SC22: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-6654-5444-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/SC41404.2022.00016

HammingMesh: A Network Topology for Large-Scale Deep Learning

Torsten Hoefler*†, Tommaso Bonato*, Daniele De Sensi*, Salvatore Di Girolamo*, Shigang Li*, Marco Heddes†, Jon Belk†, Deepak Goel†, Miguel Castro†, and Steve Scott†

*Department of Computer Science, ETH Zürich, Rämistrasse 101, 8092 Zürich, Switzerland
†Microsoft Corporation, One Microsoft Way, Redmond, Washington 98052, United States of America

*{torsten.hoefler, tommaso.bonato, daniele.desensi, salvatore.digirolamo}@inf.ethz.ch, {shigangli.cs}@gmail.com
†{mattheus.heddes, jon.belk, mcastro, steve.scott}@microsoft.com, {degoel}@gmail.com

***Abstract*—Numerous microarchitectural optimizations unlocked tremendous processing power for deep neural networks that in turn fueled the AI revolution. With the exhaustion of such optimizations, the growth of modern AI is now gated by the performance of training systems, especially their data movement. Instead of focusing on single accelerators, we investigate data-movement characteristics of large-scale training at full system scale. Based on our workload analysis, we design HammingMesh, a novel network topology that provides high bandwidth at low cost with high job scheduling flexibility. Specifically, HammingMesh can support full bandwidth and isolation to deep learning training jobs with two dimensions of parallelism. Furthermore, it also supports high global bandwidth for generic traffic. Thus, HammingMesh will power future large-scale deep learning systems with extreme bandwidth requirements.**

***Index Terms*—Network architecture, Deep Learning, Clusters, Software defined networking**

I. Motivation

Artificial intelligence is experiencing unprecedented growth providing seemingly open-ended opportunity. *Deep learning* models combine many layers of operators into a complex function that is *trained* by optimizing its parameters to large datasets. Given the abundance of sensor, simulation, and human artifact data, this new model of designing computer programs, also known as data-driven programming or "software 2.0" [1], is mainly limited by the capability of machines to perform the compute- and data-intensive training jobs [2]. In fact, the predictive quality of models improves as their size and training data grow to unprecedented scales [3]. Building *deep learning supercomputers*, to both explore the limits of artificial intelligence and commoditize it, is becoming not only interesting to big industry but also humanity as a whole.

A plethora of different model types exist in deep learning and new major models are developed every two to three years. Yet, their computational structure is similar—they consist of layers of operators and they are fundamentally *data-intensive* [4]. Many domain-specific accelerators take advantage of peculiarities of deep learning workloads be it matrix multiply units ("tensor cores") [5], specialized vector cores [6], or specific low-precision datatypes [7]. Those optimizations can lead to orders of magnitude efficiency improvements [8], [9]. Yet, as we are approaching the limits of such microarchitectural improvements, we need to direct our focus to the system level.

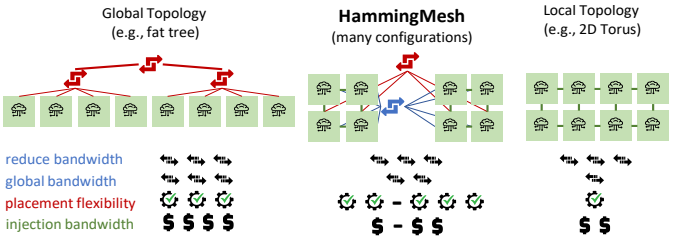


Fig. 1: HammingMesh's bandwidth-cost-flexibility tradeoff.

Today's training jobs are already limited by data movement [4]. In addition, trends in deep neural networks, such as sparsity, further increase those bandwidth demands in the near future [10]. Memory and network bandwidth are expensive—in fact, they form the largest cost component in today's systems [11]. Standard HPC systems with the newest InfiniBand adapters can offer 400 Gb/s but modern deep learning training systems offer much higher bandwidths. Google's TPUv2, designed seven years ago, has 1 Tbps off-chip bandwidth [12], AWS' Trainium has up to 1.6 Tbps per Tm1n instance [13], and Nvidia A100 and H100 chips have 4.8 and 7.2 Tbps (local) NVLINK connectivity, respectively [14], [15]. The chips in Tesla's Dojo deep learning supercomputer even have 128 Tbps off-chip bandwidth—*more than a network switch* [16]. Connecting these extreme-bandwidth chips at reasonable cost is a daunting task and today's solutions, such as NVLINK, provide only local islands of high bandwidth.

We argue that general-purpose HPC and datacenter topologies are not cost-effective at these endpoint injection bandwidths. Yet, workload specialization, similar to existing microarchitectural optimizations, can lead to an efficient design that provides the needed high-bandwidth networking. We begin with developing a generic model that accurately represents the fundamental data movement characteristics of deep learning workloads. Our model shows the inadequacy of the simplistic view that the main communication in deep learning is allreduce [17], [18]. In fact, we show that commu-

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

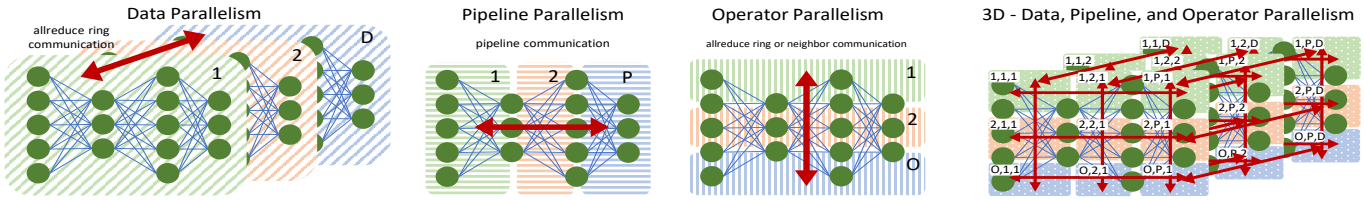


Fig. 2: Distribution strategies for parallel deep neural network training.

nication can be expressed as a concurrent mixture of pipelines and orthogonal reductions forming toroidal data movement patterns. This formulation shows that today's HPC networks, optimized for full global (bisection) bandwidth, are inefficient for deep learning workloads. Specifically, their *global bandwidth is overprovisioned while their local bandwidth is underprovisioned*.

