SYNTHESIZING OPTIMAL PARALLELISM PLACEMENT AND REDUCTION STRATEGIES ON HIERARCHICAL SYSTEMS FOR DEEP LEARNING

Ningning Xie [1] **Tamara Norman** [2] **Dominik Grewe** [2] **Dimitrios Vytiniotis** [2]

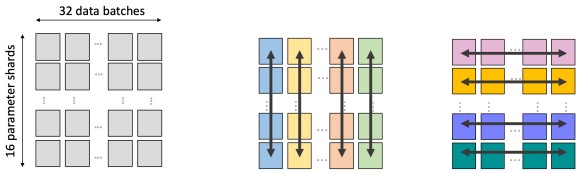
ABSTRACT

We present a novel characterization of the mapping of multiple parallelism forms (e.g. data and model parallelism) onto hierarchical accelerator systems that is hierarchy-aware and greatly reduces the space of software-to-hardware mapping. We experimentally verify the substantial effect of these mappings on all-reduce performance (up to 448×). We offer a novel syntax-guided program synthesis framework that is able to decompose reductions over one or more parallelism axes to sequences of collectives in a hierarchy- and mapping-aware way. For 69% of parallelism placements and user requested reductions, our framework synthesizes programs that outperform the default all-reduce implementation when evaluated on different GPU hierarchies (max 2.04×, average 1.27×). We complement our synthesis tool with a simulator exceeding 90% top-10 accuracy, which therefore reduces the need for massive evaluations of synthesis results to determine a small set of optimal programs and mappings.

1 INTRODUCTION

To facilitate efficient training of large-scale deep learning models, numerous parallelism techniques have been successfully employed. Common forms of parallelism include *data parallelism* (Krizhevsky et al., 2012), where each device has a copy of the full model to process a portion of the training data, and *model parallelism* (Dean et al., 2012), which partitions a training model over available devices, such as *parameter sharding* (Shoeybi et al., 2020). More recent studies explore combinations of parallelism forms to maximize training throughput (Jia et al., 2019; Narayanan et al., 2021), where each form of parallelism is referred to as a *parallelism axis*.

While the aforementioned forms of parallelism and their combinations have greatly improved training throughput, they may still incur significant *communication cost*. For example, in the simplest form of data parallelism, parameter gradients for each device must be reduced and replicated for each iteration (Amodei et al., 2016), which is typically implemented using the *collective operation* AllReduce (Thakur et al., 2005). State-of-the-art parameter sharding for transformers (Shoeybi et al., 2020) introduces sharded layers where each involves several AllReduce operations. Communication overhead is especially important for distributed deep learning, as the more devices we have, computation time reduces, and the communication cost becomes more prominent (Sergeev & Balso, 2018; Goyal et al., 2018).



(a) Combining parameter sharding and data parallelism

(b) Reduction along the axis of parameter sharding

(c) Reduction along the axis of data parallelism

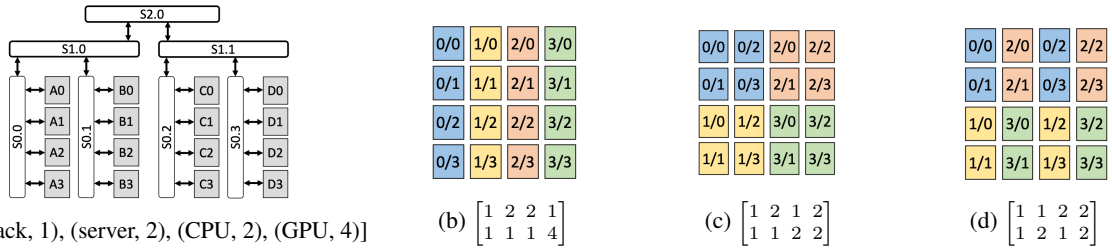
Figure 1: Parallelism combination

To reduce communication overhead, one particular challenge posed by multiple parallelism axes is *parallelism placement*. That is, how we map parallelism over devices decides which devices communicate with each other along each parallelism axis, and therefore decides the communication overhead. For example, Figure 1a presents a combination of parameter sharding and data parallelism, for which reduction along the axis of parameter sharding (or data parallelism), referred to as the *reduction axis*, is shown in Figure 1b (or Figure 1c, respectively). Now, suppose we map each box in the figure to devices. In that case, different mappings correspond to different reduction device groups, which can have a significant impact on the communication overhead depending on the network topology.

In this work, we present P^2, a tool for parallelism placement and placement-aware synthesis of recduction strategies. In particular, we offer the following contributions:

[1]University of Cambridge [2]DeepMind. Correspondence to: Ningning Xie <ningningxie@cl.cam.ac.uk>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022.



(a) [(rack, 1), (server, 2), (CPU, 2), (GPU, 4)]

(b) $\begin{bmatrix} 1 & 2 & 2 & 1 \\ 1 & 1 & 1 & 4 \end{bmatrix}$ (c) $\begin{bmatrix} 1 & 2 & 1 & 2 \\ 1 & 1 & 2 & 2 \end{bmatrix}$ (d) $\begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 2 & 1 & 2 \end{bmatrix}$

Figure 2: (a): A system. (b), (c), (d): Possible (non-exhaustive) parallelism placements for (a) under data parallelism of size 4 and 4 parameter shards. For clarity, we show only the 16 GPUs but omit interconnects. Device marker n/m indicates data batch n and parameter shard m.

- *Parallelism placement synthesis*: Given the parallelism axes, the reduction axes, and a hierarchical system topology, P^2 automatically synthesizes *hierarchical* parallelism placements, where a parallelism placement is modelled as a *parallelism matrix* mapping from parallelism axes to the system hierarchy (Section 3.1). The notion of parallelism matrices greatly reduces the space of parallelism placements contrary to a naive implementation.

- *Reduction strategy synthesis*: For each parallelism placement, P^2 utilizes the system hierarchy to further synthesize a wide variety of *reduction strategies* to implement reductions using common collective operations. To achieve this, we introduce: (a) a formal semantics for collectives (Section 3.2) based on *Hoare* triples (Hoare, 1969); (2) a domain-specific language (DSL) that can express possibly simultaneous reductions amongst groups of devices based on the system hierarchy (Section 3.3); and (b) a lowering of our DSL into sequences of collective operations. We use the formal semantics to guide a syntax-directed synthesis procedure on our DSL.

- *Synthesis hierarchy*: We show how the parallelism matrix, which determines a candidate parallelism placement, can be put to good use by the synthesizer to massively reduce the space of programs considered *without missing any semantically valid* programs – provably (Section 3.4).

- *Evaluation*: We evaluate the parallelism matrices and reduction strategies synthesized by P^2 on two different GPU systems available on Google Cloud Platform (GCP) (Section 4). We use collective operations as implemented by NVIDIA's NCCL communication library (NVidia, 2021), exposed through XLA. The evaluation demonstrates (1) the impact of parallelism placement: the performance of a single AllReduce across different parallelism matrices differs up to 448.5×; and (2) the effectiveness of custom reduction strategies: for 69% of all parallelism mapping matrices, a synthesized reduction outperforms AllReduce with up to 2.04× speedup (average 1.27×).

- *Simulation*: P^2 synthesizes all mapping and hierarchy-aware reduction strategies, but evaluating hundreds or thousands of them to identify the best can be expensive. We therefore introduce a simulator for predicting the end-to-end performance of a parallelism matrix and reduction strategy (Section 5). The simulator is aware of the network topology including different bandwidths for different interconnects and networks (e.g., NVLink and ethernet / data-center network in GPU topologies), predicting with reasonable accuracy the communication overhead for each parallelism placement and reduction strategy. The validation – over all mappings and synthesized programs for each mapping, and for each of the two GPU systems we considered – demonstrates that the simulator has 52%, 72%, and 92% of top-1, top-5 and top-10 accuracy, respectively, making it practical for identifying a much smaller subset of programs for actual evaluation.

P^2 is helpful for ML practitioners to speed up their models by improving placement and synthesizing reduction strategies tailored to their system hierarchies. For instance, we have used P^2 to improve ResNet-50 (He et al., 2016) data-parallel training by 15% across 4 nodes, each with 8 V100 GPUs. (See Section 4 for the details of this system.)

2 Overview

This section outlines the key design in P^2. First, a *system* consists of two entities: (1) a hardware hierarchy, where each level has a name and a cardinality; and (2) a set of switched interconnects. The system hierarchy is expected to reflect how devices are arranged. Figure 2a describes an example system with 16 GPUs (Cho et al., 2019). The hierarchy is one-dimensional: a rack has 2 servers, each with 2 CPUs connecting 4 GPUs. Interconnects specify how devices are connected with each other and the latency and bandwidth constraints. In this case, we have exactly

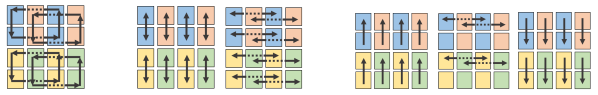
one kind of interconnect in each level, but, in general, the interconnect topology can be more complex: there can be multiple interconnects in one level, and an interconnect can connect devices (and other interconnects) across levels.

2.1 Parallelism Placement

Parallel placement decides which parts of a partitioned program will execute on which parts of a system. However, synthesizing all arbitrary device mappings, as well as running experiments with them, can be extremely expensive if implemented naively. For example, if we have data parallelism of size 4 and 4 parameter shards for the system in Figure 2a, then there will be $(4 * 4)! > 2^{44}$ possibilities to decide which partitioned program maps to which GPU.

To explore the search space efficiently, the critical idea of P^2 is to *partition parallelism axes over the system hierarchy* to generate topology-aware parallelism placements, while still being able to systematically generate a wide range of parallelism placements. Specifically, a result of parallelism placement synthesis is a *parallelism matrix*, where each element is a *parallelism factor* representing the number of a specific level in the hierarchy that a parallelism form splits the computation across. Figures 2b, 2c and 2d show examples of parallelism matrices synthesized by P^2, where we have data parallelism of size 4 and 4 parameter shards. In Figure 2b, the first row $\begin{bmatrix}1 & 2 & 2 & 1\end{bmatrix}$ corresponds to a factorization of data parallelism on each system level. Specifically, we first assign all data parallelism (of size 4) into 1 rack (each with data parallelism of size $4/1 = 4$). Then each rack assigns data parallelism of size 4 into 2 servers (each with data parallelism of size $4/2 = 2$). Next, each server assigns data parallelism of size 2 into 2 CPUs (each with data parallelism of size $2/2 = 1$). Finally, each CPU assigns data parallelism of size 1 into 1 GPU. The second row $\begin{bmatrix}1 & 1 & 1 & 4\end{bmatrix}$ corresponds to a factorization of parameter sharding: each rack, server, and CPU gets assigned all parameter shards (of size 4), and each CPU then assigns 4 parameter shards into 4 GPUs, each GPU level with $4/4 = 1$ shard. Therefore, in the resulting placement, each CPU corresponds to one replica (data parallelism) where each GPU has one parameter shard. We can interpret Figure 2c and 2d accordingly.

