HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis

Shiwei Zhang
The University of Hong Kong
swzhang@cs.hku.hk

Lansong Diao
Alibaba Group
lansong.dls@alibaba-inc.com

Chuan Wu
The University of Hong Kong
cwu@cs.hku.hk

Zongyan Cao
Alibaba Group
zongyan.cao@alibaba-inc.com

Siyu Wang
Alibaba Group
siyu.wsy@alibaba-inc.com

Wei Lin
Alibaba Group
weilin.lw@alibaba-inc.com

Abstract

Single-Program-Multiple-Data (SPMD) parallelism has recently been adopted to train large deep neural networks (DNNs). Few studies have explored its applicability on heterogeneous clusters, to fully exploit available resources for large model learning. This paper presents *HAP*, an automated system designed to expedite SPMD DNN training on heterogeneous clusters. *HAP* jointly optimizes the tensor sharding strategy, sharding ratios across heterogeneous devices and the communication methods for tensor exchanges for optimized distributed training with SPMD parallelism. We novelly formulate model partitioning as a program synthesis problem, in which we generate a distributed program from scratch on a distributed instruction set that semantically resembles the program designed for a single device, and systematically explore the solution space with an A*-based search algorithm. We derive the optimal tensor sharding ratios by formulating it as a linear programming problem. Additionally, *HAP* explores tensor communication optimization in a heterogeneous cluster and integrates it as part of the program synthesis process, for automatically choosing optimal collective communication primitives and applying sufficient factor broadcasting technique. Extensive experiments on representative workloads demonstrate that *HAP* achieves up to 2.41x speed-up on heterogeneous clusters.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**.

Keywords: Distributed system, Neural networks, Program synthesis

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629580

ACM Reference Format:
Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. 2024. *HAP*: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3627703.3629580

1 Introduction

Recent machine learning research has demonstrated that scaling up deep neural network (DNN) models not only improves their prediction performance but also expands their capabilities. For instance, a language model can perform a task with few-shot prompting after reaching a certain model scale [49]. Massive models with billions or trillions of parameters have emerged [6, 11]. Training these models necessitates the use of large clusters of accelerator devices (dominantly GPUs), as well as sophisticated parallelization paradigms. Consequently, a number of parallelization schemes have been proposed and adopted in distributed training of large DNNs, including data/model/pipeline parallelism [17, 19, 29, 36, 44, 45, 47, 52, 55]. However, the majority of the existing proposals concentrate on DNN training on homogeneous clusters, where all devices are of the same type and interconnect network links have identical bandwidth.

On the other hand, the rapid evolution of accelerate devices (e.g., GPUs, TPUs, Habana chips) and multi-tenant resource sharing have resulted in mixed device types and uneven interconnect bandwidth in many clusters. Efficient exploitation of available heterogeneous resources for DNN training has piqued significant interest from AI practitioners [16, 25, 27, 30, 43, 53, 56]. A prevalent approach for making use of heterogeneous clusters is to distribute multiple training jobs onto different homogeneous subsets of the cluster [50, 57]. However, this approach imposes a constraint on the maximum model size, as it is bound by the capacity of the subsets. With the emergence of large language models, the demand for employing the whole heterogeneous cluster to train a large model has become increasingly crucial.

The limited existing heterogeneity-aware DNN training designs mostly support data parallelism and inter-op model parallelism [27, 53, 56]. For data parallelism, each device

trains the DNN model for a distinct portion of the dataset, allocated according to capacity of the device, and synchronizes model gradients among each other at the end of each training iteration. Inter-op model parallelism involves placing different operations in the DNN model across devices, with intermediate tensors transmitted among devices during training. For better GPU utilization, inter-op model-parallel training is often scheduled in a pipelined manner, with micro-batches of data processed on different devices at the same time. Due to constrained memory capacity of GPUs, pure data parallelism or inter-op model parallelism may not be feasible or efficient for training large DNN models. For instance, in models with Mixture-of-Expert (MoE) [40] layers, a single tensor may surpass the GPU's memory limit. In these cases, it becomes necessary to employ intra-op model parallelism, which partitions the operations/tensors and distributes the shards to different devices. It is especially challenging to devise efficient strategies on tensor sharding and deployment across a cluster of heterogeneous devices, due to the large strategy space.

This work studies Single-Program-Multiple-Data (SPMD) parallelism for efficient training of large DNNs on heterogeneous clusters. SPMD parallelism generalizes data parallelism and intra-layer model parallelism with tensor sharding along any of its dimensions and input data partitioning across the devices. It has been proven effective in training various state-of-the-art models. For example, GShard [22] uses model parallelism for MoE layers and data parallelism for other layers, and Megatron [41] designs an SPMD strategy for the Transformer layers [46]. One of the key benefits of SPMD parallelism is that each device executes the same program, thereby enabling scaling to a large number of devices with a constant program compilation time.

SPMD parallelism has so far been exploited on homogeneous clusters. Enabling efficient SPMD training on a set of heterogeneous resources facilitates better utilization of available resources for substantially lowered cost of large model learning. Three key decisions are involved in applying SPMD parallelism in heterogeneous clusters: (i) the *sharding strategy*, i.e., deciding which dimension to partition (sharding dimension) for each tensor; (ii) the *sharding ratios* across the devices, i.e., different tensor partition sizes to assign to heterogeneous devices according to their computation and memory capacities, to optimize device utilization; and (iii) selection of the *communication methods*, which decides the implementation of each collective communication operation for each tensor, to best cater to different tensor sizes and different interconnect bandwidths across devices. The three decisions are co-related. For example, if the sharding ratios are relatively even among the devices, inter-device communication pattern resembles that of homogeneous clusters, and standard collective communication generally performs well. In contrast, if the sharding ratio differs significantly across devices, heterogeneity-aware communication are needed.

We propose *HAP*, an SPMD DNN training system for heterogeneous clusters, that automatically decides optimal tensor sharding dimension/ratios and communication methods for expedited training and optimized resource utilization. We make the following contributions in designing *HAP*:

▹ We design an iterative optimization process that alternatively optimizes the SPMD sharding strategy and sharding ratios while fixing the other one. In comparison to existing methods that only optimize each aspect once [43], our iterative optimization enables us to approach the global optimum while still maintaining an acceptable optimization time (Sec. 3).

▹ We novelly formulate SPMD model sharding as a program synthesis problem, to construct a distributed program on a *distributed instruction set* to emulate a given tensor program implemented on a *single-device instruction set*. We analyze the single-device program to build a background theory $\mathcal{T}$ of semantic constraints, and then employ syntax-guided synthesis [3] with an A*-based search algorithm to automatically synthesize a distributed program to achieve minimal training time, that is equivalent to the single-device program under the theory $\mathcal{T}$ (Sec. 4).

▹ We design a linear cost model and formulate sharding ratio optimization as a linear programming problem, and solve it optimally with off-the-shelf solvers (Sec. 5).

▹ We explore two communication optimization techniques and integrate them into the program synthesis, to optimize communication on heterogeneous clusters jointly with SPMD sharding. The first optimization involves the trade-off between two `All-Gather` implementations on heterogeneous clusters and the second is to automatically apply sufficient factor broadcasting [51] that reduces the communication volume for certain operators (Sec. 4.4).

▹ We implement *HAP* on PyTorch and evaluate it on a 64-GPU heterogeneous cluster on a public cloud. The user API of *HAP* is analogous to the built-in DDP module of PyTorch. Experiments with four representative image classification and language models demonstrate that *HAP* consistently outperforms existing systems on heterogeneous clusters and show competitive performance on homogeneous clusters while introducing only seconds of overhead in program synthesis.

2 Background and Motivation

2.1 Large Neural Networks and SPMD Parallelism

There has been a noticeable increase in size of state-of-the-art DNN models, with examples such as GPT-3 [6] containing 175 billion parameters and PaLM [11] containing 540 billion parameters. These large neural networks are often constructed using Transformer [46] layers. Many of them incorporate Mixture-of-Expert (MoE) [40] layers as well, which contain sparsely activated experts (each input token is processed by a fixed number of experts, regardless of the total

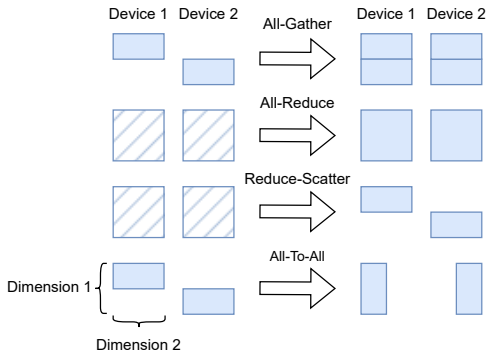

Figure 1. Common collective communication operations.

number of experts in an MoE layer). The tensors (parameters, gradients, activation, optimizer states, etc.) in these models can be sizable that a single tensor exceeds the memory capacity of a modern GPU. For example, the parameter size of an MoE layer with 2048 experts is about 36GB. Thus, partitioning the model and deploying its shards on multiple devices is indispensable for effective training of these models.

A tensor in a DNN model is a multi-dimension array of floating-point numbers. A tensor can be partitioned (aka sharded) into smaller tensors by splitting on any of its dimensions. For example, most of the activation tensors (intermediate results calculated from the input to the model) in mini-batch training has a "batch size" dimension. Partitioning this dimension results in the so-called data parallelism. With SPMD parallelism, the same program is executed on multiple devices, which collectively produces a result that is identical to that of a single-device program. Existing SPMD training systems typically partition the operators in a single-device program by choosing a sharding strategy out of a fixed set for each operator. For example, AccPar [43] considers three sharding strategies (sharding on the "batch size" dimension, "hidden feature" dimension, and "reduction" dimension) for each `MatMul` operator. Flexflow [19] supports 4 dimensions of parallelism by sharding operators on the sample, operator, attribute and parameter dimensions. When producers and consumers of a tensor are sharded in an incompatible way (i.e., a tensor is expected by the consumer to be sharded on a different dimension than what it is produced on), collective communication operators are inserted to switch the sharding dimension of the tensors according to pre-defined rules. For instance, if a tensor is sharded on the batch size dimension and the consumer is expecting a full-sized tensor, `All-Gather` can be used to gather the shards of the tensor across devices to recover the same full-sized tensor on all devices.

