin} \times T_{bin})\ |\ Leaf \tag{14}$$

In order to define homomorphisms, we need to be able to describe contexts and paths in the tree. These will be represented thanks to a zipper [23]. A path in the binary tree would be a sequence of left (*Left*) or right (*Right*) choices that we represent in Eq. 15. The relation between a tree path and a zipper is illustrated in Fig. 5.

$$\textbf{data}\ zip = (Left\ (V \times T_{bin})\ |\ Right\ (V \times T_{bin}))list \tag{15}$$

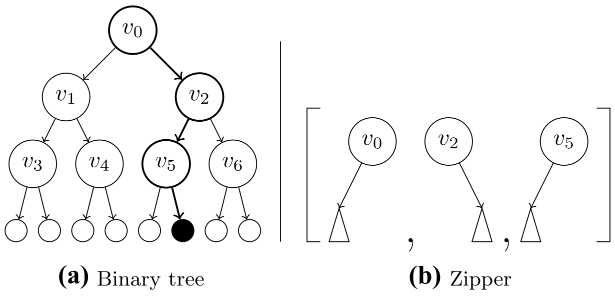


Fig. 5 A zipper structure representing the path highlighted of the binary tree

Applying the third theorem of homomorphism on trees requires the definition of an upward and downward computation. The downward version is represented Eq. 16 and the upward version in Eq. 17.

$$\begin{array}{ll} cost_{\downarrow} & = fst \circ h_{\downarrow} \\ h_{\downarrow}([\,]) & = (\emptyset, Leaf) \\ h_{\downarrow}([Left(v_i, l)] +\!\!+ x) & = \textbf{let } (S, T) = h_{\downarrow}(x) \\ & \quad \textbf{in } (search(v_i, S, T),\ Node(v_0, l, T)) \\ h_{\downarrow}([Right(v_i, r)] +\!\!+ x) & = \textbf{let } (S, T) = h_{\downarrow}(x) \\ & \quad \textbf{in } (search(v_i, S, T),\ Node(v_0, T, r)) \end{array} \tag{16}$$

$$\begin{array}{ll} cost_{\uparrow} & = fst \circ h_{\uparrow} \\ h_{\uparrow}(x +\!\!+ [v]) & = h_{\uparrow}(x) \otimes v \\ h_{\uparrow}([\,]) & = (\emptyset, Leaf) \\ (S, T) \otimes [Left(v_i, l)] & = (search\ (v_i, S, T),\ Node\ (v_i, l, T)) \\ (S, T) \otimes [Right(v_i, r)] & = (search\ (v_i, S, T),\ Node\ (v_i, T, r)) \end{array} \tag{17}$$

Remark that going from a binary tree to any (bounded) degree trees requires a minor adjustment to the tree, zipper and computation definition by adding the required number of additional cases.

3.5 From Tree to Graph

Trees differ from DAGs in the sense that a node may have several parents in the case of a DAG. To address this issue, we treat the DAG as its spanning tree (only picking one parent of each node) while computing the redistribution cost with respect to all of the DAG edges. The number of parents (inputs) of each node can easily be bounded by the maximum number of inputs an operator in neural networks can have. This allows us to treat any DAG cases, but our proof that the reordering of the vertices preserves the cost cannot be extended to general DAGs in its current form.

Although trees are not general enough to cover any DAGs, they are already a substantial improvement over other methods that treat linear graphs [8, 11]. Those methods are consequently suitable for coarser grained graph representation, whereas a finer-grained representation provides more possibilities.

This work does not provide implementation details because a more practical point of view was already given in previous work [19].

4 Evaluation

4.1 Experiment Environment

DNN Models: We evaluate SMSG on real-world DNN models:

- Computer Vision:
  - ResNet50 with the Cifar10 dataset,
  - ResNet50/101/152 [2] with the ImageNet dataset,
  - Fully Convolution Network (FCN) [24] with a dedicated remove sensor dataset;
- Recommendation Systems:
  - Wide & Deep [3] with the Criteo dataset;
- Neural Language Processing:
  - BERT [14] with the Wiki-en dataset,
  - PanGu-Alpha 2.6B and 13B [25] (B stands for billion, which signifies the size of parameters),
  - T5 [15] (Text-to-Text Transfer Transformer) with a dedicated text dataset.

Evaluation Metric: We choose the average *step time* to compare the performance of the hybrid parallel strategies. Step time is the training time of one batch of data, including the FPG and BPG, which is inversely proportional to throughput. Shorter step time denotes better performance.

Hardware Environments: The experiments are conducted on an Atlas900 AI cluster [26]. Each Atlas node is composed of eight Ascend910 accelerators. Our experiments test until 64 accelerators, where the four nodes are connected with a 64-port switch. All the Ascend910 clusters are inter-connected directly, even from a different node. We also implemented an 8 NVIDIA-V100 GPU cluster as a control group to show the better portability of our approach.