We use our insights to develop HammingMesh, a flexible topology that can adjust the ratio of local and global bandwidth for deep learning workloads. HammingMesh combines ideas from torus and global-bandwidth topologies (e.g., fat tree) to enable a flexibility-cost tradeoff shown schematically in Figure 1. Inspired by machine learning traffic patterns, HammingMesh connects local high-bandwidth 2D meshes using row and column (blue and red) switches into global networks[1].

In summary, we show how deep learning communication can be modeled as sets of orthogonal and parallel Hamiltonian cycles to simplify mapping and reasoning. Based on this observation, we define principles for network design for deep learning workloads. Specifically, our HammingMesh topology

- uses technology-optimized local (e.g., PCB board) and global (optical, switched) connectivity.
- utilizes limited packet forwarding capabilities in the network endpoints to achieve lower cost and higher flexibility.
- enables full-bandwidth embedding of virtual topologies with deep learning traffic characteristics.
- supports flexible job allocation even with failed nodes.
- enables flexible configuration of oversubscription factors to adjust global bandwidth.

With those principles, HammingMesh enables extreme off-chip bandwidths to nearest neighbors at more than 8x cheaper allreduce bandwidth compared to standard HPC topologies such as fat trees. HammingMesh reduces the number of external switches and cables and thus reduces overall system cost. Furthermore, it provides significantly higher flexibility than torus networks. HammingMesh also enables seamless scaling to larger domains without separation between on- and off-chassis programming models (like NVLINK vs. InfiniBand). And last but not least, we believe that HammingMesh topologies extend to other machine learning, (multi)linear algebra, parallel solvers, and many other workloads with similar traffic characteristics.

We start with a characterization of parallel deep learning and the related data movement patterns. For reference, we provide an overview of symbols used in this paper in Table I.

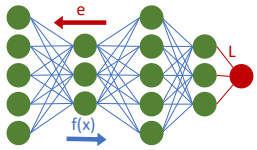
[1]The name *HammingMesh* is inspired by the structural similarity to 2D Hamming Graphs with Meshes as vertices.

Symbol	Description
M	number of examples per minibatch
$N_{\mathcal{P}}$	number of network parameters
W	size of a word
D, P, O	degree of data, pipeline, operator parallelism
a, b and x, y	2D HammingMesh board and global sizes

TABLE I: Symbols used in the paper

II. Communication in Distributed Deep Learning

One iteration of deep learning training with Stochastic Gradient Descent (SGD) consists of two phases: the forward pass and the backward pass. The forward pass evaluates the network function $f(x)$ on a set of M examples, also called a "minibatch". The backward pass of SGD computes the average loss L and propagates the errors e backwards through the network to adapt the parameters $\mathcal{P}$. This training process proceeds through multiple (computationally identical) iterations until the model achieves the desired accuracy.



Parallelism and data distribution can fundamentally be arranged along three axes: *data parallelism*, *pipeline parallelism*, and *operator parallelism* [19]. The latter two are often summarized as *model parallelism* and operator parallelism is sometimes called tensor parallelism [13]. We now briefly discuss their main characteristics.

A. Data parallelism

When parallelizing over the training data, we train D separate copies of the model, each with different examples. To achieve exactly the same result as in serial training, we sum the distributed gradients before applying them to the weights at the end of each iteration. If the network has $N_{\mathcal{P}}$ parameters, then the communication volume of this step is $WN_{\mathcal{P}}$.

Modern deep neural networks have millions or billions of parameters, making this communication step expensive. Thus, many optimizations target gradient summation [18]—some even change convergence properties during the training process but maintain final result quality [20]. Dozens of different techniques have been developed to optimize this communication—however, all perform some form of distributed summation operation like *MPI_Allreduce*. Data-parallelism differs thus mostly in the details such as invocation frequency, consistency, and sparsity [18], [21]–[28]. We describe various schemes in Appendix A in more detail.[2]

[2]The appendix is included in the supplementary material.