2.2 Collective Communication

Four MPI-style [28] collective communication operations are commonly used in distributed DNN training. As illustrated in Fig. 1, `All-Gather`$(\bar{e}, d)$ concatenates the slices of tensor $\bar{e}$ across devices along the d dimension. `All-Reduce`$(\bar{e})$ sums the replicas of a tensor $\bar{e}$ across devices, element-wisely. `Reduce-Scatter`$(\bar{e})$ is equivalent to performing `All-Reduce` and then sharding the results on each device, but is implemented in a more efficient way. `All-To-All`$(\bar{e}, d_1, d_2)$ takes as input the tensor $\bar{e}$ that is sharded on its d_1 dimension and outputs the tensor that is sharded on the d_2 dimension.

The current collective communication libraries have been developed for homogeneous clusters and may not exhibit optimal performance when applied to tensors of different sizes in a cluster of different inter-device bandwidths. For instance, NCCL [18] requires all tensors to be of the same size for `All-Gather`. In order to perform this communication operation on unevenly sharded tensors, the tensor shards must be first padded to the same size and subsequently trimmed upon completion of the operation, resulting in wasted bandwidth and extra memory access. Alternatively, `All-Gather` can be implemented with multiple `Broadcast` operations to support unevenly sliced tensor shards without padding, at the cost of a higher kernel launching overhead.

2.3 Syntax-Guided Synthesis

Syntax-guided synthesis [3] is a type of program synthesis, where the inputs include a syntax specification that defines the program space, a semantic correctness specification that describes the desired properties of the synthesized program, and a background theory to verify whether a given program satisfies the semantic correctness specification. Automated program synthesis has been successfully applied to optimize various kinds of programs, such as SQL [38] and Datalog [48]. Compared to the programs implemented in general-purpose programming languages, these programs typically have simpler structures that allow easier semantic analysis. Tensor programs that implement DNNs share the same characteristics as they are non-recursive and have no side-effects.

We exploit syntax-guided synthesis to systematically generate feasible distributed programs for SPMD parallelism, in order to identify the best one that maximizes training speed. Most literature uses manually defined background theories, such as the linear integer arithmetic (LIA) [4]. We construct a background theory for each single-device program in the form of Hoare triples [15] by automatically analyzing the single-device program. During the synthesis of the distributed program, only the mathematical relations between the model output and the model inputs are utilized. As a result, we decouple the performance of the distributed program from the implementation details of the provided single-device program. Moreover, we automatically explore alternative implementations of operations that achieve the same mathematical results during program synthesis.

Some existing distributed DNN training systems may be categorized as a form of "transformational program synthesis", wherein an initial program undergoes successive modifications based on a set of rewriting rules. As an example, Unity [45] performs pattern matching and substitution on a

proposed parallel computation graph to transform programs. Each optimized step in this process yields a complete and correct program, by the correctness of the applied rewriting rules. In contrast, our approach diverges by exploring incomplete and potentially incorrect programs throughout the synthesis process, and validate the semantics of the resulting distributed program using the background theory derived from the single-device program.

2.4 Optimal Sharding Ratios

With pure data parallelism, the optimal sharding ratios can be readily decided by profiling the computation speed of different devices and setting the ratios in proportion to computation speeds of devices. This does not work optimally for SPMD parallelism where `All-Gather` and `Reduce-Scatter` are utilized to concatenate tensor shards or aggregate tensor replicas and then shard the result across devices. In these communication operations, the devices send and receive tensor shards of sizes proportional to the sharding ratios. The communication time therefore depends on the size of the largest shard. Minimal communication time is achieved when the tensors are sharded evenly, as the size of the largest shard is minimized in this case.

To demonstrate this, we train a Transformer model with intra-op model parallelism on two machines, one equipped with two P100 GPUs and the other with two A100 GPUs. Tensors in the model are sharded across the GPUs in two ways: *CP*, with sharding ratios proportional to the computational power of the devices, and *EV*, evenly sharding the tensors. We manipulate the hidden feature dimension of the model to alter the computation-to-communication ratio. Fig. 2 shows that when the computation time dominates, CP performs better as it balances computation time on different devices; EV performs better when communication is the bottleneck. EV leads to lower communication time for `All-Gather` and `Reduce-Scatter` operations and is preferable when the computation-to-communication ratio is low. When computation time and communication time are similar, a "sweet point" may exist between CP and EV that achieves the optimal trade-off between load balance on different devices and fast communication operations. Further, since different layers of a model may exhibit different computation-to-communication ratios, the optimal sharding ratios may vary for each layer. In *HAP*, we formulate sharding ratio optimization as a linear programming problem to determine the optimal ratios for each part of the model.

2.5 Communication Optimization

When a tensor is partitioned unevenly, standard collective communication routines that assume homogeneous clusters do not perform optimally. To explore opportunities for heterogeneity-aware communication, we study a few techniques that can potentially benefit collective communication in heterogeneous environments and show that these optimizations need to be jointly decided with the sharding strategy and sharding ratio selection.

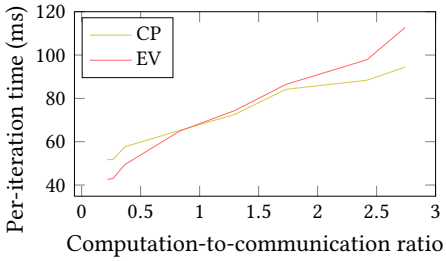

Figure 2. Performance with different sharding ratios under different computation-to-communication ratios (computed for P100 GPUs).

2.5.1 Padded All-Gather and Grouped Broadcast.

The `All-Gather` and `Reduce-Scatter` operations implemented in NCCL require all shards to be of the same size. To perform these operations for unevenly partitioned tensors, we can either pad the shards to the same size before communication, or broadcast each shard seperately using a NCCL group call. Fig. 3 visualizes the two approaches.

Selection between the two approaches is highly contingent upon the tensor sharding ratios employed. When the tensors are nearly evenly partitioned, the required padding is minimal, and the padded `All-Gather` method outperforms other approaches owing to optimizations in NCCL. In contrast, when a tensor is sharded using heavily skewed ratios among devices, the grouped `Broadcast` approach yields better performance. Fig. 4 illustrates this phenomenon as we test the two approaches on a 4MB tensor in a cluster of two machines, each equipped with two NVIDIA A100 GPUs. We allocate the largest shard to the first device and evenly partition the remaining among the other devices. The sharding ratio on the first device then decides the skewness of sharding, depicted as the x-axis in Fig. 4. The bandwidth is computed by dividing the full tensor size by the communication time, without taking into account any padding.

As performance of the two implementations depends on the sharding ratios, when we optimize the sharding ratios, the communication methods should be updated accordingly to achieve the optimal overall performance. In *HAP*, we include the selection of the two methods into the program synthesis process and interleave its optimization with the sharding ratio optimization.

2.5.2 Sufficient Factor Broadcasting (SFB).

SFB [51] exploits low-rank structures in gradient tensors to reduce parameter synchronization communication by replacing the `All-Reduce` operation on gradient tensors with `All-Gather` operations on smaller tensors called sufficient factors, which are sufficient to calculate the gradients. Fig. 5 gives an example of SFB for an `MatMul` operation. The output is an $f \times h$ tensor, which is the gradient of the parameter in a fully-connected layer. The inputs are the activation tensor

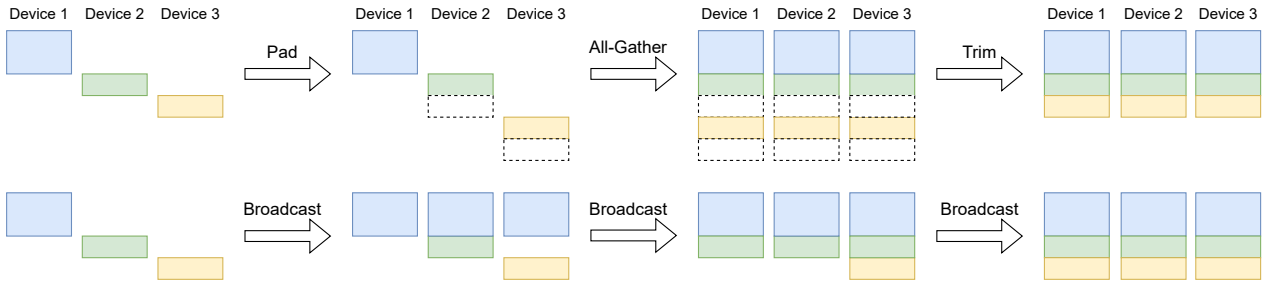

Figure 3. Different implementations of `All-Gather` for uneven shards.

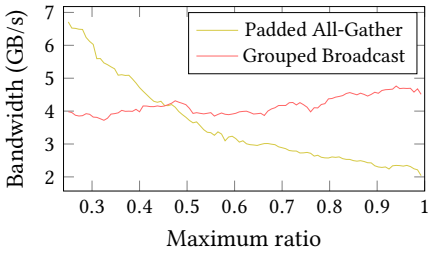

Figure 4. Performance of padded `All-Gather` and grouped `Broadcast` under different tensor sharding ratios.

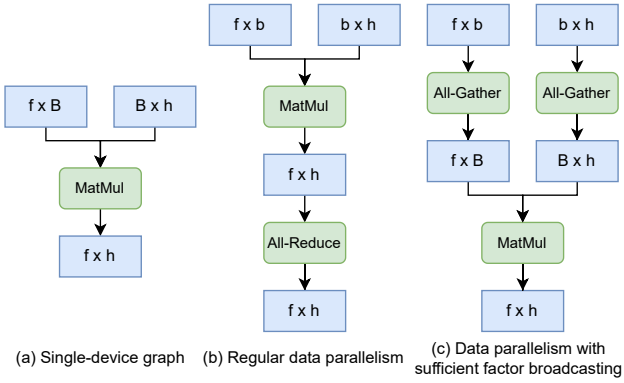

Figure 5. Sufficient Factor Broadcasting. (b) and (c) depict the SPMD program on each device.

of shape $b \times h$ and the gradient of the output with shape $f \times b$, where b is the local batch size on each device and B is the global batch size. When b is small, the gradient (the $f \times h$ tensor) is not full rank, and the two aggregated tensors (the $B \times h$ and $f \times B$ tensors) are its sufficient factors as the gradient can be calculated from the two tensors without further communication. With standard data parallelism, the gradients are aggregated among devices with `All-Reduce`, as shown in Fig. 5(b). With SFB, the input tensors are first collected with `All-Gather` and then each device calculates the complete gradient independently. SFB changes the communication process from transferring the gradient to transferring the sufficient factors, which can be of smaller sizes when the global batch size B is small.

