ParaGraph: An application-simulator interface and toolkit for hardware-software co-design

Mikhail Isaev*
michael.v.isaev@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Nic McDonald
nimcdonald@nvidia.com
NVIDIA
Salt Lake City, Utah, USA

Jeffrey Young
jyoung9@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Richard Vuduc
richie@cc.gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

ParaGraph is an open-source toolkit for use in co-designing hardware and software for supercomputer-scale systems. It bridges an infrastructure gap between an application target and existing high-fidelity computer-network simulators. The first component of ParaGraph is a high-level graph representation of a parallel program, which a) faithfully represents parallelism and communication, b) can be extracted automatically from a compiler, and c) is "tuned" for use with network simulators. The second is a runtime that can emulate the representation's dynamic execution for a simulator. User-extensible mechanisms are available for modeling on-node performance and transforming high-level communication into operations that backend simulators understand. Case studies include deep learning workloads that are extracted automatically from programs written in JAX and TensorFlow and interfaced with several event-driven network simulators. These studies show how system designers can use ParaGraph to build flexible end-to-end software-hardware co-design workflows to tweak communication libraries, find future hardware bottlenecks, and validate simulations with traces.

ACM Reference Format:
Mikhail Isaev, Nic McDonald, Jeffrey Young, and Richard Vuduc. 2022. ParaGraph: An application-simulator interface and toolkit for hardware-software co-design. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3545008.3545069

1 INTRODUCTION

We consider the co-design process for future high-performance multi-node cluster and supercomputer systems. In that setting, designers build workflows that necessarily employ simulators of the interconnection network [11]. We perceive a technical gap in this process, namely, the lack of a flexible and efficient means of representing application workloads and composing them with simulation tools. The challenge is how to model the high-level behaviors of an application while still making the model easy to integrate with network simulators.

By way of explanation, consider a typical simulator of a high-performance interconnection network [8, 19, 32, 43]. It is a discrete-event simulator with a limited interface for specifying workloads: a workload "enters" the simulation as a sequence of timestamped network message injection or extraction events at certain *endpoints* (or *terminals*), typically corresponding to a network interface controller (NIC). Between such events, whatever happens on the compute node is abstracted away by delays between timestamps.

To conform to this interface, there are three basic strategies: analytical application models, synthetic traffic generators, and network or communication traces. Analytical models are fast to evaluate but challenging to design for anything other than small kernels. Generators can represent traffic patterns but ignore all other aspects of a computation that might significantly affect those patterns. (This limitation typically holds, we would argue, even for motifs [43], which may be viewed as complex traffic generators.) And traces can represent communication in great detail but can also be extremely large, "overfit" to existing systems on which they were captured, and typically lack any node-level detail. These limitations restrict the hypothetical designs or scenarios one might otherwise explore.

This situation is problematic for co-design because applications embody many design choices, which current simple simulator interfaces betray (Section 2). Consider two examples. First, communication patterns may be much more complex than what simple sequences of send and receive events convey. An application might invoke collective communication operations, like all-to-all or all-reduce operations, through Message Passing Interface (MPI) middleware [53]. Such collectives have many possible implementations, with choices made according to the network's characteristics and the application's characteristics, including message frequency and sizes. For example, an optimal implementation of a collective communication operation on a fat tree topology might differ from one on a torus. Second, node design and network design interact, fundamentally. For instance, one result for multidimensional distributed fast Fourier transforms (FFTs) says that one can tradeoff the volume of intranode (memory hierarchy) communication with network communication [56]. This scenario reflects an algorithmic design choice as well as the importance of the node itself: one might want

*Portions of this work were performed during internships at Google and NVIDIA.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08.
https://doi.org/10.1145/3545008.3545069

to adjust the parameters of the node that affect its intrinsic flop-to-byte ratio against the parameters of the network that affect its latency and bandwidth. These examples suggest that the structure of applications is rich and fluid, yet in practice, the conventional ways of representing workloads—via analytical models, generators, or trace-driven methods—have intrinsic limitations that will frustrate attempts to incorporate such structure.

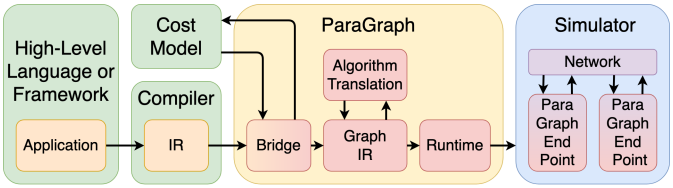


Figure 1: ParaGraph organization, including bridge with high level compiler, instruction translation, and interfacing with simulators through end-point model running ParaGraph runtime.

ParaGraph is our attempt to improve how applications are modeled and integrated into the simulation workflow for multi-node systems co-design (Section 3). It sits between a "frontend" application and a "backend" simulator, as illustrated in Fig. 1. ParaGraph defines a native graph-based intermediate representation (IR) of the application and a runtime that can interpret this IR for the hardware simulator. ParaGraph's IR comes from a direct translation of the compiler's IR of the application, and the runtime is easy for the simulator to use. Thus, ParaGraph provides a natural bridge between the application and the simulator. Importantly, ParaGraph is *not* itself a simulator; rather, it is a framework for modeling distributed applications for use with simulators. One may view it as a productivity tool for people who design large-scale systems.

The ParaGraph IR statically expresses the structure of high-level control and data dependences in the program (Section 3.1). It explicitly represents parallelism and communication, including collective operations, and is single-program multiple-data (SPMD) in style. It optionally permits inclusion of profile information. ParaGraph's IR derives from Google's High-Level Operations (HLO) IR, and it is straightforward to convert programs compatible with the Accelerated Linear Algebra (XLA) domain-specific compiler into ParaGraph's IR [15]. Today, such programs include large-scale distributed deep learning computations written in TensorFlow or PyTorch [1, 39], which is one target of our work; there is also an emerging generation of scientific computing programs being written to use JAX, a NumPy-compatible drop-in for graphics co-processor (GPU) and Tensor Processing Unit (TPU) systems [4, 16, 24, 46]. ParaGraph's scope of application targets is the class of SPMD applications represented as graphs, e.g., XLA HLO or CUDA graphs.

But where HLO and the like are traditional compiler IRs designed to assist in program analysis, optimization, and code generation for a backend processor [7, 15, 28, 44], ParaGraph's IR complements compiler IRs: it is simplified and designed for interfacing with network simulators. For example, ParaGraph provides extensible mechanisms for lowering its IR to substitute different collective communication algorithms and can also store profile data or generate delay information needed by the backend simulator (Section 3.2). It does so using existing compiler infrastructure to perform IR-to-IR transformations, not source-to-source translation. It expects applications that are already represented as computational graphs. This aspect of ParaGraph confers additional benefits: it requires no code alteration, no code annotation, and no domain expertise, unlike other methods. A designer runs the application, stores its graph, and then is ready to model its execution.

To use this IR to emulate a workload, a simulator invokes ParaGraph's runtime through a thin application programmer interface (API) (Section 3.3). The design of this API is informed directly by the needs of simulators, so interfacing with it needs only a modicum of code. ParaGraph's runtime interprets the IR and translates its operations into events for the simulator as needed.

Using ParaGraph, one can flexibly build any number of modeling workflows, a claim which we demonstrate through a series of case studies (Section 4). These include an exploration of hardware and software configurations to optimize the performance of all-reduce; TPU trace matching on MLPerf training benchmarks; and running a deep learning application on a vendor-specific simulator, utilizing multiple frontend frameworks and backend simulators across tasks. The case studies consider the system-level co-design tasks where GPUs and accelerators like TPUs are modeled, whereas prior work has tended to focus on combinations of CPU + MPI, single node co-design, or software optimizations for existing hardware. We also discuss validation of ParaGraph against TPU traces (Section 4.2). As far as we know, no comparable infrastructure to bridge applications and network simulation in support of such multi-node co-design tasks exists today.

2 RELATED WORK

Many methods exist to model hardware for future large-scale systems. They differ in modeling fidelity, flexibility, scalability, and ease of use. The main categories of methods are analytical performance modeling, modeling with cycle-accurate or cycle-approximate simulators, and hardware prototyping. We summarize these techniques below except for hardware prototyping, as our focus is on analytical and software-based simulation techniques.

Analytical models such as LogP predict performance given the amount of communication, available injection bandwidth, and latency [9]. These models assume idealized network behavior and typically yield asymptotically accurate scaling predictions. Roofline models are another category that can include network communication [5, 58]. These models are the easiest to use but provide limited fidelity. They only consider injection and ejection link utilization and cannot model in-network congestion, arbitration, or adaptive routing. ParaGraph uses rooflines to estimate node performance (computation instructions and memory access time) and interfaces with a simulator for modeling of dynamic network behavior. Future work will explore the use of event-driven models for other system components, like the memory system, to provide higher fidelity on-node performance predictions.