Note how parallelism matrices decide communication requirements. Consider reduction along parameter sharding (i.e., reduce devices n/m with the same n but different m). In Figure 2b, this can be done by communication over only S0, while in 2c, half of the data can be reduced by only S0, but the rest of the reduction requires communication over S0/S1/S2. We discuss the impact of parallelism placements on communication cost in detail in Section 4.



(a) AllReduce (b) AllReduce-AllReduce (c) Reduce-AllReduce-Broadcast

Figure 3: Example reduction strategies.

(a) ReduceScatter - AllReduce (b) AllReduce - AllReduce

Figure 4: Semantically invalid reduction. (a): Reduce data that should not be reduced. (b): Reduce the same data twice.

2.2 Reduction Strategy

For each parallelism matrix, P^2 further synthesizes topology-aware reduction strategies using common collective operations, which allows us to find the optimal reduction strategy for any given parallelism matrix.

To illustrate the idea, consider the parallelism matrix in Figure 2d, and the goal is to reduce along parameter sharding. As shown in Figure 3a, an obvious choice to perform the reduction is a single AllReduce within reduction groups. However, such reduction may be suboptimal, as it does not utilize the topology of the system. Figure 3b and 3c show two reduction strategies, among others, synthesized by P^2. Figure 3b first performs a step of AllReduce which communicates over only S0, and then AllReduce that communicates over S0/S1/S2. Figure 3c first performs Reduce that puts the reduction result in the root device, then AllReduce between root devices, and finally Broadcast that broadcasts data from the root device. Of particular interest in these two reduction strategies is that no one is strictly better than the other, as the communication overhead depends on the network: 3c takes more steps, but has fewer data to be transferred over S1/S2, which may outperform 3b if S0 has high bandwidth while communication over S1/S2 is expensive.

P^2 gives us a systematic way to synthesize and compare a wide range of topology-aware reduction strategies. In particular, synthesized reduction strategies can outperform a single step AllReduce, with speedup up to 2.05×. However, synthesizing reduction strategies also imposes challenges, which we outline in the rest of this section.

2.3 Formalism of Collective Operations

To synthesize reduction strategies, we first need to formalize the semantics of collective operations, since not all sequences of operationally valid collective operations correspond to semantically correct implementations of the end-to-end reduction requested by the user. For example, consider

Figure 5

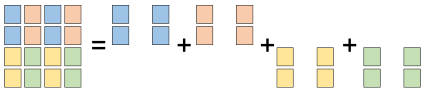


Figure 6: Device structured as the reduction axis

the (incomplete) reduction steps given in Figure 4 for the requested reduction across parameter shards. Both programs can be executed successfully, e.g., by NCCL (NVidia, 2021). Unfortunately, they are both *semantically invalid*. In particular, *we consider reduction steps which result in device states that can never reach the final desired state to be semantically invalid*. Specifically, in 4a, the first ReduceScatter will reduce, among others, device A0 and A1 (recall that GPUs are named in Figure 2a), and put the first half of the result on A0 and the second half on A1. Then the second AllReduce will reduce A0 and A1 – so the first and the second half of the result get reduced while they should not! Now, we can never reach the desired final state. 4b is also invalid as it reduces the data on A0 and C0 twice.

P^2 provides a concise and novel formalism of common collective operations (Section 3.2) that captures semantic correctness and rules out semantically invalid programs, massively reducing the synthesis space. Specifically, each device state is defined as a *state matrix* describing what kind of data a device has. The semantics of collective operations is defined with Hoare triples (Hoare, 1969), where a collective takes the state of each device as a *pre-condition* and returns a new state as a *post-condition*.

2.4 Reduction Communication Patterns

Even though the formalism of collective operations rules out semantically invalid reduction steps, the search space of reduction strategies is still quite large. One reason is that we need to decide which devices form a reduction group for each reduction step. For example, the first step in Figure 3b reduces over $\{A0, A1\}, \{A2, A3\}$ (among others). We may randomly generate all possible groups, but that would significantly increase the search space. Also, many of them would be immediately thrown away after semantic checks.

To synthesize reduction strategies effectively, P^2 uses a domain-specific language (Section 3.3) that explores the hierarchy to generate *hierarchical* communication patterns. The reduction language, together with the *synthesis hierarchy* (explained next), can model many common communication patterns, including those in Figure 3 and 4.

2.5 Synthesis Hierarchy

To generate hierarchical reduction communication patterns, the DSL reduction instruction needs to know the synthesis hierarchy. For example, a possible instruction is to reduce

$\begin{bmatrix}1&1&2&2\\1&2&1&2\end{bmatrix}$	column-based	$[1\ 1\ 1\ 2\ 2\ 1\ 2\ 2]$	①
	row-based	$[1\ 1\ 2\ 2\ 1\ 2\ 1\ 2]$	②
	reduction axis	$[1\ 2\ 1\ 2]$	③
$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}$	column-based	$[1\ 4\ 7\ 2\ 5\ 8\ 3\ 6\ 9]$	
	row-based	$[1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$	
	reduction axes	$[1\ 2\ 3\ 7\ 8\ 9]$	
	collapsed	$[7\ 16\ 27]$	

Table 1: Synthesis hierarchy (reduction axes highlighted)

all "GPUs" connected to the same "CPU". Now, an important design decision to consider is *which* hierarchy to use for synthesis, as it decides what kind of instructions we can produce. One obvious choice is the hardware hierarchy, i.e., $[1\ 2\ 2\ 4]$ for our running example (we ignore level names as that can be randomly generated). But the system hierarchy is not fine-grained enough for the requested reduction: for example, the reduction in Figure 3 requires reducing half of the GPUs connected to a CPU. To do that, we need to take parallelism axes into consideration. In this case, the parallelism matrix is $\begin{bmatrix}1&1&2&2\\1&2&1&2\end{bmatrix}$, which splits GPU into 2 by 2, allowing us to reduce 2 GPUs connected to a CPU. However, to form a synthesis hierarchy from the parallelism matrix, we have two options (Table 1): ① puts parallelism factors by columns, which essentially expands the system hierarchy corresponding to the parallelism matrix; or ② puts factors by rows, which expands the parallelism axes. In this work, we *prove* that ② is more *expressive* than ①, i.e., ② can generate all semantically valid reduction strategies that can be generated by ①. The result might be somewhat counter-intuitive, as ① seems a natural way to expand the system hierarchy. The critical insight is that as ② puts parallelism axes consecutively, it can more easily generate semantically correct reduction, while ① can more easily reduce devices laid out consequently but those reduction can be semantically invalid as it partitions parallelism axes.

It turns out we can further optimize the synthesis hierarchy. In particular, note that we reduce along the axis $[1\ 2\ 1\ 2]$, but ② includes the full matrix (i.e., including $[1\ 1\ 2\ 2]$). With a full matrix, we can generate reduction like Figure 5, which, however, is not useful for this specific case, as we should not reduce device A0 and A2. The key observation here is that each reduction group is essentially structured according to the reduction axis $[1\ 2\ 1\ 2]$, and this structure is repeated for the rest of the matrix, as shown in Figure 6. Based on this observation, P^2 uses the synthesis hierarchy ③ formed by parallelism factors from only the reduction axis and then *lowers* synthesized programs to the full system hierarchy. We *prove* that ③, while largely reducing the search space, is actually more expressive than ②.

Exploration with multiple reduction axes. The same obser-

vation applies for reduction over multiple axes. An example is given in the second half of Table 1. In this case, the reduction axes based synthesis hierarchy is $[1\ 2\ 3\ 7\ 8\ 9]$. Note that some parallelism factors are from the same hardware level: 1 and 7, 2 and 8, and 3 and 9. Since for switched networks, splitting hardware hierarchies does not bring benefits in most cases, we can *collapse* parallelism factors of the same hardware hierarchies. In this example, the final synthesis hierarchy is $[7\ 16\ 27]$.

3 Program Synthesis

We now present the program synthesis algorithm in P^2.

3.1 Parallelism Placement

Parallelism placement partitions parallelism axes over the system hierarchy. With the novel notion of the *parallelism matrix* and its interpretation (Section 2.1), synthesizing parallelism matrices is straightforward. Consider
$\mathbf{H} = [h_0 \cdots h_n]$ is the system hierarchy (e.g., $[1\ 2\ 2\ 4]$),
$\mathbf{P} = [p_0 \cdots p_m]$ is the parallelism axes (e.g., $[4\ 4]$),
then a parallelism matrix is

$$\begin{bmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,0} & x_{m,1} & \cdots & x_{m,n} \end{bmatrix}$$

subject to:

$$\prod_{i=0}^{m} x_{i,j} = h_j, \quad j = 0, ..., n \ (1)$$

$$\prod_{j=0}^{n} x_{i,j} = p_i, \quad i = 0, ..., m \ (2)$$

Equation (1) requires the product of a column to be equivalent to the corresponding system hierarchy cardinality, while Equation (2) requires the product of a row to be equivalent to the corresponding parallelism axis.

3.2 Collective Operations

This section defines the semantics of collective operations. In this work we focus on the common ones: AllReduce, ReduceScatter, AllGather, Reduce and Broadcast.

Notations We first define the notations.

d			device
s	$\in$	$\mathbb{B}^{k \times k}$	device state
$\mathcal{G}$	$:=$	$\overline{d_i : s_i}$	state context
$\mathcal{C}$	$:=$	AllReduce | ReduceScatter	
	|	AllGather | Reduce | Broadcast	

We use d to denote a device, whose state s is represented as a boolean matrix of dimensions $k \times k$; k being the number of devices. In particular we treat the data as being split in k chunks. The ith row of a state matrix represents the ith chunk. $s[i][j] = 1$ means that device j has contributed its original ith chunk to the reduction result. Figure 7 gives an example. A state context $\mathcal{G}$ maps devices to their states.

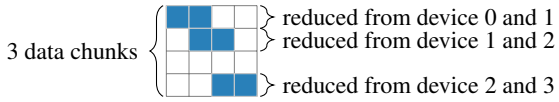


Figure 7: A device state. Assume we have in total 4 devices (i.e., device 0,1,2 and 3), so each device state is a 4×4 matrix. $s[i][j]$ is colored if $s[i][j] = 1$. The device state has 3 non-empty rows, meaning that it has 3 data chunks. Each data chunk describes where the data is reduced from. For example, the first data chunk is the reduction result between the original first data chunk of device 0 and 1.

Note finally that Reduce and Broadcast typically take a root device to reduce to or broadcast from. Since we focus on hierarchical systems, we always use the first device in a reduction group as the root without loss of generality.