The performance of SFB is primarily determined by the batch size and the number of devices involved [7, 54]. TAG [56] proposes an integer linear programming-based technique to automatically identify beneficial application of SFB to tensors in a DNN model trained in a homogeneous cluster. However, uneven tensor partitioning across heterogeneous resources introduces additional complication to this problem. As analyzed in Sec. 2.5.1, performance of `All-Gather` is influenced by the sharding ratios, on which whether SFB is beneficial depends as well. Moreover, in the case of SFB, every device performs the `MatMul` operation with a full batch size B, which may pose substantial computation overhead on slower devices. Therefore, SFB presents a different trade-off in heterogeneous clusters. In *HAP*, we integrate SFB into program synthesis to ensure optimal application of SFB as we update the sharding ratios.

3 Design Overview

We propose *HAP* that jointly decides the sharding strategy, sharding ratios, and communication methods of all tensors in a DNN model for effective SPMD model training on a heterogeneous cluster. The input to *HAP* consists of a single-device DNN model (DNN training program written for a single device), represented as a computation graph (V, E), and a cluster specification comprising m virtual devices. A virtual device can refer to a solitary computation unit (such as a GPU) or a small homogeneous group of physical devices (such as a machine containing multiple GPUs). We consider the distributed training strategy at the virtual device level. In the latter case, we assume that data parallelism is employed within each virtual device, as inter-connections within a machine typically exhibits high bandwidth (e.g., NVLink) and data parallelism tends to yield reasonable performance.

3.1 Main Components

HAP comprises two pivotal components: a program synthesizer and a load balancer. The program synthesizer generates the optimal distributed program Q for given sharding ratios B of the tensors in the DNN model, while the load balancer produces the optimal sharding ratios B for a fixed distributed program Q. The distributed program Q is a program on a distributed instruction set that can be executed on all devices for distributed DNN training with the SPMD parallelism. The tensor sharding strategies and communication methods are implicitly embedded in Q.

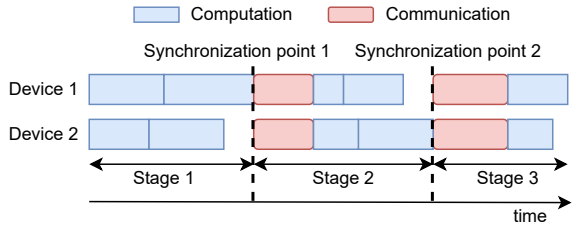

Figure 6. Stages and synchronization points.

The goal is to find the optimal combination of distributed program and sharding ratios (Q^*, B^*) that minimizes the DNN per-iteration training time $t(Q, B)$. *HAP* adopts an iterative optimization approach. During each step s, we fix one of the two decision aspects and identify the optimal solution for the other:

$$Q^{(s)} = \arg\min_{Q} \ t(Q, B^{(s-1)}) \tag{1}$$

$$B^{(s)} = \arg\min_{B} \ t(Q^{(s)}, B) \tag{2}$$

Starting with the initial sharding ratios $B^{(0)}$, which are selected to be proportional to the computation power of devices, we first execute the program synthesizer to generate $Q^{(1)}$ following Eqn. (1). Then we compute $B^{(1)}$ based on $Q^{(1)}$ utilizing Eqn. (2). We proceed to calculate $Q^{(2)}$ based on $B^{(1)}$, and so on, until convergence or oscillation of the solutions is attained. In the case of oscillation, we use the pair of Q and B achieving the lowest cost within the optimization process.

3.2 Cost Modeling

Directly profiling the training performance for each combination of (Q, B) would be resource expensive. We provide an estimate of the per-iteration training time $t(Q, B)$ by simulating the execution of the distributed program Q with B on the heterogeneous cluster.

Collective communication typically requires every participant device to both send and receive data to and from all other participants. It is hence reasonable to assume that all devices are synchronized before communication operations commence. This allows us to divide the execution of a distributed program into *stages*, with each stage starting with a communication operation followed by a series of computation operations (except for the first stage which only contains computation), as illustrated in Fig. 6. For example, the first two instructions in the program ⑦ in Fig. 11 are in the first stage of this program and the other two instructions are in the second stage. All devices are synchronized at the beginning of a stage.

Let $\text{comm}^{(i)}$ and $\text{comp}_j^{(i)}$ denote the communication time and computation time of the i-th stage on the j-th device, respectively. As each stage is globally synchronized, the iteration time is the sum of execution time of all stages. The execution time of a stage is determined by the maximum running time of that stage on all devices. Therefore, we have

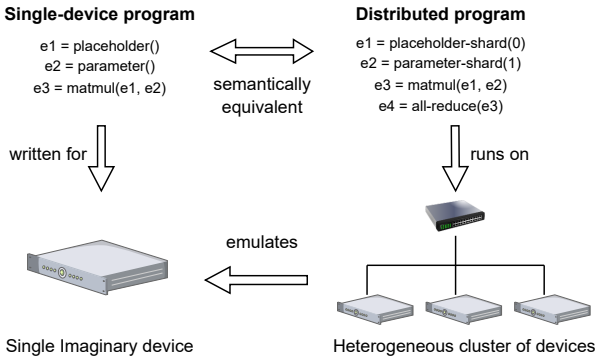

Figure 7. A heterogeneous cluster runs distributed programs to emulate a single-device program.

$$t(Q, B) = \sum_{i \in \text{stages}(Q)} (\text{comm}^{(i)}(B) + \max_{j \in [m]} \text{comp}_j^{(i)}(B_j))$$

where $[m] = \{1, \ldots, m\}$ is the list of devices. $\text{comp}_j^{(i)}(B_j)$ can be calculated based on the profiled flops-per-second of the j-th device and the estimated number of flops of the computation operations in stage i. Specifically, common operations in DNNs have numbers of flops that are linear to some of the dimensions of the input tensors. If one of these dimensions are sharded, the number of flops of this operation on a device is proportional to the sharding ratio of this device; otherwise, the number of flops does not change. The computation time $\text{comp}_j^{(i)}(B_j)$, which is calculated for each operator in the i-th stage by dividing the flops with the flops-per-second of the j-th device, is therefore a linear function of the sharding ratio B_j. In the case of running *HAP* on virtual devices which represent machines (each may contain multiple GPUs), we add the internal communication time estimated by the internal bandwidth and parameter sizes in the stage into $\text{comp}_j^{(i)}(B_j)$. $\text{comm}^{(i)}(B)$ is determined based on the collective operation type, the sharding ratio B, and NCCL's profiling data on the cluster's network. We run each collective operation on the cluster with tensors of different sizes and fit the latency and bandwidth in a linear model. $\text{comm}^{(i)}(B)$ is then estimated using the fitted model, with the input of the tensor size of the largest shard.

4 Distributed Program Synthesis

Various optimizations such as SFB (Sec. 2.5.2) and different implementations of collective communications (Sec. 2.5.1) can be considered in tensor sharding strategy search, which has not been comprehensively investigated in previous SPMD training frameworks [19, 44, 47, 58]. In *HAP*, we systematically approach tensor sharding strategy design in a novel way by formulating it as a program synthesis problem. Instead of selecting a partitioning method for each operator to modify the single-device program, we synthesize a distributed program from scratch on a distributed instruction

program	∈	[*instruction*]
instruction	:=	*computation* | *communication*
computation	:=	*tensor* ← *optype*([*tensor*])
communication	:=	*tensor* ← *collective*(*tensor*, *dim*)
dim	∈	{0, 1, ...}
collective	:=	`All-Reduce` | `All-Gather` | ...
optype	:=	`MatMul` | `Sigmoid` | ...

Figure 8. Syntax of a distributed program.

set that emulates the single-device program, as illustrated in Fig. 7. To ensure that the synthesized program is equivalent to the original single-device program, we formalize the semantics of the single-device program and then generate the distributed program under the constraint that the generated program must produce a semantically equivalent output as the single-device program for any inputs.

4.1 Distributed Programs

A *distributed program* Q is defined as a sequence of symbols that follows the syntax in Fig. 8. An *instruction* is a computation operation or a collective communication operation with a set of tensors as inputs and produces a tensor as the output. The computation instructions in the distributed instruction set are largely similar to the single-device instruction set, i.e., the tensor operators provided by DNN frameworks like PyTorch [33], except for some specialized operations like `Placeholder-Shard`, which is akin to the `Placeholder` operation utilized in single-device programs to read model inputs, but assumes that the input tensor is partitioned along a specific dimension. Executing a distributed instruction involves executing the same instruction on all devices, where the inputs are local tensors on each device.

4.2 Program Semantics

To produce distributed programs that are equivalent to the single-device program, we first analyze the semantics of the single-device program to form a background theory $\mathcal{T}$, which is utilized to express the semantic constraints during program synthesis.

The semantics of a program are expressed as a set of *properties*. To formally define these properties, we introduce *distributed tensors*, which are tensors produced by and used in distributed programs. A distributed tensor is a collection of instances (i.e., shards of sharded tensors and replicas of replicated tensors) of the same tensor on all devices. A property of a distributed tensor describes its mathematical relationship with a reference tensor, referring to a tensor in the single-device graph (V, E). The properties of a distributed tensor $\bar{e}$ are expressed as $e \mid I$, where $e \in E$ is a reference tensor and I is an instruction such that executing I with $\bar{e}$ as input produces a distributed tensor whose instances on all devices are equal to e. For example, if a distributed tensor $\bar{e}$ has the property $e \mid$ `All-Gather`(0), then after executing `All-Gather`$(\bar{e}, 0)$ (where 0 denotes the sharding dimension), all devices will have a tensor that is equivalent to e. The set of tensors generated by a program Q is denoted by $E(Q)$, and the properties of a program Q are defined as the properties of all tensors in $E(Q)$, denoted as $P(Q)$.

The background theory $\mathcal{T}$ is expressed as a set of Hoare triples [15]. A Hoare triple is represented as:

$$\{\, precondition \,\}\ instruction\ \{\, postcondition \,\}$$

When the precondition is satisfied, executing the instruction establishes the postcondition. The instruction is either a computation operation or a collective communication operation that runs simultaneously across all devices. The precondition and postcondition are expressed as sets of properties. If a program contains all properties in the precondition, appending the instruction to the program results in a new program that contains the properties in the postcondition.

