ComScribe: A Communication Monitoring Tool for Multi-GPU Platforms

by

Palwisha Akhtar

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

January 18, 2021

ComScribe: A Communication Monitoring Tool for Multi-GPU Platforms

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Palwisha Akhtar

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Asst. Prof. Didem Unat (Advisor)

_____

Prof. Öznur Özkasap

_____

Asst. Prof. Kamer Kaya

Date: _____

I dedicate this work to my Dad and Mom

ABSTRACT

ComScribe: A Communication Monitoring Tool for Multi-GPU Platforms
Palwisha Akhtar
Master of Science in Computer Science and Engineering
January 18, 2021

GPU communication plays a critical role in performance and scalability of multi-GPU accelerated applications. With the ever increasing methods and types of communication, it is often hard for the programmer to know the exact amount and type of communication taking place in an application. Though there are prior works that detect communication in distributed systems for MPI and multi-threaded applications on shared memory systems, to our knowledge, none of these works identify intra-node GPU communication.

In this work we present ComScribe, a tool that identifies and categorizes types of communication among all GPU-GPU and CPU-GPU pairs in a node. Our tool is built on top of NVIDIA's profiler *nvprof* for capturing intra-node point-to-point communication resulting from explicit communication primitives, Unified Memory operations, and Zero-copy Memory transfers. For monitoring collective GPU-GPU communication in a node, ComScribe intercepts NCCL's collective primitives at runtime and records data transfers among GPUs. It visualizes data movement as a communication matrices for both number of bytes transferred and the number of transfers.

To validate our tool on 16 GPUs, we present communication patterns of 13 micro- and 3 macro-benchmarks from NVIDIA, Comm|Scope, and MGBench benchmark suites. To demonstrate tool's capabilities in real-life applications, we also present insightful communication matrices of three deep neural network models. All in all, ComScribe can guide the programmer in identifying groups of communicating GPUs, the volume of communication, and types of primitives used. This offers avenues to detect performance bottlenecks and more importantly communication bugs in an application.

ÖZETÇE

Yüksek Lisans Tez Başlığı
Palwisha Akhtar
Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans
18 Ocak 2021

GPU iletişimi, çoklu GPU hızlandırmalı uygulamaların performansında ve ölçeklenebilirliğinde kritik bir rol oynar. Giderek artan iletişim yöntemleri ve türleriyle, programcının bir uygulamada gerçekleşen iletişim miktarını ve türünü bilmesi genellikle zordur. MPI için dağıtık sistemlerdeki iletişimi ve paylaşılan bellek sistemlerinde çok iş parçacıklı uygulamaları algılayan önceki çalışmalar olsa da, bildiğimiz kadarıyla, bu çalışmaların hiçbiri düğüm içi GPU iletişimini tanımlamamaktadır.

Bu çalışmada, bir düğümdeki tüm GPU-GPU ve CPU-GPU çiftleri arasındaki iletişim türlerini tanımlayan ve sınıflandıran bir araç olan ComScribe'u sunuyoruz. Aracımız, açık iletişim ilkelleri, Birleşik Bellek işlemleri ve Sıfır Kopyalı Bellek aktarımlarından kaynaklanan düğüm içi noktadan noktaya iletişimi yakalamak için NVIDIA'nın profil oluşturucusu *nvprof* üzerine inşa edilmiştir. Bir düğümde toplu GPU-GPU iletişimini izlemek için ComScribe, NCCL'nin toplu ilkelerini çalışma zamanında GPU'lar arasındaki veri aktarımlarını kaydeder. Veri hareketini hem aktarılan bayt sayısı hem de aktarım sayısı için bir iletişim matrisi olarak görselleştirir.

Aracımızı 16 GPU'da doğrulamak için, NVIDIA, Comm|Scope ve MGBench benchmark paketlerinden 13 mikro ve 3 makro karşılaştırmalı iletişim modelleri oluşturduk. Aracın yeteneklerini gerçek hayattaki uygulamalarda göstermek için, üç derin sinir ağı modelinin içgörülü iletişim matrislerini oluşturduk. Sonuç olarak, ComScribe programcıya iletişim kuran GPU gruplarını, iletişim hacmini ve kullanılan ilkel türlerini belirlemede rehberlik eder. Bu, performans darboğazlarını ve daha da önemlisi bir uygulamadaki iletişim hatalarını tespit etmek için yollar sunar.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor *Asst. Prof. Didem Unat* without whose valuable insights and consistent support this feat would not have been achievable. My colleagues from ParCoreLab *Najeeb Ahmad, Muhammad Aditya Sasongko, Fareed Mohammad Qararyah, Ilyas Turimbetov, Erhan Tezcan, Mandana Bagheri Marzijarani*, and *Muhammet Abdullah Soytürk* deserve an honorable mention who have always been a source of motivation and inspiration in a journey that often feels lonesome. I would also like to thank *Waris Gill, Khunsha Mehmood, Farjad Zafar, Ayesha Gulzar*, and *Munam Arshad* who made Koç University and Istanbul a home away from home during my time here.

I owe my success to my family; *Abu, Mama, Fareha*, and *Farwa*, whose ardent love and unflinching support has always been a source of strength to this day. Last, but not the least, I would like to thank my husband *Muneeb* for his patience and love that helped me persevere through our time apart.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

API	Application Program Interface
CUDA	Compute Unified Device Architecture
CL	Collective
CPU	Central Processing Unit
D2H.ll	Device to Host
DNN	Deep Neural Network
DP	Data Parallelism
GPU	Graphics Processing Unit
H2D	Host to Device
MPI	Message Passing Interface
NCCL	NVIDIA Collective Communications Library
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
P2P	Point-to-Point
PCIe	Peripheral Component Interconnect Express
QPI	QuickPath Interconnect
UM	Unified Memory
UVA	Unified Virtual Addressing

Chapter 1

INTRODUCTION

1.1 Motivation

Graphical Processing Units (GPUs) are increasingly becoming common and vital in compute-intensive applications such as deep learning, big data and scientific computing. With just a single GPU and exceeding working sets, memory becomes a bottleneck, inevitably shifting the trend towards multi-GPU computing. Currently, a variety of single node multi-GPU systems are available such as NVIDIA's TU102, V100-DGX2 and DGX A100 with varying number of GPUs ranging from 2 to 16, connected via NVLink based interconnects [Li et al., 2020, NVIDIA, 2017, NVIDIA, 2018a, NVIDIA, 2020c].

Traditionally, multiple GPUs in a node are interconnected via PCIe such as in DGX1 with PCIe Gen3 [NVIDIA, 2017]. A balanced tree structure is formed by the PCIe network where one GPU is connected to another through a PCIe switch. In such systems, multi-GPU application performance is constrained by the lower bandwidth of PCIe and the indirect connections between GPUs. With the introduction of GPUDirect Peer-to-Peer that enables direct data access and transfer capability among multiple GPUs and GPU-oriented interconnects such as NVLink, programmers can directly read or write local graphics memory, peer GPU memory or the system memory, all in a common shared address space [Foley and Danskin, 2017]. The higher bandwidth of such interconnects improve the performance of applications utilizing inter-GPU communication.

There are two inter-GPU communication patterns: Point-to-Point and Collective. To handle point-to-point communication between multiple GPUs, CUDA API offers various data transfer options to the programmer under the hood of Unified

Virtual Addressing (UVA), Zero-copy Memory and Unified Memory paradigms. Explicit data transfers can be carried out in the traditional way using CUDA's memory copy functions but with Zero-copy Memory or Unified Memory operations, the memory management is handled by the CUDA driver and programmer is free from the burden of tracking data allocations and transfers among host and the GPUs in a node. However, in such a scenario, users are unaware of the underlying communication taking place between the GPUs. Similarly, for collective multi-GPU communication, NVIDIA provides an optimized library for CUDA devices called the NVIDIA Collective Communication Library (NCCL) [NVIDIA, 2020e, NVIDIA, a, NVIDIA, b]. Inspired by MPI (Message Passing Interface) that defines the collective routines, NCCL implements 5 collective primitives including All-Gather, All-Reduce, Broadcast, Reduce and Reduce-Scatter. These collective communication primitives can used by the programmer to carry out data transfers among a group of devices.

In this pool of varying communication types and methods for single node multi-GPU systems, communication remains as a critical programming component and performance contributor [Sourouri et al., 2014]. As a result, identifying communication between GPUs can guide the programmer in many aspects. Firstly there are many cases of communication that result in implicit data transfers, and these unforeseen communications will occur if the programmer is indifferent towards the topology and device capabilities. For example, an application might not have any implicit communication with an all-to-all GPU topology, but might have them with a Hyper-cube Mesh topology such as in DGX-1, where some GPUs do not have direct interconnect to some others, and transfers among them may require explicit routing, as NVLink is not self-routed [Li et al., 2018]. Secondly, it is possible for there to be bugs in an application such that the result is not affected, however the bug causes an unexpected communication pattern, which may be hard to infer from debugging alone. Finally, monitoring and identifying communication among multiple GPUs can help reason about scalability issues and performance divergence between different implementations of the same application, and guide the programmer to utilize the GPU interconnects for better performance [Buono et al., 2017]. For instance, a single GPU application when naively scaled up to multiple GPUs, may follow a

master-slave communication pattern, which would mean that it is not effectively using the GPU interconnects. All in all, identifying which groups of GPUs communicate in what volume and their quantitative comparison offer avenues to detect performance bugs and tune software for scalability.

1.2 ComScribe *Overview*

Even though there are communication identification tools for MPI applications on distributed systems (e.g. EZTrace [Trahay et al., 2011]) and for multi-threaded applications on shared memory systems (e.g. ComDetective [Sasongko et al., 2019]), to our knowledge, there is no communication monitoring tool designed for single node multi-GPU systems. In this work, we propose ComScribe, a tool that can monitor, identify, and quantify different types of communication among GPU devices. The tool can generate a communication matrix that shows the amount of data movement between two pairs of GPUs or with host. In addition, it can automatically identify the types of communication i.e for point-to-point communication whether data transfers resulted from explicit transfers using CUDA primitives or implicit transfers using Zero-copy Memory or Unified Memory operations and which among the 5 collective primitives from NCCL were used for collective communication.

To enable monitoring point-to-point communication, ComScribe is built on top of NVIDIA's profiling tool *nvprof* [NVIDIA, 2020b] that provides a timeline of all activities taking place on a GPU such as kernel execution, memory copy or memory set, data transmitted or received through NVLink. However, *nvprof* does not readily generate communication matrices and the *nvprof* trace consists of extraneous information that is unnecessary for a user who is concerned with communication among GPUs. Our tool overcomes this limitation and works in two steps: First, it collects intra-node multi-GPU and CPU-GPU memory transfer information during execution with *nvprof*. Then, it performs post-processing to quantify communication among GPUs as well as the host, and identify communication types.

ComScribe also captures collective communication by intercepting NCCL primitives at runtime using LD_PRELOAD [Pulo, 2009]. It generates communication

matrices as well as timeline of all the data transfers with the type of collective communication in chronological order.

Contributions of this thesis are summarized as follows:

- We present ComScribe, a tool for generating communication matrices that show both the number of transfers and amount of data transferred between every GPU-GPU and CPU-GPU pair in a node.

- The tool can identify communication as point-to-point or collective. Furthermore,

  - For point-to-point communication, it can categorize data transfers into explicit transfers using CUDA primitives and implicit transfers using Zero-copy Memory or Unified Memory operations.
  - For collective communication, it can identify NCCL collective primitives used for communication.