Semantics Figure 8 defines the semantics of collective operations, which is closely based on Hoare rules (Hoare, 1969). Each reduction takes the form of a Hoare triple $\{\,\mathcal{G}_1\,\}\,\mathcal{C}\,\{\,\mathcal{G}_2\,\}$, which means that *from the pre-condition state $\mathcal{G}_1$, a step of reduction $\mathcal{C}$ yields to the post-condition state $\mathcal{G}_2$*. Explanations of auxiliary functions are given in the figure. To better illustrate the semantics, the right of Figure 8 provides examples of each collective operation.

At a high level, these rules capture the constraints for a reduction step to be semantically correct. Rule R-AllReduce first checks that the data contained in each device (denoted as rows representing the *non-empty rows*) should have the same data chunks. Moreover, columns in any specific chunk should be disjoint. Both constraints are essential for the reduction result to be valid: we should not reduce data from different chunks or reduce the same data twice (as discussed in Section 2.3). Finally, we generate the resulting state $\uplus\overline{s_i}$ for each device by adding up all matrices. Rule S-ReduceScatter and S-Reduce are similar to rule S-AllReduce, except that S-ReduceScatter scatters the reduction result over devices, where scatter raises an error if the number of data chunks in s is not divisible by the number of devices; and S-Reduce puts the result only in the first device and clears up the rest of the devices. Rule S-AllGather simply needs all data rows to be disjoint. Rule S-Broadcast overrides the data of every device with the data from the first one. As an optimization, the rule enforces *information increase*, i.e., the data to be broadcasted must be as informative as data in other devices and more informative than at least one other device.

$\boxed{\{\,\mathcal{G}_1\,\}\,\mathcal{C}\,\{\,\mathcal{G}_2\,\}}$ *(Reduction: from the pre-condition state $\mathcal{G}_1$, $\mathcal{C}$ yields to the post-condition state $\mathcal{G}_2$)* before after

R-AllReduce
$$\frac{\forall i\,j,\ s_i.\mathsf{rows} = s_j.\mathsf{rows} \qquad \forall i\,j\,k,\ i \neq j \implies s_i[k] \circledast s_j[k] \qquad s = \uplus\overline{s_i}}{\{\,\overline{d_i : s_i}\,\}\ \mathsf{AllReduce}\ \{\,\overline{d_i : s}\,\}}$$

R-ReduceScatter
$$\frac{\forall i\,j,\ s_i.\mathsf{rows} = s_j.\mathsf{rows} \qquad \forall i\,j\,k,\ i \neq j \implies s_i[k] \circledast s_j[k] \qquad s = \uplus\overline{s_i} \qquad s'_i = \mathsf{scatter}(s, \bar{i})[i]}{\{\,\overline{d_i : s_i}\,\}\ \mathsf{ReduceScatter}\ \{\,\overline{d_i : s'_i}\,\}}$$

R-AllGather
$$\frac{\forall i\,j,\ i \neq j \implies s_i.\mathsf{rows} \circledast s_j.\mathsf{rows} \qquad \forall i\,j,\ |s_i.\mathsf{rows}| = |s_j.\mathsf{rows}| \qquad s = \uplus\overline{s_i}}{\{\,\overline{d_i : s_i}\,\}\ \mathsf{AllGather}\ \{\,\overline{d_i : s}\,\}}$$

R-Reduce
$$\frac{\forall i\,j,\ s_i.\mathsf{rows} = s_j.\mathsf{rows} \qquad \forall i\,j\,k,\ i \neq j \implies s_i[k] \circledast s_j[k] \qquad s = \uplus\overline{s_i}}{\{\,\overline{d_i : s_i}\,\}\ \mathsf{Reduce}\ \{\,d_0 : s, \overline{d_i : \{\}}^{\,i\neq 0}\,\}}$$

$\circledast$	*disjoint*	rows	*non-empty rows*
$\uplus$	*addition*	$\lvert\cdot\rvert$	*length*

$\mathsf{scatter}(s, \bar{i})$ *scatters non-empty rows in s over devices $\bar{i}$*

R-Broadcast
$$\frac{\forall i,\ s_i \leq s_0 \qquad \exists i,\ s_i < s_0}{\{\,\overline{d_i : s_i}\,\}\ \mathsf{Broadcast}\ \{\,\overline{d_i : s_0}\,\}}$$

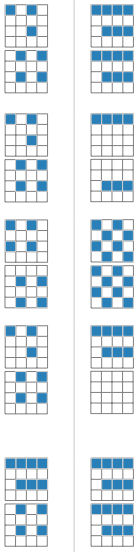


Figure 8: Semantics of collective operations. with the right presents examples of each operation. For those examples, we have in total 4 devices .e., device 0,1,2, and 3), so each device state is a 4×4 matrix. We assume the reduction happens between only device 0 and 1. The pre-condition states of device 0 (top) and 1 (bottom) are on the left, and after a step of reduction, their states turn into the post-condition states on the right.

3.3 Reduction Programs

We now turn to our reduction language which is built on top of the formalism of collective operations.

$program$	$\in$	$[reduction]$
$reduction$	$\in$	$slice \times form \times \mathcal{C}$
$slice$	$:=$	e
$form$	$:=$	$\mathsf{InsideGroup} \mid \mathsf{Parallel}(e) \mid \mathsf{Master}(e)$

A reduction strategy is represented as a $program$, which is essentially a list of reduction instructions. A $reduction$ instruction consists of a $slice$, a $form$, and a collective operation $\mathcal{C}$. We use e to represent a level in the synthesis hierarchy. The $slice$ chooses a level. The $form$ has three patterns: $\mathsf{InsideGroup}$, $\mathsf{Parallel}(e)$, and $\mathsf{Master}(e)$. Inside a $reduction$, the e carried in the form must be an ancestor of the one carried in the slice. The slice and the form together decide the device groups that will perform the operation $\mathcal{C}$.

It turns out that $slice$ and $form$ are quite expressive and can encode many common hierarchical communication patterns. Table 2 demonstrates several examples using the system hierarchy in Figure 2a. Specifically, a $slice$ divides devices into *reduction groups*, and $form$ decides the reduction form happening for the reduction groups. For example, consider that the slice is CPU, then we get reduction groups within each CPU, i.e., $\{A_0, A_1, A_2, A3\}, \{B_0, B_1, B_2, B_3\}, \{C_0, C_1, C_2, C_3\}, \{D_0, D_1, D_2, D_3\}$. Now, if the form is InsideGroup, then we perform reduction within each reduction group. If the form is $\mathsf{Parallel}(e)$, we perform reduction over the first device in each group, the second device in each group, etc, if they connect to the same e. Thus, $\mathsf{Parallel}(\text{server})$ generates $\{A_0, B_0\}$, $\{A_1, B_1\}$, etc., whereas $\mathsf{Parallel}(\text{rack})$ generates $\{A_0, B_0, C_0, D_0\}$ etc. Master generates the device groups in the same way as Parallel, but only reduces over the first device group.

Note that Table 2 presents device groups for reduction over the system hierarchy [(rack, 1), (server, 2), (CPU, 2), (GPU, 4)]. As we discussed in Section 2.5, reduction over specific parallelism axes will use the synthesis hierarchy formed by parallelism factors, and we will get reduction groups for that particular reduction axis like $\{A_0, A_1\}, \{A2, A3\}$ etc.

Supposing $slice$ and $form$ derive the device groups $\overline{\mathcal{G}_i}$, which are disjoint by construction, we define the semantics of a $reduction$ instruction as:

$$\frac{(slice, form)\ \textit{derives}\ \overline{\mathcal{G}_i} \qquad \overline{\{\,\mathcal{G}_i\,\}\,\mathcal{C}\,\{\,\mathcal{G}'_i\,\}}}{\{\,\overline{\mathcal{G}_i}, \mathcal{G}\,\}\,(slice, form, \mathcal{C})\,\{\,\overline{\mathcal{G}'_i}, \mathcal{G}\,\}}$$

where each device group participating in the reduction gets the device states updated according to the semantics of collective operations, and devices not participating in the reduction have their states unchanged. A reduction program

slice	*form*	groups(*slice*, *form*)
CPU	InsideGroup	$\{A_0, A_1, A_2, A_3\}, \{B_0, B_1, B_2, B_3\}, \{C_0, C_1, C_2, C_3\}, \{D_0, D_1, D_2, D_3\}$
	Parallel(server)	$\{A_0, B_0\}, \{A_1, B_1\}, \{A_2, B_2\}, \{A_3, B_3\}$ $\{C_0, D_0\}, \{C_1, D_1\}, \{C_2, D_2\}, \{C_3, D_3\}$
	Parallel(rack)	$\{A_0, B_0, C_0, D_0\}, \{A_1, B_1, C_1, D_1\}, \{A_2, B_2, C_2, D_2\}, \{A_3, B_3, C_3, D_3\}$
	Master(rack)	$\{A_0, B_0, C_0, D_0\}$
server	InsideGroup	$\{A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3\}, \{C_0, C_1, C_2, C_3, D_0, D_1, D_2, D_3\}$
	Parallel(rack)	$\{A_0, C_0\}, \{A_1, C_1\}, \{A_2, C_2\}, \{A_3, C_3\}$ $\{B_0, D_0\}, \{B_1, D_1\}, \{B_2, D_2\}, \{B_3, D_3\}$
rack	InsideGroup	$\{A_0, A_1, A_2, A_3, B_0, B_1, B_2, B_3, C_0, C_1, C_2, C_3, D_0, D_1, D_2, D_3\}$

Table 2: Hierarchical communication patterns for Figure 2a.

then iteratively applies each reduction:

$$\frac{program = \overline{reduction}^{i \in n} \qquad \overline{\{\,\mathcal{G}_i\,\}\, reduction_i\, \{\,\mathcal{G}_{i+1}\,\}}}{\{\,\mathcal{G}_0\,\}\, program\, \{\,\mathcal{G}_{n+1}\,\}}$$

3.4 Synthesis Hierarchy

In Section 2.5, we have proposed and compared different synthesis hierarchies for synthesizing reduction programs:

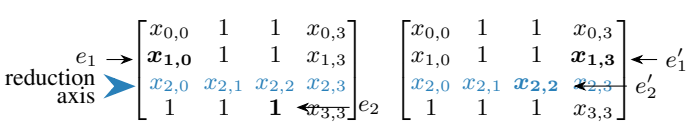
(a) System hierarchy ($[1\ 2\ 2\ 4]$)
(b) Column-based parallelism factors ($[1\ 1\ 1\ 2\ 2\ 1\ 2\ 2]$)
(c) Row-based parallelism factors ($[1\ 1\ 2\ 2\ 1\ 2\ 1\ 2]$)
(d) Reduction axis parallelism factors ($[1\ 2\ 1\ 2]$)

P^2 uses (d). Here, we formally prove the theorem that justifies our choice. First, every reduction instruction essentially decides the device groups $\overline{\mathcal{G}}$ and the operation $\mathcal{C}$. Therefore, a program can be lowered to a sequence $(\overline{\mathcal{G}_1}, \mathcal{C}_1)$, $(\overline{\mathcal{G}_2}, \mathcal{C}_2)$, ..., $(\overline{\mathcal{G}_n}, \mathcal{C}_n)$. Since (d) includes only the reduction axis, lowering for (d) applies the generated grouping patterns to non-reduction axes when forming device groups.