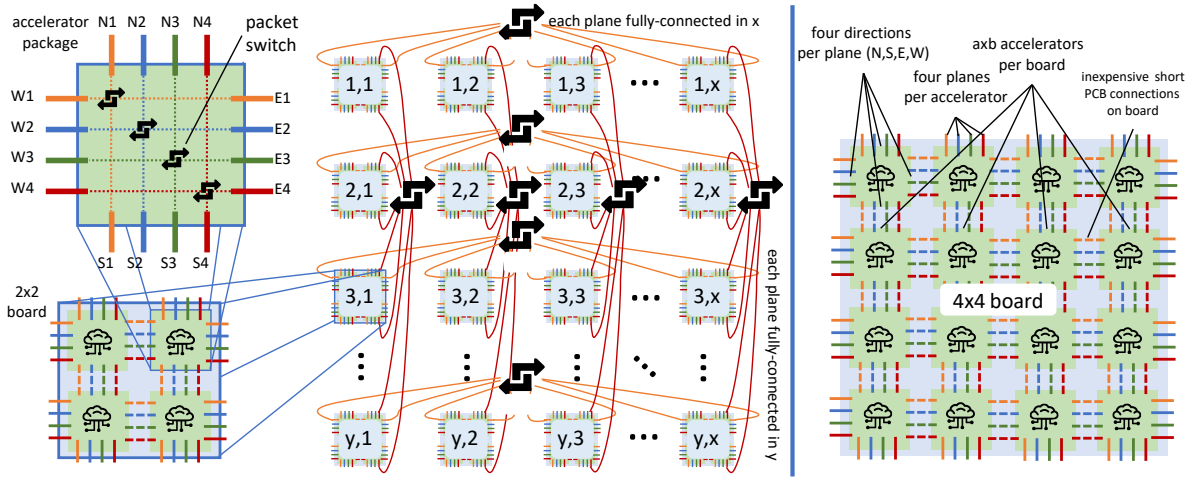


Fig. 3: HammingMesh structure: left $x \times y$ Hx2Mesh, right Hx4Mesh board, both with four planes.

B. Pipeline parallelism

Deep neural networks are evaluated layer by layer with the outputs of layer i feeding as inputs into layer $i+1$. Back-propagation is performed along the reverse direction starting at the loss function L after the last layer and proceeding from layer $i+1$ to layer i. We can model the network as a pipeline with P stages with one or more layers per stage [29]. Forward and backward passes can be interleaved at each processing element to form a bidirectional training pipeline [30]. Pipelines suffer from characteristic start-up and tear-down overheads. These can be reduced by running two pipelines in both directions [31] or by using asynchronous schemes that impact convergence [32].

Overall, pipelining schemes can use P processors with a nearest-neighbor communication volume proportional to the number of output activations at the cut layers.

C. Operator parallelism

Very large layer computations (operators) can be distributed to O processors. Most deep learning layer operators follow computational schedules of (multi-)linear algebra and tensor contractions and require either (tightly-coupled) distributed reductions or nearest-neighbor communications. We discuss communication schemes of typical operators in Appendix B.

D. Overall communication pattern

When all forms of parallelism are used, then the resulting job comprises $D \times P \times O$ accelerators; each accelerator in a job has a logical address $(1..D, 1..P, 1..O)$. The data-, pipeline-, and operator-parallel communication can be arranged as one-dimensional slices (rings) by varying only one coordinate of the Cartesian structure. Pipelines would leave one connection of the ring unused. For example, the data-parallel dimension consists of $P \cdot O$ rings of length D each. Each of those rings represents a single allreduce. We show efficient ring-based reduction and broadcast algorithms for large data volumes in Section V-A2.

The overall composition of communication patterns forms a torus as illustrated in the right part of Figure 2 for a $3 \times 3 \times 3$ example: Both the operator and the data parallel dimensions use nine simultaneous allreductions of size three each. The pipeline parallel dimension uses nine three-deep pipelines on three different model replicas, each split in three pieces.

While we can map such a logical torus to a full-bandwidth network topology, it seems wasteful to provide full bandwidth for sparse communication. For example, a 400 Gb/s non-blocking fat tree with 16,384 endpoints provides full bisection bandwidth of more than $\frac{16{,}384 \cdot 50\,\text{GB/s}}{2} = 410$ TB/s. A bi-directional 32x32x16 torus communication pattern requires at most $32 \cdot 16 \cdot 2 \cdot 50$ GB/s$= 51.2$ TB/s bisections (cutting one dimension of size 32) - a mere 12.5% of the offered bandwidth. In other words, *88% of the available bandwidth will remain unused and is wasted*. Furthermore, it is not always simple to map such torus communication patterns efficiently to full-bandwidth low-diameter topologies in practice [33].

III. HammingMesh

Based on the communication workload analysis, we now design a flexible and efficient network topology. The basic requirements are to support highest injection bandwidth for a set of jobs, each following a virtual toroidal communication topology. We note that medium-size models are often decomposed only in two dimensions in practice (usually data and pipeline or data and operator). Only extreme-scale workloads require all three dimensions—even then, communication along the data parallel dimension only happens after one complete iteration. Thus, we use a two-dimensional physical topology.

As a case study, we assume a modern deep learning accelerator package with 16 400 Gb/s off-chip network links, a total network injection bandwidth of 800 GB/s (top left in Figure 3). Our topology design also takes technology costs into account: Similar to Dragonfly, which combines local short copper cables with global long fiber cables to design a cost-effective overall topology, we combine such local groups with a global topology. Different from Dragonfly, we choose two quite distinct topologies: The local groups are formed by a local inexpensive high-bandwidth 2D mesh using short metal traces on PCB boards. This is the opposite of Dragonfly designs, which combine densely-connected local groups ("virtual switches") and connect those fully globally. HammingMesh combines

Topology	Small Cluster (≈1,000 accelerators) cost [M$]	glob. BW [% inject]	global saving	ared. BW [% peak]	ared. saving	diam.	Large Cluster (≈16,000 accelerators) cost [M$]	glob. BW [% inject]	global saving	ared. BW [% peak]	ared. saving	diam.
nonbl. FT	25.3	99.9	1.0x	98.9	1.0x	4	680	98.9	1.0x	99.8	1.0x	6
50% tap. FT	17.6	51.2	0.7x	98.9	1.4x	4	419	47.6	0.8x	99.8	1.6x	6
75% tap. FT	13.2	25.7	0.5x	98.9	1.9x	4	271	24.0	0.6x	99.8	2.5x	6
Dragonfly	27.9	62.9	0.6x	98.8	0.9x	3	429	71.5	1.2x	98.6	1.6x	5
2D HyperX[3]	10.8	91.6	**2.1x**	98.1	2.3x	4	448	95.8	1.5x	91.4	1.4x	8
Hx2Mesh	5.4	25.4	1.2x	98.3	4.7x	4	224	25.0	0.8x	92.3	2.8x	8
Hx4Mesh	2.7	11.3	1.0x	98.4	**9.3x**	8	43.3	10.5	**1.7x**	92.2	**14.5x**	8
2D torus	2.5	2.0	0.2x	98.1	10.1x	32	39.5	1.1	0.2x	91.4	15.7x	128