Deep Learning Framework: Our experiments are conducted on MindSpore, which automatically supports lancer distributed training with a given strategy. The strategy found by SMSG will be taken as input for Mindspore, and the training will be conducted on this DL framework.

Searching Algorithm: SMSG offers the ability to model the cost and compute it with homomorphism. As for the searching algorithm, we implemented a linear-complexity searching algorithm D-Rec [19] for the experiments in this section.

Baseline: The research on automatic parallel plan search is still in its infancy: the generality of related methods is not complete, such as OptCNN, Flexflow, and

TensorOpt's search algorithms can only handle linear computational graphs; in addition, on classical neural networks, it is difficult for automatic parallel plan search to find manual parallel plans that experts have intensively studied for months. Therefore, the comparison baseline chosen in this section is the expert-designed parallel plan.

The following lists the *expert-designed parallel plans* used for different networks:

- *ResNet:* For all variants of ResNet, parallel plans are respectively analyzed for the convolutional and fully connected layers by taking into account the cost of redistribution. Fine-tuned OWT [27] for networks with different layers and input data is chosen as the expert-designed parallel plan here.
- *FCN:* The FCN network tested here is trained with a high-precision image such that model parallel (operator-level) should be put more importance. There is no published paper to describe them, but these expert-designed parallel plans were studied internally by experienced researchers for more than one month.
- *Wide & Deep:* HugeCTR [28] is a dedicated distributed training framework developed by NVIDIA that can support typical CTR network deployments such as Wide & Deep. The expert-designed parallel plan here follows the HugeCTR idea.
- *BERT, T5:* The baseline is a fine-tuned expert-defined parallel plan based on Megatron-LM [7], the well-known transformer-based manual parallel plan.
- *PanGu-alpha:* Pangu-alpha's expert-designed parallel plan is introduced in its published paper [29].

Competitive Approach: To show the better portability of SMSG compared with the previous approaches, we choose TensorOpt [8] as the competitive approach, which is a representative approach proposed in 2021 and can be regarded as the state-of-the-art approach for operator-level search.

4.2 Generality

We tested the quality of the hybrid parallel strategy found by SMSG in an extensive range of real-world DNN models. Table 1 shows the average step times of training varieties of DNN models with expert-designed strategies and the strategies found by SMSG. This table does not include the results of TensorOpt for two reasons. First, the baseline of expert-designed strategies has their optimality guaranteed by the efforts spent by the researchers. Secondly, TensorOpt lacks generality: (1) it cannot deal with non-linear graphs; (2) it requires manually configuring the search space for a new unknown DNN model.

The last column shows the performance percentage of SMSG to Baseline (high than 100% denotes a better performance). It can be found that the minimum performance percentage of SMSG to the baseline is 90.40% for the BERT model. The experiments show that SMSG can find good hybrid strategies for varieties of real-world DNN models with a correct performance higher than 90% to the

Table 1 Performance on varieties of DNN models

Performance: step time/ms (8 Ascends)

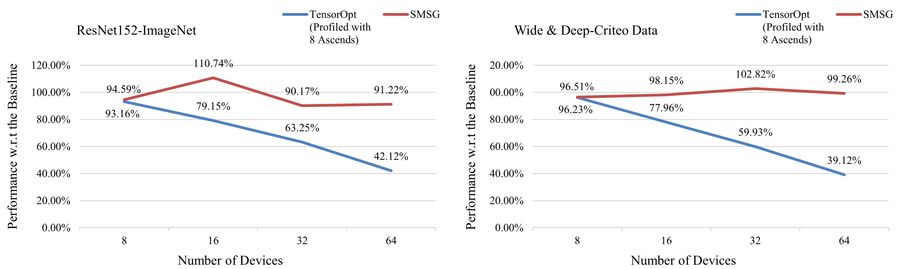
	DNN models	Baseline	SMSG	Percentage (%)
CV	ResNet50-Cifar	48.58	45.91	105.83
	ResNet50-ImageNet	57.53	61.18	94.03
	ResNet101-ImageNet	86.73	93.38	92.88
	ResNet152-ImageNet	120.57	127.46	94.59
	FCN	485	512	94.72
Rec.Syst.	Wide & Deep	21.6	22.38	96.51
NLP	BERT	110.63	122.38	90.40
	PanGu-Alpha 2.6B	4826	4876	98.91
	PanGu-Alpha 13B	13990	13988	100.01
	T5	1288	1279	100.70

expert-designed strategies. The −10% variations are not statistically significant and can be said to be a good performance. Even though the results of TensorOpt are not given in Table 1, it can be conducted from Fig. 6 that an inappropriate parallel plan leads to more than −60% performance decrease.