Network simulators are the most popular tools for large-scale system research, as they model networks with high fidelity, which is the distinguishing feature of multi-node systems. However, it is typical for network simulators to oversimplify application and node models. Simulators for HPC networks, like booksim [19] and SuperSim [32], rely on simplistic synthetic traffic generators. Many simulators, e.g., the CODES simulator [8] with the TraceR trace reader [18], provide trace-driven application modeling by replaying

dependency-oriented MPI trace logs. Simulators for internet and data center networks, like NS [35] and OMNeT++ [54], implement communication protocols, e.g., TCP, IP, parts of system-level software implemented in the Linux stack. However, they also rely on artificial traffic generators to produce simulation inputs, and there is no direct modeling support for the structured (e.g., collective) communication typical in HPC and deep learning (DL). ParaGraph does not compete with these simulators directly. Instead, it is designed to provide application models and communication protocols to such simulators.

The Structural Simulation Toolkit (SST) permits simple artificial traffic patterns, but also allows MPI and SHMEM application models called motifs [43]. Motifs are skeletal versions of the program with fixed compute times and predefined communication patterns, typically hand-crafted with the help of an application domain expert. ParaGraph can describe application models using its IR, similar to how motifs work. However, it also allows automatic workload extraction if used with existing compiler infrastructure. SST also provides source-to-source skeletonization via the LLVM compiler infrastructure [57], but this feature requires domain experts to annotate the code with pragmas manually. ParaGraph builds on XLA HLO [15], a higher-level compiler infrastructure, as described in Section 3. ParaGraph overcomes several limitations of SST. It supports multiple frontends and backends, e.g., for TPUs, GPUs on the backend, non-MPI and non-SHMEM applications, DL applications on the frontend. It provides *automatic* workload extraction via IR-to-IR transformation, with *no annotation* and *no domain expertise* to alternate the code. This approach also provides better correlation between real applications execution and their model as exactly the same code is executed in both hardware and simulator.

ASTRA-SIM facilitates HW/SW co-design by providing application models and describing system-level software such as communication libraries [42]. However, it focuses on DL application models exclusively and implements them using parameterized traffic generators that are similar to SST motifs. By contrast, ParaGraph can extract DL application models directly from high-level frameworks like TensorFlow and JAX [1, 4], which allows ParaGraph to use new models and support more types of applications.

Detailed models, such as those described with register transfer language (RTL) or cycle-accurate models, provide the best application and node fidelity. Many of them, such as the gem5 full-system simulator, can execute full programs [3], albeit with a large slowdown and a focus on modeling a single node. Gem5 uses Garnet [2], the network simulator designed to model Networks-on-Chip (NoC) with limited scalability. This fact and the focus on fine-grained CPU instructions make gem5 hard to use in large-scale system modeling. There are several fast FPGA-driven simulators available for system modeling tools. QFlex [38] supports full-system modeling and multi-node systems but focuses on a smaller scale and needs QEMU integration. DIABLO [52] provides an FPGA-based evaluation methodology for warehouse-scale computing (WSC), allowing the modeling of systems of size $O(1{,}000)$ to $O(10{,}000)$ nodes, but it is also tied to a CPU-based simulator that only implements the SPARC v8 architecture. ParaGraph's representation works with coarser-grained HW-agnostic instructions, such as a "matrix multiply instruction." That approach allows ParaGraph to reduce modeling time significantly and provides flexibility to model systems built with GPUs and other accelerators.

There is existing work to couple compiler-derived IRs, like LLVM, with cycle-accurate simulators [27, 29, 30, 50]. These focus primarily on single-accelerator and networks-on-chip design, as opposed to multi-node system-level networks as with ParaGraph. The distinct problem ParaGraph addresses is the development of an IR that is higher-level than a compiler IR and being easy to integrate with simulators.

Being adapted from HLO's IR, the ParaGraph IR shares features with other parallel IRs, like PDG [13], SPIRE [21], LLVM PIR [22], HPVM [25], Intel's ISPC [40], and HSAIL [45]. However, most of these focus on code generation for SIMD/SIMT units and address SPMD hardware that utilizes a shared address space. SPIRE models explicit `send` and `recv` instructions but lacks collectives like `AllReduce` and `AllToAll` as first-class entities. Several DL-focused IRs, like TVM, Glow, and Halid, focus on a single-processor performance and lack native support for collectives [7, 41, 44]. ProGraML focuses on optimizing single-processor code using machine learning algorithms [10]. Finally, XLA HLO and Multi-Level Intermediate Representation (MLIR) are similar to ParaGraph in their support of communication instructions but lack point-to-point communications [15, 28]. Compared to all of these efforts, ParaGraph is distinct in targeting network simulators for distributed systems, which permits aggressive simplification of the program representation. For instance, ParaGraph needs no support for optimizing transformations and does not need to maintain a fully executable program representation.

3 PARAGRAPH ARCHITECTURE

The high-level workflow of ParaGraph that we sketched in Fig. 1 is further expanded and refined in Fig. 2. Its three key components are the ParaGraph IR, its translator, and a runtime, summarized next and then elaborated upon in Sections 3.1 to 3.3.

To start, an application code (Fig. 2a) is compiled using existing tools to produce a standard IR. For example, the XLA compiler might produce the HLO IR shown in Fig. 2b. *(The code fragments of Fig. 2 have been simplified for ease of presentation.)* This IR is a parallel, near-source level, graphical representation, and is designed primarily for program analysis and optimization. Importantly, HLO's IR expresses a distributed-memory parallel computation in a SPMD style; regard the IR as representing the program process-code that is replicated and run on every node of a distributed-parallel system. In this example, all data arrays refer to local data, e.g., the single-precision 64×1024 array named `%input.1` is local to each process. (HLO has additional data attributes, not shown, for describing distributed arrays.) Operations are expressed at a high level, e.g., `%FeedForward.4` is the local result a call to `dot`, which multiplies two local matrices (here, 32-bit floating-point operands of size 64×1024 and 1024×10). Communication is explicit. Here, the call to `all-reduce` performs a collective matrix-all-reduce among three processors. (In HLO, the number of processors is instantiated in the IR; one sees this fact in this example as the "replica group" with three ranks, 0, 1, and 2.) Lastly, the IR expresses control or data dependences among the operations, e.g., `dot` produces `%FeedForward.4`, which a subsequent `add` consumes.