TABLE II: Overview of our example networks (small and large cluster) using the cost model in Section III-C. All bandwidths are the result of the packet-level simulations detailed in Section V-A. Global alltoall bandwidth is reported as share of the injection bandwidth for large messages (1.6 Tb/s). Allreduce bandwidth is reported as share of the theoretical optimum (1/2 of the injection bandwidth) for large messages. The cost savings for global and allreduce bandwidth are relative to the corresponding network cost of the nonblocking fat tree. Note that the diameter counts all cables and its derivation is explained in Section III-B.

sparsely connected boards in a dimension-wise (not globally) fully-connected topology. Those boards are connected by a two-dimensional Hamming graph, in which each dimension is logically fully connected (e.g., by a fat tree). All accelerator ports are arranged in *planes* with four directions each. Our example accelerator has four planes (top left in Figure 3), e.g., plane 1 has ports E1, W1, N1, and S1. We assume that each accelerator can forward packets within a plane like any network switch. Accelerators do not have to forward packets between planes, e.g., packets arriving at N1 may only be forwarded to E1, W1, or S1 but none of the other ports. Thus, only simple 4x4 switches are needed at each accelerator. Figure 3 illustrates the structure in detail.

A 2D HammingMesh is parameterized by its number of planes and four additional numbers: (a, b), the dimensions of the board, and (x, y), the dimensions of the global topology. It connects a total of $abxy$ accelerators. We abbreviate HammingMesh with HxMesh in the following. Furthermore, an HxMesh with an $a \times b$ accelerator board is called HaxbMesh, e.g., for a 2x2 board, H2x2Mesh. For square board topologies, we skip the first number, e.g., an H2x2Mesh that connects 10x10 boards is called a 10x10 Hx2Mesh.

HxMesh has a large design space: We can combine different board and global topologies, e.g., 3D mesh boards with global Slim Fly topologies [34]. In this work, we consider 2D boards as most practical for PCB traces. The board arrangement could be reduced to a 1D HxMesh, where $y = 1$ and each Nk link is connected to the corresponding Sk link ("wrapped around"). The same global topology can also span multiple rows or columns (e.g., full boards in a single fat tree). For ease of exposition, we limit ourselves to 2D HxMeshes using 2D boards and row/column-separated global topologies. We use two-level fat trees as global topologies to connect the boards column and row wise. If the boards can be connected with a single 64-port switch, we use that instead of a fat tree.

A. Bisection and global bandwidth

Bisection cut is defined as the minimal number of connections that would need to be cut in order to bisect the network into two pieces, each with an equal number of accelerators. The bisection bandwidth is the cut multiplied by the link bandwidth. Let us assume a single-plane of an $x \times y$ HxaMesh (square board) with $x \leq y$ and y even, wlog. We now consider the $xy/2$ "lower" half boards with y coordinates $1, 2, \ldots y/2$. We split the HxMesh into two equal pieces by cutting the $2a$ links in y direction of each of the lower half of the boards. This results in a total cut width of axy. Each accelerator has four network links per plane, a total injection bandwidth of $4a^2$ per board. We have $xy/2$ boards with a total injection bandwidth of $4a^2xy/2 = 2xya^2$ in each partition. Thus, the relative bisection bandwidth is $axy/2xya^2 = 1/2a$.

In a bisection traffic pattern, all traffic crosses the network bisection (any two communicating endpoints are in different sets of the bisection). Such (worst-case) patterns are rare in practice. A more useful pattern, more often observed in practice is alltoall, where each process sends to all other processes. This pattern is the basis of parallel transpositions, Fast Fourier Transforms, and many graph algorithms. The achievable theoretical bandwidth for such alltoall patterns is often called "global bandwidth". Some topology constructions take advantage of the fact that global bandwidth is higher than bisection bandwidth. Prisacari et al. [35] shows that full-global bandwidth (alltoall) fat trees can be constructed with 25% less switches than nonblocking fat trees. Dragonfly [36], Slim Fly [34], or other low-diameter topologies [37] can further reduce the number of switches in very large installations while maintaining full global bandwidth. As is customary for low-diameter topologies [34], [36], we assess it using packet-level simulations of alltoall traffic.

B. Network diameter

We now analyze the diameter of the different topologies by counting cables between source and destination. For example, a two-level fat tree has diameter four, while many works discount the cable to/from the endpoint, we count it to ensure fairness with direct topologies such as simple torus networks.

To compute the diameter of HxMesh, we consider the distance between two accelerators on two different boards. Both accelerators are as distant as possible from the edges of the corresponding boards, and source and destination boards

[3]Note that a 2D HyperX is identical to an Hx1Mesh

are on different rows and columns. $\lfloor (a-1)/2 \rfloor$ hops lead from an inner accelerator on a board to the East or West edge. The switch allows to take two cables to an intermediate board (sharing the same HxMesh row of the source board, and the same HxMesh column of the destination board). Once on the intermediate board, we need another $\lfloor (a-1)/2 \rfloor$ hops to reach the correct board column. From there, we need $\lfloor (b-1)/2 \rfloor$ hops to reach the North or South edge, and then two cables (through the switch) to reach the destination board. Eventually, other $\lfloor (b-1)/2 \rfloor$ hops are needed on the destination board to reach the destination accelerator.