We derive the background theory $\mathcal{T}$ by analyzing the single-device computation graph with a set of pre-defined rules that encodes mathematical characteristics of common tensor operations. Fig. 9 provides some examples of such rules on four collective communication operations and the `MatMul` operation. For conciseness, we do not explicitly name the tensors produced by an operation, but just use $\bar{e}$ to refer to the distributed tensor that has the property in the precondition associated with a reference tensor e. For example, in the first rule in Fig. 9, the precondition $e \mid$ `All-Reduce` means that there is a tensor $\bar{e}$ in the program running `All-Reduce` with which produces a distributed tensor that equals a reference tensor e. When this condition is met, appending the instruction `All-Reduce`$(\bar{e})$ to the program leads to a new program that meets the postcondition $e \mid$ `Identity`, which means that a distributed tensor produced by the new program is equivalent to the reference tensor e (`Identity` is an operator that returns its input). As another example, the last rule in Fig. 9 describes what is usually called reduction parallelism for `MatMul`: if e_1 is sharded on its second dimension and e_2 is sharded on its first dimension, the `MatMul` operation can be executed with instances of the two tensors on all devices, but an extra `All-Reduce` needs to be used on the result to obtain a tensor that equals the single-device `MatMul`.

The background theory $\mathcal{T}$ is obtained for the single-device graph (V, E) by enumerating all rules and finding matches in the single-device graph, and then gathering the Hoare triples from the matched rules. The semantic constraint of the distributed program Q is defined as $(l \mid$ `All-Reduce`$) \in P(Q)$, where l is the output tensor (typically the training loss) of the single-device graph. If a distributed program can be proved to have property $l \mid$ `All-Reduce` under theory $\mathcal{T}$, it is deemed equivalent to the single-device graph as it produces the same output as the single-device graph. We will use such semantic constraints to produce equivalent distributed programs with different tensor sharding and communication strategies.

$$\frac{\forall e \in E}{\{\, e \mid \texttt{All-Reduce} \,\}\ \texttt{All-Reduce}(\bar{e})\ \{\, e \mid \texttt{Identity} \,\}}$$

$$\frac{\forall e \in E,\ \forall d \in \dim s(e)}{\{\, e \mid \texttt{All-Reduce} \,\}\ \texttt{Reduce-Scatter}(\bar{e}, d)\ \{\, e \mid \texttt{All-Gather}(d) \,\}}$$

$$\frac{\forall e \in E,\ \forall d_1, d_2 \in \dim s(e),\ d_1 \neq d_2}{\{\, e \mid \texttt{All-Gather}(d_1) \,\}\ \texttt{All-To-All}(\bar{e}, d_1, d_2)\ \{\, e \mid \texttt{All-Gather}(d_2) \,\}}$$

$$\frac{\forall e \in E,\ \forall d \in \dim s(e)}{\{\, e \mid \texttt{All-Gather}(d) \,\}\ \texttt{All-Gather}(\bar{e}, d)\ \{\, e \mid \texttt{Identity} \,\}}$$

$$\frac{\forall e_1, e_2, e_3 \in E,\ e_3 = \texttt{MatMul}(e_1, e_2)}{\{\, e_1 \mid \texttt{All-Gather(0)},\ e_2 \mid \texttt{Identity} \,\}\ \texttt{MatMul}(\bar{e}_1, \bar{e}_2)\ \{\, e_3 \mid \texttt{All-Gather(0)} \,\}}$$

$$\frac{\forall e_1, e_2, e_3 \in E,\ e_3 = \texttt{MatMul}(e_1, e_2)}{\{\, e_1 \mid \texttt{Identity},\ e_2 \mid \texttt{All-Gather(1)} \,\}\ \texttt{MatMul}(\bar{e}_1, \bar{e}_2)\ \{\, e_3 \mid \texttt{All-Gather(1)} \,\}}$$

$$\frac{\forall e_1, e_2, e_3 \in E,\ e_3 = \texttt{MatMul}(e_1, e_2)}{\{\, e_1 \mid \texttt{All-Gather(1)},\ e_2 \mid \texttt{All-Gather(0)} \,\}\ \texttt{MatMul}(\bar{e}_1, \bar{e}_2)\ \{\, e_3 \mid \texttt{All-Reduce} \,\}}$$

Figure 9. Examples of the semantics of common collective communication operations and the `MatMul` operation.

4.3 Program Search Algorithm

A naive way to generate the best distributed program Q (that minimizes the iteration time with B) is to enumerate all possible programs, produced by sharding tensors along different dimensions and using different suitable collective communication following the syntax in Fig. 8. We can produce the programs in a breadth-first search manner and verify if a result program is semantically equivalent to the single-device program. However, the number of possible distributed programs grows exponential with the number of instructions. The exhaustive search is impractical for DNN models with hundreds or more operators.

We propose a more efficient program search algorithm based on the following ideas: (i) we estimate a cost lower-bound (execution time) for a partial program and stop searching further based on this partial program if its cost lower-bound is higher than the current best program; (ii) if two programs lead to the same set of properties, we discard the one with a higher cost. A program Q is considered *complete* if $l \mid \texttt{All-Reduce} \in P(Q)$, indicating that it is already semantically equivalent to the single-device program and no additional instructions are required. Programs constructed during the search that are not complete are called partial programs.

We use A* algorithm combined with the idea of dynamic programming to search for the optimal distributed program Q^*, as given in Fig. 10. We maintain a priority queue S that contains partial programs and their scores. The score of a partial program Q is an estimate of the per-iteration execution time of the optimal complete program Q_c that starts with Q: $\text{score}(Q) = \text{cost}(Q) + \text{ecost}(Q)$, where $\text{cost}(Q)$ is the execution time of the partial program Q and $\text{ecost}(Q)$ is a heuristic function that estimates the future cost of the program $\text{cost}(Q_c) - \text{cost}(Q)$. In order for the A* algorithm to find the optimal program, the heuristic function ecost must not overestimate the future cost (i.e., we need $\text{ecost}(Q) \leq \text{cost}(Q_c) - \text{cost}(Q)$), as otherwise the program will be excluded from the potential solutions. We use the minimum required execution time of program Q as its ecost, assuming infinite communication bandwidth among the devices. For a complete program Q, $\text{cost}(Q) = t(Q, B)$, obtained via our cost modeling in Sec. 3.2. For an incomplete program, $\text{cost}(Q)$ is calculated similarly, but only includes the time to reach the last synchronization point (e.g., synchronization point 2 in Fig. 6).

1: **Input:** computation graph (V, E), sharding ratios B
2: **Output:** Optimal distributed program Q^*

3: Initialize a priority queue S with an empty program $Q_\emptyset$
4: Initialize best program Q^* = **null** and set $\text{cost}(Q^*) = \infty$
5: **while** $\exists Q \in S, \text{score}(Q) < \text{cost}(Q^*)$ **do**
6: Remove the program Q with lowest score from S
7: **for** $\{\, precondition \,\}\ instruction\ \{\, postcondition \,\} \in \mathcal{T}$ of the single-device program where $precondition \subseteq P(Q)$ and $postcondition \nsubseteq P(Q)$ **do**
8: $Q' = Q \cup instruction$; $P(Q') = P(Q) \cup postcondition$
9: **if** $\exists Q_s \in S$ s.t. $P(Q_s) \supseteq P(Q')$ and $\text{cost}(Q_s) \leq \text{cost}(Q')$ **then**
10: **continue**
11: **end if**
12: **for** $Q_s \in S$ where $P(Q') \supseteq P(Q_s)$ and $\text{cost}(Q') \leq \text{cost}(Q_s)$ **do**
13: remove Q_s from S
14: **end for**
15: **if** Q' is *complete* **then**
16: $Q^* \leftarrow Q'$ if $\text{cost}(Q') < \text{cost}(Q^*)$
17: **else**
18: add Q' into S
19: **end if**
20: **end for**
21: **end while**

Figure 10. A* algorithm for sharding strategy search

In each loop of the algorithm, we retrieve the program Q with the lowest score from the priority queue S (Line 6) and find an instruction that can be appended to it (Line 7). We want the resulting program to have more properties than Q; otherwise, the resulting program would be strictly worse than Q because it contains more instructions (therefore a higher cost) but is not closer to a complete program. Therefore, we enumerate the Hoare triples in $\mathcal{T}$ and find instructions whose precondition is met but postcondition contains properties not in $P(Q)$ (Line 7), so appending the instructions to Q is guaranteed to produce a new program Q' that contains more properties than Q. Next, we check if there are programs that are strictly better than Q' and stops further constructing programs based on Q' if so (Lines 9 to 11). We also remove any programs in S that are strictly worse than Q' (Lines 12 to 14). Finally, if Q' is complete, we compare it with the current best complete program and replace the best program if Q' is better (Line 16). If Q' is not complete, we add it to S and proceed to next loop (Line 18).

The resulting program contains both computation operations and communication operations (as in instructions). The sharding strategy is implicitly included in the generated program when synthesizing communication operations and special operations like `Placeholder-Shard`. Following our semantic constraints, the generated collective communication is guaranteed to properly handle the tensors the computation produces. For example, $\texttt{All-Gather}(\bar{e_1}, 0)$ will only be generated upon a tensor $\bar{e_1}$ that is previously sharded on its 0-th dimension. Similarly, $\texttt{All-Reduce}(\bar{e_2})$ will only be generated if performing this operation produces a tensor that equals a tensor e_2 in the single-device graph.

We give an example of the searching process in Fig. 11. Suppose that the single-device program contains 4 instructions, as given on the top left in the figure. The `Placeholder` operation retrieves a batch of input samples. The `Parameter` operation loads a parameter tensor of the model. Then the matrix product of the two tensors are computed and its element sum is computed as the loss. Starting with an empty distributed program, assume that we find the following matching rule:

$$\frac{\forall e \in E,\ e = \texttt{Placeholder}()}{\{\,\emptyset\,\}\ \texttt{Placeholder-Shard}(0)\ \{\,e \mid \texttt{All-Gather}(0)\,\}}$$

whose precondition is met (as it has no properties). We append the instruction `Placeholder-Shard(0)` to the empty program and obtain program ①. Suppose there is no other rule with an empty precondition; ① is now the only element in the priority queue. For brevity, we only consider partitioning e_1 on the first dimension (dimension 0) and e_2 on its second dimension (dimension 1), and do not include communication optimizations in this example. In the second loop, we retrieve ① from the priority queue and append different instructions to it by enumerating $\mathcal{T}$, leading to programs ② and ③. In the third loop, we retrieve ② from the priority queue and append a `MatMul` operation and obtain program ④. Note that we remove the properties regarding e_1 and e_2 as the two tensors will no longer be used in the rest of the program. We will introduce details of this optimization in Sec. 4.5. Then in the fourth loop, we find ⑤ and ⑥ based on ③. In the fifth loop, we obtain program ⑦ which is a complete program as it has property $loss \mid \texttt{All-Reduce}$. Since its cost is no higher than the scores of ⑤ and ⑥ (the two programs in the priority queue), the search terminates and returns ⑦ as the optimal program.

