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

ParaGraph is a novel open-source toolkit for use in co-designing hardware and software for supercomputer-scale systems. It is directly motivated by our prior experience, namely, that building a high-fidelity HPC workload model to interface with an existing network simulator can be onerous and unproductive [4]. ParaGraph aims to decouple application modeling from hardware modeling. The first component is a high-level graph intermediate representation (IR) of a parallel program, which faithfully represents parallelism and communication and can be extracted automatically from a compiler. This IR is distinct from others in that it is "tuned" for use with network simulators. The second component is a runtime that can emulate this representation's dynamic execution for a simulator. User-extensible mechanisms are available for modeling on-node performance and lowering high-level communication into alternatives that backend simulators understand.

When co-designing interconnection networks for future systems, designers employ discrete-event simulators [2]. There are typically two basic approaches to model the workload: synthetic traffic generators and traces. Generators can represent traffic patterns, but ignore all other aspects of a computation that might significantly affect those patterns, even with motif-based generators [5]. Traces can be faithful to the application but can also be extremely large and "overfit" to the existing systems on which they were captured.

Regardless of the approach, a workload "enters" the simulation as a sequence of timestamped network message injection or extraction events at *endpoints* (terminals) Between such events, whatever happens on the compute node is abstracted away by delays between timestamps. This is problematic for co-design because applications embody many design choices, which such a simple simulator interface betrays. For example, communication or math libraries are typically tuned for particular systems, with choices made according to both the network's and the application's characteristics.

ParaGraph tries to improve how applications are modeled and integrated into the simulation workflow for multi-node systems co-design. It sits between a "frontend" application and a "backend" simulator, as illustrated in Fig. 1 and in Fig. 2. ParaGraph defines a native graph-based IR of the application and a runtime that can interpret this IR for the hardware simulator.

ParaGraph's IR (PGIR) statically expresses the structure of high-level control and data dependences in the program. It explicitly represents parallelism and communication, including collective operations, and is single-program multiple-data (SPMD) in

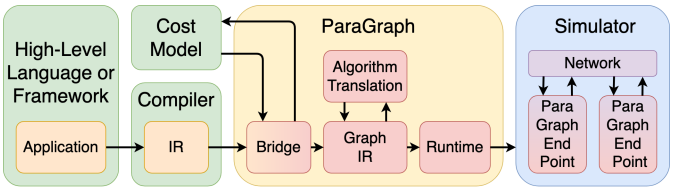

Figure 1: ParaGraph organization, including bridge with high level compiler, instruction translation, and interfacing with simulators through end-point model running ParaGraph runtime.

style. PGIR 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 PGIR [3]. Today, such programs include large-scale distributed deep learning computations written in TensorFlow [6]; there is also an emerging generation of HPC programs written in JAX, a NumPy-compatible drop-in for graphics co-processor (GPU) and Tensor Processing Unit (TPU) systems [1].

PGIR complements traditional compiler IRs. Whereas the latter are designed to aid in program analysis, optimization, and code generation for a backend processor, PGIR targets network simulators and so is simpler. PGIR includes extensible mechanisms to substitute different collective communication algorithms and can also store profile data or generate delay information needed by the backend simulator.

A simulator uses PGIR to emulate a workload by invoking ParaGraph's runtime through a thin application programmer interface (API). The design of this API is also informed by the needs of simulators, so interfacing with it needs just a modicum of code. ParaGraph's runtime interprets the IR and translates its operations into simulator events as needed.

We have conducted case studies to show that one can use ParaGraph to flexibly build different modeling workflows. 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. 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. Moving forward, we plan to extend this work to support additional simulators and develop more case studies and support for HPC applications. For more information, see: github.com/paragraph-sim/.