Definition 3.1 (Expressive power of synthesis hierarchy)**.** *A synthesis hierarchy is more expressive than ($\geq$) another, if every valid lowered program $\mathcal{L}$ synthesized using the latter can be synthesized using the former.*

Theorem 3.2. *(d) $\geq$ (c) $\geq$ (b) $\geq$ (a).*

Proof. For space reasons, we show one example that illustrates the key proof strategy, and we refer the reader to the appendix for the full proof. Consider proving (c) $\geq$ (b), where (b) synthesizes a valid reduction step $(e_2, \mathsf{Parallel}(e_1), \mathcal{C})$. For the reduction to be valid, every reduction group must be partitioned only over the reduction axis. Therefore, all non-reduction parallelism factors column-wisely between e_1 (exclusive) and e_2 (inclusive) can only be 1. An example is given below on the left.
Now we construct a reduction instruction for (c) that expresses the same reduction. Suppose the reduction step in (b) covers parallelism factors $e_i, ..., e_j$ on the reduction axis. Let e'_1 be the level corresponding to the parallelism factor right before e_i row-wisely, and let e'_2 be e_j. Then $(e'_2, \mathsf{Parallel}(e'_1), \mathcal{C})$ is a desired reduction instruction.



□

3.5 Program Synthesis for Reduction Programs

So far, we have given the constraint for generating parallelism matrices (Section 3.1) and how we can obtain a synthesis hierarchy from a parallelism matrix (Section 3.4). The last missing piece is how to synthesize reduction programs.

To formalize the synthesis problem, we need an initial precondition state as the beginning state and a post-condition state as the final desired state. Initially, every device only holds its own data, and therefore device i has 1 in the ith column, and 0 in any other position. In the final desired state, a device should have 1 in all columns corresponding to devices in its reduction group, and 0 in any other position. An extra indirection is caused from using the reduction axis parallelism factors as the synthesis hierarchy (Section 3.4), which only includes part of the system, and then lowers the program to the full system. Therefore, our goal is to synthesize a $program$, whose lowering $\mathcal{L}$ subjects to:

$$\left\{ \overline{d_i : \begin{bmatrix} & & \overset{i}{} & & \\ 0 & \dots & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & \cdots & 0 \end{bmatrix}} \right\} \mathcal{L} \left\{ \overline{d_i : \begin{bmatrix} & & \overset{i}{} & & & & \overset{\overline{j}}{} & & \\ 0 & \dots & 1 & \dots & 0 & \dots & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & \cdots & 0 & \cdots & 1 & \cdots & 0 \end{bmatrix}} \right\}$$

supposing d_i reduces with devices $\overline{j}$.

Given the syntax and the semantics of reduction programs, we use *syntax-guided program synthesis* (Alur et al., 2013) to synthesize programs in increasing order of program size.

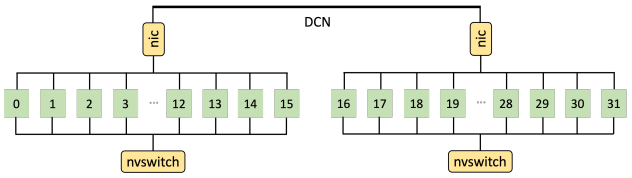
4 Experiments

We implement P^2 to synthesize parallelism matrices and reduction programs, and lower the programs into sequences of XLA collective operations, which in turn result in sequences of NCCL calls on the XLA GPU backend. We measure the execution time of the compiled programs. The experiments aim to answer the following research questions:

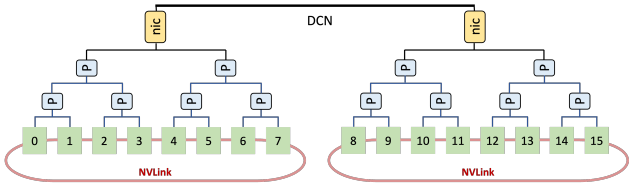
RQ1 What is the impact of parallelism placement on reduction algorithms?

RQ2 Are our various techniques for taming the search space effective so that we can quickly enumerate a wide variety of reduction programs?

RQ3 Given a parallelism placement, can we find reduction strategies that outperform the default implementation



(a) 2 nodes, each with 16 A100 GPUs sharing one NVSwitch and one NIC, and all NICs are connected in a data center



(b) 2 nodes, each with 8 V100 GPUs forming a ring via NVLink and connected via PCIe switches. Each node consists of two CPUs (each owning 4 GPUs) with one NIC to the DCN. A shared NIC connecting the two CPUs is a modeling simplification – in reality cross-domain communication is through shared memory.

Figure 9: System topology models for 2 nodes. For the experiments, we run on both 2 and 4 nodes.

(i.e., AllReduce), and if so what is their form?

The experiments ran on two different GPU system configurations available on Google Cloud Platform (GCP, 2021) (see Figure 9): (i) NVIDIA A100, where each node consists of 16 GPUs sharing one NVSwitch and one NIC connecting to the data center network; and (ii) NVIDIA Tesla V100, where each node consists of 8 GPUs forming a *ring* via NVLink; each pair of GPUs are connected via PCIe switches, and each of the two CPUs of the node has 4 GPUS in its PCIe domain. We experiment with both NCCL ring reduction and tree reduction (Sanders et al., 2009), set by NCCL_ALGO.

We run experiments with 2 and 4 nodes. For A100, the system hierarchy is $[2\ 16]$ or $[4\ 16]$. For V100, since the NVLink ring connects all 8 GPUs, and the NVLink ring has much higher bandwidth than PCIe bridges, we put 8 GPUs inside one layer, and so with 2 or 4 nodes, the system hierarchy is $[2\ 8]$ or $[4\ 8]$, respectively. Each GPU carries a large amount of data ($(2^{29}\times$ nodes) of float32) to reduce the impact of latency, and each program runs 10 times to reduce the impact of network noise.

For each system, we synthesize parallelism mappings and reduction programs for (1) a single parallelism axis; (2) all combinations of two parallelism axes, with reduction on one of the axes; and (3) three parallelism axes, with reduction on the first and the third axes. We can easily scale to more axes, though up to three axes are quite common in practical settings, and many observations can already be illustrated.

Next, we discuss the results and insights from the experiments. For space reasons, we present only representative cases, and we put the full experiment results in the appendix.

	Parallelism axes	Parallelism matrix	Reduction on the 0th axis Ring	Reduction on the 0th axis Tree	Reduction on the 1st axis Ring	Reduction on the 1st axis Tree
4 nodes, each with 16 A100						
A1	$[2\ 32]$	$[[1\ 2]\ [4\ 8]]$	0.12	0.17	8.74	9.89
A2		$[[2\ 1]\ [2\ 16]]$	37.16	36.94	4.81	3.41
B1	$[4\ 16]$	$[[1\ 4]\ [4\ 4]]$	0.15	0.20	17.70	19.03
B2		$[[2\ 2]\ [2\ 8]]$	28.77	19.81	8.39	4.99
B3		$[[4\ 1]\ [1\ 16]]$	56.13	89.70	0.18	0.22
C1	$[8\ 8]$	$[[1\ 8]\ [4\ 2]]$	0.17	0.21	33.92	41.06
C2		$[[2\ 4]\ [2\ 4]]$	16.52	9.18	15.68	9.43
C3		$[[4\ 2]\ [1\ 8]]$	34.05	41.23	0.17	0.21
4 nodes, each with 8 V100						
E1	$[8\ 4]$	$[[1\ 8]\ [4\ 1]]$	0.28	0.39	21.74	30.42
E2		$[[2\ 4]\ [2\ 2]]$	14.25	15.48	10.98	7.34
E3		$[[4\ 2]\ [1\ 4]]$	14.84	19.90	2.96	0.43

Table 3: Reduction time in seconds of running AllReduce.

4.1 Synthesizing Parallelism Placement

Result 1 (**RQ 1**): *The performance of* AllReduce *differs significantly among parallelism matrices, up to 448.5×.*

The experiment results are given in Table 3. For a particular parallelism axis (e.g., A), we compare the reduction time for difference parallelism matrices (e.g., A1 and A2) with each NCCL algorithm and the reduction axis. Notably, for reducion on the 0th axis and with the Tree algorithm, B3 (89.70s) is slower than B1 (0.20s) by 448.5×.

The difference is due to the fact that different parallelism matrices lead to different data placement. In B1, the first row of the the matrix ($[1\ 4]$) means that devices to be reduced are inside a single node, where the local NVSwitch can perform the reduction efficiently. For B3, the first row ($[4\ 1]$) puts reduction groups across nodes, going through the slow data-center network. However, B3 can still be useful *for a diffferent reduction*: since it puts the 1st reduction axis inside a single node, for a reduction on the 1st axis, B3 (0.22s on Tree) is 86.5× faster than B1 (19.03s). In practice, models with multiple parallelism forms (e.g., Shoeybi et al. (2020)) involve reductions across both axes, and the selection of a mapping should take all of them into account.

4.2 Synthesizing Reduction Programs

Now we turn to the reduction programs synthesized for each parallelism matrix. Table 4 presents experiment results.

Result 2 (**RQ 2**): *Our pruning techniques are effective for the synthesizer to achieve fast synthesis time.*

With our formalism, a program cannot be arbitrarily large, since our carefully crafted semantics of collective operations enforces a form of information increase for every operation.

	NCCL algo	Parallelism axes	Synthesis time (s)	Programs outperforming AllReduce / total programs	Parallelism matrix	AllReduce (bold if the optimal AllReduce)	Optimal (bold if overall optimal)	Speedup
2 nodes, each with 16 A100								
F1	Ring	[8 4]	0.03	14/47	[[1 8] [2 2]]	**0.17**	**0.17**	1×
F2					[[2 4] [1 4]]	16.84	9.19	1.83×
4 nodes, each with 16 A100								
G1	Tree	[4 16]	0.04	10/53	[[1 4] [4 4]]	**0.20**	**0.17**	1.17×
G2					[[4 1] [1 16]]	89.70	56.13	1.60×
H1	Ring	[16 2 2]	0.97	25/235	[[1 16] [2 1] [2 1]]	**4.79**	4.63	1.03×
H2					[[2 8] [2 1] [1 2]]	4.91	**3.10**	1.58×
I1	Ring	[2 2 16]	0.93	29/235	[[2 1] [2 1] [1 16]]	**4.82**	**2.99**	1.61×
I2					[[1 2] [2 1] [2 8]]	5.28	4.77	1.11×
J1	Tree	[64]	1.16	5/47	[[4 16]]	**5.75**	**4.74**	1.21×
4 nodes, each with 8 V100								
K1	Ring	[8 2 2]	0.24	17/188	[[2 4] [2 1] [1 2]]	4.80	**2.35**	2.04×
K2					[[1 8] [2 1] [2 1]]	**4.40**	4.40	1×
L1	Ring	[32]	0.06	11/47	[[4 8]]	**4.83**	**3.45**	1.4×

Table 4: Reduction time in seconds for running AllReduce and the synthesized optimal reduction strategy (reduction on the 0th axis for parallelism axes of size 1 and 2, and on the 0th and 2rd axes for parallelism axes of size 3).