4.4 Communication Optimization

With our program synthesis approach, we can readily incorporate the two communication optimization techniques (Sec. 2.5) into the distributed program search, to jointly optimize communication on heterogeneous clusters with the sharding strategy.

As discussed in Sec. 2.5.1, there can be two implementations of `All-Gather` on a heterogeneous cluster, which exhibit different performance under different sharding ratios. To automatically decide the better implementation under a given sharding ratio B, we can add the following rule during program search:

$$\frac{\forall e \in E,\ \forall d \in \dim(e)}{\{\,e \mid \texttt{All-Gather}(d)\,\}\ \texttt{Grouped-Broadcast}(\bar{e}, d)\ \{\,e \mid \texttt{Identity}\,\}}$$

The rule has the same precondition and postcondition as the `All-Gather` instruction in Fig. 9 (which refers to the padded `All-Gather` implementation), but indicates using multiple Broadcast operations to implement `All-Gather`. Whenever a partial program meets the precondition, our A* search will attempt both instructions and retain only the one with better estimated performance, using lines 9 to 14 in Fig. 10.

To support sufficient factor broadcasting (Fig. 5(c)), we only need to add the following rule in additional to those in Fig. 9:

$$\frac{\forall e_1, e_2, e_3 \in E,\ e_3 = \texttt{MatMul}(e_1, e_2)}{\{\,e_1 \mid \texttt{Identity},\ e_2 \mid \texttt{Identity}\,\}\ \texttt{MatMul}(\bar{e_1}, \bar{e_2})\ \{\,e_3 \mid \texttt{Identity}\,\}}$$

which denotes that all devices duplicate the same computation with identical input data. By applying this rule and the fifth rule in Fig. 9 to the single-device program in Fig. 5(a), which is inside the search space of our A* algorithm, we can generate the program depicted in Fig. 5(c). By adding similar rules to common operators in DNNs, *HAP*'s program synthesis process can automatically explore other possible applications of SFB.

4.5 Search Time Optimization

As the number of operations increases, the execution time of our A* algorithm may still be substantial. We further propose three heuristics to balance the search time and performance of the obtained sharding strategy.

Our first optimization involves fusing Hoare triples that have empty preconditions with their consumers. For instance, for a `Placeholder` operation in the single-device graph that produces reference tensor e, we may create a Hoare triple $\{\,\emptyset\,\}$ `Placeholder` $\{\,e \mid \texttt{Identity}\,\}$. Since it has an empty precondition, the code may appear at any position in the program before the first consumer of e, and our search algorithm would explore all possible positions of such instructions during the search. To reduce the overhead, we fuse those Hoare triples with their consumers to generate new Hoare triples with two consecutive instructions. Specifically, for two Hoare triples $\{\,Pre_1\,\}\ Inst_1\ \{\,Post_1\,\}$ and $\{\,Pre_2\,\}\ Inst_2\ \{\,Post_2\,\}$, if $Pre_1 = \emptyset$ and $Post_1 \subseteq Pre_2$, we remove the first triple from $\mathcal{T}$ and insert a new Hoare triple $\{\,Pre_2 \setminus Post_1\,\}\ Inst_1 \cup Inst_2\ \{\,Post_1 \cup Post_2\,\}$. This ensures that all instructions with empty preconditions occur directly before their first consumers and eliminates the enumeration of their positions in the program.

Our second optimization is to disallow repeated communications of the same reference tensor. We also disallow communication of tensors produced by `Placeholder` and `Parameter`, which can directly produce sharded tensors with specialized instructions, `Placeholder-Shard` and `Parameter-Shard`. Without this optimization, *HAP* attempts

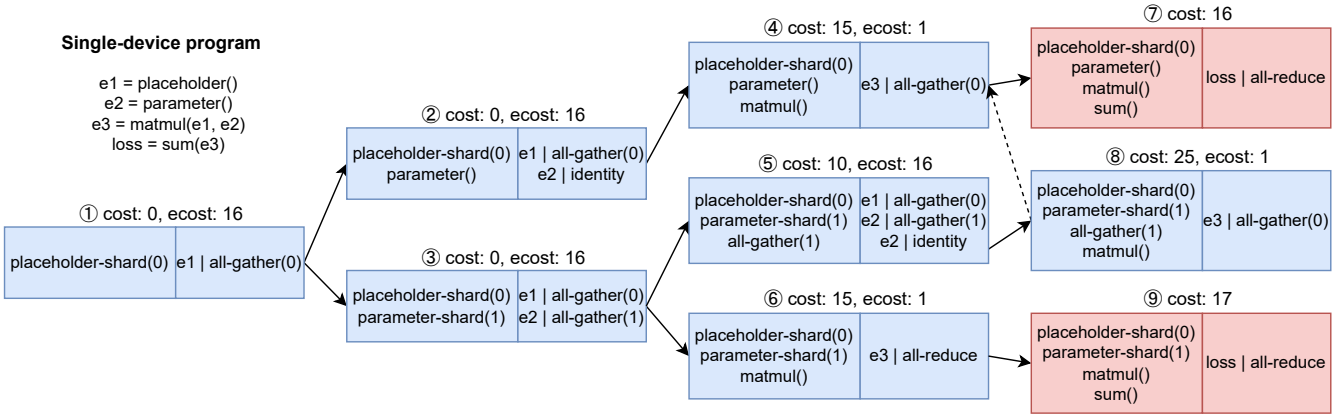

Figure 11. A* search example. Names of distributed tensors (e.g., $\bar{e}_1$) are omitted.

to append multiple communication instructions for each tensor because they introduce new properties to the program, even though most of these properties are not utilized. For a Hoare triple that generates a communication instruction of reference tensor e, we append a special property $e \mid \neg\texttt{Communicated}$ to its precondition and $e \mid \texttt{Communicated}$ to its postcondition. This makes communication instructions of the same reference tensor conflict with each other, so that at most one of them can appear in one distributed program.

Our third optimization is about removing redundant properties from partial programs to increase the number of programs we can prune in lines 9 to 14 in Fig. 10. A property is redundant to a partial program Q if it does not appear in the precondition of any instruction whose *postcondition* $\notin P(Q)$. For example, the property $e_1 \mid \texttt{All-Gather}(0)$ in the partial program ② only appears in the preconditions of two kinds of instructions: communication operations of e_1 and $\texttt{MatMul}(e_1, e_2)$. The former are not considered as a result of our second optimization. Therefore, after inserting the MatMul instruction to form ④, no instruction with $e_1 \mid \texttt{All-Gather}(0)$ in its precondition satisfies *postcondition* $\notin P(④)$, and we can safely remove this property from $P(④)$ without affecting the final result.

5 Load Balancing

We next detail our design of the load balancer that produces the optimal sharding ratios B for a fixed distributed program Q, i.e., solve $\arg\min_B t(Q, B)$.

5.1 A Base Case

We first consider a basic case where the same sharding ratios across the devices are used for each tensor in the DNN model. As a collective communication operation involves all devices and is bottlenecked by the slowest participant, the communication time in the i-th stage (Sec. 3.2), $\text{comm}^{(i)}(B)$, is decided by the largest communication time of a tensor shard, i.e., $\text{comm}^{(i)}(B)$ is a linear function of $\max_{j\in[m]}(B_j)$. Since we synchronize all devices at the beginning of each stage and the collective communication operations take the same time across devices (Fig. 6), the computation time of the i-th stage is the maximum computation time among the devices, i.e., $\max_j \text{comp}_j^{(i)}(B_j)$. We then solve the following problem to obtain B:

$$\min \sum_{i\in\text{stages}(Q)} (\text{comm}^{(i)}(B) + \max_{j\in[m]} \text{comp}_j^{(i)}(B_j)) \quad (3)$$
$$\text{subject to:} \quad \sum_{j=1}^{m} B_j = 1,$$
$$B_j \geq 0, \quad \forall j \in [m]$$

The objective function is the sum of the communication time and computation time of all stages, which is the per-iteration training time that we are minimizing. The constraints state that sharding ratios are non-negative and sum to 1. In *HAP*, the functions $\text{comm}^{(i)}$ and $\text{comp}_j^{(i)}$ are modeled as linear functions on bandwidth and flops (Sec. 3.2). Therefore, the optimization problem is a linear program and can be solved efficiently with off-the-shelf solvers.

After obtaining the optimal fractional solutions, we round the sharded sizes of each tensor to integers and ensure they add up to the total length of the dimension that is sharded on. We first set the sharded sizes to their nearest integers. If the sum is larger or smaller than the original size, we repeatedly reduce/increase the size by one for a shard that introduces smallest rounding errors, until the sizes of the sharded tensors sum to the original tensor.

5.2 Different Sharding Ratios across the Model

If the DNN model contains many layers and the computation-communication ratio differs across layers, using the same sharding ratios throughout the model may not be ideal. Due to the large number of tensors in a model, computing a different set of sharding ratios for each tensor may incur high computation overhead. Instead, we partition the tensors in the model, E, into g segments, denoted by E_k, $1 \leq k \leq g$, and identify the sharding ratios for each model segment.

The segment division can be either specified by the user (such as using the layers of the model) or determined using a partition algorithm such as METIS [20] (which minimizes the tensor size on the cuts while balancing the size of partitions). The sharding ratios B subsequently become a $g \times m$ matrix, where $B_{k,j}$ represents the sharding ratio for tensors in the k-th model segment on the j-th device.

For a tensor in the k-th segment that is produced by tensors from other segments (whose sharding ratios may not be B_k), an `All-To-All` operation is inserted to coordinate between the sharding ratios. To simplify implementation, we always insert `All-To-All` operations at the boundaries of segments, regardless of whether the sharding ratios are the same or not between the two segments. As a result, there are synchronization points at segment boundaries, and each stage is entirely within a model segment. In this way, for each segment, we can solve the optimization problem in (3) independently to determine the optimal sharding ratios for tensors in that segment.

6 Implementation

HAP is implemented on PyTorch 1.13.1 [33] with 2789 lines of Rust code for the program synthesizer, 69 lines of Rust code for the load balancer, and 362 lines of Python code for the profiler, collective operations, and the user API. Fig. 12 shows the modules we implemented in *HAP*. The single-device program is represented as a PyTorch fx [37] graph. The cluster specification contains the information of the virtual devices (GPUs and machines), including the profiled flops-per-second of the devices and the latency and bandwidth of each collective primitives on this cluster. The program synthesizer (Sec. 4) and the load balancer (Sec. 5) are run on CPU to identify the optimal distributed program Q and sharding ratio B. We use CBC [13] to solve the sharding ratio optimization problem.