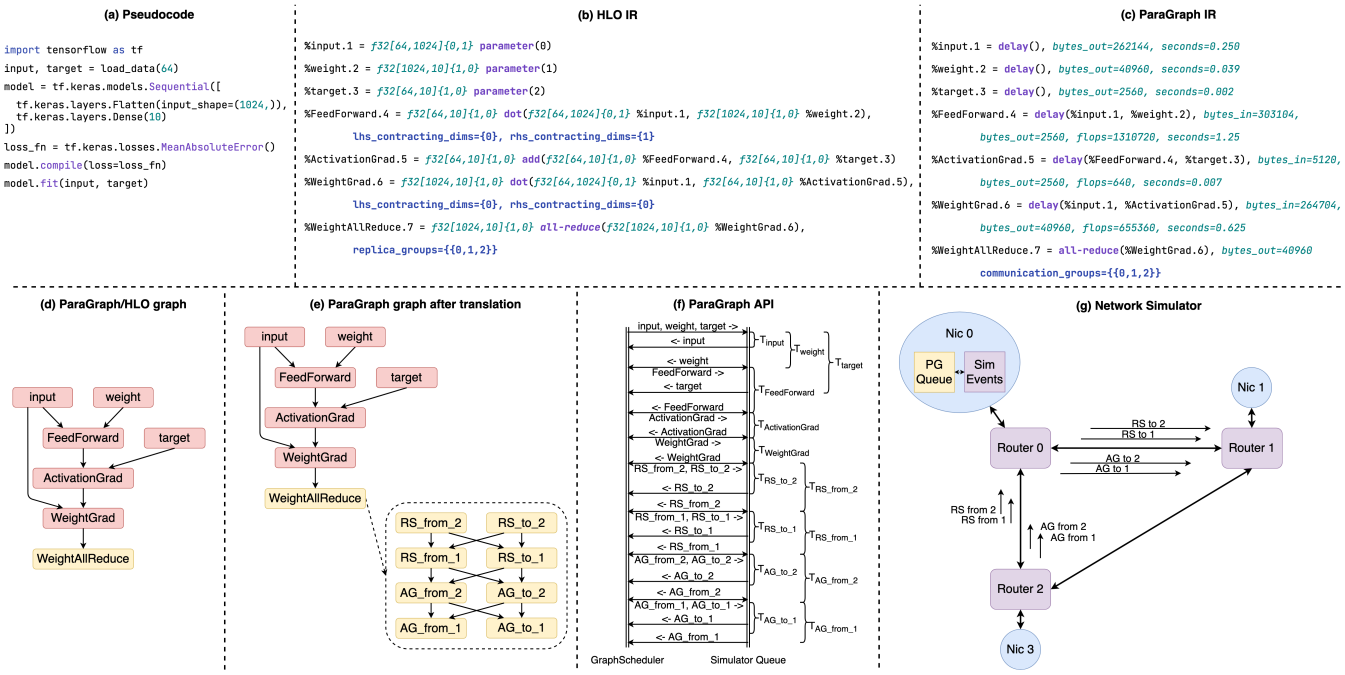


Figure 2: The ParaGraph workflow. (a) Application (pseudocode); (b) compiler IR; (c) ParaGraph IR, derived from the compiler IR; (d) a visual graph representation of the dependences encoded by an IR; (e) an example of substitution and lowering of an all-reduce primitive to a concrete implementation (RS stands for reduce-scatter, AG stands for all-gather); (f) schematic of ParaGraph and simulator interaction, via ParaGraph's scheduler while interpreting the graph (left) and initiation of simulation events (right); (g) illustration of a simulator executing. The code fragments have been simplified for ease of presentation.

The compiler's IR must then be converted to ParaGraph's IR (PGIR), Fig. 2c (Section 3.1). PGIR preserves the high-level structure of operations and dependences and adopts the same SPMD structure. However, the ParaGraph IR is simpler. It removes many attributes related to code optimization and execution and adds just a few according to what information the simulation will need. For example, the simulator might only need to know that a given operation induces a delay of a certain length between network-message events, which is represented in PGIR as `delay` "operations." The close correspondence between the HLO IR and PGIR makes it straightforward to translate the former to the latter.[1]

Both the compiler IR and PGIR are graphical, that is, the textual representations encode a computation graph, and hierarchical to support nested structures. It is arguably easier to see and understand these visually, as illustrated in Fig. 2d. Vertices are operations and directed edges indicate that a sink has a control or data dependence on the source. Following HLO's terminology, we refer to vertices in either the HLO or ParaGraph IRs as "instructions," although this should generally be understood to mean macro-operations like matrix multiply or all-reduce rather than processor-level instructions. We use the visual IR forms in the rest of the paper but emphasize that they have the more precise forms.

ParaGraph's *translator* can further rewrite the PGIR, as seen in Fig. 2e (Section 3.2). It does so to prepare for interfacing with a simulator backend. Here, an all-reduce collective has been lowered into a specific implementation. This translation occurs either because the co-design task includes modeling of a communication library and analyzing alternative implementations for collective operations or because the backend simulator does not directly support them (thereby necessitating a lower-level implementation). Also observe from this example that PGIR is hierarchical: in this case, the lowered form is nested within the collective's vertex. The depth of such nesting is arbitrary. Finally, the substitution templates are composable and the translator is extensible, so users can add rewrite rules or create entirely new rewrite libraries for other primitives and operations beyond collective communication. The composability is supported by using the hierarchical structure of PGIR to mimic object or function composition that appears in programming languages.

The last two panels, Fig. 2f and Fig. 2g, illustrate ParaGraph's *runtime* and its interaction with the backend simulator (Section 3.3). The runtime defines a thin API that a simulator can use. The simulator calls the API as part of event-processing at the simulated network endpoints. Meanwhile, the ParaGraph runtime interprets the IR to emulate the program's execution and responds to the simulator with new events. Thus, ParaGraph maintains its own internal scheduling queue associated with interpreting and emulating the IR (left of Fig. 2f) and interacts with the simulator via its event queue (right of Fig. 2f). The complexity of integration is low, per Section 3.3.

A critical design feature of ParaGraph is that its workflow leverages existing infrastructure, namely, the application and compiler on the frontend and a network simulator on the backend. They remain decoupled, with ParaGraph creating a bridge between them. This architecture provides flexibility for implementing different modeling and simulation workflows (see Section 4).

[1] At the time we began work on ParaGraph, the MLIR was still in its infancy [28]. However, we believe it would also be possible to convert MLIR to PGIR or to implement PGIR within MLIR.

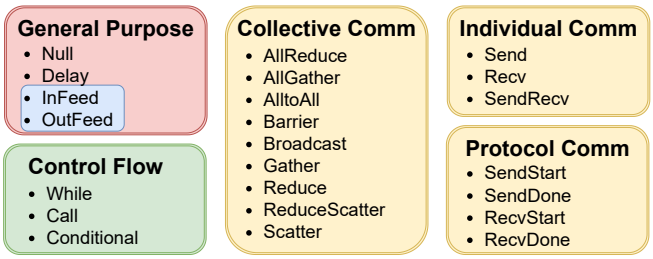


Figure 3: ParaGraph instruction classes (IR vertex types).

3.1 Graph representation

According to the taxonomy of Stanier and Watson [51], PGIR is a graphical hierarchical IR with macro-operations as the graph nodes and hybrid (control and data flow) dependencies as the edges. It is based on the HLO IR, which is used by XLA, the domain-specific compiler for linear algebra programs [15]. However, PGIR is defined by its purpose of *connecting the actual compiler IR with simulators, which cannot execute code,* and hence differs from other compiler IRs. Program operations or instructions in PGIR need only be mapped to events that network simulators can understand, such as time delays or communication over the network. Statements are not executed to produce any results, but modeled to predict execution time. As such, PGIR streamlines the compiler's IR, discarding notions of registers, memory, or ability to optimize or generate executable code. Nevertheless, by maintaining a close correspondence between PGIR and the input compiler's IR, ParaGraph reduces the gap between the application's code and what is interpreted and emulated during simulation (see Section 3.3). For example, ParaGraph can accept optimized compiler IR as input to increase the modeling accuracy.

HLO's hierarchical SPMD form targets a broad class of programs that may be expressed as computational dataflow graphs. Such programs include XLA's primary target, TensorFlow-based deep learning programs, as well as emerging support for PyTorch [33]. Beyond deep learning, XLA also supports general-purpose frameworks and languages, e.g., JAX [4] and Julia [14], as well as a growing number of scientific computing applications (Section 1). We estimate that the gap between PGIR and MLIR is small due to MLIR's similarity to HLO [28] and interfacing with it, as well as Glow or LLVM, would be opportunities for future work [27, 44].

Recall that vertices in PGIR represent macro-operations or "instructions" (following HLO IR's terminology). There are five broad categories of instructions, as summarized in Fig. 3. *General purpose* instructions represent anything that is not communication or control flow. These become time delays or service instructions (`Null`). *Control* instructions represent simple control flow constructs, such as conditionals, loops, and function calls. *Individual communication* instructions represent point-to-point communication between processors; *Collective communication* instructions represent collective communication primitives within a group of processors; *Protocol-level communication* instructions support different messaging protocols used in low-level software or hardware.

PGIR instructions may carry additional attributes or metadata related to performance and networking. Performance modeling attributes include information needed to synthesize delays. For instance, they can be the number of floating-point operations (flops) and bytes in or out of the vertex if a particular analysis wishes to use a roofline model to model compute-node performance [58]. Network attributes include communication groups, tags, and IDs for matching send and receive operations. Attributes are extensible if, say, a given analysis needs more complex performance models or a particular network simulator needs other information.

PGIR can be used to represent the program's computational structure hierarchically. That is, some instruction vertices may include pointers to subgraphs of instructions. The main purpose is to support graph translation (rewriting and lowering) as described in Section 3.2, performing composability in a way similar to object or function composition in programming languages. However, it also helps to support other nested structures, such as control flow instructions. Like pure functional languages, subroutines have a single entry point and a single return point, with no wormholes between subroutines. However, subroutines can execute concurrently if they belong to different instructions with no interdependencies.

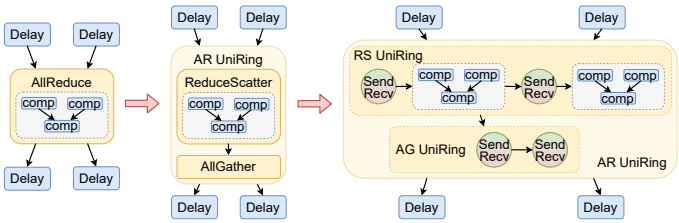


Figure 4: Phases of all-reduce translation for unidirectional ring (UniRing). New instructions are nested inside the translated ones expanding the hierarchy of the graph.