The previous discussion assumes a flat (alltoall) global topology, which is not practical for large x and y. If we instead use a full bandwidth fat tree built from routers with k ports for global connectivity, it has a diameter of $2(\lceil \log_{k/2} q/k \rceil + 1)$, where q is the number of endpoints. Because in HxMesh a tree connects either boards on the same column or on the same row, and because each tree is connected to two opposite edges of a board, we either have $q = 2x$ or $q = 2y$. Thus, the HxMesh diameter is $2(\lfloor (a-1)/2 \rfloor + \lfloor (b-1)/2 \rfloor) + 2(\lceil \log_{k/2} 2x/k \rceil + 1) + +2(\lceil \log_{k/2} 2y/k \rceil + 1)$.

C. Cost model

We analyze capital expenditure when purchasing the system in the following. We assume that the accelerator NICs and ports as well as the PCB are included in the endpoint packaging cost for all topologies. We charge for the network equipment: optical transceivers, cables, and switches. We use three types of cables and a single type of switch with costs from Colfaxdirect (in April 2022). A single 64-port switch costs $14,280. One 20m Adaptive optical Cable (AoC) costs $603 and one 5m Direct Attach Cables (DAC) costs $272. All topologies can be built with these two types of cables using standard 19 inch racks. To compute the costs in Table II, we use DAC cables to connect to the endpoints and AoC cables between switches. More details are found in Appendix E.

D. Example topologies

We consider a small cluster with approximately 1,000 accelerators and a large cluster with approximately 16,000 accelerators as specific design points to compare realistic networks. We compare various fat trees (nonblocking, 50%, 75% tapered), full bandwidth Dragonfly, and two-dimensional torus, with Hx2Mesh and Hx4Mesh example topologies.

Table II summarizes the main cost and bandwidth results. Global and allreduce bandwidth are determined using packet-level simulations (see Section V) for large messages. *For all experiments, we simulated a single plane of HammingMesh and four planes for all other topologies, i.e., a total injection bandwidth of 4×400 Gb/s.* We use industry-standard layouts and cable configurations for the cost estimates: fat trees are tapered beginning from the second level and connect all endpoints using DAC and all switches using AoC. Dragonfly topologies use full-bandwidth groups with $a = 16$ routers each, $p = 8$ endpoints per router, and $h = 8$ links to other groups with DAC links inside the groups and AoC links between groups. The torus uses 2×2 board topologies with discounted local PCB connectivity, similar to Hx2Mesh and only DAC cables between the boards. For HxMeshes, we use DAC links to connect endpoints to switches along one dimension, and AoC links for the other dimension. All inter-switch links are AoC as in fat trees. All details and exact cable counts are described in Appendix C.

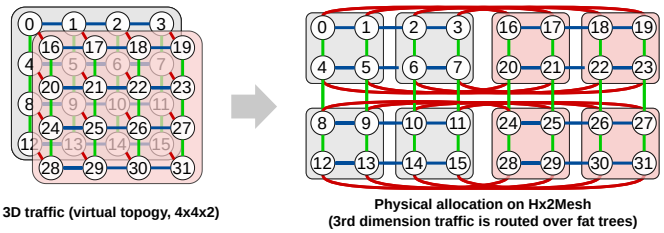


Fig. 4: 3D workload mapping onto Hx2Mesh example. Left: virtual 4x4x2 topology. Right: mapping on Hx2Mesh.

E. Logical job topologies and failures in HxMesh

As we discussed in Section II-D, communication patterns in deep learning can be modeled as sets of cycles. Typical learning jobs use either logical 1D cycles for small models with only data parallelism or 2D tori that combine data and pipeline parallelism for medium-scale models or combining pipeline and model parallelism for very large models. Each specific training job will have a different optimal decomposition resulting in 1D, 2D, or sometimes even 3D logical communication topologies.

We use logical 2D topologies for our training jobs. Each job uses several boards and requests a $u \times v$ layout (i.e., a, b divides u, v, respectively). If the application topology follows a 1D or 3D scheme, then users use standard folding techniques to embed it into two dimensional jobs [38]. Figure 4 shows an example of 3D virtual topology mapped on an Hx2Mesh physical topology. Processes can be sliced on the third dimension and mapped on different boards. Communications between different slices of the third dimension are routed over the per-column or per-row fat trees, depending how different slices are mapped. To minimize communication latency between slices, consecutive slices should be adjacent to each other. Furthermore, we will show in Section V-A2 how to accelerate allreduce on a 2D torus instead of a ring.

It is easy to see that any consecutive $u \times v$ block of boards in a 2D HxMesh has the same properties as a full $u \times v$ HxMesh. We call such subnetworks *virtual sub-HxMeshes*. They are a major strength of HxMesh compared to torus networks in terms of fault tolerance as well as for allocating jobs. In fact, HxMeshes major strength compared to torus networks is that virtual subnetworks can be formed with non-consecutive sets of boards (not only blocks): any set of boards in an HxMesh where all boards that are in the same row have the same sequence of column coordinates can form a virtual subnetwork. We will show examples below together with a motivation for subnetworks—faults.

a) Fault-tolerance: We assume that a board is the unit of failure in an HxMesh, i.e., if an accelerator or link in a board fail, the whole board is considered failed. This simplifies system design and service. Partial failure modes (e.g., per plane) are outside the scope of this work.