In our experiments, we set 5 as the program size limit for the synthesizer, which turns out to be sufficient to generate interesting reduction patterns. With this setup, the longest synthesis time is under 2 seconds (for up to 235 programs). Increasing the size limit makes the synthesis *slightly* slower, but, for most cases, does not generate new programs.

Result 3 (**RQ 3**): *If the reduction axes can be put within one node, then a single step* AllReduce *inside that node is the most performant reduction due to fast local bandwidth.*

We observe this result from the difference between F1 and F2. F1 assigns the reduction axis to the GPU level, and thus AllReduce is the most performant reduction, outperforming F2, which requires cross-node reduction, by 99.06×.

Result 4 (**RQ3**): *Synthesized programs can help mitigate the impact of parallelism placement.*

Consider G1 and G2. As discussed before, G2's AllReduce (89.7s) is 448.5× slower than G1 (0.20s). Synthesized programs have helped bridge the gap: G2's optimal program is only 330.2× slower. However, the performance difference here is significant and the help is limited. The case of H1 and H2 is more interesting: H1's AllReduce is 1.03× slower than H2, but its optimal program is 1.49× faster than H2!

On the other hand, it is also possible that synthesis aggravates the impact of parallelism placement. For example, for I1 and I2, the performance difference jumps from 1.10× for AllReduce to 1.60× for the optimal program.

Result 5 (**RQ3**): *For reduction across nodes, a topology-aware reduction program tends to outperform a single step* AllReduce*, with speedup on average 1.28×, upto 2.04×.*

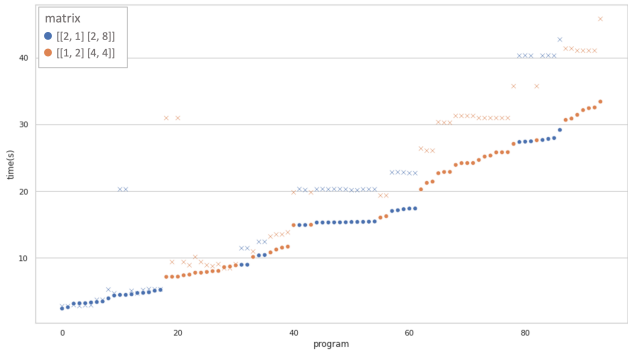
(i) Reduce-AllReduce-Broadcast (ii) ReduceScatter-AllReduce-AllGather

Figure 10: Common optimal reduction programs

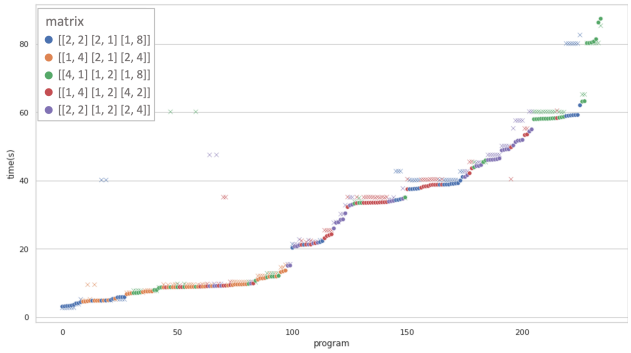
Table 4 shows that when cross-node communication is needed the optimal program tends to outperform AllReduce. For example, the speedup is 1.84× in F2, and 2.04× in K2. For 69% of all mappings across both systems, synthesized programs outperform AllReduce by 1.27× on average.

We present common optimal reduction programs applied to our running examples (Section 2) in Figure 10. (i) Figure 10i first reduces local data to a root device, performs AllReduce between root devices, and broadcasts the result from the root device to each device. (ii) Figure 10ii first performs ReduceScatter between local devices, and then AllReduce between remote devices, and finally AllGather between local devices. Both reduction programs utilize the topology, by performing local communication first, which is often more efficient due to local high bandwidth. Now, the data to be reduced across nodes in the intermediate step is significantly smaller. The final step is again local communication. Thus, the reduction programs have overall better performance than AllReduce. It turns out that both reduction programs have been recently proposed: program (i) has been used in Goyal et al. (2018); Jia et al. (2018a), and program (ii) has been proposed by Cho et al. (2019).

Furthermore, the experiments suggest that program (ii) is more often the optimal one and outperforms (i) by a larger



(a) 4 nodes of V100 with NCCL Ring and parallelism axes $\begin{bmatrix}2 & 16\end{bmatrix}$, reduction on the 1st axis. Synthesis 0.12s, and simulation 0.54s.



(b) 4 nodes of A100 with NCCL Tree and parallelism axes $\begin{bmatrix}4 & 2 & 8\end{bmatrix}$, reduction on 0th and 2rd axes. Synthesis 2.86s, simulation 3.09s.

Figure 11: Simulation results, in increasing order of experiment time. Measurements are '•' and solid, and simulations are '×' and translucent. Colors denote parallelism matrices.

speedup. Specifically, when (i) is optimal, it outperforms (ii) by only about 1.1× (up to 1.12×); when (ii) is optimal, it outperforms (i) by about 1.3× (up to 2.73×).

4.3 ResNet-50

The extensive experiments have demonstrated that P^2 is effective in synthesizing efficient parallelism placements and reduction strategies for a wide variety of reductions and choices of parallelism axes. We further applied P^2 to ResNet-50 as an end-to-end evaluation. The ResNet-50 baseline is implemented in Haiku (DeepMind, 2020) with data parallelism spanning 4 nodes, each with 8 V100 GPUs. Since the reduction involves all devices in the system, the parallelism mapping is trivial. P^2 finds the optimal reduction strategy as described in Figure 10ii. By applying the optimal reduction strategy, the reduced communication overhead achieves 15% overall training speedup compared to the baseline. Since the training time includes the computation time, the speedup indicates the significant improvement on the communication overhead.

	Top-1	Top-2	Top-3	Top-5	Top-6	Top-10
A100	46.8%	71.0%	72.6%	74.2%	90.3%	94.4%
V100	60.5%	67.1%	71.1%	76.3%	76.3%	88.1%
Total	52.0%	69.5%	72.0%	75.0%	85.0%	92.0%

Table 5: Prediction accuracy.

5 Simulation

In this section, we apply simulation to the system topologies in Figure 9, and show that P^2 can identify near-optimal programs, to reduce the need for evaluation over hundreds or thousands of synthesized mappings and strategies.

Assumptions. We use 100Gbps NICS, which we assume were utilized at 60%, yielding an effective 8GB/s. For the PCIe switches we assumed 32GB/s. For the V100 NVLink ring we assume 135GB/s in each direction – an optimistic 90% of the nominal uni-directional bandwidth (150GB/s) (Jia et al., 2018c). For the A100 NVLink switch, we assume 270GB/s uni-directional bandwidth – again 90% of the nominal value (300GB/s in each direction) (NVIDIA, 2021).

Results. Figure 11 shows that P^2 predictions follow the same trend as the V100 experiments (11a), and are very close to the absolute performance for A100 (11b). The main reason for reduced absolute accuracy in V100 is the imperfect modeling of cross-domain communication. Note that there exist a few programs for which the simulation result is notably slower than the experiments, mostly due to XLA optimizations. For example, program 10 and 11 (in 11a) are all 2 steps of AllReduce, which XLA optimizes to a single step AllReduce. We do not extend P^2 with any optimizations, as optimized programs are themselves valid *synthesizable* programs; in this specific example program 9. P^2 successfully predicts the performance of program 9.

Table 5 summarizes the accuracy over all experiments for the two GPU systems. Overall, P^2 delivers 52% top-1 accuracy, 75% top-5 accuracy, and 92% top-10 accuracy.

6 Related Work

Parallelism forms. Recent work has explored combinations of parallelism forms. Jia et al. (2018b) propose layer-wise parallelism that allows each network layer to use a different parallelization strategy. FlexFlow (Jia et al., 2019) uses guided randomized search to find a fast parallelism combination. Narayanan et al. (2021) combine pipeline, tensor and data parallelism to scale transformer training to thousands of GPUs, extending previous model parallelism work (Shoeybi et al., 2020). ZeRO-DP (Rajbhandari et al., 2020) includes three optimization stages partitioning over optimizer state, gradient, and parameters. To our best knowledge, no prior work has discussed parallelism placement, and typically

commit to a specific placement (e.g. model parallelism within a host, batch parallelism across (Narayanan et al., 2021).) which can get involved when multiple axes occur. Finally, there exists a rich body of work on operator mapping, e.g. (Addanki et al., 2019; Gao et al., 2018; Mirhoseini et al., 2017), but the focus there does not include the structured forms of parallelism and reductions we address.

Program synthesis for communication. For a given topology, SCCL (Cai et al., 2021) synthesizes optimal collective algorithms as a sequence of sends, but the work has focused on the single-node setting. SCCL takes a more fine-grained system topology as a graph with bandwidth constraints on GPUs and edges, and uses an SMT solver to synthesize algorithms. TACCL (Shah et al., 2021), similarly, models the synthesis problem as a mixed integer linear programming (MILP) problem, improving on the SCCL formalism by better supporting heterogeneous link characteristics, and uses heuristics to solve the resulting problem. It is possible for P^2 to use SMT or a MILP formulation, but we found that the structure we imposed on the problem already enables fast enumerative syntax-guided synthesis. Blink (Wang et al., 2020) performs the synthesis based on an approximate spanning-tree packing algorithm for intra-node communication, but always uses program (i) (Figure 10i) for inter-node communication. While both SCCL and Blink focus on synthesizing collective operations using communication primitives (i.e., sends and receives) in a single node, P^2 synthesizes reduction algorithms using collective operations in multi-node systems. Moreover, while both frameworks reply on low-level implementation details (e.g., CUDA), P^2 enjoys being platform-independent, as it targets XLA collectives and can thus support all backends that XLA readily supports. PLink (Luo et al., 2020) probes the network locality and groups nodes by physical affinity, and performs program (i) (Figure 10i) for intra-group reduction. By contrast, P^2 synthesizes hierarchical strategies using collectives.