3.2 ParaGraph algorithm translation

ParaGraph provides an instruction translation mechanism, or *translator*. The main purpose is to allow substitution and lowering of instructions or subgraphs into other subgraphs. For example, the application may invoke `AllReduce` while the simulator only understands `Send` and `Recv` operations. Or, as part of a co-design workflow, a user may wish to test different implementations of `AllReduce`. Importantly, substitutions are composable. For instance, an all-gather operation can be implemented as a gather followed by a broadcast, where there are different implementations of gather and broadcast [6]. A user of ParaGraph can extend the library of available substitutions.

As a concrete example of this process, consider the `AllReduce` on a unidirectional ring shown in Fig. 4 for a network simulator that only supports send and receive operations. Initially, the PGIR might contain only the vertex corresponding to the `AllReduce` instruction. The translation library might contain a possible substitution consisting of `ReduceScatter` (RS) followed by `AllGather` (AG). On a ring of P processors, these would correspond to $P-1$ neighbor exchanges for both RS and AG implementations. Dependence edges present in the substitution ensure that neighbor exchange performs sequentially and that RS precedes AG. One of our case studies (Section 4.2) exploits this hierarchical compositionality of substitutions to implement both bidirectional ring and optimized 2-D torus algorithms for `AllReduce` [26]. This example is just one of many possible opportunities afforded by the ParaGraph infrastructure. Other system libraries, such as math libraries for FFT or matrix multiplication, could be implemented and modeled similarly [49].

3.3 ParaGraph runtime

The ParaGraph runtime is its component that interfaces with the network simulator. The runtime reads and interprets the PGIR. It maintains, for each logical process of the application, a queue of *ready instructions*, i.e., operations ready for execution. The interface to this queue of ready instructions is the thin API shown in listing 1. The API can be used to select the next ready instruction, query its logical cost (simulated execution time), and "execute" it. The runtime checks for newly satisfied dependencies and enqueues any new ready instructions.

The simulator is commonly implemented as a discrete event simulation (DES) engine. System level network simulators model both network devices and the endpoints generating and consuming network traffic. ParaGraph interfaces with the endpoint model of the simulator. ParaGraph associates application processes with endpoints modeled by the simulator. The endpoint can use the ParaGraph API to execute the IR, by fetching ready instructions and taking an appropriate action for each one. For instance, the simulator might update the simulation clock to reflect the delay of a computational instruction or schedule a communication event for communication instructions.

The case studies in Section 4 include two examples of integrating ParaGraph with real simulators, namely, the open-source simulator SuperSim [32] and a proprietary NVIDIA simulator, N10, that uses bksim [23]. These integrations only took about 500-1,000 lines of code. (We leave integration with other popular simulators, e.g., SST [43] and NS-3 [35], to future work.)

```
GraphScheduler* Create(Graph* graph);
void Initialize(double current_time);
std::vector<Instruction*> GetReadyInstructions();
InstructionStarted(Instruction* instruction,
                   double current_time);
InstructionFinished(Instruction* instruction,
                    double current_time);
```

Listing 1: ParaGraph API with simulators

Internally, the ParaGraph runtime instantiates a copy of the PGIR for every logical process, each of which will become associated with a simulator endpoint. The runtime implements a state machine for all the instructions and subroutines. Each instruction may be in one of five possible states: (1) **Blocked**, meaning the instruction has unsatisfied operands (dependences); (2) **Ready**, meaning all dependences are satisfied and the instruction is waiting to be scheduled; (3) **Scheduled**, meaning the instruction is moved to the queue of instructions available to the simulator endpoint; (4) **Executing**, meaning the simulator has fetched this instruction but has not yet marked it as completed; (5) **Finished**, meaning the simulator has marked the instruction as completed. The state of instructions is updated at each API call.

The simulator instantiates a *GraphScheduler* object to associate with a logical process. Since the ParaGraph runtime maintains separate instances of the PGIR for each logical process, it is thread-safe for parallel simulators. The simulator uses `GetReadyInstructions` to get a list of ready instructions; calls `InstructionStarted` to signal to the ParaGraph runtime that the instruction has "started execution;" performs any simulator actions, like advancing simulation time or scheduling new events; and calls `InstructionFinished` when the instruction should be marked as complete. If the simulator can only handle a specific subset of instructions (Fig. 3), substitutions may be performed during translation (Section 3.2).

The `Conditional` and `While` instructions involve special handling by the runtime. For `Conditional` instructions, the runtime picks one of several inner subroutines using deterministic pseudorandom number generator based on the probability field of each subroutine registered with the inner subroutine. This field may be set manually or using instrumentation. For `While` instructions, the instruction node carries an explicit iteration counter, so runtime executes the body a specific number of times. Both the branching probabilities and iteration counters would come from profiling or other attributes available in the compiler IR (and otherwise assume default values configurable by the user).

4 CASE STUDIES

We conducted four co-design case studies. They reflect simulation-based "what-if" analysis workflows. The scenarios involve hypothetical systems that might otherwise be harder to model without ParaGraph's ability to connect the application's software to a high-fidelity hardware model. We emphasize these *methodological* contributions of ParaGraph over the specific simulation results. In particular, the case studies show how ParaGraph can accommodate a variety of workflows through combining different tools and employing different modeling approaches.

Unless stated otherwise, the simulations use SuperSim, a flit-level network simulator designed to model large-scale networks accurately [32]. SuperSim's interface accepts only `Delay`, `Send`, and `Recv` instructions. ParaGraph's translator (Section 3.2) supplies the collectives on top of that interface, and SuperSim performs packetization and forwarding of messages through the network. The simulations use the parameters shown in Table 1, where the router cycle time was picked to match TPUv3 network bandwidth [20].

Among the four case studies, the one in Section 4.2 validates modeled time from ParaGraph against a TPUv3 trace. The study of 4.3 validates a motif-like stencil application implementation translated by ParaGraph against the same application implemented natively in the SuperSim simulator.

Table 1: SuperSim simulation parameters

Network topology	2D torus 4x4, concentration 2, 32 terminals up to 16x16, concentration 2, 512 terminals
Network channel latency	1.561 ns (i.e., 0.3 meter cables)
Router architecture	output queued (OQ)
Router cycle time	1.561 ns
Router radix	12 (4 torus links, 8 links to 2 cores
Router core latency	109.27 ns (70 cycles)
Number of VCs	2
Routing algorithm	dimension order routing
Input buffer size	200 flits
Output buffer size	infinite
Flit size	128 bytes
Message size	1 flit

4.1 Optimal all-reduce algorithm search

This case study co-designs an all-reduce algorithm with a TPUv3-like system of deep learning accelerators connected by a 2D-torus. ParaGraph's translator supplies the two collective algorithms we wish to compare: a ring-based all-reduce, which is widely used in communication libraries and deep learning frameworks [36, 47] and bandwidth-optimal [6], against a "2-D all-reduce algorithm" designed for a 2-D torus [26]. In this workflow, we used an all-reduce generator that emits ParaGraph IR directly rather than translating from a compiler's IR. For the hardware, we consider two 32-core systems: a 4x4 torus of dual-core accelerators and an 8x4 system of single-core accelerators. We also vary local bandwidth between each core and the on-chip router connected to the network.

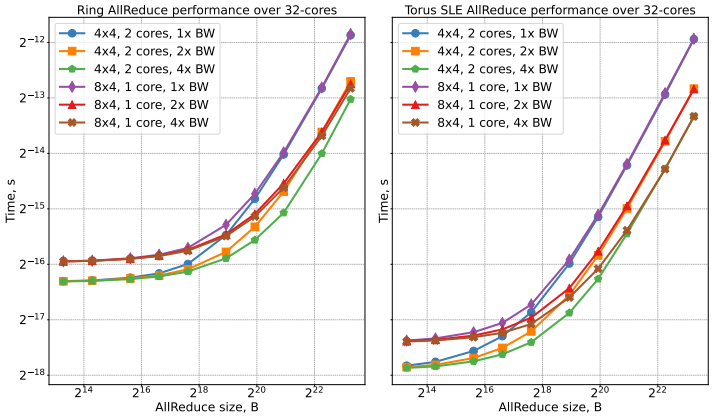


Figure 5: Optimal hardware configuration search for ring-based (left) and torus-based (right) all-reduce. Dual core configurations are seen to provide better latency. Ring and torus-based all-reduce require local bandwidth matching two external links for ring-based all-reduce and four for torus-based.

We start by analyzing the benefits of combining accelerator cores onto the same chip for ring-based all-reduce. Figure 5 (left) shows that a 4x4 torus with dual-core accelerators outperforms an 8x4 single-core configuration with respect to the latency (small message sizes). Communication between cores does not require full router arbitration and happens much faster.