The left part of Figure 5 shows a 4x4 Hx2Mesh and three board failures. We show two different subnetworks (many more are possible): a 2x4 subnetwork (blue) with the physical boards $(1,1),(1,4),(2,1),(2,4),(3,1),(3,4),(4,1),(4,4)$ and a 3x3 subnetwork (yellow) with the physical boards $(1,1),(1,2),(1,4),(2,1),(2,2),(2,4),(4,1),(4,2),(4,4)$. We also annotate the new coordinates of boards in the virtual subnetworks. Remapping can be performed transparently to the user application, which does not observe a difference between a virtual and physical HxMesh in terms of network performance. The right part of the figure shows the output of our automatic mapping tool for a more complex configuration of jobs (top, read job ids 1-3 are 3×3 logical jobs etc.). We analyze in detail the effects of fragmentation due to failed boards in Section IV-B

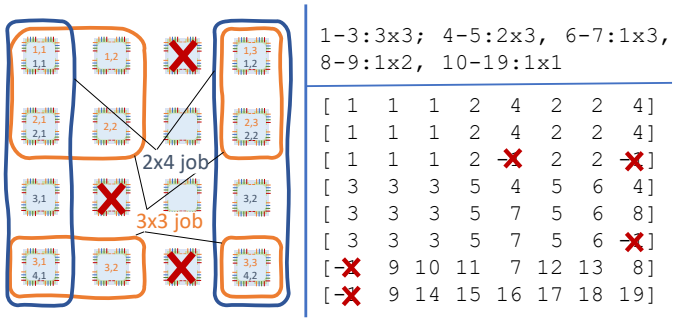


Fig. 5: Subnetworks in the case of failures

F. Tapering the dimensions

As we discussed in Section II-D, deep learning workloads do not require much global bandwidth. We can thus reduce the global bandwidth of HxMeshes further by tapering each dimension to reduce the cost.

First, we observe that for mapping a single ring along one dimension of an HxMesh, we only need two ports between neighboring switches in that direction. Consider the Hx2Mesh in Figure 6 with a two-level fat tree to implement the global connections and a ring spanning all accelerators in x direction (we show only one of the 8 identical physical rings for clarity):

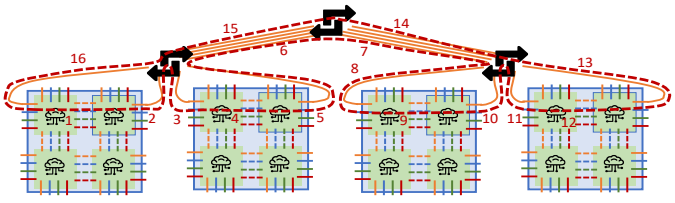


Fig. 6: Ring mapping example in a two-level fat tree.

We observe that switches in the top level only route two connections between neighboring switches. In this example, only 50% of the links are utilized. More generally, we rarely need full bandwidth in the global topology. We only need it for global traffic and to support fragmented (non-consecutive) allocations. For example, for a large deep learning system, we may only need 20% global bandwidth to support global traffic and moderate fragmentation. This provides a second dial in addition to the board size to adjust the tradeoff between global bandwidth and cost.

IV. Using HammingMesh in Practice

Training jobs request two-dimensional sets of boards. The allocation of rectangular jobs on a 2D torus is equivalent to the strongly NP-hard 2D bin packing problem [39]. Yet, HxMesh allocations support splitting blocks as illustrated in Figure 5, significantly simplifying the problem. We now show a simple and effective greedy allocation strategy.

A. Allocating jobs on HammingMesh

We develop the following simple greedy allocation strategy for an $au \times bv$ job to an $x \times y$ HxMesh:

1) Identify all available indexes in each row, resulting in y sets of at most x indexes.
2) Set $\mathcal{S}$ ("selected") to the first row with at least v indexes.
3) Add another row whose intersection with all rows in $\mathcal{S}$ has at least v indexes to $\mathcal{S}$.
4) Repeat the last step until $\mathcal{S}$ contains u rows, fail if no such set exists.

We implemented this procedure in less than 50 lines of Python and allocated an extremely large 1,000x1,000 HxMesh in less than one second on a laptop. We now outline a set of simple optimization heuristics that can be used to improve utilization of HxMeshes.

Transpose If it fails to find an $u \times v$ block, then we retry to find a (transposed) $v \times u$ block.

Aspect ratio Jobs could also allow to change their aspect ratio. For example, a job requesting 4x16 boards may also function well with 2x32 boards.

Sorting If the jobs to be allocated are known in advance, they can be allocated from the largest to the smallest to reduce fragmentation.

Locality The allocation algorithm can evaluate different aspect ratios and select the one that, for alltoall traffic, minimizes the traffic in the upper levels of the trees connecting the boards (so that performance is less affected by tapering).

We analyze the impact of these optimization on the system utilization below.

a) Job interference: A valuable property of this allocation scheme is that it avoids network interference between different jobs. Because each board is used at most by one job at a time, interference can never occur within the board. Even when source and destination are on two different boards on the same row (or on the same column) a nonblocking fat tree with packet spraying can avoid interference. When source and destination boards are on different rows and different columns, packets will traverse an intermediate board (Section IV-C). Our allocation scheme ensures that each board shares at least the row or the column with another board belonging to the same job (e.g., see Figure 5). Thus, packets never cross boards belonging to a different job to reach any destination.

b) Defragmentation: Modern cluster and cloud systems supports efficient checkpoint/restart. Thus, we assume that we can *defragment* the system by checkpointing jobs, shuffling them, and restarting them in a better permutation. Our example accelerator can send 64 GiB in less than 80 ms. Thus, a system with reasonable global bandwidth (e.g., 10%) can be defragmented in less than a second. Given the long running times of deep learning workloads (hours to days), we expect that such defragmentation will not impact the user experience or utilization of the overall system during operation.