- For point-to-point communication, we validate our tool against known communication patterns using 11 benchmarks from Comm|Scope [Pearson et al., 2019], MGBench [Ben-Nuun, 2017] and NVIDIA on a multi-GPU V100-DGX2 system and demonstrate how ComScribe is used for detecting a communication bug in one of these benchmarks. We also present communication matrices for 2 deep neural network models that uses CUDA explicit data transfer primitives and demonstrate how ComScribe can be used for explaining different implementations of data parallelism in deep learning.

- For collective communication, we verify collective ring-based protocols [Xu, 2018] using 5 micro-benchmarks from NVIDIA's NCCL Tests [NVIDIA, 2020d] on a multi-GPU V100-DGX2 system.

- We demonstrate ComScribe's capability to capture both point-to-point and collective GPU communication in a node on a deep neural network model from

PyTorch framework that uses NCCL's library for collective GPU communication as well as Host-Device communication.

- ComScribe is available as an open source tool at `https://github.com/ParCoreLab/ComScribe`.

The rest of the thesis is organized as follows: Chapter 2 provides necessary background on the evolution of NVIDIA GPUs in terms of device capabilities and memory management for inter-GPU communication. In Chapter 3, we categorize various data transfer options available in CUDA API for point-to-point communication and discuss collective primitives available through NCCL library for collective communication. ComScribe's design and workflow for capturing intra-node GPU communication is explained in Chapter 4. In Chapter 5, we discuss the additional features that can be added to the tool to improve its productivity and usability. We validate our tool on a number of micro- and macro-benchmarks and present its results on several deep learning applications as use-cases in Chapter 6. Related work is discussed in Chapter 7 followed by conclusion in Chapter 8 to summarize this work.

Chapter 2

BACKGROUND

In this chapter we will discuss various data exchange scenarios with CUDA data transfer primitives that are supported by NVIDIA GPUs. Traditionally, for point-to-point CPU-GPU and GPU-GPU communication, programmers initiate explicit data transfers using `cudaMemcpy`. GPU-GPU transfers involve copying through host memory, which not only requires careful tracking of device pointer allocations but may also result in performance degradation [Micikevicius, 2012]. With the introduction of GPUDirect Peer-to-Peer and GPU-oriented interconnects among multiple GPUs, one GPU can directly transfer or access data at another GPU's memory within a shared memory node [NVIDIA, 2012, Foley and Danskin, 2017, Li et al., 2018, Li et al., 2020]. For host-initiated explicit peer-to-peer memory copies, `cudaMemcpyPeer` is used, which also requires passing device IDs as arguments, so that CUDA can infer the direction of transfer.

With CUDA 4.0, NVIDIA introduced a new "unified addressing space" mode called Unified Virtual Addressing (UVA), allowing all CUDA execution – CPU and GPU – in a single address space. Applications using UVA with peer-access do not require device IDs to be specified to indicate the direction of transfer for an explicit host-initiated copy. Instead, by using `cudaMemcpy` with transfer kind `cudaMemcpyDefault`, CUDA runtime can infer the location of the data from its pointer and carry out the copy. However, if the peer access is disabled, implicit transfers through the host will occur, resulting in communication overhead.

Similar to CUDA explicit bulk transfers, data accesses between host and device are also possible with Zero-copy Memory, which requires mapped pinned memory. Pinned allocations on the host are mapped into the unified virtual address space and device kernels can directly access them [NVIDIA, 2020a]. Similarly, implicit peer-to-peer data accesses between GPUs are possible during kernel execution in

Zero-copy Memory paradigm.

In CUDA 6.0, NVIDIA introduced a single address space called Unified Memory. With data allocated using Unified Memory, CUDA becomes responsible for migrating memory pages to the memory of the accessing CPU or GPU on demand. Pascal GPU architecture via its Page Migration Engine is the first that supports virtual memory page faulting and migration [Harris, 2017]. Unified Memory offers ease of programming as managed memory is accessible to CPU and GPUs with a single pointer and does not require explicit memory transfers among them.

Collective multi-GPU communication for CUDA devices was enabled when NVIDIA introduced its Collective Communication Library (NCCL). It implements 5 collective operations including All-Gather, All-Reduce, Broadcast, Reduce and Reduce-Scatter. Each routine is implemented as CUDA kernel that ensures faster access to device memory, tight synchronization, NVLink usage and parallel reductions [Jeaugey, 2017]. NCCL library has two versions in which NCCL-V1 only supports intra-node collective communication over PCIe. Whereas, NCCL-V2 also enables inter-node collective communication and supports modern interconnects such as NVLink, NVSwitch, InfiniBand and IP networks [Li et al., 2020].

Multi-GPU applications are developed using any of the above data transfer primitives with respect to the GPU hardware, CUDA version and application model. We cover all types of communication options available for intra-node GPU-GPU of CPU-GPU communication, and present a tool for generating a communication matrix and identifying the types of communication for single node multi-GPU applications.

Chapter 3

TYPES OF COMMUNICATION

This chapter discusses point-to-point and collective GPU communication patterns in detail as they form the basis for understanding which communication types are monitored by ComScribe.

3.1 Point-to-Point Communication

For point-to-point communication, various data transfer primitives are available in the CUDA programming model, each requiring different software or hardware support. Therefore, we have divided these communication types into two categories: Peer-to-Peer and Host-Device shown in Figure 3.1. These categories are further divided into sub-categories based on the data transfer options utilized, e.g. explicit transfers using `cudaMemcpy` and `cudaMemcpyPeer`, or implicit transfers using Zero-copy Memory or Unified Memory operations. In order to make reference to the sub-categories easier, we have assigned them a case number. We should also note that synchrony is not a consideration here, because we are interested in the type and amount of the transfer. For example, whether we use `cudaMemcpy` or `cudaMemcpyAsync` does not matter, as both methods will result in the same amount of data transfer, though at different times. In the following sections, we explain each category and their sub-categories as cases.

3.1.1 Peer-to-Peer Communication

Peer-to-Peer communication refers to a data transfer between two GPUs that support peer-to-peer memory access. This type of communication depends on whether peer access is enabled or disabled and whether the application is using UVA, Zero-copy Memory or Unified Memory.

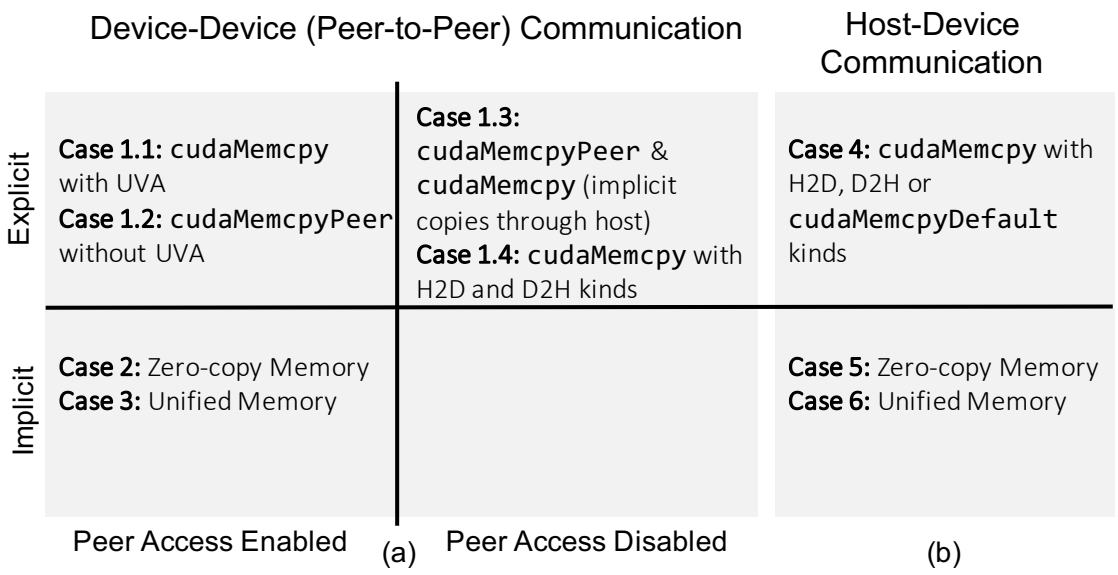


Figure 3.1: Types of point-to-point communication in a multi-GPU system. D2H stands for "Device to Host", H2D stands for "Host to Device".

Peer-to-Peer Explicit Transfers:

The host can be explicitly programmed to carry out peer-to-peer transfers using CUDA API. GPU's support for peer access determines the communication pattern in the node, for instance, in a node where GPUs do not have peer access, GPU-GPU communication will happen through the host. Furthermore, the choice of CUDA primitives depend on the utilization of UVA in data transfers. For example, `cudaMemcpyDefault` transfer kind can not be used without UVA.

Case 1: cudaMemcpy and cudaMemcpyPeer. The host can explicitly initiate a peer-to-peer transfer using either `cudaMemcpy` or `cudaMemcpyPeer`. These functions have several cases to consider:

Case 1.1: cudaMemcpy with UVA. UVA enables a single address space for all CPU and GPU memories. It allows determining the physical memory location from the pointer value and simplifies CUDA memory copy functions. For GPUs with peer access and UVA, data transfers between devices are initiated by the host, using `cudaMemcpyDefault` as the transfer kind with `cudaMemcpy`, also known as a UVA

memory copy. `cudaMemcpyDefault` is only allowed on systems that support UVA and it infers the direction of transfer from pointer values, e.g. if two pointers point to different devices, then a peer-to-peer memory transfer occurs without specifying in which memory space source and destination pointers are. An alternative to this would be to use `cudaMemcpy` with the transfer kind `cudaMemcpyDeviceToDevice` instead. Listing 1 shows an example of ping-pong data copy between two GPUs for this case.

```
1  // For UVA it is important to enable peer access
2  cudaSetDevice(0);
3  cudaDeviceEnablePeerAccess(1, 0); //Device 0 can access Device 1
4  cudaSetDevice(1);
5  cudaDeviceEnablePeerAccess(0, 0); //Device 1 can access Device 0
6
7  // Allocate buffers
8  const size_t buf_size = 1024 * 1024 * 16 * sizeof(float);
9  cudaSetDevice(0);
10 float *g0; //g0 allocated on Device 0
11 cudaMalloc(&g0, buf_size);
12 cudaSetDevice(1);
13 float *g1; //g1 allocated on Device 1
14 cudaMalloc(&g1, buf_size);
15
16 // Ping-pong copy between GPUs
17 for (int i=0; i<100; i++){
18     if (i % 2 == 0)
19         cudaMemcpy(g1, g0, buf_size, cudaMemcpyDefault);
20     else
21         cudaMemcpy(g0, g1, buf_size, cudaMemcpyDefault);
22 }
```

Listing 3.1: cudaMemcpy with UVA

First peer access is enabled between the GPUs for utilizing UVA (lines 2-5) followed by buffer allocations on each GPU (lines 8-14). For ping-pong copy, in every iteration, data is copied from buffer of one GPU to another using `cudaMemcpy` with `cudaMemcpyDefault` that can infer the direction of transfer and pointer locations

because of UVA (lines 17-22).

Case 1.2: cudaMemcpyPeer without UVA. For CUDA devices that do not support UVA but have peer access enabled, the programmer uses `cudaMemcpyPeer` and explicitly specifies the source device ID where the data to be transferred resides, as well as the destination device ID, in addition to the buffer size, source and destination pointer arguments.

Case 1.3: cudaMemcpy and cudaMemcpyPeer without Peer Access. For GPUs without peer access support or applications that explicitly disable peer access, using `cudaMemcpy` or `cudaMemcpyPeer` will result in implicit transfers through host. CUDA will transparently copy the data from source device to the host and then transfers it to the destination device.