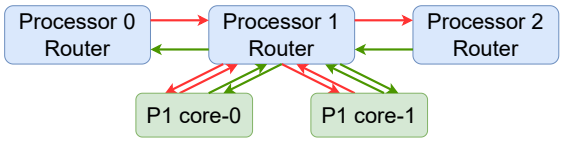


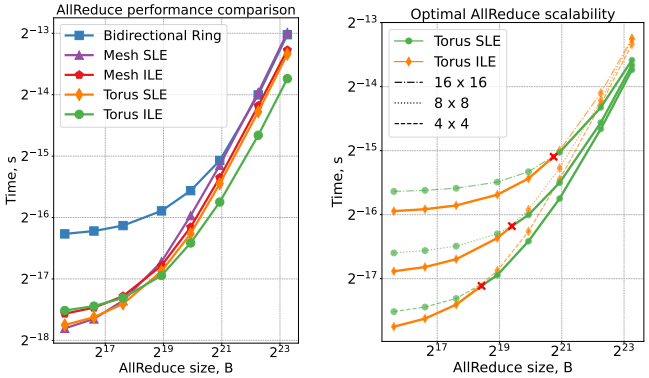
Figure 6: Schematics of communication flows in dual-core accelerator on a bidirectional ring. Bandwidth between cores and router has to match full bandwidth between router and its peers for full performance.

However, large message performance depends on the available network bandwidth and its utilization. To fully utilize bandwidth between routers, the bidirectional ring algorithm maintains two streams, clockwise and counterclockwise, as shown in Fig. 6. We need to match this bandwidth locally, doubling the bandwidth of the links between the cores and the router. Further increase in bandwidth improves performance a little more as link undersubscription eases bottlenecks in queueing and arbitration.

Despite being bandwidth optimal, a ring-based all-reduce can only inject traffic into two links out of four in the 2D torus. The 2-D all-reduce algorithm improves link utilization via concurrent bidirectional ring reduce-scatter followed by all-gather. The right of Fig. 5 shows that increasing local bandwidth by 4x improves the performance of all-reduce as it allows full utilization of all four torus links.

From these results, we focus on a configuration with a 4x4 torus of dual-core accelerators and 4× local bandwidth and then compare ring, torus, and mesh-based algorithms. Slicing a part of the system results in a mesh, so it is essential to consider mesh-based all-reduce in addition to the torus. Without the wraparound links, processors on a "line" can only reach another processor via one direction. That doubles the message size compared to the exchange possible on a bidirectional ring, but in that case processors are busy only half of the time. For a line P processors, edge processors send data only once but receive it $P-1$ times. The processor in the middle will send and receive data only $P/2+1$ times, as half of the processors are to its left and half of the processors are to its right.

In the context of multicore design, we also need to decide how we perform communication between the cores. We consider two different local exchange configurations. The first is a separate local exchange (SLE), which is performed after the exchange in both torus dimensions during the separate exchange step. We have three smaller rings, and the links between routers are not active during the local exchange. The second is integrated local exchange (ILE), which adds cores to the dimension iterated the last, resulting in two rings, with the second ring being larger. This approach utilizes links better than SLE.



(a) Performance comparison of various all-reduce algorithms on 4x4 torus of dual-core accelerators. There is no single best algorithm for all message sizes, so optimal algorithm should be a composition of several.

(b) Optimal all-reduce algorithm for torus topologies of various sizes. Each system size requires its own optimal parameter search.

Figure 7: Optimal all-reduce search.

The results in Fig. 7a indicate that the optimized 2D torus/mesh all-reduce outperforms a ring-based one due to fewer exchanges and better link utilization. Probing deeper, we consider all-reduce performance separately for small (10 KB) messages and large (10 MB) messages in Fig. 8. A small all-reduce SLE outperforms ILE for both mesh and torus due to fewer neighbor exchanges. On average, the mesh-based all-reduce appears to be more effective than the torus-based one. However, the error bars indicate that the spread of all-reduce completion times is wider for various processors than with both ring and torus-based implementations. Better performance

stability makes torus SLE a top choice, as network performance usually depends on the tail latency. For a large all-reduce, the torus ILE algorithm is a clear winner due to better link utilization.

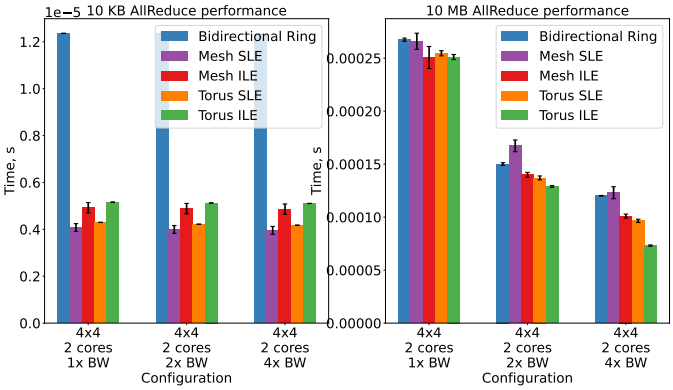


Figure 8: Small (left) and large (right) size all-reduce algorithms comparison on 4x4 torus of dual-core accelerators. Mesh-ILE achieve best small all-reduce performance on average, but larger error bar makes torus ILE a better option. Torus-SLE is a clear winner for large size all-reduce.

An optimal all-reduce algorithm should combine torus-SLE for small messages and torus-ILE for large messages. However, we need to determine the crossover point, which might vary with system scale. Comparing all-reduce performance on 4×4, 8×8, and 16×16 torus systems, Fig. 7b indicates that the larger the system size, the larger the crossover size. Ideally, these crossover points would be made available to the compiler or communication library so that they always choose the optimal algorithm to use.

Hardware and software configurations interact, and this study exemplifies the process of optimizing software library using a hardware model. Without the ability to co-design, it might otherwise have taken several generations of hardware implementations and subsequent software tuning to discover the best system configuration. ParaGraph can assist in tuning both hardware and software during a single optimization process over a space of candidate hardware configurations.

4.2 Matching an application trace

This case study compares measurements from an application trace that was recorded on real hardware against a variety of simulated results. The trace comes from the MLPerf training benchmark (v0.7) [31] running on a system with 32 TPUv3 dual-core accelerators organized in an 8x4 slice of a 2-D torus network and is measured at the IR level, so communication details are not resolved below the collectives level. The workflow uses ParaGraph to connect the TensorFlow application extracted from XLA, a production-grade machine learning compiler, with SuperSim configured to match the TPUv3 system (Table 1). Overall modeled time for MLPerf applications amounted to $O(100 - 1000)$ ms, with communication time in the range $O(1 - 10)$ ms.

First, we compare the trace's timing measurements with two simulations, one that uses a simple roofline as the node model and another that uses the trace's instruction timings as the node model. We consider the difference between modeling the node using roofline model for computation instructions, the node model utilizing computation statistics extracted from the trace, and an

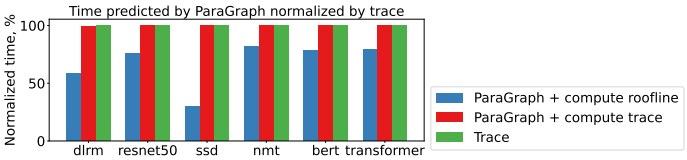


Figure 9: Comparison between ParaGraph model driven by roofline model for computation instructions, ParaGraph model that extracts computation statistics from trace, and the actual trace. Most of the modeling error is coming from using the roofline model.

actual benchmark trace to validate ParaGraph's approach. Figure 9 shows that MLPerf is sensitive to the node model. Whereas a roofline optimistically assumes 100% TPU utilization, MLPerf in practice attains just 30-75%. That corresponds to the TPU utilization one can see profiling MLPerf applications, e.g. by using cloud TPU profile tools. Substituting actual timings for the computational instructions (i.e., non-communication instructions) nearly eliminates the gap in performance estimates from simulation.

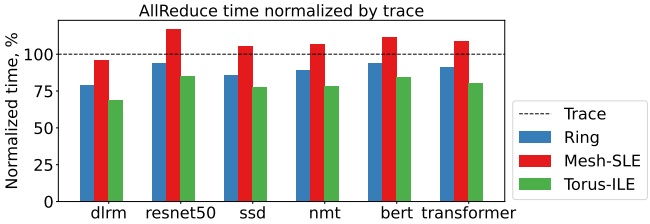


Figure 10: Comparison of modeled time for various all-reduce algorithms and the traced time, normalized by trace time.

Next, we focus on just the communication time, comparing simulated times against trace measurements. As we use the information about non-communication instructions timings directly from the trace, their timestamps coincide with the trace itself, and only communication instructions' timestamps differ between modeling and trace. Since the workloads only utilize data-parallelism and do not communicate much, as seen in Fig. 11, we report times only for the all-reduce operations to improve readability. The trace measures a closed-source all-reduce implementation, whereas our simulations use ParaGraph's. Figure 10 shows that the closest correspondence to the trace option comes from the ring or mesh-SLE algorithms, both of which are within 10% of the traced time. Reducing the gap further would require more detailed knowledge of the actual implementation and possibly also other networking details.