Reduction strategies. BlueConnect (Cho et al., 2019) propose program (ii) (Figure 10ii) for hierarchical systems, which is later generalized by FlexReduce (Lee et al., 2020) to asymmetric topologies. On the other hand, P^2 systematically generates and compares a wide range of hierarchical reduction strategies for different parallelism placements.

7 Conclusion and Outlook

We have presented a framework P^2 to synthesize structured forms of parallelism mappings and hierarchy-aware reduction strategies. Our tool can be useful for speeding up ML models, and also for establishing projections about communication costs when investigating new system hierarchies. P^2 focuses on hierarchical reduction, and future work includes extending P^2 to include mappings for pipeline parallelism (Huang et al., 2019); and also finding ways to model and synthesize for non-regular network configurations, e.g. where GPU hosts are spread across the data center with different communication characteristics across subsets of these hosts.

Acknowledgements

We thank Chris Jones and Tom Hennigan for their help with the experiment settings, and Adam Paszke and the anonymous reviewers for their constructive feedback.

References

Addanki, R., Venkatakrishnan, S. B., Gupta, S., Mao, H., and Alizadeh, M. *Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Alur, R., Bodik, R., Juniwal, G., Martin, M. M., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. *Syntax-guided synthesis*. IEEE, 2013.

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., et al. Deep speech 2: end-to-end speech recognition in english and mandarin. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*, pp. 173–182, 2016.

Cai, Z., Liu, Z., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., and Saarikivi, O. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pp. 62–75, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382946. doi: 10.1145/3437801.3441620.

Cho, M., Finkler, U., and Kung, D. Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning. In *Proceedings of the 2nd SysML Conference*, 2019.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pp. 1223–1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

DeepMind. Resnet50 on imagenet2012, 2020. URL https://github.com/deepmind/dm-haiku/blob/main/examples/imagenet/train.py.

Gao, Y., Chen, L., and Li, B. Spotlight: Optimizing device placement for training deep neural networks. In Dy, J. G. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1662–1670. PMLR, 2018. URL http://proceedings.mlr.press/v80/gao18a.html.

GCP. Gpus on compute engine. https://cloud.google.com/compute/docs/gpus, 2021. Accessed: 2021-10-07.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32:103–112, 2019.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018a.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring hidden dimensions in accelerating convolutional neural networks. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2274–2283. PMLR, 10–15 Jul 2018b. URL https://proceedings.mlr.press/v80/jia18a.html.

Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826, 2018c. URL http://arxiv.org/abs/1804.06826.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 1–13, 2019.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25: 1097–1105, 2012.

Lee, J., Hwang, I., Shah, S., and Cho, M. Flexreduce: Flexible all-reduce for distributed deep learning on asymmetric network topology. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020. doi: 10.1109/DAC18072.2020.9218538.

Luo, L., West, P., Krishnamurthy, A., Ceze, L., and Nelson, J. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. In Dhillon, I. S., Papailiopoulos, D. S., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020. URL https://proceedings.mlsys.org/book/293.pdf.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 2430–2439. JMLR.org, 2017.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on gpu clusters using megatron-lm. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2021.

NVidia. The nvidia collective communication library (nccl), 2021. URL https://developer.nvidia.com/nccl.

NVIDIA. Nvidia nvlink and nvswitch. https://www.nvidia.com/en-gb/data-center/nvlink/, 2021. Accessed: 2021-10-07.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models, 2020.

Sanders, P., Speck, J., and Träff, J. L. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

Sergeev, A. and Balso, M. D. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

Shah, A., Chidambaram, V., Cowan, M., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., Saarikivi, O., and Singh, R. Synthesizing collective communication algorithms for heterogeneous networks with TACCL. *CoRR*, abs/2111.04867, 2021. URL https://arxiv.org/abs/2111.04867.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

Wang, G., Venkataraman, S., Phanishayee, A., Thelin, J., Devanur, N. R., and Stoica, I. Blink: Fast and generic collectives for distributed ML. In Dhillon, I. S., Papailiopoulos, D. S., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020. URL https://proceedings.mlsys.org/book/299.pdf.

A EXPERIMENTS

Parallelism axes	Reduce axes	Synthesis time	Simulation time		Programs		Parallelism matrix	AllReduce		Optimal		Speedup	
			Ring	Tree	Ring	Tree		Ring	Tree	Ring	Tree	Ring	Tree
2 nodes each with 16 A100													
[32]	[0]	0.244	0.458	0.860	9/47	3/47	[2 16]	4.74	3.57	3.18	2.83	1.49	1.26
[2 16]	[0]	0.004	0.014	0.019	0/6	1/6	[[1 2] [2 8]]	0.12	0.17	0.12	0.15	1	1.13
							[[2 1] [1 16]]	36.69	37.18	36.69	37.18	1	1
	[1]	0.065	0.019	0.208	10/50	1/50	[[2 1] [1 16]]	0.18	0.22	0.18	0.20	1	1.1
							[[1 2] [2 8]]	8.44	4.82	4.77	4.82	1.77	1
[4 8]	[0]	0.026	0.125	0.137	11/50	5/50	[[1 4] [2 4]]	0.15	0.20	0.15	0.17	1	1.18
							[[2 2] [1 8]]	28.91	19.61	18.48	18.5	1.56	1.06
	[1]	0.030	0.135	0.149	12/50	3/50	[[2 2] [1 8]]	0.17	0.21	0.17	0.18	1	1.17
							[[1 4] [2 4]]	16.04	9.15	8.97	9.01	1.79	1.02
[8 4]	[0]	0.033	0.136	0.151	14/50	2/50	[[1 8] [2 2]]	0.17	0.21	0.17	0.19	1	1.11
							[[2 4] [1 4]]	16.84	9.26	9.19	9.10	1.83	1.02
	[1]	0.026	0.124	0.136	11/50	7/50	[[2 4] [1 4]]	0.15	0.20	0.15	0.17	1	1.18
							[[1 8] [2 2]]	28.78	18.93	18.48	18.00	1.56	1.05
[16 2]	[0]	0.067	0.195	0.214	10/50	2/50	[[1 16] [2 1]]	0.18	0.22	0.18	0.19	1	1.16
							[[2 8] [1 2]]	8.86	5.34	5.05	5.21	1.75	1.02
	[1]	0.005	0.011	0.012	0/6	1/6	[[2 8] [1 2]]	0.11	0.17	0.11	0.14	1	1.21
							[[1 16] [2 1]]	36.84	36.82	36.84	36.82	1	1
4 nodes each with 16 A100													
[64]	[0]	1.161	1.868	2.01	6/47	5/47	[[4 16]]	5.18	5.75	4.29	4.74	1.21	1.21
[2 32]	[0]	0.006	0.019	0.022	0/6	1/6	[[1 2] [4 8]]	0.12	0.17	0.12	0.15	1	1.13
							[[2 1] [2 16]]	37.16	36.94	37.04	36.94	1.003	1
	[1]	0.463	0.160	1.27	18/94	10/94	[[2 1] [2 16]]	4.81	3.41	3.05	3.07	1.58	1.11
							[[1 2] [4 8]]	8.74	9.89	6.91	8.00	1.26	1.24
[4 16]	[0]	0.043	0.259	0.287	12/53	10/53	[[1 4] [4 4]]	0.15	0.20	0.15	0.17	1	1.18
							[[2 2] [2 8]]	28.77	19.81	18.24	18.55	1.57	1.07
							[[4 1] [1 16]]	56.13	89.70	55.99	56.13	1.003	1.60
	[1]	0.133	0.619	0.704	21/97	9/97	[[4 1] [1 16]]	0.18	0.22	0.18	0.19	1	1.16
							[[2 2] [2 8]]	8.39	4.99	4.81	4.82	1.74	1.04
							[[1 4] [4 4]]	17.70	19.03	13.38	14.85	1.32	1.28
[8 8]	[0]	0.091	0.512	0.590	20/97	7/97	[[1 8] [4 2]]	0.17	0.21	0.17	0.18	1	1.17
							[[2 4] [2 4]]	16.52	9.18	9.21	9.18	1.79	1
							[[4 2] [1 8]]	34.05	41.23	28.86	29.79	1.18	1.38
	[1]	0.084	0.508	0.598	19/97	17/97	[[4 2] [1 8]]	0.17	0.21	0.17	0.18	1	1.17
							[[2 4] [2 4]]	15.68	9.43	8.92	9.22	1.76	1.02
							[[1 8] [4 2]]	33.92	41.06	27.93	29.36	1.21	1.40
[16 4]	[0]	0.149	0.631	0.712	21/97	17/97	[[1 16] [4 1]]	0.18	0.22	0.18	0.2	1	1.1
							[[2 8] [2 2]]	8.81	5.42	5.01	5.25	1.76	1.03
							[[4 4] [1 4]]	18.30	20.13	14.13	14.90	1.30	1.35
	[1]	0.042	0.261	0.297	13/53	5/53	[[4 4] [1 4]]	0.15	0.20	0.15	0.18	1	1.11
							[[2 8] [2 2]]	28.68	18.47	19.09	18.41	1.50	1.003
							[[1 16] [4 1]]	57.13	85.22	56.23	55.63	1.02	1.53
[32 2]	[0]	0.483	1.183	1.30	20/94	15/94	[[2 16] [2 1]]	4.74	3.99	3.14	3.13	1.51	1.27
							[[4 8] [1 2]]	9.37	10.41	7.23	7.71	1.30	1.35
	[1]	0.008	0.020	0.022	0/6	1/6	[[4 8] [1 2]]	0.11	0.17	0.11	0.15	1	1.13
							[[2 16] [2 1]]	37.18	37.10	37.18	37.10	1	1
[16 2 2]	[0 2]	0.968	2.36	2.55	25/188	21/188	[[1 16] [2 1] [2 1]]	4.79	3.69	4.63	2.71	1.03	1.36
							[[2 8] [2 1] [1 2]]	4.91	3.97	3.10	2.93	1.58	1.35
							[[2 8] [1 2] [2 1]]	9.05	10.29	9.03	9.46	1.002	1.09
							[[4 4] [1 2] [1 2]]	9.14	10.32	7.08	7.87	1.29	1.31
[8 2 4]	[0 2]	1.107	2.88	3.08	28/235	22/235	[[1 8] [2 1] [2 2]]	4.80	3.62	4.64	2.67	1.03	1.36
							[[2 4] [2 1] [1 4]]	4.82	3.87	3.12	3.08	1.54	1.26
							[[2 4] [1 2] [2 2]]	8.91	9.68	8.91	9.29	1	1.04
							[[4 2] [1 2] [1 4]]	9.19	10.24	7.02	7.69	1.31	1.33
							[[1 8] [1 2] [4 1]]	9.21	10.37	5.50	8.72	1.7	1.19
[4 2 8]	[0 2]	1.143	2.86	3.09	32/235	24/235	[[2 2] [2 1] [1 8]]	4.74	3.54	3.04	2.99	1.56	1.18
							[[1 4] [2 1] [2 4]]	4.77	3.77	4.48	3.54	1.06	1.06
							[[4 1] [1 2] [1 8]]	8.73	9.81	7.00	8.12	1.25	1.21
							[[1 4] [1 2] [4 2]]	9.12	9.98	8.65	8.80	1.05	1.13