Case 1.4: cudaMemcpy through Host. GPUs that do not have peer-to-peer access or hardware support for Unified Memory require programmer to transfer data among GPUs through the host. `cudaMemcpy` with transfer kind `cudaMemcpyDevi ceToHost` is called to copy data from the device to host, and then again with transfer kind `cudaMemcpyHostToDevice` to transfer data from host to another device. This movement of data through host introduces additional overhead, similar to *Case 1.3*.

Peer-to-Peer Implicit Transfers:

One can use UVA or Unified Memory to transfer data among GPUs with peer access with very little programming effort. For a kernel, its data can reside in device memory, in another GPU or in system memory. During the kernel execution, CUDA can infer the location of kernel's data and implicitly migrate it to the local device memory if required [NVIDIA, 2019a].

Case 2: Zero-Copy Memory. Devices that support UVA and have peer access enabled can access each others' data through the same pointer. During execution, device kernels can implicitly read or write data into the memory of their peer GPUs. Zero-copy Memory transfers occur over the device interconnects without any explicit

control by the programmer. The interconnect speed, access pattern and degree of parallelism affect the performance of these transfers [Pearson et al., 2019].

Case 3: Unified Memory. With Unified Memory, data is allocated with `cudaMallocManaged`, which returns a pointer that is accessible from CPU and any GPU in the shared memory node. In this case, if a single pointer is allocated on one device using CUDA managed memory and another device accesses it during kernel execution, then the GPU driver will migrate the page from one device memory to another [Harris, 2017]. Page migrations occur between devices in the event of a page fault, which results in an implicit data transfer. Listing 2 illustrates usage of Unified Memory on two GPUs.

```
__global__ void scalarMultiplication(float *src, float *dst){
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    dst[idx] = src[idx] * 2.0f;
}

void main(){
    // Allocate buffers
    const size_t buf_size = 1024 * 1024 * 16 * sizeof(float);
    float *g0;
    cudaMallocManaged(&g0, buf_size);
    float *g1;
    cudaMallocManaged(&g1, buf_size);

    for (int i=0; i<buf_size / sizeof(float); i++)
    { g0[i] = float(i % 4096); }

    // Kernel launch configuration
    const dim3 threads(512, 1);
    const dim3 blocks((buf_size / sizeof(float)) / threads.x, 1);

    // Run kernel on GPU 1
    cudaSetDevice(1)); // Set to Device 1
    scalarMultiplication<<<blocks, threads>>>(g0, g1);
    cudaDeviceSynchronize();
```

```
25
26      // Run kernel on GPU 0
27      cudaSetDevice(0); //Set to Device 0
28      scalarMultiplication<<<blocks, threads>>>(g1, g0);
29      cudaDeviceSynchronize();
30
31      // Verification
32      int error_count = 0;
33      for (int i=0; i<buf_size / sizeof(float); i++){
34          if (g0[i] != float(i % 4096) * 2.0f * 2.0f){
35              { if (error_count++ > 10) break;}
36          }
37      }
38  }
```

Listing 3.2: Unified Memory

Two buffers *g0* and *g1* are allocated in the managed memory using `cudaMallocManaged` (lines 8-12) followed by initializing *g0* and setting kernel configurations (lines 14-19). First, *scalarMultiplication* kernel is launched on GPU1 with *g0* and *g1* as *src* and *dst* parameters, respectively (lines 22-24). During kernel execution, data migrations from Host to GPU1 occur when the kernel accesses *src* and *dst* buffers (line 3). Then, *scalarMultiplication* is launched on GPU0 with *g1* and *g0* as *src* and *dst* parameters (lines 27-29). Now, data migrates from GPU1 to GPU0 during kernel execution (line 3). Lastly, the results of the kernel executions are verified on Host (lines 32-38).

3.1.2 Host-Device Communication

Similar to peer-to-peer communication between GPUs, CUDA explicit data transfer primitives, Zero-copy Memory and Unified Memory operations exist for communication between host and device. The following cases describe each option in more detail.

Case 4: cudaMemcpy. Traditionally, after initializing data on the host, the data is transferred to the device using `cudaMemcpy` with the transfer kind of `cudaMemcpyH`

`ostToDevice`. Data is transferred from device to host using `cudaMemcpy` with the transfer kind of `cudaMemcpyDeviceToHost`. Using transfer kinds such as `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` mean that the direction of transfer is explicitly specified by the programmer. Or, with the support of UVA, these transfers can happen without explicitly describing the direction by calling `cudaMemcpy` with `cudaMemcpy`
`Default` as the transfer kind. CUDA will infer the direction of transfer by analyzing the pointer, similar to *Case 1.1*.

Case 5: Zero-Copy Memory. Zero-copy Memory paradigm uses mapped pinned memory that allows a GPU to directly access host memory over PCIe or NVLink interconnect. Page-locked mapped host memory is allocated using `cudaHostAlloc` or `cudaHostRegister` with `cudaHostRegisterMapped`. Next, the device pointer referring to the same memory is acquired with `cudaHostGetDevicePointer` and therefore no explicit data transfers are needed. During kernel execution, the device pointer directly accesses the mapped pinned host memory [NVIDIA, 2020a]. However, it must be noted that only unidirectional data transfers that are `read` and `write` operations by the device are possible in this case as the host is unable to perform `write` operations in Zero-copy Memory paradigm [Pearson et al., 2019].

Case 6: Unified Memory. As mentioned in *Case 3*, for Unified Memory, data is allocated via `cudaMallocManaged` which returns a pointer accessible from any processor. If a pointer is allocated on host using `cudaMallocManaged` and the device accesses it during kernel execution, then driver will migrate the page from host to device memory. This holds true for the host as well. In the event of a page fault, page migration occur.

3.2 Collective Communication

Collective communication refers to a data transfer among a group of GPUs. On CUDA devices, NVIDIA's Collective Communication Library (NCCL) can be used to perform multi-GPU collective communication. It has two versions, where the first

version only supports intra-node collective communication over interconnects such as PCIe or QPI. The second version is extended to perform inter-node collective communication as well and support modern interconnects and network cards such as NVLink, NVSwitch, InfiniBand and IP networks. NCCL library relies on GPU Direct for inter-GPU communication that includes GPU-to-GPU communication as well as GPU-to-NIC communication [NVIDIA, a]. It implements 5 collective operations including All-Gather, All-Reduce, Broadcast, Reduce and Reduce-Scatter as CUDA kernels. In order to use collective primitive on a group of GPUs, each GPU within the communication group also known as *communicator* is assigned a zero-based rank. All ranks belonging to the same communicator can send and receive data upon calling a collective primitive on that group. The functionality of each collective primitive is described below:

Broadcast

Using Broadcast collective primitive results in sending a copy of data from one rank referred as *root* rank to all other ranks in the communicator.

Reduce

During Reduce operation, only the specified *root* rank receives the reduced form of data across all ranks in the communicator. The reduction operations that can be performed on the input data include sum, product, min and max.

All-Reduce

Similar to Reduce collective primitive, All-Reduce also applies reduction operation on data across all ranks in the communicator. However, instead of sending it to only one rank, the reduced data is copied to the receiving buffers all ranks in the communicator.

Reduce-Scatter

In Reduce-Scatter operation, first the reduction operation from Reduce collective primitive is applied on the input data across all ranks. Then, according to the rank index, each rank receives an equal chunk of data.

All-Gather

All-Gather operation results in aggregation of chunks of data across all ranks in the communicator and is received by each rank. The order of aggregation is determined by the rank index.

NCCL collective primitives can use one of the following algorithms for receiving and sending data in NCCL library: Ring, Tree or Collnet. As of now for intra-node communication, NCCL collective primitives explained above follow Ring-based algorithm. It detects the topology of the system and forms a logical ring network of the devices that are part of the communicator. Instead of sending all data at once, NCCL divides data into chunks where each chunk is equal to total data size divided by the number of GPUs in the ring. Each device only sends data to the next clockwise device in the ring, and receives from the previous one. All collectives except Broadcast and Reduce have complete ring communication. In Broadcast, a copy of data is sent to all ranks in the communicator except the root rank, therefore in the ring algorithm, the last rank does not send data to the root rank. Similarly, for Reduce collective primitive, the root rank receives data of all other ranks and the performs reduction operation on it. Consequently, in ring-based implementation, all ranks except root rank sends data to the next rank in clockwise direction. NCCL Library currently implements Tree and Collnet algorithm for All-Reduce primitive for inter-node communication only[NVIDIA, b].

Listing 3 presents an example of All-Reduce collective primitive for single process with single thread managing multiple GPUs.

```
1 ncclComm_t comms[8];
2 int num_devices = 8;
3 int size = 32*1024*1024;
4 int devices[8] = { 0, 1, 2, 3, 4, 5, 6, 7};
```

```
ncclCommInitAll(comms, num_devices, devices);

float** send_buff = (float**)malloc(num_devices * sizeof(float*));
float** recv_buff = (float**)malloc(num_devices * sizeof(float*));
cudaStream_t* streams = (cudaStream_t*)malloc(sizeof(cudaStream_t)*
    num_devices);


for (int i = 0; i < num_devices; ++i) {
cudaSetDevice(i);
cudaMalloc(send_buff + i, size * sizeof(float));
cudaMalloc(recv_buff + i, size * sizeof(float));
cudaMemset(send_buff[i], 1, size * sizeof(float));
cudaMemset(recv_buff[i], 0, size * sizeof(float));
cudaStreamCreate(streams+i);
}

ncclGroupStart();
for (int i = 0; i < num_devices; ++i)
ncclAllReduce((const void*)send_buff[i], (void*)recv_buff[i], size,
              ncclFloat, ncclSum,
              comms[i], streams[i]);
ncclGroupEnd();

for (int i = 0; i < num_devices; ++i) {
cudaSetDevice(i);
cudaStreamSynchronize(streams[i]);
}

for(int i = 0; i < num_devices; ++i)
  ncclCommDestroy(comms[i]);

for (int i = 0; i < num_devices; ++i) {
cudaSetDevice(i);
cudaFree(send_buff[i]);
cudaFree(recv_buff[i]);
```

```
40 }
```

Listing 3.3: AllReduce in NCCL

In this example, we are using 8 GPUs with a single process and thread, therefore, 8 communicator objects are created (lines 1-5). For each device, send and receive buffers as well as CUDA streams are created (lines 7-19). Next each device calls an All-Reduce collective primitive with *sum* reduce operation inside `ncclGroupStart` and `ncclGroupEnd` to inform NCCL that a collective operation for multiple devices is being called in a single thread and that the first device does not have to wait for others to arrive (lines 21-26). Using `cudaStreamSynchronize`, all streams are synchronized for the NCCL collective operation complete (lines 28 - 31). At the end of the program. all communicator objects are destroyed and buffers are freed on devices (lines 33-40).

Chapter 4

DESIGN AND IMPLEMENTATION OF COMSCRIBE

This chapter discusses the design and implementation of ComScribe that monitors both intra-node point-to-point and collective GPU communication. It uses different methods to monitor each communication pattern. Therefore, we explain the design and workflow of ComScribe for point-to-point and collective communication under separate sections.

4.1 Intra-node Point-to-Point Communication

We have developed ComScribe on top of *nvprof* to identify intra-node point-to-point communication between multi-GPUs and CPU-GPU. *nvprof* is a light-weight command-line profiler available since CUDA 5.0 [NVIDIA, 2020b]. Although NVIDIA's latest profiling tool *Nsight System* is a system-wide performance analysis tool designed to visualize an application's behaviour [NVIDIA, 2020f], it does not provide memory-copy information for all types of communication in its machine-readable format. On the other hand, *nvprof*'s profiling output contains data transfer information for explicit CUDA data transfers as well as implicit transfers involving Unified Memory and Zero-Copy memory transfers. Though, *nvprof* profiles data transfers between CPU-GPU and GPU-GPU during the execution of an application, the data required for generating intra-node communication matrices i.e. total amount of data shared between each pair of devices in a node for each type of communication is not readily available, requiring extra effort by the programmer to extract such information. For example, for each kernel or memory copy, detailed information such as kernel parameters, shared memory usage and memory transfer throughput are recorded. As a result, it becomes difficult for the user to observe the total data movement and types of communication between each pair of GPUs

in a node. ComScribe represents the communication pattern in a compact though descriptive manner to the user, in the light of the data obtained and parsed from *nvprof*.