From the validation perspective, we can see that discrepancy between modeled and traced timestamps for communication instructions is rather small. This difference owes to the lack of knowledge on specific hardware timings and the exact all-reduce algorithm implementation.

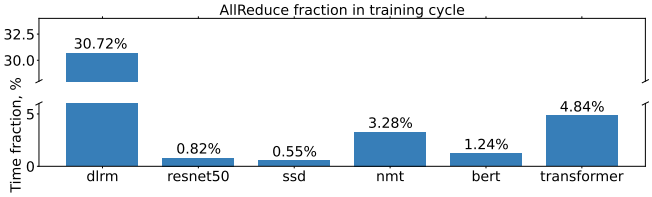


Figure 11: Fraction of training cycle that all-reduce takes for considered DL applications. Except DLRM, it is normally under 5% time.

Lastly, we examine the effect of changing the collectives algorithm on the performance of selected deep learning applications. We perform MLPerf models weak scaling study for system sizes, from 2×2 to 8×8 torus scaling from 8 to 128 cores. Figure 12 shows the differences between mesh-SLE as a realistic option and torus-ILE as an idealistic one. We know that the TPU system has a 2D-torus topology, and a slice of it is a 2D-mesh with no wrap-around torus link, so using torus-based all-reduce is impossible in practice, but we want to see how much performance we lose compared to torus-based system. We can see that torus ILE can achieve up to 50% performance improvement for all-reduce communication time. The impact is larger for DLRM [34] and large language models like BERT [12] and Transformer [55].

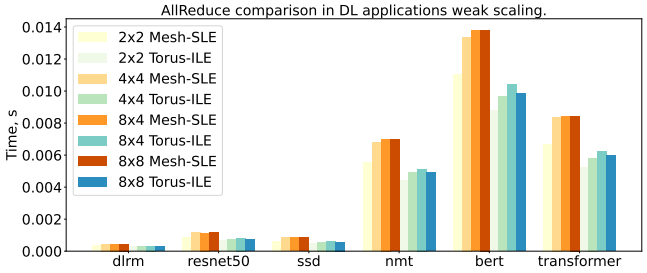


Figure 12: Comparison of mesh-SLE and torus-ILE all-reduce algorithms on DL applications weak scaling from 2×2 to 8×8 dual-core accelerators. We can see that significance of all-reduce algorithm increases with the system size increase.

Overall, this case study highlights the ability to analyze or incorporate hardware traces in ParaGraph. Also, ParaGraph does not preclude the use of more accurate node models, and mixing types of models is certainly possible.

4.3 Simplifying application modeling

This case study showcases ParaGraph's ability to simplify workload generation compared to what is typically required by "native" application implementation for network simulators. The workflow is similar to that of Section 4.1 but for a 27-point stencil benchmark consisting of three phases: compute, point-to-point neighbor exchange, and synchronization to disseminate results via all-reduce. There was a prior native implementation of this benchmark written specifically for SuperSim, which needed about 750 lines of C++. Interestingly, the generic ParaGraph endpoint, which works with any PGIR input, is about 30% smaller than that. For the ParaGraph workflow, we implemented a generator that constructs a PGIR for the stencil. For the aggregate exchange of 100 KB, the ParaGraph model reports 45.88 ms, while the native SuperSim model reports 46.66 ms, resulting in less than 2% discrepancy in modeled time.

One side-effect of using ParaGraph evident in this case study is the ability to amortize the cost of workload development. For instance, here we developed an SST-style motif for the workload with comparable effort [43]. However, since PGIR is decoupled from the simulator, the same workload could be reused with a different simulator. Furthermore, with fewer lines of code than the original application model, SuperSim can now simulate any workload represented by ParaGraph.

4.4 In-network reduction analysis

This case study concerns how to configure bandwidth for a fat-tree network with hardware support for reduction operations, or *in-network reductions* [23] for deep learning workloads. The workloads are full JAX-based DL applications compiled with XLA, and the ParaGraph workflow interfaces them to N10, a proprietary NVIDIA network simulator. N10 includes a model of a switch that supports in-network reductions and implements all-reduce operations directly. As such, lowering by ParaGraph's translator is not required. For our experiments, N10 is configured to match the NVSwitch network [37].

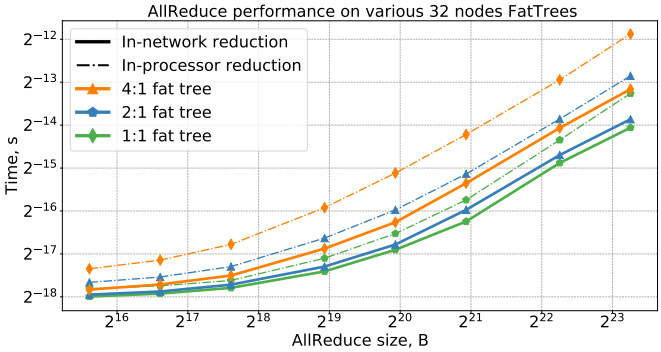


Figure 13: All-reduce performance analysis on different tapered fat tree topologies with in-network reductions. In-network reduction consistently provides a 2× performance gain over in-processor reduction for all topologies. However, it is surprising that all topologies do not perform exactly the same.

We consider in-network reductions for three topologies: full, 2:1 tapered, and 4:1 tapered fat trees. We test all-reduce using the generator of Section 4.1 for a simulated system with 32 GPUs. Regarding the fat tree topology, there are two levels of switches: edge switches and core switches. Each of the four edge switches connects to 8 GPUs (so 4×8 = 32 GPUs total); the four edge switches are all connected to two core switches. A tapering ratio of $r : 1$ means that the total bandwidth between an edge switch and all of its connected GPUs is r times larger than the total link bandwidth between the edge switch and all of its connected core switches.

In-network reductions provide about 2x performance improvement over conventional reductions, as shown in Fig. 13. This observation matches our expectations. However, tapering the fat tree degrades performance slightly for 2:1 oversubscription and significantly for 4:1. That contradicts Section 4.1, which shows that all-reduce can only depend on neighbor exchange performance, in which case tapering the fat tree topology should have no effect. That is, the in-network reduction implementation *behaves* as if it is based on an all-reduce that uses some kind of all-to-all pattern, which would be clearly suboptimal and should be further optimized.

Next, we ask what happens for three popular neural networks, data-parallel ResNet-50 and Transformer [17, 55], and hybrid model- and data-parallel Megatron [48]. The results in Fig. 14 show that in-network reductions do not improve performance much for data-parallel models, but we see that tapering worsens their performance, and in-network reductions help to reduce the gap. In contrast, the hybrid-parallel Megatron experiences a performance boost from in-network reductions while being indifferent to network tapering.

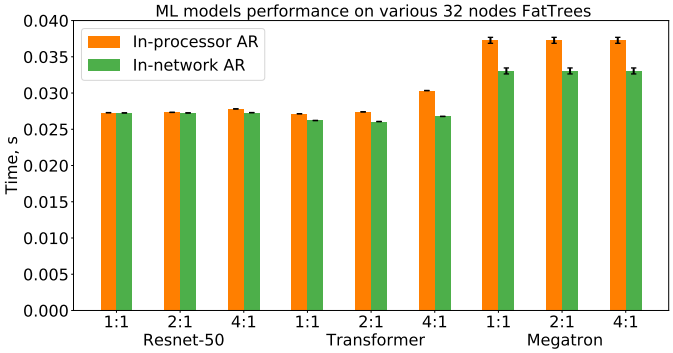


Figure 14: Performance analysis of various deep learning applications on different tapered fat tree topologies with in-network reductions. Data parallel Resnet-50 and Transformer show smaller performance gains from in-network reductions but larger performance variation between topologies, while hybrid-parallel Megatron demonstrates the opposite behavior.

With respect to in-network reductions, the differences between Transformer and Megatron are especially striking: despite having a similar architecture, they employ different parallelization strategies, thus highlighting the importance of modeling hardware and software *interactions*. The preceding results reflect that the data-parallel strategy enables overlap of computation and communication, whereas this strategy is not implemented in the application for model parallelism which puts communication on the critical path. A typical approach to model communication and computation overlap is "all or nothing"—it is either not modeled at all or modeled for each layer of the neural network. By contrast, these opportunities (or lack thereof) across the full workload are exposed directly in PGIR and thus can be simulated via the ParaGraph runtime.