							[[2 2] [1 2] [2 4]]	9.12	10.36	9.02	9.77	1.01	1.06
[2 2 16]	[0 2]	0.927	2.32	2.51	29/188	16/188	[[2 1] [2 1] [1 16]]	4.82	3.91	2.99	3.00	1.61	1.30
							[[1 2] [2 1] [2 8]]	5.28	4.29	4.77	3.66	1.11	1.17
							[[2 1] [1 2] [2 8]]	9.32	9.61	7.10	7.97	1.31	1.21
							[[1 2] [1 2] [4 4]]	9.81	9.79	9.81	9.37	1	1.04
2 nodes each with 8 V100													
[16]	[0]	0.058	0.158	0.178	12/47	0/47	[[2 8]]	4.58	2.30	2.41	2.3	1.90	1
[2 8]	[0]	0.0035	0.014	0.134	0/6	0/6	[[1 2] [2 4]]	9.37	8.44	9.37	8.44	1	1
							[[2 1] [1 8]]	14.53	14.55	14.53	14.55	1	1
	[1]	0.0257	0.206	0.128	6/50	2/50	[[2 1] [1 8]]	0.28	0.40	0.28	0.30	1	1.33
							[[1 2] [2 4]]	6.59	3.70	6	3.70	1.10	1
[4 4]	[0]	0.0181	0.243	0.161	0/50	8/50	[[1 4] [2 2]]	12.55	13.01	12.55	13.01	1	1
							[[2 2] [1 4]]	13.11	16.11	13.11	13.5	1	1.19
	[1]	0.0182	0.181	0.116	10/50	2/50	[[2 2] [1 4]]	2.96	0.43	2.29	0.43	1.29	1
							[[1 4] [2 2]]	10.95	7.42	7.52	7.26	1.46	1.02
[8 2]	[0]	0.0272	0.137	0.304	1/50	4/50	[[1 8] [2 1]]	0.28	0.40	0.28	0.30	1	1.33
							[[2 4] [1 2]]	14.24	15.47	14.2	14.48	1.003	1.07
	[1]	0.0036	0.009	0.023	0/6	1/6	[[2 4] [1 2]]	0.32	0.33	0.32	0.33	1	1
							[[1 8] [2 1]]	14.51	14.81	14.51	14.47	1	1.02
4 nodes each with 8 V100													
[32]	[0]	0.209	0.507	0.770	11/47	7/47	[[4 8]]	4.83	4.57	4.83	3.66	1	1.25
[2 16]	[0]	0.0043	0.024	0.027	0/6	1/6	[[1 2] [4 4]]	9.17	8.42	9.17	8.42	1	1
							[[2 1] [2 8]]	14.47	14.52	14.47	14.51	1	1.0007
[2 16]	[1]	0.121	0.536	0.621	10/94	7/94	[[2 1] [2 8]]	4.37	2.26	2.42	2.25	1.81	1.004
							[[1 2] [4 4]]	7.18	8.10	7.16	7.10	1.003	1.14
[4 8]	[0]	0.027	0.303	0.356	1/53	9/53	[[1 4] [4 2]]	12.60	13.00	12.60	13.00	1	1
							[[2 2] [2 4]]	13.69	16.07	13.69	13.56	1	1.19
							[[4 1] [1 8]]	22.21	30.55	22.04	21.49	1.001	1.42
[4 8]	[1]	0.096	0.421	0.498	15/97	11/97	[[4 1] [1 8]]	0.28	0.40	0.28	0.30	1	1.33
							[[2 2] [2 4]]	6.60	3.82	5.78	3.74	1.14	1.02
							[[1 4] [4 2]]	12.94	15.47	11.21	12.12	1.15	1.28
[8 4]	[0]	0.103	0.579	0.701	2/97	13/97	[[1 8] [4 1]]	0.28	0.39	0.28	0.3	1	1.3
							[[2 4] [2 2]]	14.25	15.48	14.23	14.49	1.001	1.07
							[[4 2] [1 4]]	14.84	19.90	1.84	15.33	1	1.30
	[1]	0.0435	0.213	0.241	11/53	2/53	[[4 2] [1 4]]	2.96	0.43	2.29	0.43	1.29	1
							[[2 4] [2 2]]	10.98	7.34	7.58	7.34	1.45	1
							[[1 8] [4 1]]	21.74	30.42	21.74	21.71	1	1.40
[16 2]	[0]	0.168	0.597	0.719	12/94	7/94	[[2 8] [2 1]]	4.47	2.25	2.44	2.25	1.83	1
							[[4 4] [1 2]]	15.00	17.58	14.99	15.01	1.0007	1.17
	[1]	0.004	0.016	0.019	0/6	1/6	[[4 4] [1 2]]	0.32	0.33	0.32	0.33	1	1
							[[2 8] [2 1]]	14.53	14.89	14.53	14.66	1	1.02
[2 2 8]	[0 2]	0.229	1.105		14/188		[[1 2] [2 1] [2 4]]	4.29		4.29		1	
							[[2 1] [2 1] [1 8]]	4.57		2.4		1.90	
							[[2 1] [1 2] [2 4]]	7.17		6.95		1.03	
							[[1 2] [1 2] [4 2]]	9.41		9.41		1	
[8 2 2]	[0 2]	0.243	1.184		17/188		[[1 8] [2 1] [2 1]]	4.40		4.40		1	
							[[2 4] [2 1] [1 2]]	4.80		2.35		2.04	
							[[4 2] [1 2] [1 2]]	9.36		9.35		1.001	
							[[2 4] [1 2] [2 1]]	15.02		14.95		1.005	

B PROOF OF EXPRESSIVENESS BETWEEN SYNTHESIS HIERARCHIES

Recall our definitions of synthesis hierarchies:

(a) System hierarchy ($[1\ 2\ 2\ 4]$)
(b) Column-based parallelism factors ($[1\ 1\ 1\ 2\ 2\ 1\ 2\ 2]$)
(c) Row-based parallelism factors ($[1\ 1\ 2\ 2\ 1\ 2\ 1\ 2]$)
(d) Reduction axis parallelism factors ($[1\ 2\ 1\ 2]$)

Note that while we can assume all system hierarchies (i.e., (a), and thus (b) and (c)) start with a level of 1, e.g., [(rack, 1), (server, 2), (CPU, 2), (GPU, 4)], it may not be the case for (d). To make it a complete synthesis hierarchy, we will append a (root, 1) as the root of (d).

Our goal is to prove

Theorem 3.2. *(d) $\geq$ (c) $\geq$ (b) $\geq$ (a).*

We split our goal into three parts: (1) (b) $\geq$ (a) (Appendix B.1 Lemma B.1); (2) (c) $\geq$ (b) (Appendix B.2 Lemma B.7); (2) (d) $\geq$ (c) (Appendix B.3 Lemma B.8), and prove them separately.

During the proof, we often use parallelism factors (x) and their corresponding levels (e) interchangeably.

B.1 Part 1

Lemma B.1. *(b) $\geq$ (a).*

Proof. This lemma is intuitive, as (b) expands (a), so can be used to express any communication patterns that can be formed by (a).

Suppose (a) is $[x_0\ x_1\ \cdots\ x_n]$,
and the parallelism axes has size $0..m$,
then (b) is $[x_{00}\ \cdots\ x_{0m}\ x_{10}\ \cdots\ x_{1m}\ \cdots\ x_{n0}\ \cdots\ x_{nm}]$,
with $\prod_{j=0}^{m} x_{ij} = x_i$.

Now we show that every reduction instruction given by (a), a reduction instruction can be formed by (b) expressing the same reduction.

Case 1 : (a) uses (x_i, InsideGroup, $\mathcal{C}$).

Then (b) can use (x_{i0}, InsideGroup, $\mathcal{C}$) to express the same reduction.

To see why this is true, note that x_{ij}, forall j, represents the same system hierarchy as x_i. So x_{i0} and InsideGroup form the same device groups as x_i and InsideGroup.

Case 2 : (a) uses (x_i, Parallel(x_j), $\mathcal{C}$).

Then (b) can use (x_{i0}, Parallel(x_{jm}), $\mathcal{C}$) to express the same reduction.

Here, the use of x_{i0} ensures that we form the same reduction groups, and then we use x_{jm} (instead of x_{j0}) to get the first (second/etc) device from each reduction group that connects to the same x_{jm}. As x_{jm} is the last parallelism factor for x_j, this makes sure that we only connect devices that connect to the same x_j.

Case 3 : (a) uses (x_i, Master(x_j), $\mathcal{C}$).

Then (b) can use (x_{i0}, Master(x_{jm}), $\mathcal{C}$) to express the same reduction. The case is similar as the above one.

□

B.2 Part 2

B.2.1 Semantically valid reduction

Lemma B.2 (Parallel reduction)**.** *Consider a synthesis hierarchy $[x_0\ x_1\ \cdots\ x_j\ \cdots\ x_i\ \cdots x_n]$, and the device groups formed by (x_i, Parallel(x_j), $\mathcal{C}$), we have:*

1. *a device group to be reduced has size $x_{j+1} \times ... \times x_i$, and we have $x_0 \times x_1 \times ... \times x_j \times x_{i+1} \times ... \times x_n$ number of such groups.*
2. *a device group has been partitioned over x_{j+1}, ..., x_i.*

Proof. We can derive the observation from the semantics of Parallel. In particular, we first form reduction groups for devices connected to the same x_i. So each reduction group has size $size_1 = x_{i+1} \times ... \times x_n$.

Then, because of Parallel, we reduce all first (second/etc) devices in the reduction group if they are connected to the same x_j. Since each x_j owns $size_2 = x_{j+1} \times ... \times x_i$ different reduction groups, the device group we form is exactly of $size_2$. And for each x_j, we have $size_1$ of such device groups. Since we have $size_3 = x_0 \times ... \times x_j$ different x_j, we have in total $size_3 \times size_1$ device groups.

The second observation is similar. In particular, since device groups are formed by each x_j, which owns $size_2$ different x_is with different $x_{j+1}, ..., x_i$, the device groups we form are exactly partitioned over $x_{j+1}, ..., x_i$.

□

Lemma B.3 (Partitioning over reduction axes)**.** *Given reduction axes, for a reduction instruction to be semantically valid, all device groups to be reduced must only be partitioned over the reduction axes.*

Proof. Suppose we want to reduce over reduction axes A, and we reduce between d_j and d_j that have been, possibly among others, partitioned over B (different from A). Then, after reduction both devices contain data that differs in B.