B. Experimental workloads

We analyze the quality of our allocation algorithm using the distribution of job sizes extracted from a two-month workload log of Alibaba's ML-as-a-service (MLaaS) cluster with 6,742 GPUs [40], [41]. We aim to simulate how well a representative job mix that completely fills a full global bandwidth topology can be allocated on an HxMesh. For this, we draw 1,000 such job mixes from the job size distribution. We do this by sampling a job size, multiply it by the size of the board, and add it to the cluster. We repeat this step until we fill the target cluster fully (we carry samples that do not fit to the next jobs mix). We store the (random) order of drawn samples in a job trace. Figure 7 shows the cumulative distribution function of the proportion of boards allocated to jobs of a specific size for both the original distribution and the sampled distribution that fully occupy the Alibaba cluster.

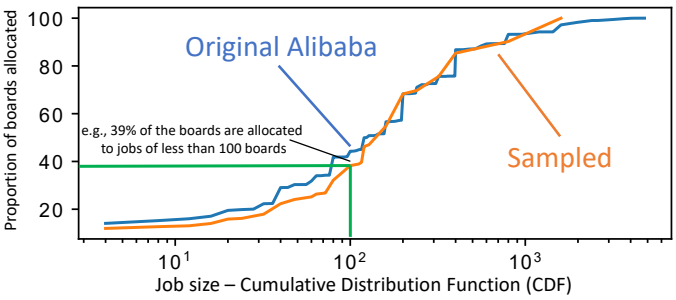


Fig. 7: Proportion of boards allocated to a certain size job.

Figure 8 shows the system utilization of our greedy allocation algorithm and different optimization heuristics on the different HxMeshes described in Table II. The figure shows the distribution of 1,000 allocations of random job traces. Full-global bandwidth topologies achieve 100%. By default, we make jobs as square as possible. The aspect ratio optimization allows changing the aspect ratio up to eight.

We observe that, even without any optimization, the greedy algorithm leads to a 90% system utilization. When transposing the jobs this further increases by an additional 5-8%. If jobs are further sorted by their size, we observe a mean and median utilization higher than 98%, with the 99th percentile higher than 95%.

We now investigate the expected load on the upper levels of the fat tree in the large cluster variants. Figure 9 shows the fraction of traffic crossing the upper levels of the fat trees, both for jobs running alltoall traffic or allreduce traffic, respectively.

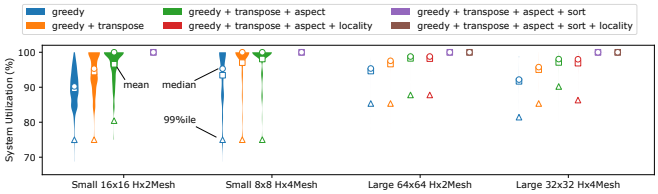


Fig. 8: System utilization using different optimizations. Each color corresponds to a specific set of optimizations.

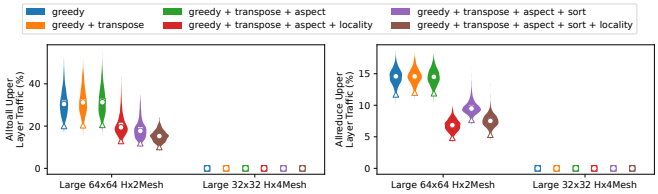


Fig. 9: Fraction of traffic crossing upper fat tree levels.

We observe that in both cases the percentage of traffic crossing the upper level is lower than 50%, thus justifying a 2:1 tapering for these types of workloads. For the large 32x32 Hx4Mesh all the boards on the same column (or on the same row) can be connected through a single 64-ports switch, thus not requiring a fat tree. Moreover, when the algorithm tries to allocate jobs to improve the communication locality, the traffic on the upper levels drops down to less than 25% for Hx4Mesh, meaning that high tapering would not reduce performance. We see in Figure 8 that the locality allocation heuristic does not reduce overall system utilization.

Last, we investigate how random failures reduce the system utilization (due to fragmentation). Figure 10 repeats our allocation experiment but now with a varying number of randomly failed boards. We report the system utilization as the number of non-faulted boards allocated to jobs.

Fig. 10: HxMesh utilization for different numbers of failures.

The left shows the small clusters (256-board Hx2Mesh and 64-board Hx4Mesh) and the right shows the large clusters (4,096-board Hx2Mesh and 1,024-board Hx4Mesh). In almost all the cases, our allocation algorithm achieves a median utilization of working boards (white dot inside the violin) higher than 70%. In particular, this also holds with 40 failed boards, 62% of the small Hx4Mesh. We also observe that meshes with fewer boards are more affected by random failures. Last, we can see that allocating the jobs in their random arrival order (rather than ordered by their size) decreases the utilization at most by 10% on large networks.

C. Routing on HammingMesh

On HxMesh, packets are routed adaptively along all shortest paths. We assume packet-level adaptive routing as implemented in High-Performance Interconnects such as Slingshot [42] or InfiniBand [43]. To minimize packet reordering, flowlet-level adaptive routing can be used as an approximation on Ethernet networks [44]. Without loss of generality, we assume input buffered switches and credit-based flow control. We describe first how packets are routed when both source and destination accelerators are on the same board, and then how to route packets when source and destination are on two different boards.

1) Routing on the same board: If both source and destination are on the same board, we use adaptive routing on the torus network: The algorithm chooses the least loaded output port at each accelerator (4x4 switch) along all shortest paths between the two endpoints. Packets may be routed through fat tree switches (similar to what happens in a 2D torus) or only through on-board links.

2) Routing between different boards: If the source and destination boards are on the same row, they are adaptively routed in the source board to the closest edge (*west* or *east*), and then on the fat tree to the destination board's port closest to the destination. On the fat tree, packets are forwarded using up/down adaptive routing [45]. Once in the destination board, the packet is adaptively routed to the destination. It is worth noting that the packet might also need to be forwarded *south* or *north* within the source or destination board to reach the destination. Whether this happens in the source or destination board (or in both) depends on the local load. When the source and destination boards are on the same HxMesh column a similar process is applied.