The hardware parameters and quantity functions are separated thanks to the profiling-free modeling. SMSG only profiles the communication and computation of the 8 Ascend cluster once (it takes a few minutes), and the hardware parameter values can generally be used for all the DNN models. The DNN models are different compositions of the 20 kinds of operators. SMSG shows its generality in searching for varieties of DNN models with the predefined quantity functions and the one-time profiled cluster parameters.

4.3 Portability

The experiments in this subsection demonstrate the portability of SMSG and the profiling-based approach when the training environments are changed. For



(a) ResNet152-ImageNet **(b)** W&D-Criteo Data

Fig. 6 Portability w.r.t. the scale of cluster

profiling-based approaches like TensorOpt, profiling the operators under different parallel configurations of typical DNN models usually takes more than one day. On the contrary, for SMSG, profiling a hardware configuration takes only some minutes. In this section, the results of SMSG shown in Fig. 6 and Table 2 are obtained with the profiled communication and computation capacity on targeted hardware architecture because it only takes some minutes. However, for the TensorOpt, we kept one profiling base and varied the training configurations to show the impact of the profiling data. We choose two typical DNN models, ResNet152-ImageNet and Wide & Deep, to test.

Figure 6 shows the percentage performance of TensorOpt and SMSG w.r.t. the number of cluster devices. Both for ResNet and Wide & Deep, it can be easily conducted that with the increase in device numbers, the quality of strategies found by TensorOpt decreases because they do not have enough profiling data of the possible partition dimensions, so they missed the optimal strategies. However, the profiling-free approach SMSG can keep a good strategy quality because of the leveraged profiling time.

The same conduction can be made from Table 2 that searching the strategy with different profiled data from a different architecture for TensorOpt, the decrease of strategy quality is evident, while SMSG keeps good results. The cost model of TensorOpt is based on the profiled execution time of operators on the actual hardware. The execution time of an operator with the same parallel strategy is different on GPUs and Ascends. That is why the strategy quality of TensorOpt decreases when executed on GPUs with profiling data on Ascends. Heavy profiling tasks (a few days) limit the portability of these profiling-based approaches, while SMSG with a lightened profiling job is more practical.

5 Conclusion

This paper proposes SMSG, a profiling-free strategy generation method for distributed DNN training. The main idea of SMSG is that its symbolic cost model is built based on the relative cost instead of the execution time. The hardware parameters and similar data quantity functions are separated. The cost functions can be optimized independently of the hardware. Besides, we introduce homomorphism

Table 2 Portability w.r.t. hardware architecture

Performance: percentage w.r.t the baseline

Profiling base	DNN models	Tensoropt (8 Ascends) (%)	Tensoropt (8 GPUs) (%)	SMSG (%)
8 GPUS	ResNet152-ImageNet	62.12	99.18	99.53
	Wide & Deep	49.55	98.25	98.33
8 Ascend	ResNet152-ImageNet	98.16	71.56	94.59
	Wide & Deep	97.23	66.89	96.51

to re-organize the redistribution cost so that the searching algorithm does not have graph topology dependency. These two contributions offer generality and portability for the DNN parallel strategy generation.

Acknowledgements This work was partially supported by the National Key R&D Program of China under Grand No. 2021ZD0110403 and by the French industrial doctoral framework CIFRE under No.2019/2451.

Author Contributions Haoran Wang, Thibaut Tachon, Chong Li contributed equally and wrote the main manuscript text. All authors reviewed the manuscript.

Declarations

Competing interests The authors declare no competing interests.

References

1. Brown, T., Mann, B., Ryder, N.: Language models are few-shot learners. Adv. Neural Inf. Process. Syst. **33**, 1877–1901 (2020)
2. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
3. Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., Shah, H.: Wide & deep learning for recommender systems. In: Proceedings of the 1st Workshop on Deep Learning for Recommender Systems. DLRS 2016, pp. 7–10. Association for Computing Machinery, New York, (2016)
4. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. **60**(6), 1097–1105 (2012)
5. Dean, J., Corrado, G.S., Monga, R.: Large scale distributed deep networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems - Vol. 1. NIPS'12, pp. 1223–1231. Curran Associates Inc., Red Hook, (2012)
6. Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M.X., Chen, D., Lee, H., Ngiam, J., Le, Q.V., Wu, Y., Chen, Z.: GPipe: Efficient training of giant neural networks using pipeline parallelism. Curran Associates Inc., Red Hook (2019)
7. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-LM: training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053 (2019)
8. Cai, Z., Yan, X., Ma, K., Wu, Y., Huang, Y., Cheng, J., Su, T., Yu, F.: Tensoropt: exploring the tradeoffs in distributed DNN training with auto-parallelism. IEEE Trans. Parallel. Distrib. Syst. **33**(8), 1967–1981 (2022)
9. Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., Diao, L., Liu, X., Lin, W.: DAPPLE: a pipelined data parallel approach for training large models.