Now the desired final state (where data should only be reduced if they have different A) becomes unreachable for both devices, as we can never recover the state of the device since according to the semantics, once a row has grown it will never get reduced back. □

Corollary B.4 (Semantically valid parallel reduction)**.** *Consider a synthesis hierarchy* $[x_0\ x_1\ \cdots\ x_j\ \cdots\ x_i\ \cdots x_n]$*, given some reduction axes, a reduction instruction* $(x_i, \mathsf{Parallel}(x_j), \mathcal{C})$ *is only semantically valid if all of* $x_{j+1}, ..., x_i$ *are either 1, or on the reduction axes.*

Proof. By Lemma B.2, we know each device group is partitioned over $x_{j+1}, ..., x_i$. However, Lemma B.3 shows that for each device group to be semantically valid, they can only be partitioned over the reduction axes. Therefore, all xs in $x_{j+1}, ..., x_i$. should either be 1, or on the reduction axes. □

Lemma B.5 (Semantically valid InsideGroup reduction)**.** *Consider a synthesis hierarchy* $[x_0\ x_1\ \cdots\ x_j\ \cdots\ x_i\ \cdots x_n]$*, given some reduction axes, a reduction instruction* $(x_i, \mathsf{InsideGroup}, \mathcal{C})$ *is only semantically valid if all of* $x_{i+1}, ..., x_n$ *are either 1, or on the reduction axes.*

Proof. Similar as Lemma B.2, except in this case we know that we are reducing devices over $x_{i+1}, ..., x_n$.

□

Lemma B.6 (Semantically valid master reduction)**.** *Consider a synthesis hierarchy* $[x_0\ x_1\ \cdots\ x_j\ \cdots\ x_i\ \cdots x_n]$*, given some reduction axes, a reduction instruction* $(x_i, \mathsf{Master}(x_j), \mathcal{C})$ *is only semantically valid if all of* $x_{j+1}, ..., x_n$ *are either 1, or on the reduction axes.*

Proof. Note that in this case we require all parallelism factors up until x_n (as with InsideGroup, rather than x_i as with Parallel). The trickiness here is that the case of Master is different from Parallel: while in Parallel we know that we will reduce in parallel everything that is not in the range of Parallel (Lemma B.2), with Master we reduce only the *first* reduction group. So it is important to guarantee that we form exactly the same first inner reduction groups.

We prove the result by contradiction. Suppose $x_{j+1}, ..., x_n$ contains a parallelism factor x' that is not 1 nor is on the reduction axes. Then there are two possibilities.

(1) x' is part of $x_{j+1}, ..., x_i$, i.e., it is covered by the reduction. Then we will reduce devices of different x'. According to lemma B.3, the reduction is invalid.

(2) x' is part of $x_{i+1}, ..., x_n$, i.e., it is not covered by the reduction, but it affects the reduction groups we form. Suppose reduction axes are A, and x' is on level B, with $B \neq A$. Then since x' is part of $x_{i+1}, ..., x_n$, we can find within $x_{i+1}, ..., x_n$ a device d_{0B}, that has only different B with the very first device d_0.

Suppose d_0 reduces with some device d_1 in this master reduction. Since the reduction is valid, d_0 and d_1 differ only in A.

Then in the same way we find d_{dB} to d_0, we can find a device d_{1B}, that is only different with d_1 in B.

However, note that this is a master reduction, and d_0 and d_{0B} belong to the same $x_{i+1}, ..., x_n$ group, so the master reduction will only reduce d_0 and d_1, but only d_{0B} and d_{1B}.

Now we can show that the final desired state becomes unreachable. In particular, note that d_{0B} and d_{1B} will never get reduced: every reduction that reduces d_{0B} and d_{1B} will reduce d_0 and d_1 as well (Figure 6 is useful here). But since d_0 and d_1 has been reduced already, re-reducing the devices is an invalid reduction step. So we will never be able to reduce d_{0B} and d_{1B} again. And thus the master reduction is invalid.

□

Lemma B.7. *(c) ≥ (b).*

Proof. **Case 1** (b) has $(e_2, \mathsf{Parallel}(e_1), \mathcal{C})$.

Based on Corollary B.4, all non-reduction parallelism factors column-wisely between e_1 (exclusive) and e_2 (inclusive) can only be 1.

Now we construct a reduction instruction for (c) that expresses the same reduction. Suppose the reduction step in (b) covers parallelism factors $e_i, ..., e_j$ on the reduction axis. Note that if the reduction step in (b) does not cover any parallelism factors, that means it forms groups of one device, and in that case this does not form a reduction and won't be generated by P^2.

Let e'_1 be the level right before e_i row-wisely in the synthesis hierarchy (c), and let e'_2 be e_j. Then $(e'_2, \mathsf{Parallel}(e'_1), \mathcal{C})$ is a desired reduction instruction. Indeed, we can derive from Lemma B.2 that this reduction instruction forms the same device groups as (b).

The example from the paper is repeated below, with the reduction axis highlighted.

$$e_1 \rightarrow \begin{bmatrix} x_{0,0} & 1 & 1 & x_{0,3} \\ \boldsymbol{x_{1,0}} & 1 & 1 & x_{1,3} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ 1 & 1 & \mathbf{1} & x_{3,3} \end{bmatrix} e_2 \qquad \begin{bmatrix} x_{0,0} & 1 & 1 & x_{0,3} \\ x_{1,0} & 1 & 1 & \boldsymbol{x_{1,3}} \\ x_{2,0} & x_{2,1} & \boldsymbol{x_{2,2}} & x_{2,3} \\ 1 & 1 & 1 & x_{3,3} \end{bmatrix} \begin{matrix} \leftarrow e'_1 \\ e'_2 \end{matrix}$$

Case 2 (b) has $(e, \mathsf{InsideGroup}, \mathcal{C})$.

This case can be reasoned in a similar way as the previous case, by using Lemma B.5.

We construct a reduction instruction for (c) that expresses the same reduction. Suppose the reduction step in (b) covers parallelism factors $e_i, ..., e_j$ on the reduction axis. Let e' be the level corresponding to the parallelism factor right before e_i row-wisely. Then $(e', \mathsf{InsideGroup}, \mathcal{C})$ is a desired reduction instruction.

Below we give an example, with the reduction axis highlighted.

$$e \rightarrow \begin{bmatrix} x_{0,0} & 1 & 1 & 1 \\ \boldsymbol{x_{1,0}} & 1 & 1 & 1 \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ 1 & 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} x_{0,0} & 1 & 1 & 1 \\ x_{1,0} & 1 & 1 & \mathbf{1} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ 1 & 1 & 1 & 1 \end{bmatrix} \leftarrow e'$$

On the other hand, we can also show a counterexample of (b) $\geq$ (c), where in the following hierarchy, $(e', \mathsf{InsideGroup}, \mathcal{C})$ is a valid reduction in (c) but there is no way in (b) that can simulate the same reduction.

$$\begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & \boldsymbol{x_{1,3}} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ 1 & 1 & 1 & 1 \end{bmatrix} \leftarrow e'$$

Similar counterexamples can also be shown for other cases.

Case 3 (b) has $(e_2, \mathsf{Master}(e_1), \mathcal{C})$.

This case can be reasoned in a similar way as the previous case, by using Lemma B.6.

We construct a reduction instruction for (c) that expresses the same reduction. The choice of e'_1 and e'_2 is the same as Case 1. Then $(e'_2, \mathsf{Parallel}(e'_1), \mathcal{C})$ is a desired reduction instruction.

Below we give an example, with the reduction axis highlighted.

$$e_1 \rightarrow \begin{bmatrix} x_{0,0} & 1 & 1 & 1 \\ \boldsymbol{x_{1,0}} & 1 & 1 & 1 \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ 1 & 1 & \mathbf{1} & \cancel{\leftarrow 1} \end{bmatrix} e_2 \quad \begin{bmatrix} x_{0,0} & 1 & 1 & 1 \\ x_{1,0} & 1 & 1 & \mathbf{1} \\ x_{2,0} & x_{2,1} & \boldsymbol{x_{2,2}} & \cancel{x_{2,3}} \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{matrix} \leftarrow e'_1 \\ e'_2 \end{matrix}$$

□

B.3 Part3

Lemma B.8. *(d) $\geq$ (c).*

Proof. Most reasoning is the same as Lemma B.7. For each case of a (c) reduction instruction, we show how we construct the reduction instruction for (d) that expresses the same reduction. Remember that we attach a $(root, 1)$ to synthesis hierarchy (d).

Case 1 (c) has $(e, \mathsf{InsideGroup}, \mathcal{C})$.

We construct a reduction instruction for (d) that expresses the same reduction.

If the reduction covers the whole parallelism factor, then (root, $\mathsf{InsideGroup}, \mathcal{C}$) is a desired reduction instruction.

$$e \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ \cancel{x_{1,0}} & \boldsymbol{x_{1,1}} & 1 & 1 \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

If the reduction covers some (but not all) parallelism factors on the reduction axis, then e itself is on the reduction aixs.

Then $(e, \mathsf{InsideGroup}, \mathcal{C})$ is a desired reduction instruction.

$$e \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & 1 & 1 \\ \cancel{x_{2,0}} & \boldsymbol{x_{2,1}} & x_{2,2} & x_{2,3} \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Again, if the reduction does not cover any parallelism factors on the reduction axis, then it forms reduction groups of size 1, and thus this does not form a reduction and won't be generated by P^2.

Below we also show a counterexample of (c) $\geq$ (d), where in the following hierarachy, $(e, \mathsf{InsideGroup}, \mathcal{C})$ is a valid reduction in (d) but there is no way in (c) that can simulate the same reduction since x_3 can be arbitrary numbers.

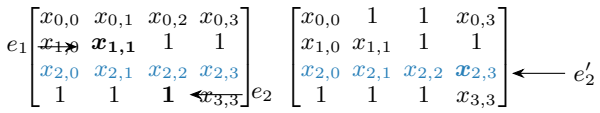
$$e \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & 1 & 1 \\ \cancel{x_{2,0}} & \boldsymbol{x_{2,1}} & x_{2,2} & x_{2,3} \\ x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}$$

Case 2 (c) has $(e_2, \mathsf{Parallel}(e_1), \mathcal{C})$.

Similar as the previous case, we construct a reduction instruction for (d) that expresses the same reduction.

Suppose the reduction step in (c) covers parallelism factors $e_i, ..., e_j$ on the reduction axis. Let e'_1 be the level in the synthesis hierarchy (d) right before e_i row-wisely. If the reduction covers the whole parallelism factor, then e' would be the root. Let e'_2 be e_j. Then $(e'_2, \mathsf{Parallel}(e'_1), \mathcal{C})$ is a desired reduction instruction.

In the following example, e'_1 = root.



Case 3 (c) has $(e_2, \mathsf{Master}(e_1), \mathcal{C})$.

This case is exactly the same as the case for Parallel.

□