The *nvprof* profiler has 5 modes namely *Summary*, *GPU-Trace*, *API-Trace*, *Event/Metric Summary* and *Event/Metric Trace*. Among these only GPU-Trace and Event/Metric Trace modes are necessary for our tool. The GPU-Trace mode provides information of all activities taking place on a GPU such as kernel execution, memory copy and memory set and is used by ComScribe for generating communication matrices for explicit and Unified Memory data transfers. For Zero-copy Memory transfers, the amount of data transmitted and received through NVLink by a single GPU in a node is drawn from *nvprof*'s Metric Trace mode. Since *nvprof* operates in one mode at a time and the information for all types of communication is not available through a single mode, ComScribe runs the application twice with *nvprof* to collect necessary information.

ComScribe parses *nvprof*'s output for memory copies, accumulates the communication amount for each GPU-GPU and CPU-GPU pair, infers the types of communication and then generates intra-node communication matrices for an application using CUDA's explicit data transfer primitives or Unified Memory. ComScribe can generate these matrices for both the number of bytes transferred, and the number of data transfers. For Zero-copy Memory, *nvprof's* Metric Trace mode lacks information about the sending or receiving GPU, making it infeasible to generate communication matrix. Instead our tool constructs a bar chart that represents the data transmitted to and received from another GPU or CPU for write and read operations, respectively for each GPU.

4.1.1 Workflow

To generate communication matrices, our tool runs the application code with *nvprof*'s GPU-Trace and Metric Trace modes. The profiling output is parsed only for memory copies by our tool. The type of memory copy helps ComScribe identify the CUDA data transfer primitive. The workflow is shown in Figure 4.1 and explained below:

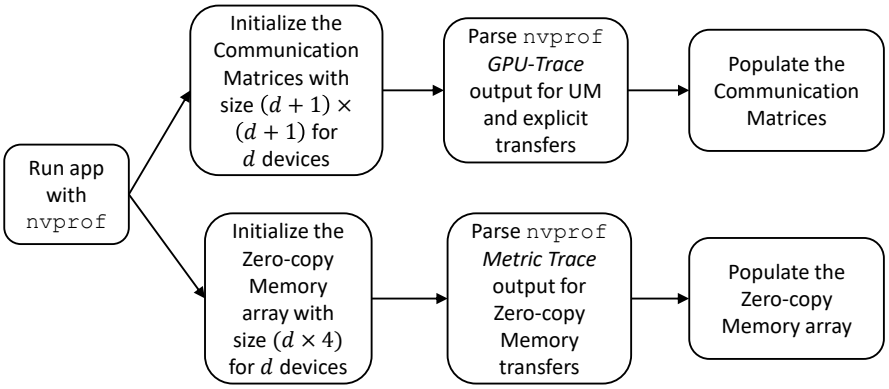


Figure 4.1: Workflow diagram of ComScribe for capturing Intra-node Point-to-Point Communication

1. **Running application with nvprof.** By running the application with *nvprof*'s GPU-Trace and Metric Trace modes, all activities taking place in the node such as kernel executions and memory copies are recorded in a chronological order.

2. **Initialization of communication matrices and Zero-copy Memory array.** Depending on the number of devices, d, available in the node, communication matrices represented as two-dimensional arrays of size $(d+1) \times (d+1)$ are initialized to zero where $+1$ accounts for CPU. Communication matrices are generated for both explicit data transfers and Unified Memory transfers. For Zero-Copy Memory transfers, a one-dimensional array of size $(d \times (4)$ is initialized to recorded data transferred and received through NVLink and system memory. A Zero-copy Memory read or write can be identified for a GPU's memory and host memory, so in total this gives 4 types of transfer for every device.

3. **Recording data transfers.** *nvprof's* GPU-Trace mode output is parsed by ComScribe, which reads each recorded event to detect if it is an event of data transfer between GPU-GPU or CPU-GPU. On each memory copy, the source device ID, destination device ID, size of data transferred is extracted. The size is converted into bytes, then by looking at the source and destina-

tion device IDs, the communication event is recorded to the communication matrix, where y-axis indicates receivers and x-axis indicates senders. In this matrix, we can also see the host as a sender or receiver, at the 0^{th} index with the letter H. Data movements within a GPU such as local memory copies are also recorded, which appear as non-zero entries in the diagonal of the communication matrix. Moreover, memory copies are characterized to identify the types of communication. For Zero-copy Memory transfers the Metric Trace mode output is parsed to record data movement for each GPU.

4. **Generating Results.** After the memory copy information are recorded in communication matrices, these matrices are plotted for each identified type of communication, in the case of explicit and Unified Memory transfers. Zero-copy Memory transfers are represented in the form of a bar chart.

4.2 Intra-node Collective Communication

ComScribe also supports monitoring collective communication by intercepting NCCL primitives at runtime using LD_PRELOAD as *nvprof* does not profile NCCL library. On the other hand, NVIDIA's *Nsight System* profiles NCCL's collective primitives but as CUDA kernels and records only the kernel parameters. Therefore, ComScribe comes with a shared library that binds collective primitive's symbols in NCCL with a new definition that includes calling the original NCCL function which is defined in NCCL's shared library followed by recording memory copy information between different pairs of GPUs. At runtime, the NCCL collective primitive calls are intercepted by the function with the same NCCL primitive name in the preloaded shared library of our tool and data transfers are recorded. This recorded information is used to generate communication matrices as well as a timeline of all the data transfers with the type of collective communication in chronological order.

4.2.1 Workflow

Figure 4.2 shows the workflow of our tool for profiling intra-node collective communication. The workflow is explained in detail below:

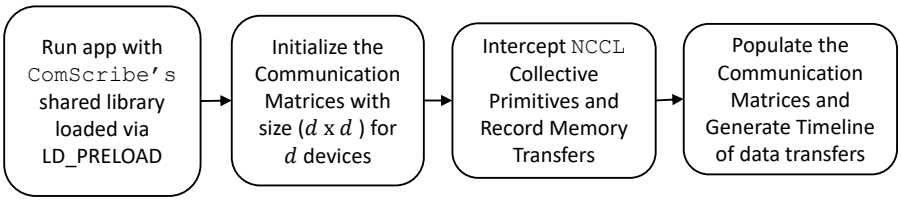


Figure 4.2: Workflow diagram of ComScribe for capturing Intra-node Collective Communication.

1. **Running application with NCCL's modified shared library.** ComScribe preloads its shared library at runtime using LD_PRELOAD and then launches the application.

2. **Initialization of communication matrices.** Similar to intra-node point-to-point communication, communication matrices are initialized to zero. However, the size of the matrices is $d \times d$ where d is the number of devices in the node because collective communication only involves GPU-GPU communication.

3. **Intercepting NCCL primitive and recording data transfers.** During application execution, when any NCCL collective primitive (All-Gather, All-Reduce, Reduce-Scatter, Reduce and Broadcast) is called for a group of GPUs (communicator), ComScribe intercepts this primitive by calling its respective new definition from the preloaded shared library. Upon successful completion of the original collective primitive, data transfer information is recorded that includes source device ID, destination device ID and the number of bytes transferred between different pair of GPUs in the communicator.

4. **Generating Results.** Memory transfers for each collective primitive that is

intercepted during execution are recorded into their respective communication matrices and plotted. Also, a timeline of all the data transfers is generated by combining the memory copy information of each collective primitive into chronological order.

Although ComScribe uses different methods to monitor point-to-point and collective communications in a node, their results are merged to present communication matrices for all data transfers that result from both communication patterns.

Chapter 5

ENHANCEMENTS WITH ADVANCED FEATURES

This chapter discusses some additional features of ComScribe that are in progress to increase it's performance, usability, and productivity.

5.1 Leveraging NVIDIA Nsight Systems

For monitoring intra-node point-to-point communication, ComScribe is currently built on top of NVIDIA's old profiling tool *nvprof*. However, NVIDIA's latest profiling tool is *Nsight Systems*, which is a system-wide performance analysis tool designed to visualize an application's behaviour. *nvprof* is soon going to be deprecated and it is better to move ComScribe's support from *nvprof* to *Nsight Systems* [Wilper, 2019]. However, *Nsight Systems* provides only partial information. It does not provide memory-copy information for all types of communication in its machine-readable format. In *Nsight Systems* latest version (2020.5), CUDA memory-copy information including source device ID, destination device ID, and the number of bytes transferred were added only for explicit data transfers. For implicit data transfers, partial information i.e only source device ID and the number of bytes are recorded for peer-to-peer Unified Memory transfers and Zero-copy memory transfers are not profiled. Since, *nvprof's* GPU-Trace mode provides information for both explicit and Unified Memory data transfers, it is not feasible to move ComScribe to *Nsight Systems* for explicit transfers only and use *nvprof's* GPU-Trace for Unified Memory memory transfers and *nvprof's* Metric Trace for Zero-copy Memory transfers. When *Nsight Systems* will have complete information for implicit data transfers, we will make the necessary transition.

5.2 Increasing Productivity

To increase the productivity of the programmer, each data transfer recorded by ComScribe can be linked back to the source code via stack backtracking. Knowing the function name and line number in the source code at which memory copies were invoked can help programmers locate which parts of the code cause large bulk of data movements. For this purpose, *libunwind* [Mosberger, 2011], a platform-independent unwind API, can be used to determine call stack for functions. Explicit data transfers are initiated by `cudaMemcpy` and `cudaMemcpyPeer` functions in CUDA and can be intercepted with LD_PRELOAD to use libunwind which generates backtrace of caller functions with PC addresses only. Therefore, additional work is required to convert these addresses to function symbol names by using `addr2line` function from Linux toolbox. Also, line numbers cannot be extracted as addresses are relative to main program at runtime and not the shared library. Moreover, *Nsight System*'s GUI gives the call trace of all the functions that led to triggering explicit data transfers. However, this information is missing in its machine-readable output format. On the other hand, HPCToolkit [Adhianto et al., 2010] can give the function name and source code line where CUDA explicit data transfers are initiated. Using LD_PRELOAD option in HPCToolkit with function wrappers for `cudaMemcpy` and `cudaMemcpyPeer`, the stack trace can be retrieved.

5.3 Supporting Inter-node GPU-GPU Communication

ComScribe currently supports only intra-node communication and it will prove beneficial to extend its support to monitor scale-out applications that use multi-node multi-GPU communication. GPUDirect-RDMA enables inter-node GPU-GPU communication by allowing third-party devices such as InfiniBand's Host Channel Adapter (HCA) to directly access GPU memory over PCIe bus [Li et al., 2020]. Two nodes are connected to each other's HCA via InfiniBand interconnect [Potluri et al., 2013]. The challenge is to profile GPU-GPU communication across nodes such that each data transfer recorded includes source node ID, destination node ID, source device ID, destination device ID and the number of bytes transferred. Successfully

monitoring the aforementioned information enables ComScribe to generate communication matrices for inter-node point-to-point and collective communications.