At the begining of model training, *HAP* broadcasts Q and B to all workers (virtual devices), which run them on the PyTorch runtime. Each worker first initializes the original single-device model in CPU using the same seed. For each `Parameter-Shard`(d) operation in Q, the j-th worker shards the corresponding parameter along its d dimension and only keeps the slice corresponding to the portion of $\sum_{x=1}^{j-1} B_x$ to $\sum_{x=1}^{j} B_x$. The sharded parameters are loaded to GPU for training.

In each training iteration, the workers each load a mini-batch of input data according to their sharding ratios. Then they execute Q and synchronize with each other when executing collective communications. After running Q, each worker applies the gradient to its own parameter shards. The collective communication operations are implemented using PyTorch's API with the NCCL [18] backend.

When *HAP* is run on a virtual device which represents a machine, program Q sent to the machine is replicated to all GPUs in the machine. Regular data-parallel training is carried out among GPUs in the machine. Each collective operation in Q is replaced by a three-step communication operation: the tensors for communication are first aggregated from all GPUs to GPU 0 using `Gather` or `Reduce`; GPU 0 participants in the global collective communication using the aggregated tensor; and then GPU 0 broadcasts the result to other GPUs in the machine using `Scatter` or `Broadcast`.

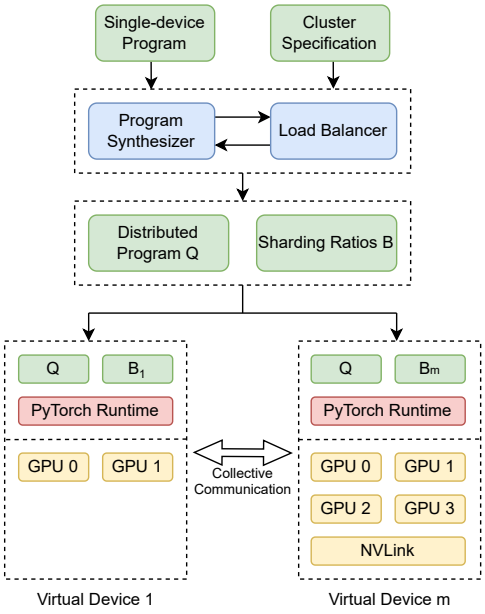

Figure 12. *HAP* Implementation.

The user API of *HAP* is analogous to the built-in DDP module of PyTorch: the user calls `hap.HAP` function with a single-device PyTorch model and a Python Dict of device specification, and the function returns a distributed model that can run on the cluster with PyTorch's `torch.distributed` module. We plan to open-source *HAP* to the community.

7 Evaluation

7.1 Experimental Setup

Testbed. We conduct experiments on 8 machines in a public cloud with 64 GPUs in total. Two machines are each equipped with 8 NVIDIA V100 GPUs and NVLink. The others are each equipped with 8 NVIDIA P100 GPUs. Inter-machine bandwidth is about 10.4Gbps, as measured with iperf3 [12]. The cluster provides network isolation and stable bandwidths.

Benchmarks. We train 4 representative DNN models as listed in Table 1. VGG19 [42] is a convolutional neural network (CNN) for image classification. ViT[10] is a Transformer-based neural network for image classification. BERT-Base [9] is a Transformer-based model for language modeling. Bert-MoE adds MoE layers to the BERT-Base model by replacing a feed-forward module every two layers in a similar way as in GShard [22]. We follow the convention of scaling MoE models with the number of devices, thus the model size

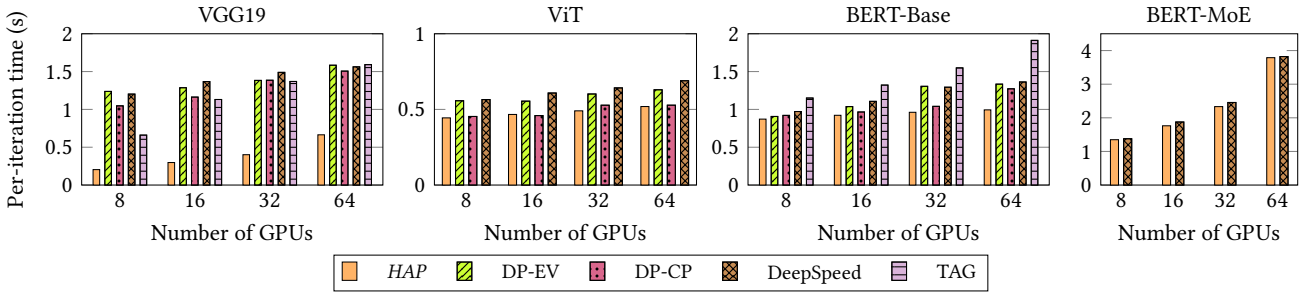

Figure 13. Per-iteration training time on heterogeneous clusters.

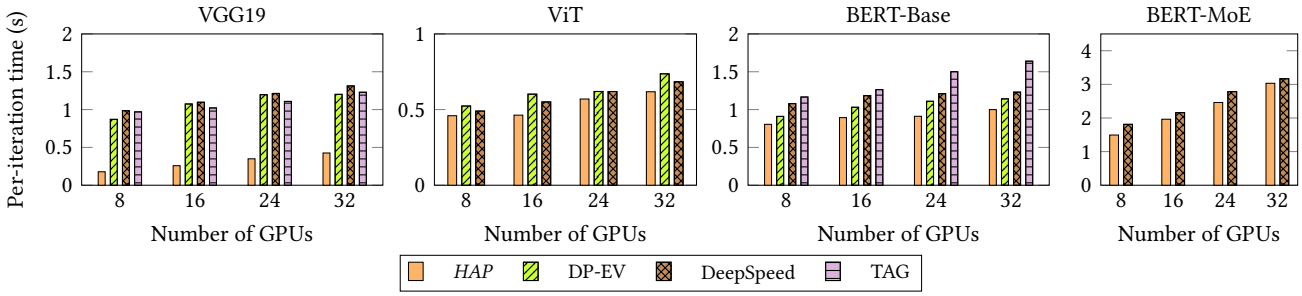

Figure 14. Per-iteration training time on homogeneous clusters.

Table 1. Benchmark models

Model	Task	Parameters (Millions)
VGG19[42]	Image Classification	133
ViT[10]	Image Classification	54
BERT-Base[9]	Language Model	102
BERT-MoE[9]	Language Model	84 + 36m

depends on the number of devices m. We adopt weak scaling and set the global batch size proportional to the number of devices, with per-device batch size 32 for BERT-MoE and 64 for other models.

We use Cifar-10 [21] dataset for image classification tasks and WikiText-2 [26] dataset for language modeling tasks.

Baselines. We compare *HAP* with four relavent designs: (1) *DP-EV* is data parallelism with even sharding ratios. (2) *DP-CP* is data parallelism with sharding ratios proportional to the computation speed of the devices. We use PyTorch's DDP module [24] to implement DP-EV and DP-CP. (3) *DeepSpeed* [36] supports ZeRO-based [35] data parallelism and implements intra-op model parallelism for MoE layers. (4) *TAG* [56] is a heterogeneity-aware DNN training system. TAG supports data parallelism and inter-op model parallelism. It optimizes communication by selecting parameter-server [23] or `All-Reduce` for gradient synchronization and automatically applying sufficient factor broadcasting.

HAP, DP-EV, DP-CP, and DeepSpeed are based on PyTorch and use the same implementation of the benchmark models. TAG is implemented on TensorFlow [1]. We were only able to train VGG19 and BERT-Base with TAG. Due to replicating the whole model on all devices, DP-CP and DP-EV causes out-of-memory errors when training BERT-MoE.

7.2 Training Speed-up on Heterogeneous Clusters

We first evaluate *HAP* and the baselines on the heterogeneous cluster with 8 machines. As shown in Fig. 13, *HAP* significantly outperforms the DP baselines when training VGG19. VGG19 comprises layers of different computation-to-communication ratios. In particular, the fully-connected layers in VGG19 is very communication-intensive as compared to the convolution layers. *HAP* adopts model parallelism to reduce the communication time and achieves up to 2.41x speedup in the case of 32 GPUs. TAG puts these layers exlusively on one device in the case of 8 GPUs to eliminate communication. However, this method does not work with more GPUs. *HAP* achieves similar performance to DP-CP when training ViT while consistently outperforming the baselines when training BERT-Base, with a 28% speedup in the case of 64 GPUs. When training BERT-MoE, *HAP* finds a strategy that performs similarly to the expert-designed MoE sharding strategy implemented in DeepSpeed.

7.3 Training Speed-up on Homogeneous Clusters

In this experiment, we assess the performance of *HAP* and baselines on a homogeneous subset of our testbed consisting of 4 machines, each equipped with 8 P100 GPUs. Since all devices have the same computational power, DP-CP is equivalent to DP-EV and therefore is not included in this experiment. As demonstrated in Fig. 14, *HAP* still outperforms all baselines across all models, achieving up to 217%, 19%, 22%, and 13% speedup when training VGG19, ViT, BERT-Base, and BERT-MoE, respectively, compared to the best baselines.

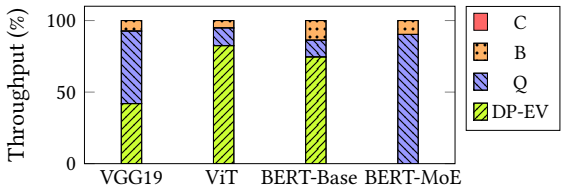

Figure 15. Ablation study.

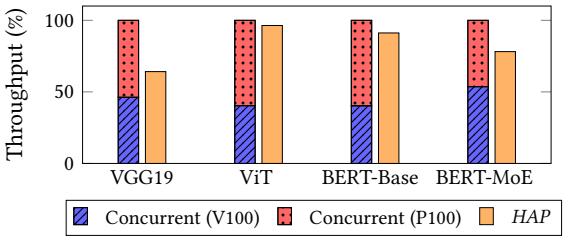

Figure 16. Training multiple models.