In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP ’21, pp. 431–445. Association for Computing Machinery, New York, (2021)

10. Tarnawski, J.M., Narayanan, D., Phanishayee, A.: Piper: Multidimensional planner for DNN parallelization. In: Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P.S., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems, vol. 34, pp. 24829–24840 (2021)
11. Jia, Z., Zaharia, M., Aiken, A.: Beyond data and model parallelism for deep neural networks. In: Talwalkar, A., Smith, V., Zaharia, M. (eds.) Proceedings of Machine Learning and Systems, vol. 1, pp. 1–13 (2019)
12. Jia, Z., Lin, S., Qi, C.R., Aiken, A.: Exploring hidden dimensions in accelerating convolutional neural networks. In: Dy, J., Krause, A. (eds.) International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 80, pp. 2274–2283 (2018)
13. Wang, M., Huang, C.-C., Li, J.: Supporting Very Large Models using Automatic Dataflow Graph Partitioning. EuroSys ’19. Association for Computing Machinery, New York (2019)
14. Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Vol. 1 (Long and Short Papers), pp. 4171–4186. Association for Computational Linguistics, Minneapolis, Minnesota (2019)
15. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. **21**(140), 1–67 (2020)
16. Abadi, M., Barham, P., Chen, J., : Tensorflow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 265–283. USENIX Association, Savannah, GA (2016)
17. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L.: Pytorch: an imperative style, high-performance deep learning library. Adv. Neural Inf. Process. Syst. **32**, 8026–8037 (2019)
18. Huawei: MindSpore. Huawei. https://www.mindspore.cn/
19. Wang, H., Li, C., Tachon, T., Wang, H., Yang, S., Limet, S., Robert, S.: Efficient and systematic partitioning of large and deep neural networks for parallelization. In: European Conference on Parallel Processing, pp. 201–216, Springer, (2021).
20. Valiant, L.G.: A bridging model for multi-core computing. J. Comput. Syst. Sci. **77**(1), 154–166 (2011)
21. Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. SIGPLAN Not. **44**(1), 177–185 (2009)
22. Gibbons, J.: Functional pearls: the third homomorphism theorem. J. Funct. Program. **6**(4), 657–665 (1996)
23. Huet, G.: The zipper. J. Funct. Program. **7**(5), 549–554 (1997)
24. Schwing, A., Urtasun, R.: Fully connected deep structured networks (2015)
25. Zeng, W., Ren, X., Su, T., Wang, H., Liao, Y., Wang, Z., Jiang, X., Yang, Z., Wang, K., Zhang, X., Li, C., Gong, Z., Yao, Y., Huang, X., Wang, J., Yu, J., Guo, Q., Yu, Y., Zhang, Y., Wang, J., Tao, H., Yan, D., Yi, Z., Peng, F., Jiang, F., Zhang, H., Deng, L., Zhang, Y., Lin, Z., Zhang, C., Zhang, S., Guo, M., Gu, S., Fan, G., Wang, Y., Jin, X., Liu, Q., Tian, Y.: PanGu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. CoRR (2021) arXiv:2104.12369
26. Huawei: Atlas900. https://e.huawei.com/en/products/cloud-computing-dc/atlas/atlas-900-ai
27. Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997 (2014)
28. Oldridge, E., Perez, J., Frederickson, B., Koumchatzky, N., Lee, M., Wang, Z., Wu, L., Yu, F., Zamora, R., Yilmaz, O., et al.: Merlin: a gpu accelerated recommendation framework. Proceedings of IRS (2020)
29. Zeng, W., Ren, X., Su, T., Wang, H., Liao, Y., Wang, Z., Jiang, X., Yang, Z., Wang, K., Zhang, X., et al.: Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. arXiv preprint arXiv:2104.12369 (2021)

Authors and Affiliations

Haoran Wang[1,2] · Thibaut Tachon[1] · Chong Li[1] · Sophie Robert[2] · Sébastien Limet[2]

Haoran Wang
haoran.wang@etu.univ-orleans.com

Thibaut Tachon
thibaut.tachon@huawei.com

Sophie Robert
sophie.robert@univ-orleans.fr

Sébastien Limet
sebastien.limet@univ-orleans.fr

1 Huawei Technologies France S.A.S.U., 18-20 Quai du Point du Jour, 92100 Boulogne-Billancourt, France

2 LIFO, Bat. 3IA, Université d'Orléans, Rue Léonard de Vinci, 45067 Orléans, France