Chapter 6

EVALUATION

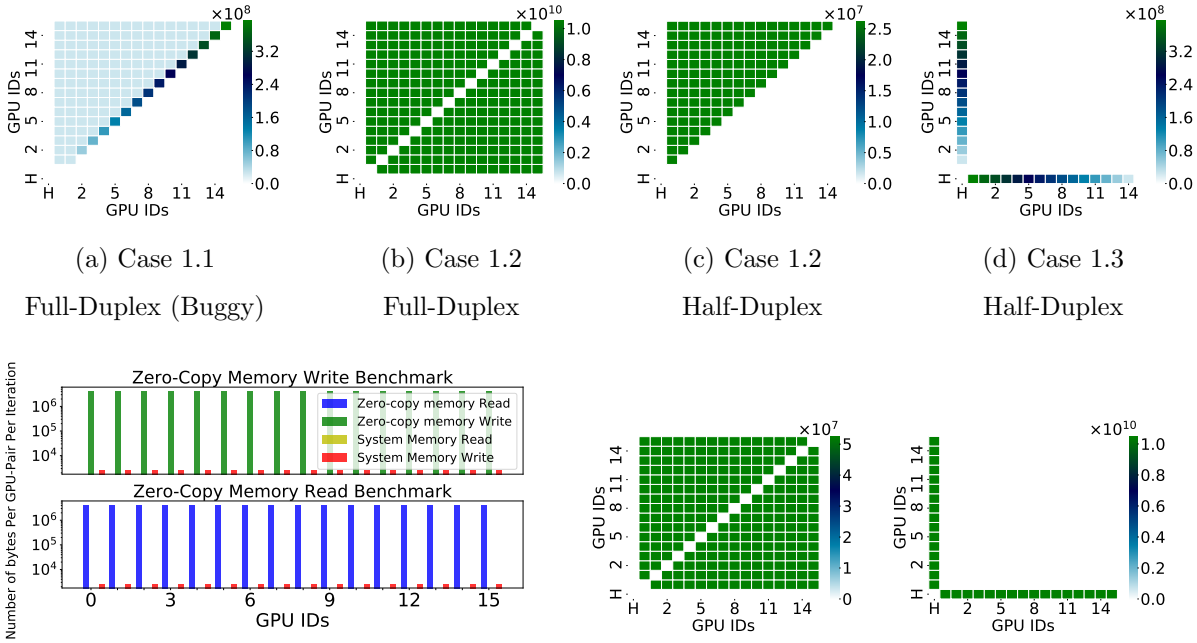
In this chapter, we present the evaluation results of ComScribe on various benchmarks and real-world applications for both intra-node communication patterns. The evaluation is conducted on a DGX-2 system consisting of 16 NVIDIA Tesla V100 GPUs, allowing simultaneous communication between 8 GPU pairs at 300 GBps through 12 integrated NVSwitches [NVIDIA, 2018a]. We use CUDA v10.0.130.

6.1 Intra-node Point-to-Point Communication

This section evaluates ComScribe for intra-node point-to-point communication on selected micro-benchmarks from the MGBench and Comm|Scope [Ben-Nuun, 2017, Pearson et al., 2019] benchmark suites. We also present communication matrices for macro-benchmarks including NVIDIA's Ising-GPU and Multi-GPU Jacobi Solver, MGBench's Game of Life (GOL) and study two deep learning applications: E3D-LSTM and Transformer [NVIDIA, 2019b, NVIDIA, 2018b, Ben-Nuun, 2017, Wang et al., 2019, Vaswani et al., 2017a] as use-cases.

6.1.1 Micro-benchmarks from Comm|Scope and MGBench

To validate our tool, we have selected 8 micro-benchmarks with known communication patterns, 6 from Comm|Scope and 2 from MGBench based on their communication types using explicit data transfers with peer access enabled and disabled, as well as implicit data transfers using Unified Memory and Zero-copy Memory operations. We have observed that the communication matrices for number of bytes transferred and number of transfers are very similar in these benchmarks, therefore we only include the former in this section. Each Comm|Scope micro-benchmark is



(a) Case 1.1 Full-Duplex (Buggy)

(b) Case 1.2 Full-Duplex

(c) Case 1.2 Half-Duplex

(d) Case 1.3 Half-Duplex

(e) Case 2 and Case 5 Half-Duplex

(f) Case 3 Full-Duplex

(g) Case 4 Half-Duplex

Figure 6.1: ComScribe results on several micro-benchmarks from Comm|Scope (Figures 6.1a, 6.1c, 6.1d, 6.1e and 6.1f) and MGBench (Figures 6.1b and 6.1g) benchmark suites.

run with 100 iterations and transfers 256KB data. Whereas, MGBench by default runs with 100MB for memory transfers and for a total of 100 iterations.

Figures 6.1a-6.1d show the communication matrices generated by ComScribe for the micro-benchmarks that use peer-to-peer explicit transfers. Benchmarks in Figures 6.1a and 6.1b use `cudaMemcpyAsync` and `cudaMemcpyPeerAsync` data transfer primitives, respectively for peer-to-peer full-duplex communication with peer access enabled (*Case 1.1 and 1.2*). However, the pattern in the communication matrix in Figure 6.1a is not as expected– should have been similar to the one in Figure 6.1b, which shows the data movement as a result of bidirectional transfers between peer GPUs. We observe data transfers in one direction between GPUs along with data movement within the GPU, which translates to local memory copies. With the help of ComScribe, we were able to spot this incorrect communication pattern and identify a communication bug in Comm|Scope's ***`cudaMemcpyAsync-duplex`*** micro-

benchmark on GitHub[1]. After investigating the benchmark code, we realized that the source and destination pointers of the second copy are unintentionally pointing to the same device, resulting in data movement within the device.

Figures 6.1c and 6.1d present results for peer-to-peer half-duplex micro-benchmarks where unidirectional communication between each pair of GPUs is expected. However, the key difference between two is that the benchmark in Figure 6.1c uses `cudaMemcpyPeerAsync` with peer access (*Case 1.2*) while the benchmark in Figure 6.1d uses `cudaMemcpyPeerAsync` without peer access (*Case 1.3*). Figure 6.1c and 6.1d demonstrate how peer access being enabled or disabled affects the communication pattern. In Figure 6.1c, the transfers are made directly to the peer GPU, whereas, in Figure 6.1d, the data is implicitly transferred through host memory as peer access is disabled.

Figures 6.1e and 6.1f show results for implicit transfers that occur through Zero-copy Memory and Unified Memory operations (*Case 2, 3 and 5*). Zero-copy Memory bar-chart generated by ComScribe in Figure 6.1e shows the number of bytes received and transmitted by a single GPU in Comm|Scope Zero-copy Memory GPU-GPU read and write half-duplex micro-benchmarks, respectively, where a small amount of system memory write is detected as well (*Case 5*). The benchmark in Figure 6.1f uses Unified Memory for peer-to-peer full-duplex communication. We can observe communication between all pairs of GPUs in the node as a result of bidirectional Unified Memory data transfers. However, the total number of bytes transferred among each pair of GPUs is twice than expected due to prefetching i.e. after GPU0 transfers data to GPU1, in the next iteration the data is prefetched from GPU1 to GPU0 before it is sent to GPU2.

Figure 6.1g illustrates the communication type of *Case 4* with the MGBench's Scatter-Gather micro-benchmark that uses `cudaMemcpyAsync` with transfer kind `cudaMemcpyHostToDevice` to scatter data from host to all GPUs and then gathers data back from GPUs to host using `cudaMemcpyAsync` with transfer kind `cudaMemcpyDeviceToHost`.

[1] https://github.com/c3sr/comm_scope/blob/master/src/cudaMemcpyAsync-duplex/gpu_gpu_peer.cpp

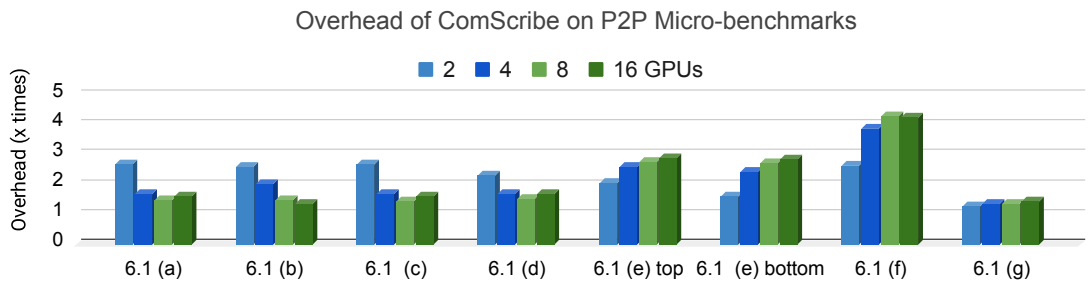


Figure 6.2: ComScribe's overhead (mostly stemming from *nvprof*) for Comm|Scope and MGBench micro-benchmarks on 2, 4, 8, and 16 GPUs.

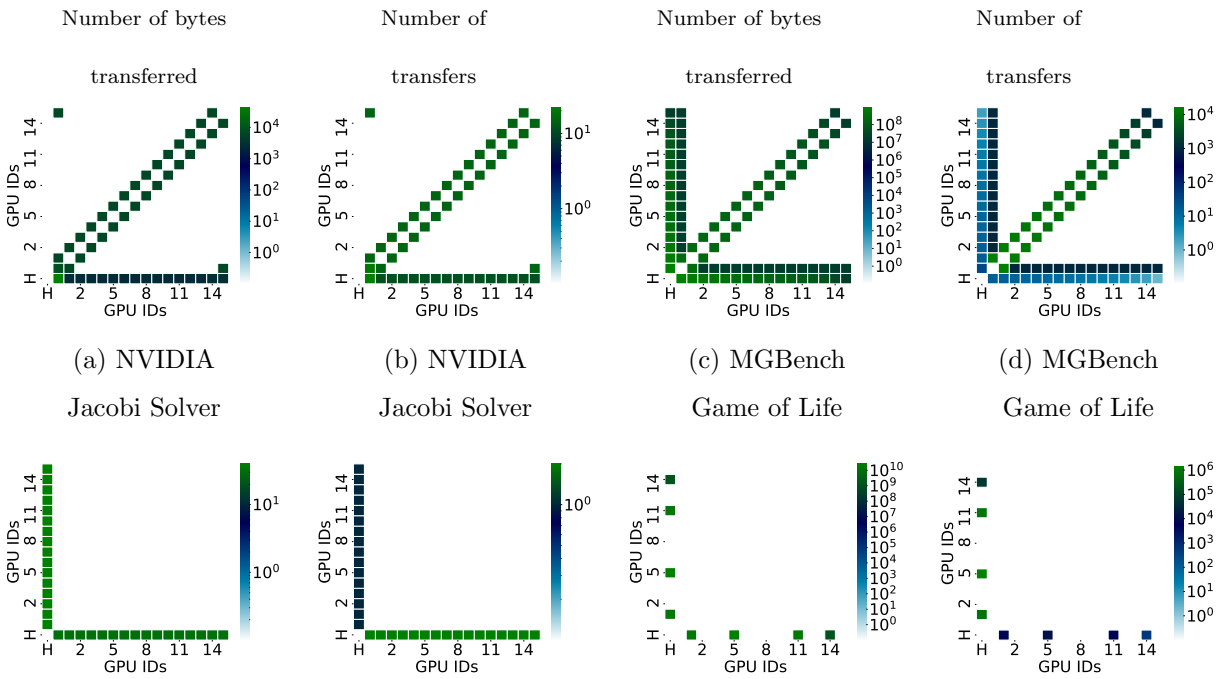
6.1.2 Overhead

Figure 6.2 presents the overhead of running ComScribe with the micro-benchmarks for monitoring intra-node point-to-point communication only. *nvprof* is the main contributor to the overhead of ComScribe as post-processing of *nvprof* profiling outputs is negligible. On average, an overhead of 2.17x is observed, due to the fact that ComScribe runs the application with *nvprof* twice, once with GPU-Trace mode for explicit and Unified Memory transfers, and then once more with Metric Trace mode for Zero-copy Memory transfers.