7.4 Ablation Study

We examine the efficacy of various components of *HAP* by comparing the throughput of benchmark models achieved through the utilization of different parts of our designs. In Fig. 15, DP-EV represents the throughput achieved without any of our designs. "Q" denotes the additional throughput obtained by employing *HAP*'s program synthesizer. "B" represents the throughput contributed by our load balancer, and "C" is the speedup provided by communication optimization. The findings indicate that the program synthesizer has the greatest impact on the performance of *HAP*, whereas the communication optimization does not yield noticeable speedup in this experimental setup. This can be attributed to the relatively small disparity in computational power between the GPUs. As discussed in Sec. 2.5, the communication optimization is mostly effective when there is a significant difference in sharding ratios between devices.

7.5 Case Study: Training Multiple Models

Hardware heterogeneity presents inherent challenges for distributed DNN training. Even with the optimizations of *HAP*, it is anticipated that there will be reduced hardware utilization on heterogeneous clusters. We estimate this overhead by simultaneously training multiple models on homogeneous subsets of the cluster and use the total throughput as an estimation of the potential throughput achievable if the cluster were homogeneous. Specifically, in this experiment, we train one model using two V100 machines while simultaneously training another model using six P100 machines. We refer to this approach as *concurrent*. We then compare the total throughput achieved using *concurrent* with that of *HAP*, as shown in Fig. 16. We normalize the throughput of different models by comparing them to the total throughput achieved by *concurrent*. Our results show that *HAP* achieves 64% to 96% throughputs compared to *concurrent* for the benchmark models. The VGG19 model exhibits suboptimal utilization of GPU resources when scaled to accommodate a higher number of devices due to its relatively small convolution layers. As we scale the MoE model with the number of devices, the BERT-MoE model trained with *HAP* is larger than the two models trained using the *concurrent* method. The results show that *HAP* facilitates the training of larger models that may exceed the capacity of homogeneous subsets while maintaining satisfactory throughput on heterogeneous clusters. Given the current trend of large models, there is a growing demand for the ability to utilize all available resources to train models of maximum size. Moreover, *HAP* enables the prioritization of time-sensitive training tasks by fully leveraging all resources for their execution. For instance, with *HAP*, users can employ the entire cluster to train a production model before lower-priority research jobs. On the other hand, *concurrent* limits the training of the production model to a homogeneous sub-cluster, leading to increased latency.

7.6 Case Study: Uneven Placement of Experts

Expert parallelism, which partitions MoE layers on the expert dimension, is the dominating training strategy for MoE models. Current training systems that adopt expert parallelism allocate the same number of experts to all devices [14, 22, 34]. If the number of devices does not evenly divide the number of experts, the related tensors must be first padded, resulting in inefficient use of computation power. With uneven partitioning, *HAP* naturally supports sharding MoE models with any number of experts onto a cluster with any number of devices, as long as the total memory capacity is sufficient.

To demonstrate the effectiveness of *HAP*, we conduct an experiment using two machines, one with 2 NVIDIA A100 GPUs and the other with 2 NVIDIA P100 GPUs. We train BERT-MoE with varying numbers of experts using *HAP* and DeepSpeed. To maintain the same load of each expert, we keep the number of tokens proportional to the number of experts. The per-iteration training time is plotted in Figure 17. DeepSpeed has to pad the number of experts to a multiplier of 4, the number of available devices, while *HAP* can provide a smooth performance curve. Further, *HAP* places more experts onto A100 GPUs to maximumly exploiting its computation power, bringing up to 64% speedup.

7.7 Cost Model Accuracy

HAP employs a cost model (Sec. 3.2) to evaluate distributed program Q and sharding ratios B during the optimization loop. The accuracy of the cost model is critical in obtaining the optimal distributed programs. In this experiment, we alter the configurations (the number of layers, hidden width, and sequence length) of the BERT-Base model to create different variants and compare the cost estimated by our cost model and the actual profiled per-iteration training time. As shown in Fig. 18, the cost model tends to under-estimate the training

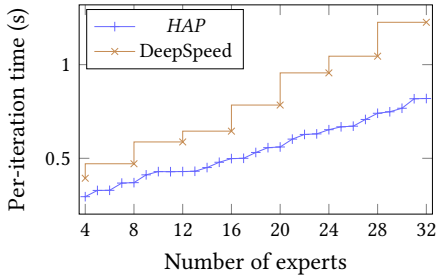

Figure 17. BERT-MoE performance with uneven placement of experts.

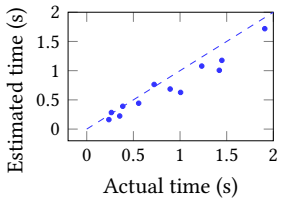

Figure 18. Cost model accuracy.

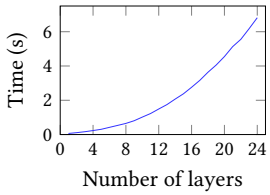

Figure 19. Program synthesis time.

time, but the estimated time is mostly linear to the actual time, with a Pearson correlation coefficient of 0.970.

7.8 Overhead of *HAP*

HAP adopts SPMD parallelism and exhibits constant search and compiling time with respect to the number of devices. Therefore, we assess the overhead of *HAP* by varying model scales. We adjust the number of layers of the ViT model and generate a distributed model with *HAP*. The sharding ratio optimization takes less than 1ms and the majority of overhead is attributed to the program synthesis process. As shown in Fig. 19, the program synthesis time increases superlinearly as the number of layers increases. Nevertheless, for a model with 24 layers, *HAP* only takes a few seconds to synthesize the distributed program, which is negligible compared to the hours or even days of model training time.

8 Related Work

Inter-Operator Parallelism. Placing different parts of the DNN model on different devices allows distributed training on heterogeneous clusters [2, 27, 59]. Pure inter-operator parallelism falls short when a single operator in a model exceeds the memory capacity of a single device, like in the increasingly common MoE models. Inter-operator parallelism may not scale well as each device is treated individually and the decision space grows with the number of devices.

Unevenly-split Data Parallelism. VirtualFlow [31] splits a mini-batch into *virtual nodes* and assigns multiple virtual nodes to a single device. HeteroG [53] uses graph neural networks and reinforcement learning to find the placement and communication strategy for each operator in a heterogeneous cluster, supporting both inter-operator parallelism and unevenly-split data parallelism. Data parallelism does not support large operations that do not fit in a single device.

Asynchronous Data-Parallel Training. HetPipe [32] divides the heterogeneous cluster into k virtual workers; each virtual worker employs pipeline parallelism internally, while asynchronous data parallel training is carried out among virtual workers using a parameter-server architecture. Prague [25] adopts partial all-reduce with only a subset of workers participating in parameter synchronization of each training iteration. Devices with different speeds synchronize with other devices at different paces. *HAP* focuses synchronous training which achieves the same model convergence as single-device training.

Heterogeneous SPMD Systems. AccPar [43] uses dynamic programming to decide tensor partitioning among heterogeneous devices, but only considers three types of partitioning for operators in CNN models. Pathways [5] places model components on different TPU pods and uses gang scheduling for asynchronously execution of these components. *HAP* systematically explores more sharding strategies with program synthesis.

Collective Communication on Heterogeneous Clusters. TACCL [39] models communication as an mixed integer linear programming problem and finds routing and scheduling of each data chunk to minimize communication. BlueConnect [8] decomposes `All-Reduce` to fit into heterogeneous network hierarchy. *HAP* uses NCCL as the communication library and automatically chooses communication primitives during program synthesis. *HAP* may be used together with the heterogeneity-aware communication optimizations to further accelerate training on heterogeneous clusters.

9 Conclusion

This paper introduces *HAP*, an automated system for SPMD-parallel training of large neural networks on heterogeneous clusters. *HAP* novelly synthesizes a distributed program on a distributed instruction set that emulates the single-device program, and identifies the best sharding strategy and communication methods in the distributed program. Tensor sharding ratios are optimally set to balance the workload across devices, through iterative optimization with the distributed program synthesis. We implement *HAP* using PyTorch and demonstrate that it achieves up to 2.41x faster training compared to existing methods on heterogeneous clusters and can automatically find feasible SPMD strategies to train large models.

Acknowledgments

This work was supported in part by Alibaba Group through Alibaba Innovative Research (AIR) Program and grants from Hong Kong RGC under the contracts HKU 17208920 and C7004-22G (CRF).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*.

[2] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv preprint arXiv:1906.08879* (2019).

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis*. IEEE.

[4] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438* (2017).

[5] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. 2022. Pathways: Asynchronous distributed dataflow for ML. *arXiv preprint arXiv:2203.12533* (2022).

[6] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[7] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 571–582.

[8] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. 2019. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *Proceedings of Machine Learning and Systems* 1 (2019), 241–251.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[11] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. 2023. PaLM-E: An Embodied Multimodal Language Model. *arXiv preprint arXiv:2303.03378* (2023).

[12] Jon Dugan, Seth Elliott, Bruce A Mah, Jeff Poskanzer, and Kaustubh Prabhu. [n. d.]. iperf3, tool for active measurements of the maximum achievable bandwidth on ip networks. *URL: https://github.com/esnet/iperf* ([n. d.]).

[13] John Forrest, Ted Ralphs, Stefan Vigerske, Lou Hafer, Bjarni Kristjansson, JP Fasano, E Straver, M Lubin, HG Santos, R Lougee, et al. 2018. coin-or/Cbc: Version 2.9. 9. *URL http://dx.doi.org/10.5281/zenodo* 1317566 (2018).

[14] Jiaao He, Jiezhong Qiu, Aohan Zeng, Zhilin Yang, Jidong Zhai, and Jie Tang. 2021. FastMoE: A Fast Mixture-of-Expert Training System. *arXiv preprint arXiv:2103.13262* (2021).

[15] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[16] Yi Hu, Chaoran Zhang, Edward Andert, Harshul Singh, Aviral Shrivastava, James Laudon, Yanqi Zhou, Bob Iannucci, and Carlee Joe-Wong. 2023. GiPH: Generalizable Placement Learning for Adaptive Heterogeneous Computing. *Proceedings of Machine Learning and Systems* 5 (2023).

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019), 103–112.

[18] Sylvain Jeaugey. 2017. Nccl 2.0. In *GPU Technology Conference (GTC)*, Vol. 2.

[19] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[20] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[21] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[22] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[23] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[24] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[25] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-performance heterogeneity-aware asynchronous decentralized training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 401–416.

[26] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[27] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A hierarchical model for device placement. In *International Conference on Learning Representations*.

[28] MPI Forum. 1994. MPI: A message-passing interface standard.

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[30] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 481–498.

[31] Andrew Or, Haoyu Zhang, and Michael None Freedman. 2022. VirtualFlow: Decoupling Deep Learning Models from the Underlying Hardware. *Proceedings of Machine Learning and Systems* 4 (2022).