Regarding tapering, the differences are similar to what we observed in Section 4.1. Data parallel workloads generate all-reduce operations over the whole network, thus bottlenecking an oversubscribed topology if an all-to-all pattern is used. Model parallel workloads generate all-reduce traffic over parts of the system and are thus less susceptible to network oversubscription.

Overall, this case study shows how ParaGraph aids in modeling end-to-end system performance for forward-looking hardware. Beyond specific insights about the impact of in-network reduction or tapering, it demonstrates key methodological features of ParaGraph, including (a) extraction of high-level application features, like opportunities to overlap computation and communication, from full JAX-based programs, and representing them directly in PGIR; and (b) being able to swap in alternative backend simulation tools.

5 CONCLUSION

A key aspect of ParaGraph is its versatility in mixing and matching components to construct end-to-end simulation workflows. The design of the lightweight IR and runtime is our main claim for novelty and potential impact in multiple areas, including network architecture, software systems research, and system co-design.

The case-study workflows of Section 4 involved motif-generators, compiler-extracted applications, and traces on the frontend; different algorithmic variations in the "mid-end"; and different simulators (SuperSim and N10) on the backend. This flexibility is enabled by ParaGraph's graph-based program representation "tuned" for interfacing with network simulators, its translation infrastructure, and its runtime. It also comes with "batteries included" to work with XLA-supported applications, whose prevalence we expect will continue to grow. To enable new use cases, ParaGraph is freely available open-source software. For our part, we plan to extend this work to support additional simulators, including SST. In the future, we also foresee ParaGraph's instruction set being extended to cover primitives necessary to model distributed math libraries for HPC applications performance analysis, e.g., various Basic Linear Algebra Subroutines (BLAS) and FFTs implementations.

ACKNOWLEDGMENTS

We thank Safeen Huda, Sameer Kumar, Nan Jiang, and Larry Dennison for the discussions that helped to shape this paper. We are also grateful to the anonymous reviewers, who made many thoughtful suggestions to improve the paper's quality. Lastly, we acknowledge Lluis Miquel Munguia, who set aside considerable time for this project, including the release of HLO graphs and traces collected from the TPUv3 run.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 33–42. https://doi.org/10.1109/ISPASS.2009.4919636

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax http://github.com/google/jax.

[5] David Cardwell and Fengguang Song. 2019. An Extended Roofline Model with Communication-Awareness for Distributed-Memory HPC Systems. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region* (Guangzhou, China) *(HPC Asia 2019)*. Association for Computing Machinery, New York, NY, USA, 26–35. https://doi.org/10.1145/3293320.3293321

[6] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. 2007. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 19, 13 (2007), 1749–1783. https://doi.org/10.1002/cpe.1206 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1206

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[8] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and Robert Ross. 2015. CODES: Enabling Co-design of Multilayer Exascale Storage Architectures. (8 2015). https://doi.org/10.2172/1311761

[9] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. *SIGPLAN Not.* 28, 7 (jul 1993), 1–12. https://doi.org/10.1145/173284.155333

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Thirty-eighth International Conference on Machine Learning* (Virtual). PMLR.

[11] William James Dally and Brian Patrick Towles. 2004. *Principles and Practice of Interconnection Networks.* Morgan Kaufmann Publishers, San Francisco, CA, USA. https://doi.org/10.5555/2821589

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[14] Keno Fischer and Elliot Saba. 2018. Automatic Full Compilation of Julia Programs and ML Models to Cloud TPUs. https://doi.org/10.48550/ARXIV.1810.09868 https://arxiv.org/abs/1810.09868.

[15] Google, LLC. 2022. XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla

[16] Dion Häfner, Roman Nuterman, and Markus Jochum. 2021. Fast, cheap, and turbulent—global ocean modeling with GPU acceleration in Python. *Journal of Advances in Modeling Earth Systems* (12 2021). https://doi.org/10.1029/2021MS002717

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

[18] Nikhil Jain, Abhinav Bhatele, Sam White, Todd Gamblin, and Laxmikant V. Kale. 2016. Evaluating HPC Networks via Simulation of Parallel Workloads. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 154–165. https://doi.org/10.1109/SC.2016.13

[19] Nan Jiang, Daniel U. Becker, George Michelogiannakis, James Balfour, Brian Towles, D. E. Shaw, John Kim, and William J. Dally. 2013. A detailed and flexible cycle-accurate Network-on-Chip simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* 86–96. https://doi.org/10.1109/ISPASS.2013.6557149

[20] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (jun 2020), 67–78. https://doi.org/10.1145/3360307

[21] Dounia Khaldi, Pierre Jouvelot, François Irigoin, and Corinne Ancourt. 2013. SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension. In *17th Workshop on Compilers for Parallel Computing (CPC 2013).* Lyon, France. https://hal-mines-paristech.archives-ouvertes.fr/hal-00823324

[22] Dounia Khaldi, Pierre Jouvelot, François Irigoin, Corinne Ancourt, and Barbara Chapman. 2015. LLVM Parallel Intermediate Representation: Design and Evaluation Using OpenSHMEM Communications. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15).* Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/2833157.2833158

[23] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. 2020. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* 996–1009. https://doi.org/10.1109/ISCA45697.2020.00085

[24] Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. 2021. Machine learning-accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences (PNAS)* 118, 21 (2021). https://doi.org/10.1073/pnas.2101784118

[25] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous Parallel Virtual Machine. *SIGPLAN Not.* 53, 1 (feb 2018), 68–80. https://doi.org/10.1145/3200691.3178493

[26] Sameer Kumar and Norm Jouppi. 2020. Highly Available Data Parallel ML training on Mesh Networks. arXiv:2011.03605 [cs.LG]

[27] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis and transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization.* 75–86.

[28] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020). arXiv:2002.11054 https://arxiv.org/abs/2002.11054

[29] Zhijing Li, Yuwei Ye, Stephen Neuendorffer, and Adrian Sampson. 2022. Compiler-Driven Simulation of Reconfigurable Hardware Accelerators. *CoRR* abs/2202.00739 (2022). arXiv:2202.00739 https://arxiv.org/abs/2202.00739

[30] Opeoluwa Matthews, Aninda Manocha, Davide Giri, Marcelo Orenes-Vera, Esin Tureci, Tyler Sorensen, Tae Jun Ham, Juan L. Aragon, Luca P. Carloni, and Margaret Martonosi. 2020. MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* 136–148. https://doi.org/10.1109/ISPASS48437.2020.00029

[31] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. arXiv:1910.01500 [cs.LG]

[32] Nic McDonald, Adriana Flores, Al Davis, Mikhail Isaev, John Kim, and Doug Gibson. 2018. SuperSim: Extensible Flit-Level Simulation of Large-Scale Interconnection Networks. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* 87–98. https://doi.org/10.1109/ISPASS.2018.00017

[33] Meta. 2020. PyTorch-XLA. https://github.com/pytorch/xla/ https://github.com/pytorch/xla/.

[34] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[35] nsnam. 2022. ns-3. https://www.nsnam.org/ https://www.nsnam.org/.

[36] NVIDIA. 2022. NCCL: NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl https://developer.nvidia.com/nccl.

[37] NVIDIA. 2022. NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/ https://www.nvidia.com/en-us/data-center/nvlink/.

[38] EPFL Parallel Systems Architecture Lab (PARSA). 2020. QFlex. https://qflex.epfl.ch https://qflex.epfl.ch.

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[40] M. Pharr and W. R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing - Foundations and Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012).* IEEE Computer Society, Los Alamitos, CA, USA, 1–13. https://doi.org/10.1109/InPar.2012.6339601

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[42] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 22-26, 2020.* IEEE.