6.1.3 Macro-Benchmarks from NVIDIA and MGBench

We present insightful communication matrices for three macro-scale applications using 16 GPUs. These applications are NVIDIA's Multi-GPU Jacobi Solver, NVIDIA's Monte Carlo Simulation 2D Ising-GPU, and MGBench's Game of Life [NVIDIA, 2018b, NVIDIA, 2019b, Ben-Nuun, 2017]. Unlike micro-benchmarks, here we include both matrices: number of bytes transferred and number of transfers. We use log scale instead of linear scale in the figures to make the small transfers visually noticeable.

Figures 6.3a and 6.3b show a communication pattern that is prevalent among stencil applications such as Jacobi solver: a GPU shares a "halo region" with it's nearest neighbors and it communicates with them in every iteration to exchange



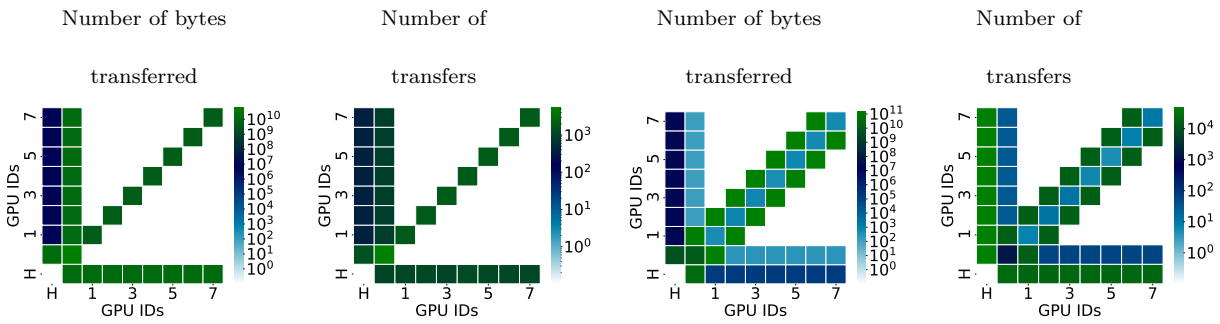
(a) NVIDIA Jacobi Solver (b) NVIDIA Jacobi Solver (c) MGBench Game of Life (d) MGBench Game of Life

(e) Monte-Carlo 2D-Ising Explicit Transfers (f) Monte-Carlo 2D-Ising Explicit Transfers (g) Monte-Carlo 2D-Ising Unified Memory (h) Monte-Carlo 2D-Ising Unified Memory

Figure 6.3: ComScribe results on macro-benchmarks from NVIDIA (Figures 6.3a, 6.3b, 6.3e, 6.3f, 6.3g and 6.3h) and MGBench (Figures 6.3c and 6.3d).

and update the halo cells. This access pattern can be thought of as a "window" [Ben-Nun et al., 2015] and shows itself as non-zero entries on the diagonal in the communication matrix. Note that this application uses circular boundary condition where every cell including the ones at the boundary has a neighbor in all directions, which presents itself as communication between GPU0 and GPU15.

Game of Life, shown in Figures 6.3c and 6.3d, is also a type of stencil application. It's domain is distributed among GPUs, and the state of the domain in the next iteration is computed by every GPU using the current state. Similarly, there is a halo region that needs to be updated every iteration, and the nearest-neighbor communication reflects itself as non-zero entries in the diagonal of the communication matrix. Unlike Jacobi, this application does not employ a circular boundary condition. Note that the communication values are not equal among the GPUs because this benchmark runs the application in increments, starting from 1 GPU, increasing



(a) Eidetic 3D LSTM (b) Eidetic 3D LSTM (c) Transformer (d) Transformer

Figure 6.4: ComScribe results on two Deep Neural Network models implemented in TensorFlow framework using data parallelism across 8 GPUs

one at a time until the desired amount of GPUs, 16 in our case, is reached. As a result, GPU0 is used in all 16 runs, GPU1 is used in 15 runs, and so on. We can observe this at the diagonals of both figures and from the L shape pattern in Figure 6.3c in which the number of bytes transferred and number of transfers decrease as as the GPU IDs increase.

Figures 6.3e-6.3h shows the results of GPU-accelerated Monte Carlo simulations of 2D Ising Model. The algorithm uses a scatter-gather pattern where independent simulations are run on each GPU and then they are gathered at the host for averaging. Scatter-gather is distinguishable as it forms an L shape pattern on the communication matrix, where the host sends data to all GPUs, and receives data from all. We observe that this application uses both explicit transfers and Unified Memory at the same time, which is identified by ComScribe. By looking at the explicit transfer's communication matrices, we can immediately see the intended communication pattern, which is also complied by the Unified Memory transfers. Moreover, this application (Figures 6.3g and 6.3h) serves as an example for the type of communication presented as *Case 6*.

6.1.4 Use-Cases: Deep Neural Network (DNN) Models

This section demonstrates the communication matrices for two DNN models. GPUs are commonly used by the deep learning frameworks to accelerate both the training

and inference. Due to its simplicity and desired weak scaling, the dominant distributed DNN training strategy is to use data parallelism, where input is split among multiple GPUs each of which holds a replica of the model [Valiant, 1990, Shazeer et al., 2018]. In stochastic gradient descent (SGD) [Amari, 1993], where the training is done in minibatches, the weights of the DNN need to be updated. In the update phase, the results of the replicas have to be averaged to obtain the gradient of the whole minibatch [Ben-Nun and Hoefler, 2019]. The averaged version then needs to be accessed by all the replicas to update their parameters. The pervasiveness of data parallelism has resulted in having multiple underlying implementations (communication patterns) for the gradient exchange. Each of these pattern's performance, and hence the training performance, vary depending on the underlying network topology. By visualizing the communication, our tool helps the user to understand the underlying communication pattern and analyze its effect on performance.

Figures 6.4 shows communication matrices generated by ComScribe for two well-known deep learning models implemented in the TensorFlow framework [Abadi et al., 2016] using data parallelism to execute on 8 GPUs. The first model is Eidetic 3D LSTM (E3D-LSTM), a spatio-temporal predictive learning model for video prediction [Wang et al., 2018]. The second is Transformer [Vaswani et al., 2017b], a widely used model that has considerably influenced the design of SoTA Transformer based models in the NLP domain. From the communication matrices of these two models, two variations of communication used for implementing data parallelism can be observed. In the E3D-LSTM case, the gradients are sent from GPUs to host, they are averaged, then sent from host to GPU0. After this, GPU0 scatters the gradients to all other GPUs. For Transformer, the gradients are exchanged using the nearest neighbor communication, as a result the largest amount of communication occurs between neighboring pairs of GPUs. While in E3D-LSTM, the second largest communication is self communication happening due to the huge memory allocated temporarily for 3D-convolutions. In Transformer, it occurs between GPU0 and host for the purpose of training checkpoint, where a copy of the model state and its parameters are sent to the host. The remaining communication takes place between the host and devices to distribute the training samples in the batch across the GPUs.

Note that the models we used in our demonstration are large models having O(1 billion) parameters that is why the last type of communication is relatively small. However, it might have higher relative importance for smaller models.

6.2 Intra-node Collective Communication

This section evaluates ComScribe for intra-node collective communication on NVIDIA's NCCL micro-benchmarks [NVIDIA, 2020d].

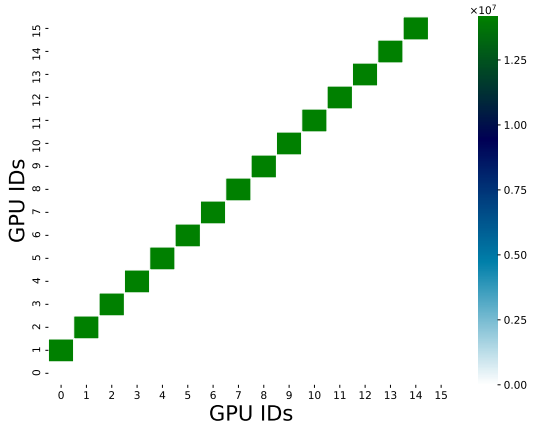
6.2.1 Micro-benchmarks from NCCL

We verify NCCL's collective ring-based protocol using micro-benchmarks from NVIDIA. Each collective primitive has its own micro-benchmark which is run with ComScribe for 25 iterations and transfers 256KB data with GPU0 set as root rank. Similar to Section 6.1.1, communication matrices for number of bytes transferred are only included in this section.

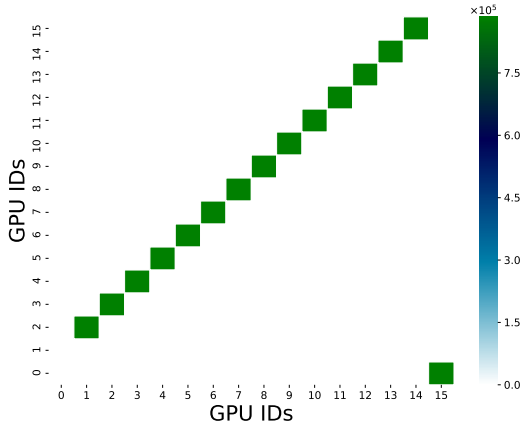
Figure 6.5 presents the communication matrices generated by ComScribe for NCCL collective primitives micro-benchmarks. The ring-based communication can be verified using these matrices. The diagonal of each figure shows one GPU sending data to next GPU in the ring. As explained in Section 3.2, all NCCL primitives implement a complete ring-based protocol except Broadcast and Reduce functions. Therefore, in Figures 6.5a and 6.5b incomplete ring-based communication can be observed. For Broadcast collective primitive, the last rank in the ring, GPU15, does not send data to the root rank, GPU0. Likewise, when Reduce primitive is called by GPU0, all GPUs send their data to next GPU except GPU0. Hence, no communication is recorded between GPU0 and GPU1.

6.2.2 Overhead

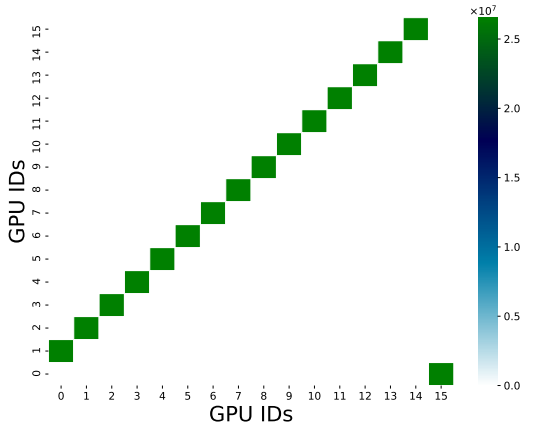
Figure 6.6 shows the overhead of running ComScribe with NCCL collective primitives micro-benchmarks. Intercepting NCCL collective primitives incurs an overhead of 0.40x on average. Also, we observe that with increasing number of GPUs, the ComScribe overhead for monitoring collective communication decreases.