[32] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. {HetPipe}: Enabling Large {DNN} Training on (Whimpy) Heterogeneous {GPU} Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 307–321.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style,

high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[34] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International Conference on Machine Learning*. PMLR, 18332–18346.

[35] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[36] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[37] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch. fx: Practical Program Capture and Transformation for Deep Learning in Python. *Proceedings of Machine Learning and Systems* 4 (2022).

[38] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. 2017. Optimizing big-data queries using program synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 631–646.

[39] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2021. Synthesizing collective communication algorithms for heterogeneous networks with taccl. *arXiv preprint arXiv:2111.04867* (2021).

[40] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[42] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[43] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2020. Accpar: Tensor partitioning for heterogeneous deep learning accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 342–355.

[44] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2019. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 56–68.

[45] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating {DNN} Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 267–284.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[47] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[48] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing recursive queries with progam synthesis. In *Proceedings of the 2022 International Conference on Management of Data*. 79–93.

[49] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[50] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 945–960.

[51] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric Xing. 2015. Distributed machine learning via sufficient factor broadcasting. *arXiv preprint arXiv:1511.08486* (2015).

[52] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. *arXiv preprint arXiv:2105.04663* (2021).

[53] Xiaodong Yi, Shiwei Zhang, Ziyue Luo, Guoping Long, Lansong Diao, Chuan Wu, Zhen Zheng, Jun Yang, and Wei Lin. 2020. Optimizing distributed training deployment in heterogeneous GPU clusters. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 93–107.

[54] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 181–193.

[55] Shiwei Zhang, Lansong Diao, Chuan Wu, Siyu Wang, and Wei Lin. 2022. Accelerating large-scale distributed neural network training with SPMD parallelism. In *Proceedings of the 13th Symposium on Cloud Computing*. 403–418.

[56] Shiwei Zhang, Xiaodong Yi, Lansong Diao, Chuan Wu, Siyu Wang, and Wei Lin. 2023. Expediting Distributed DNN Training With Device Topology-Aware Graph Deployment. *IEEE Transactions on Parallel and Distributed Systems* 34, 4 (2023), 1281–1293.

[57] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. 2020. {HiveD}: Sharing a {GPU} cluster for deep learning with guarantees. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 515–532.

[58] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. 2022. Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning. *arXiv preprint arXiv:2201.12023* (2022).

[59] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. 2019. Gdp: Generalized device placement for dataflow graphs. *arXiv preprint arXiv:1910.01578* (2019).

A Artifact Appendix

A.1 Abstract

HAP is implemented as a Python module that automatically transforms a single-device tensor program into a distributed program that can efficiently run on heterogeneous clusters. The artifacts include the source code of *HAP* and a Docker container that bundles all dependencies.

A.2 Description & Requirements

A.2.1 How to access. The source code of *HAP* is provided at https://github.com/ylxdzsw/hap. The "ae" branch contains the version for artifact evaluation. A docker container with all dependencies pre-installed is available at https://hub.docker.com/r/ylxdzsw/hap.

A.2.2 Hardware dependencies. At least two GPUs are required to run *HAP*. As *HAP* is designed for heterogeneous clusters, multiple machines with different GPU models are needed to fully show *HAP*'s capabilities. The minimum GPU memory should be at least 12GB. We recommend a similar setting as used in our experiments (Sec. 7.1), i.e., 2 machines each equipped with 8 NVIDIA V100 GPUs and 6 machines each equipped with 8 NVIDIA P100 GPUs. The inter-machine bandwidth is about 10.4Gbps.

A.2.3 Software dependencies. *HAP* is implemented on PyTorch 1.13.1. All machines should use the same versions of CUDA and NVIDIA drivers that are compatible with PyTorch 1.13.1. Rust 1.70.0-nightly and Coin CBC 2.9.9 are required to build *HAP* from source. To reproduce the results of the baselines, DeepSpeed 0.9.4 is also used.

All software dependencies are included and pre-compiled in the Docker image. However, NVIDIA driver 515.43.04 needs to be separately installed on the host machines.

A.2.4 Benchmarks. The benchmark models and datasets are included in the source code repository and Docker image.

A.3 Set-up

This set-up instruction uses the Docker image. To build *HAP* from source, we refer to the "readme" file in the source code repository.

First, ensure that NVIDIA driver 515.43.04 or higher has been installed on the host machines. The installation can be verified with the `nvidia-smi` command. All machines should use the exact same version of NVIDIA driver. The driver can be installed by following https://docs.nvidia.com/datacenter/tesla/tesla-installation-notes/index.html.

Next, install Docker engine by following https://docs.docker.com/engine/install. Then install `nvidia-container-toolkit` (e.g., with `apt-get install`) and restart the docker daemon (`systemctl restart docker`). After that, download the Docker image of *HAP* using `docker pull ylxdzsw/hap:ae`. The image is about 20GB. When finished, start a container instance with `docker run -d --shm-size="10.24gb" --name hap --gpus all --network host -it ylxdzsw/hap:ae /bin/bash`. To access the container on the host machine, run `docker exec -it hap bash`. Inside the container, run `/usr/sbin/sshd` to start an `ssh` instance on port 3922 which will later be used for communication between the containers.

Running HAP involves running the same script on all machines in the cluster. To automate this process, we provide a helper script `/root/hap/run_all`. Running this script on one of the machines starts the same script on all machines. By default it assumes 8 machines with host names `v1`, `v2`, ..., `v8`. The IP addresses of the machines are set in `/root/.ssh/config`. Before using the script, first enter `v1` and run ssh from the `v1` to all machines (including `v1` itself) with `ssh -p 3922 root@vx` and save the ssh fingerprints. Ensure that `v1` can access all workers without further interactions such as confirming fingerprints or typing passwords. When testing HAP on a single machine, one may edit `/root/hap/run_all` to keep only the line with `v1` and edit `/root/.ssh/config` to set the ip address of `v1` to 127.0.0.1.

Finally, check the set-up by running `./run_all worker.py 1` on the `v1`. It should run 100 iterations of training and reports the average per-iteration time.

A.4 Evaluation workflow[1]

A.4.1 Major Claims.

- *(C1)*: *HAP* outperforms the baselines in heterogeneous clusters in terms of the per-iteration training time when training the benchmark models. This is proven in Sec. 7.2 and the results are shown in Fig. 13.
- *(C2)*: *HAP* outperforms the baselines in homogeneous clusters in terms of the per-iteration training time when training the benchmark models. This is proven in Sec. 7.3 and the results are shown in Fig. 14.
- *(C3)*: *HAP* can generate distributed models within seconds for the benchmark models, as shown in Sec. 7.8 and Fig. 19.

A.4.2 Experiments. *Experiment (E1): [Heterogeneous Cluster] [30 human-minutes + 4 compute-hours]*: Train the benchmark models on a heterogeneous cluster and compare the per-iteration training time of *HAP* and the baseline systems.

[Preparation]

Assuming that *HAP* has been set up on a heterogeneous cluster following Sec. A.3, this experiments involves modifying `config.py` and running *HAP* and the baselines.

First, we need to collect profiling data. The device flops can be profiled by running `python profiler.py`. Execute this command for each type of GPU and replace `device_flops` in `worker.py` with the actual profiling data. `device_flops` is an array of the flops for all devices. For example, when using 2

[1]Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://sysartifacts.github.io/eurosys2024/.

V100 GPUs and 6 P100 GPUs, it should be set to an array of 8 elements, with the first two elements being the profiled flops of the V100 GPU and the last 6 elements being the profiled flops of the P100 GPU. The collective communication can be profiled by running `./run_all profiler.py 8`, which automatically runs different collective operators across all machines using 8 GPUs (the second argument to the script) on each machine. Fill the profiling data in `worker.py`.

Next, modify `config.py` and run `./run_all worker.py k` to obtain the per-iteration training time of *HAP*, where `k` is the number of GPUs to use on each machine. In `config.py`, `model_name` is the benchmark model, where `Vvgg`, `Vtransformer`, `Rtransformer` and `Rmoe` correspond to the VGG19, ViT, BERT-Base, and BERT-MoE models. `world_size` is the total number of GPUs. `master_addr` should be set to the ip address of one of the machines. `cards_per_node` is only used by the DeepSpeed baseline and should be set to the number of GPUs to use on each machine (same as `k`). Other fields should be kept unchanged to reproduce the results reported in the paper.

To run the DP-EV baseline, change the `unscaled_sharding_lengths` in `ddp.py` to an array of 1 (simulating the same device flops on each device regardless of their actual types) and run `./run_all ddp.py k` similar to running *HAP*. To run the DP-CP baseline, fill `unscaled_sharding_lengths` with the actual profiling data of each GPU type in the same way as `device_flops` in `worker.py`.

To run the DeepSpeed baseline, use `./run_all_deepspeed` instead of `./run_all`.

[Execution]

To collect the data for Fig. 13, vary `k` and the related fields in `config.py` (`model_name`, `world_size`, and `cards_per_node`), then run *HAP* and the baselines for each configuration.

[Results]

The experiment scripts print the average per-iteration time and the standard deviation on screen. As the standard deviation is relatively small in our experiments, we only report the average per-iteration time in Fig. 12. The experiment script also records the timeline in `trace.json.gz`, which can be load into Chrome Trace Profiling Tool for further inspection. The results should confirm the claim C1.

Experiment (E2): [Homogeneous Cluster] [30 human-minutes + 4 compute-hours]: Train the benchmark models on a homogeneous cluster and compare the per-iteration training time of *HAP* and the baseline systems.

[Preparation]

The preparation is same as in E1, except for that we now use a homogeneous cluster.

[Execution]

Same as in E1.

[Results]

Same as in E1. The results should confirm the claim C2.

Experiment (E3): [Overhead] [5 human-minutes + 5 compute-minutes]: Evaluate the time required by *HAP* to generate a distributed program.

[Preparation]

This experiment requires only one machine and can run without GPUs. Set `model_name` in `config.py` to `Vtransformer` for the ViT model and vary the `nlayers` field to experiment with models of different number of layers.

[Execution]

Run `python master.py`. This script compiles the model without actually running it.

[Results]

The compile time is printed on the screen. The results should confirm the claim C3.

A.5 Notes on Reusability

HAP can be extended to support new operators and custom semantic rules. To add support for a new operator, one need to edit the file `hap.rs` and add a new handler in the function `initialize_parsing_handlers` following the same structure of the existing handlers. To add new rules for generating Hoare triples, edit the `analyze_rgraph` function in `hap.rs`. The existing rules in the code can be used as examples.