[43] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (mar 2011), 37–42. https://doi.org/10.1145/1964218.1964225

[44] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2019. Glow: Graph Lowering Compiler Techniques for Neural Networks. arXiv:1805.00907 [cs.PL]

[45] Ben Sander. 2013. HSAIL: Portable compiler IR for HSA. In *2013 IEEE Hot Chips 25 Symposium (HCS).* 1–32. https://doi.org/10.1109/HOTCHIPS.2013.7478287

[46] Samuel S. Schoenholz and Ekin D. Cubuk. 2021. JAX, M.D.: A framework for differentiable physics. *J. Statistical Mechanics: Theory and Experiment* 2021, 124016 (12 2021). https://doi.org/10.1088/1742-5468/ac3ae9

[47] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799 [cs.LG]

[48] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL]

[49] Edgar Solomonik, Abhinav Bhatele, and James Demmel. 2011. Improving Communication Performance in Dense Linear Algebra via Topology Aware Collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington) *(SC '11).* Association for Computing Machinery, New York, NY, USA, Article 77, 11 pages. https://doi.org/10.1145/2063384.2063487

[50] Tyler Sorensen, Aninda Manocha, Esin Tureci, Marcelo Orenes-Vera, Juan L. Aragón, and Margaret Martonosi. 2020. A Simulator and Compiler Framework for Agile Hardware-Software Co-design Evaluation and Exploration. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[51] James Stanier and Des Watson. 2013. Intermediate Representations in Imperative Compilers: A Survey. *ACM Comput. Surv.* 45, 3, Article 26 (jul 2013), 27 pages. https://doi.org/10.1145/2480741.2480743

[52] Zhangxi Tan, Zhenghao Qian, Xi Chen, Krste Asanovic, and David Patterson. 2015. DIABLO: A Warehouse-Scale Computer Network Simulator Using FPGAs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 207–221. https://doi.org/10.1145/2694344.2694362

[53] CORPORATE The MPI Forum. 1993. MPI: A Message Passing Interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (Portland, Oregon, USA) *(Supercomputing '93)*. Association for Computing Machinery, New York, NY, USA, 878–883. https://doi.org/10.1145/169627.169855

[54] András Varga and Rudolf Hornig. 2008. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems* (Marseille, France) *(Simutools '08)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, Article 60, 10 pages.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[56] Richard Vuduc and Kent Czechowski. 2011. What GPU Computing Means for High-End Systems. *IEEE Micro* 31, 4 (2011), 74–78. https://doi.org/10.1109/MM.2011.78

[57] Jeremiah J. Wilke, Joseph P. Kenny, Samuel Knight, and Sebastien Rumley. 2018. Compiler-Assisted Source-to-Source Skeletonization of Application Models for System Simulation. In *High Performance Computing*, Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis (Eds.). Springer International Publishing, Cham, 123–143.

[58] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

A PAPER ARTIFACT DESCRIPTION APPENDIX

A.1 Computational Artifacts

Yes

A.2 Artifact Description Details

The paper presents ParaGraph, an open source framework to connect compilers with network simulators. It can be wrapped in a library and called from within the network simulators. In the Section 4 we extensively use SuperSim, an open source flit-level network simulator. We showcase it as an openly available example of ParaGraph integration.

ParaGraph is released on github as part of `ParaGraph-Sim` organization. It uses `bazel` to build and test it, but it also can be built with `cmake`, or downloaded using `spack` package manager.

ParaGraph is released with a few useful tools that help building flexible modeling workflows:

- `Graph_convertor` converts graphs between text-based and binary `protobuf` formats.
- `Graph_translator` translates graph according to provided `json` configuration file. If desirable translation does not exist, ParaGraph can be easily extended to cover new translation algorithms.
- `Simulator` provides a simple simulator that executes ParaGraph instruction using their `seconds` field as execution delay. It can be used as an analytical model for debugging or quick performance estimation.
- `Paragraph-creator` library implements motif-like application models. Currently available are allreduce generator used in Sections 4.1 and 4.4 and stencil generator used in Section 4.3.
- `Hlo_bridge` allows to model applications implemented in frameworks that support XLA compiler, e.g. TensorFlow and JAX applications.
- `Hlo_examples` has a collection of HLO graphs that can be converted to PGIR. They correspond to popular neural networks implemented in TensorFlow and used as part of MLPerf training (v0.7) benchmark, or various applications implemented in JAX. Several notebooks show how to generate HLO graphs from JAX Python applications.
- `Scripts` can be used too replicate experiments or as an example of building various modeling workflows.

To use ParaGraph application model on a simulator, we need to construct a modeling workflow. It may vary depending on the needs and chosen tools. Typical steps in such workflow are following:

(1) Obtain a composite ParaGraph graph – an SPMD view of the program translated into PGIR or implemented in it.
(2) Prepare translation configuration according to simulator ability to execute ParaGraph instructions. For example, as SuperSim understands only `send`, `recv`, and `delay` instructions, all other instructions, e.g. `all-reduce`, should be translated into this set.
(3) Individualize graph and translate it according to a specific configuration. This step prepares dedicated graph file for every network end-point that is going to be modeled during the experiment.
(4) Run simulator to obtain its results and ParaGraph log files. This step also includes preparing configuration scripts for simulator to set it up according to the modeled system design.

A.3 Software Artifact Availability

ParaGraph and its accompanied utilities are available as part of github organization `ParaGraph-Sim` at https://github.com/paragraph-sim. All the tools should work from their corresponding `main` branches according to the `README.md` files.

The easiest way to replicate the results reported in chapters Sections 4.1 and 4.2 is to use the scripts located at https://github.com/paragraph-sim/scripts. `Icpp_experiment_graph_gen.py` is used to generate graphs, run experiments, and dump raw data and ParaGraph logs. Some experiments, e.g. larger scale runs of neural network applications, require a large amount of storage space during graph translation and modeling. For example, if PGIR graph takes 5MB each, we model it for 128 cores, and use 5 different all-reduce translation options, total storage for graphs only (not including configuration scripts and log files) will be $5MB \times 5 \times 128 \approx 3GB$.

ParaGraph core library and the tools essential for typical workflows, e.g. `graph_convertor`, `graph_translator`, are available at https://github.com/paragraph-sim/paragraph-core. ParaGraph traffic generators implemented in PGIR, such as allreduce generator used in Sections 4.1 and 4.4 and stencil generator used in Section 4.3

are available as part of `paragraph-creator` library available at https://github.com/paragraph-sim/paragraph-creator. `HLO-bridge` necessary to convert compiled neural network models from Appendix A.5 is available at https://github.com/paragraph-sim/hlo-bridge.

We demonstrate integration with SuperSim network simulator available at https://github.com/ssnetsim/supersim. Integration is performed in files available in the same repository under https://github.com/ssnetsim/supersim/tree/main/src/workload/paragraph. To run a toy example with ParaGraph graph, SuperSim can be run with the json configuration file available at https://github.com/ssnetsim/supersim/blob/main/config/ring$_i$q$_p$aragraph.json.

A.4 Hardware Artifact Availability

N/a.

A.5 Data Artifact Availability

All-reduce data generator used in Section 4.1 and Section 4.4 is located at https://github.com/paragraph-sim/paragraph-creator/tree/main/src/allreduce.

MLPerf HLO graphs and TPU tracing information used in Section 4.2 are located at https://github.com/paragraph-sim/hlo-examples/tree/main/tensorflow/mlperf$_t$raining$_v$0.7.

Stencil data generator written in ParaGraph used in Section 4.3 is located at https://github.com/paragraph-sim/paragraph-creator/tree/main/src/stencil.

Stencil data generator written in SuperSim used in Section 4.3 is located at https://github.com/nicmcd/mkpg/tree/main/src.

HLO graphs of neural networks used in Section 4.4 are located at https://github.com/paragraph-sim/hlo-examples/tree/main/jax.

Megatron model is generated using script available at https://colab.research.google.com/drive/1JU-e0P4AfqSkV-MXzrrMGwaLed9PRJed.

A.6 Proprietary Artifacts

Experiments presented in Section 4.4 were run with a proprietary network simulator N10 described in [23]. It is not openly available and cannot be provided for reproducibility purposes.

A.7 General Info

A.7.1 Artifact 1.

Persistent ID: https://github.com/paragraph-sim/paragraph-core

Artifact name: ParaGraph

Relevant hardware details: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz

A.7.2 Artifact 2.

Persistent ID: https://github.com/ssnetsim/supersim

Artifact name: SuperSim

Citation of artifact:

```
@inproceedings{mcdonald2018supersim,
  title={{SuperSim: Extensible Flit-Level Simulation
    of Large-Scale Interconnection Networks}},
  author={McDonald, Nic and Flores, Adriana and Davis,
     Al and Isaev, Mikhail and Kim, John and Gibson,
    Doug},
  booktitle={International Symposium on Performance
    Analysis of Systems and Software (ISPASS)},
  year={2018},
  organization={IEEE}
}
```

Relevant hardware details: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz

A.8 System

Operating systems and versions: Ubuntu 16.04.7 LTS, newer x86 linux-based OS should work well.

Compilers and versions: bazel-3-7-2, gcc 9.4.0, c++17, Python 3.7.1.

Applications and versions:

All applications should be ParaGraph – commit `0d0f9e7`;
HLO bridge – commit `7d7eec3`;
SuperSim – commit `7d1ffbe`.

Libraries and versions:

According to bazel build requirerments.

Input datasets and versions:

HLO examples – commit `b7680b0`;
Creator library for allreduce and stencil generators – commit `02f0d37`;

URL:

https://github.com/paragraph-sim/paragraph-core
https://github.com/paragraph-sim/hlo-bridge
https://github.com/paragraph-sim/hlo-examples
https://github.com/paragraph-sim/paragraph-creator
https://github.com/ssnetsim/supersim