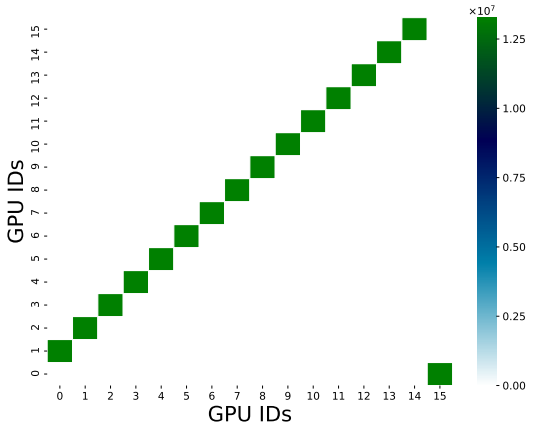
(a) Broadcast



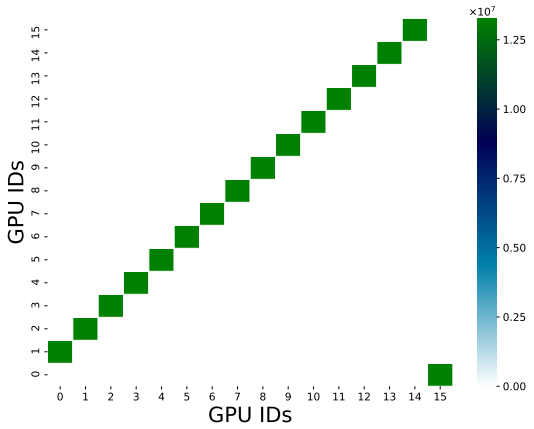
(b) Reduce



(c) All-Reduce



(d) Reduce-Scatter



(e) All-Gather

Figure 6.5: ComScribe results on micro-benchmarks from NVIDIA's NCCL tests.

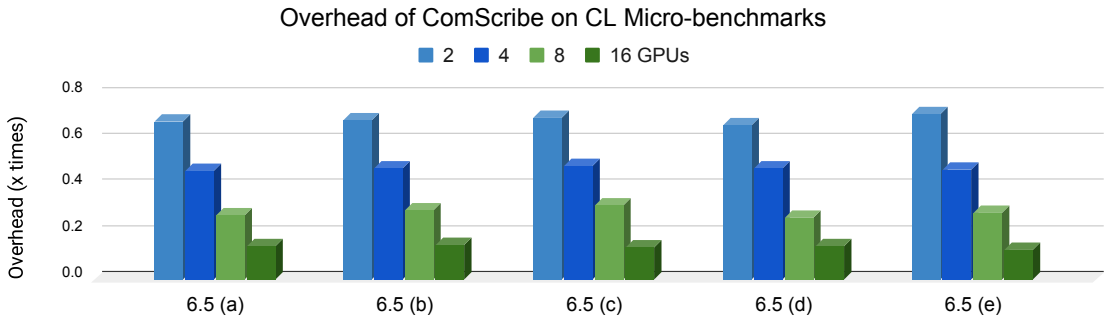


Figure 6.6: ComScribe's overhead for NCCL micro-benchmarks on 2, 4, 8, and 16 GPUs.

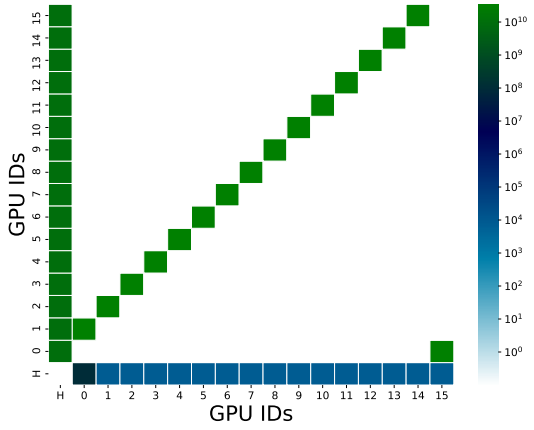
6.2.3 Use-Case: Deep Neural Network (DNN) Model

As discussed in Section 4, ComScribe can be used to monitor an application for both point-to-point and collective communication. In this section we present results of our tool on a DNN model (ResNet-18) training on ImageNet dataset [Pytorch, 2020] that uses NCCL library to carry out collective communication among GPUs.

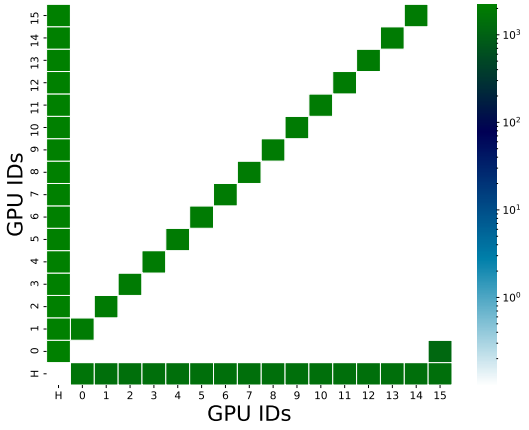
DNNs are widely used to solve the image classification problem as they can learn different features from sample images during training and later help classify the unseen images for object detection, semantic or instance segmentation. The training strategy of DNNs is explained in Section 6.1.4 including gradient exchange among GPUs to update parameters and shows how visualizing underlying communication pattern is important.

Figure 6.7 shows the communication matrices as well as the collective GPU communication timeline generated by ComScribe for a deep residual learning model, Residual neural network (ResNet), implemented in the PyTorch framework [Paszke et al., 2019] using multi-process distributed data parallelism performed on 16 GPUs. From the results, we can observe both point-to-point and collective communication captured by our tool. Host-Device communication takes place to distribute training data in batches among multiple GPUs in the node. Figure 6.7c shows the number of bytes transferred between a pair of GPUs as a result of the collective primitive. To understand the order of operations, we also present a rather zoomed in version of the

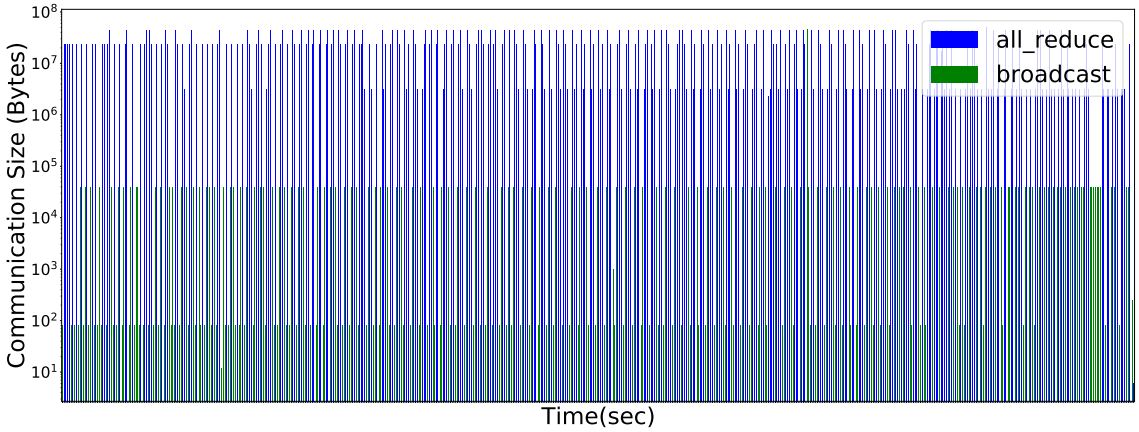
timeline in Figure 6.7d showing first 100 instances of the timeline. It can be observed that All-Reduce and Broadcast collective primitives are being used. All-Reduce is used to compute gradient summation across all the GPUs but it is not necessary to synchronize the gradients after every iteration. PyTorch uses gradient bucketing to gather gradients from multiple iterations and then call All-Reduce operations to skip synchronization after every backward pass [Li et al., 2020]. Therefore, we can observe that there is no definitive pattern in All-Reduce usage. Also, the Broadcast primitive is used to share model states such as *BatchNorm* in every iteration before forward pass [PyTorch, 2019].



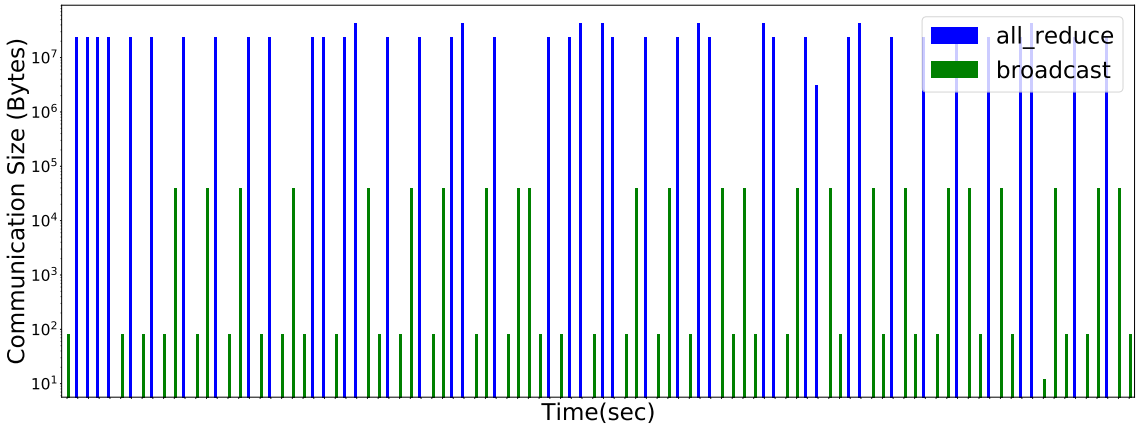
(a) Number of bytes transferred



(b) Number of transfers



(c) Complete timeline of collective communication



(d) First 100 instances in timeline of collective communication

Figure 6.7: ComScribe results on ResNet-18 DNN model implemented in PyTorch Framework using multi-process distributed data parallel training across 16 GPUs.

Chapter 7

RELATED WORK

EZTrace is a generic trace generation framework for programs that are written using OpenMP, MPI, PThreads and CUDA [Trahay et al., 2011]. It generates trace for such programs in two steps: first, it collects the necessary information by intercepting function calls and recording events during program execution using the FxT library [Danjean et al., 2005], then it performs a post-mortem analysis on the recorded events. The total message size exchanged and number of messages for peer-to-peer and collective communications can be generated for MPI with the help of EZTrace, but for CUDA this feature is limited. For CUDA applications, a log of recorded events is generated that includes the total message size exchanged and number of messages for CPU-GPU communication that uses `cudaMalloc-cudaMemcpy` pair only. In comparison, our tool detects a much diverse set of communication types and generates communication matrices for GPU-GPU communication including all CUDA explicit data transfer primitives, Zero-copy Memory and Unified Memory operations as well as collective primitives.

Comm|Scope [Pearson et al., 2019] is a set of micro-benchmarks to measure latency and bandwidth for CUDA data transfer primitives with different data placement scenarios. This includes NUMA-aware point-to-point CPU-GPU and GPU-GPU explicit data transfer primitives in CUDA as well as Zero-copy Memory and Unified Memory operations. Although our work and Comm|Scope are parallel in motivation to categorize types of communication, Comm|Scope does not focus on identifying and quantifying GPU communications. NUMA-aware communication affects the performance, however it does not change the amount of data shared among GPUs, thus it is not included in our work. Moreover, we consider additional CUDA data transfer primitives like `cudaMemcpyPeer` and transfer kind `cudaMemcpyDefault`.

Li et. al [Li et al., 2018, Li et al., 2020] developed Tartan, a multi-GPU benchmark suite consisting of micro-benchmarks for characterizing GPU interconnects and 14 application benchmarks for evaluating how multi-GPU execution performance is affected by different GPU interconnects. Tartan includes micro-benchmarks to evaluate interconnects including PCIe, NVLink 1.0, NVLink 2.0, NV-SLI, NVSwitch and Infiniband systems with GPUDirect RDMA for point-to-point and collective intra-node and inter-node GPU-GPU communication. It measures communication efficiency, bandwidth and latency of GPU-GPU explicit memory copies and the NVIDIA Collective Communications Library (NCCL) on the interconnects. However, Tartan is not a tool but rather a benchmark suite, whereas, we offer a tool that captures the total amount of data transferred for intra-node peer-to-peer and collective communication.

A number of tools that generate communication patterns for multi-core applications exist in the literature. ComDetective [Sasongko et al., 2019] is a sampling based tool that uses Performance Monitoring Units (PMUs) and debug registers to detect inter-thread data transfers and generates communication matrices for multi-threaded applications for shared memory systems. Likewise, Azimi et al. [Azimi et al., 2009] and Tam et al. [Tam et al., 2007] leverage kernel support to access PMUs and the kernel generates the communication pattern for the applications. Barrow-Williams et al. [Barrow-Williams et al., 2009] and Cruz et al. [da Cruz et al., 2011] employ simulator-based approach to collect memory access traces for generating communication patterns. Lastly, Numalize [Diener et al., 2015, Diener et al., 2016] uses binary instrumentation to intercept memory accesses and captures communication between threads accessing the same address in memory. However, none of these tools can be used to identify multi-GPU communication.

Chapter 8

CONCLUSION

Communication is a critical programming component and performance contributor in multi-GPU systems. In this thesis, we present ComScribe tool that identifies, quantifies and generates communication matrices for point-to-point and collective GPU-GPU communication in a node. For single-node multi-GPU systems, we categorize various data transfer options offered by CUDA API to the programmer, into explicit data transfer primitives and implicit transfers using Unified Memory and Zero-copy Memory operations for Peer-to-Peer and Host-Device communications. To monitor intra-node point-to-point communication, ComScribe is built on top of NVIDIA's profiling tool *nvprof*. After application execution, our tool parses nvprof's output for memory copies, accumulates the communication amount for each GPU-GPU and CPU-GPU pair and infers the types of communication for an application using CUDA's data transfer primitives. For intra-node collective GPU communication, ComScribe intercepts NCCL's collective communication primitives at application runtime and records memory transfers among GPUs. The recorded information for both communication patterns is represented in communication matrices for both the number of bytes transferred and the number of transfers. ComScribe's workflow is presented, followed by it's evaluation on several micro- and macro-benchmarks as well as three deep learning applications. Communication matrices generated by our tool can be used by programmers to differentiate types of communication, study the communication patterns and detect communication bugs in a multi-GPU application.

BIBLIOGRAPHY

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA. USENIX Association.

[Adhianto et al., 2010] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701.

[Amari, 1993] Amari, S.-i. (1993). Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196.

[Azimi et al., 2009] Azimi, R., Tam, D. K., Soares, L., and Stumm, M. (2009). Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review*, 43(2):56–65.

[Barrow-Williams et al., 2009] Barrow-Williams, N., Fensch, C., and Moore, S. (2009). A communication characterisation of splash-2 and parsec. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 86–97. IEEE.

[Ben-Nun and Hoefler, 2019] Ben-Nun, T. and Hoefler, T. (2019). Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43.

[Ben-Nun et al., 2015] Ben-Nun, T., Levy, E., Barak, A., and Rubin, E. (2015). Memory access patterns: The missing piece of the multi-GPU puzzle. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 15-20-November-2015.

[Ben-Nuun, 2017] Ben-Nuun, T. (2017). Mgbench: Multi-gpu computing benchmark suite (cuda). `https://github.com/tbennun/mgbench`. (Accessed on 07/29/2020).

[Buono et al., 2017] Buono, D., Artico, F., Checconi, F., Choi, J. W., Que, X., and Schneidenbach, L. (2017). Data analytics with NVLink: An SpMV case study. *ACM International Conference on Computing Frontiers 2017, CF 2017*, pages 89–96.

[da Cruz et al., 2011] da Cruz, E. H. M., Alves, M. A. Z., Carissimi, A., Navaux, P. O. A., Ribeiro, C. P., and Méhaut, J.-F. (2011). Using memory access traces to map threads and data on hierarchical multi-core platforms. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 551–558. IEEE.

[Danjean et al., 2005] Danjean, V., Namyst, R., and Wacrenier, P.-A. (2005). An efficient multi-level trace toolkit for multi-threaded applications. In Cunha, J. C. and Medeiros, P. D., editors, *Euro-Par 2005 Parallel Processing*, pages 166–175, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Diener et al., 2016] Diener, M., Cruz, E. H., Alves, M. A., and Navaux, P. O. (2016). Communication in shared memory: Concepts, definitions, and efficient detection. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 151–158. IEEE.

[Diener et al., 2015] Diener, M., Cruz, E. H., Pilla, L. L., Dupros, F., and Navaux, P. O. (2015). Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation*, 88:18–36.

[Foley and Danskin, 2017] Foley, D. and Danskin, J. (2017). Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17.

[Harris, 2017] Harris, M. (2017). Unified memory for cuda beginners — nvidia developer blog. `https://devblogs.nvidia.com/unified-memory-cuda-beginners/`.

[Jeaugey, 2017] Jeaugey, S. (2017). Nccl 2.0. *GTC*.

[Li et al., 2020] Li, A., Song, S. L., Chen, J., Li, J., Liu, X., Tallent, N. R., and Barker, K. J. (2020). Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110.

[Li et al., 2018] Li, A., Song, S. L., Chen, J., Liu, X., Tallent, N., and Barker, K. (2018). Tartan: Evaluating modern gpu interconnect via a multi-gpu benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–202.

[Li et al., 2020] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. (2020). Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*.

[Micikevicius, 2012] Micikevicius, P. (2012). Multi-gpu programming. `https://on-demand.gputechconf.com/gtc/2012/presentations/S0515-GTC2012-Multi-GPU-Programming.pdf`.

[Mosberger, 2011] Mosberger, D. (2011). The libunwind project. `https://www.nongnu.org/libunwind/`. (Accessed on 01/10/2021).

[NVIDIA, a] NVIDIA. Nvidia collective communication library (nccl) documentation — nccl 2.8.3 documentation. `https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html`. (Accessed on 12/29/2020).

[NVIDIA, b] NVIDIA. Nvidia/nccl: Optimized primitives for collective multi-gpu communication. `https://github.com/NVIDIA/nccl`. (Accessed on 12/29/2020).

[NVIDIA, 2012] NVIDIA (2012). Nvidia gpudirect technology. `http://developer.download.nvidia.com/dev\\zone/devcenter/cuda/docs/GPUDirect\_Technology\_Overview.pdf`.

[NVIDIA, 2017] NVIDIA (2017). Nvidia dgx-1 with tesla v100 system architecture white paper. `https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf`. (Accessed on 07/28/2020).

[NVIDIA, 2018a] NVIDIA (2018a). Dgx-2 : Ai servers for solving complex ai challenges — nvidia. `https://www.nvidia.com/en-us/data-center/dgx-2/`. (Accessed on 07/28/2020).

[NVIDIA, 2018b] NVIDIA (2018b). Multi-gpu-programming-models: Examples demonstrating available options to program multiple gpus in a single node or a cluster. `https://github.com/NVIDIA/multi-gpu-programming-models`. (Accessed on 07/29/2020).

[NVIDIA, 2019a] NVIDIA (2019a). Cuda runtime api. `https://docs.nvidia.com/cuda/pdf/CUDA\_Runtime\_\\API.pdf`. (Accessed on 07/29/2020).

[NVIDIA, 2019b] NVIDIA (2019b). Ising-gpu: Gpu-accelerated monte carlo simulations of 2d ising model. `https://github.com/NVIDIA/ising-gpu`. (Accessed on 07/29/2020).

[NVIDIA, 2020a] NVIDIA (2020a). Best practices guide :: Cuda toolkit documentation. `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html\#zero-copy`. (Accessed on 05/12/2020).

[NVIDIA, 2020b] NVIDIA (2020b). Cuda profiler user's guide. `https://docs.nvidia.com/cuda/pdf/CUDA\_Pr\\ofiler\_Users\_Guide.pdf`. (Accessed on 07/28/2020).

[NVIDIA, 2020c] NVIDIA (2020c). Dgx a100 : Universal system for ai infrastructure — nvidia. `https://www.nvidia.com/en-us/data-center/dgx-a100/`. (Accessed on 07/28/2020).

[NVIDIA, 2020d] NVIDIA (2020d). Nccl tests. `https://github.com/NVIDIA/nccl-tests`. (Accessed on 12/31/2020).

[NVIDIA, 2020e] NVIDIA (2020e). Nvidia collective communications library (nccl) — nvidia developer. `https://developer.nvidia.com/nccl`. (Accessed on 12/29/2020).

[NVIDIA, 2020f] NVIDIA (2020f). Nvidia nsight systems documentation. `https://docs.nvidia.com/nsight-systems/index.html`. (Accessed on 07/28/2020).

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library.

[Pearson et al., 2019] Pearson, C., Dakkak, A., Hashash, S., Li, C., Chung, I.-H., Xiong, J., and Hwu, W.-M. (2019). Evaluating characteristics of cuda communication primitives on high-bandwidth interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 209–218, New York, NY, USA. Association for Computing Machinery.

[Potluri et al., 2013] Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D., and Panda, D. (2013). Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus. *2013 42nd International Conference on Parallel Processing*, pages 80–89.

[Pulo, 2009] Pulo, K. (2009). Fun with ld_preload. In *linux. conf. au*, volume 153.

[PyTorch, 2019] PyTorch (2019). Distributed data parallel — pytorch 1.7.0 documentation. `https://pytorch.org/docs/stable/notes/ddp.html`. (Accessed on 01/11/2021).

[Pytorch, 2020] Pytorch (2020). Imagenet training in pytorch. `https://github.com/pytorch/examples/tree/master/imagenet`. (Accessed on 12/31/2020).

[Sasongko et al., 2019] Sasongko, M. A., Chabbi, M., Akhtar, P., and Unat, D. (2019). Comdetective: a lightweight communication detection tool for threads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–21.

[Shazeer et al., 2018] Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. (2018). Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423.

[Sourouri et al., 2014] Sourouri, M., Gillberg, T., Baden, S. B., and Cai, X. (2014). Effective multi-GPU communication using multiple CUDA streams and threads. *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, 2015-April(1):981–986.

[Tam et al., 2007] Tam, D., Azimi, R., and Stumm, M. (2007). Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *ACM SIGOPS Operating Systems Review*, 41(3):47–58.

[Trahay et al., 2011] Trahay, F., Rue, F., Faverge, M., Ishikawa, Y., Namyst, R., and Dongarra, J. (2011). Eztrace: a generic framework for performance analysis. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 618–619. IEEE.

[Valiant, 1990] Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

[Vaswani et al., 2017a] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017a). Attention is all you need. *CoRR*, abs/1706.03762.

[Vaswani et al., 2017b] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017b). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

[Wang et al., 2018] Wang, Y., Jiang, L., Yang, M.-H., Li, L.-J., Long, M., and Fei-Fei, L. (2018). Eidetic 3d lstm: A model for video prediction and beyond. In *International Conference on Learning Representations*.

[Wang et al., 2019] Wang, Y., Jiang, L., Yang, M.-H., Li, L.-J., Long, M., and Fei-Fei, L. (2019). Eidetic 3d LSTM: A model for video prediction and beyond. In *International Conference on Learning Representations*.

[Wilper, 2019] Wilper, H. (2019). Migrating to nvidia nsight tools from nvvp and nvprof — nvidia developer blog. `https://developer.nvidia.com/blog/migrating-nvidia-nsight-tools-nvvp-nvprof/`. (Accessed on 01/03/2021).

[Xu, 2018] Xu, A. (2018). Nccl based multi-gpu training_v3_alfred xu (leixu). `https://on-demand.gputechconf.com/gtc-cn/2018/pdf/CH8209.pdf`.