If source and destination boards are on different rows and columns, packets must be forwarded through an intermediate board. This board needs to be in the same row of the source board and in the same column of the destination board (or vice-versa). The path selection is adaptive and minimal and packets cross two fat trees, one in each dimension.

3) Deadlock freedom: To guarantee deadlock-freedom within the board, packets are forwarded using north-last routing [46]. Thus, the *north* direction can only be taken by switches on the same column of the destination board. When forwarding packets in the tree, up/down routing guarantees deadlock-freedom. Although both the mesh and the fat tree are deadlock-free, combining the two may result in deadlocks.

A simple solution to guarantee deadlock freedom uses multiple virtual channels by increasing the virtual channel at each hop. However, this requires a number of virtual channels equal to the network diameter. Instead we observe that, because both the board and the fat tree guarantee deadlock freedom, it is enough to increase the virtual channel when jumping from one board to another (e.g., when a board injects a packet in the fat tree). Because each packet crosses at most two fat trees, this requires at most three virtual channels.

V. Results and Comparison

We now evaluate HxMesh topology options in comparison with all topologies listed in Table II. We use the Structural Simulation Toolkit (SST [47]), a packet-level network simulator, which has been validated against the Cray Slingshot interconnect [42]. SST enables us to run (slightly modified) full MPI applications directly in the simulation environment where *they react to dynamic network changes (e.g., congestion)*. In total, we ran simulations of more than 120 billion packets using more than 0.6 million core hours with parallel simulations. The detailed configuration is described in Appendix F. We select various representative microbenchmarks and scenarios for deep learning jobs and *publish the full simulation infrastructure such that readers can simulate their own job setup*.

A. Microbenchmarks

We start by analyzing well-known microbenchmark traffic patterns to assess and compare achievable peak bandwidth.

1) Global traffic patterns: We first investigate global traffic patterns such as alltoall and random permutations as global-traffic workloads. We note that HammingMesh is not optimized for those patterns as they are rare on deep learning traffic.

a) Alltoall: Alltoall sends messages from each process to all other processes. In our implementation, each of the p processes performs $p-1$ iterations. In each iteration i, process j sends to process $j+i \mod p$ in a balanced shift pattern.

Table II shows the results for 1 MiB messages while Figure 11 shows the global bandwidth at different message sizes. Small Hx2 and Hx4Meshes achieve bandwidths around the cut width of 1/4 and 1/8, respectively (cf. Section III-A). This is because not all global traffic crosses the bisection cuts, especially for smaller clusters. The large cluster configuration performs closer to those bounds and loses some bandwidth due to adaptive routing overheads. Despite its lower bandwidth, even large HxMeshes remain competitive in terms of cost-per global bandwidth and some are even more cost effective on global bandwidth than fat trees.

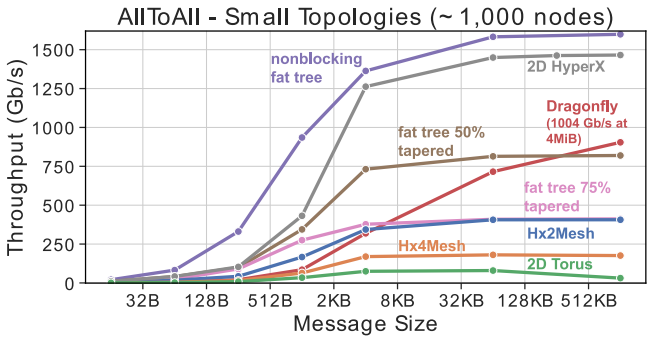


Fig. 11: Alltoall on the small topologies.

b) Random permutation: In permutation traffic, each accelerator selects a unique random peer to send to and receive from. Here, the achieved bandwidth also depends on the location of both peers. Figure 12 shows the distributions

of receive bandwidths across all of the 1k accelerators in the small cluster configurations.

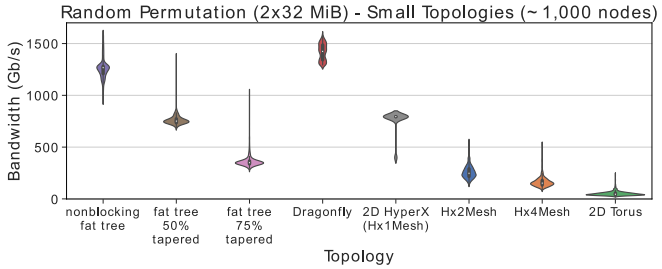


Fig. 12: Bandwidth distribution per accelerator.

The figure also shows the average bandwidth as well as the the cost per average bandwidth relative to a nonblocking fat tree in the top part. Our results indicate that all topologies have significant variance across different connections, which makes job placement and locality significant. HxMeshes are among the most cost effective topologies.

2) Reduction traffic patterns: We first describe our implementation of allreduce, enabled by the associativity of addition. We distinguish three fundamental algorithm types: trees, pipelines, and near-optimal full-global bandwidth algorithms.

a) Simple trees: For small data, simple binary or binomial tree reductions are the best choice. They perform a reduction of S bytes on p processors in time $T \approx \log_2(p)\alpha + \log_2(p)S\beta$[4]. This algorithm sends each data item a logarithmic number of times. It is thus inefficient for the large data sizes in deep learning training workloads and we do not consider trees in this work.

b) Pipelined rings: With a single network interface, large data volumes can be reduced in a simple pipelined ring. Here, the data at each process is split into p segments. The operation proceeds in two epochs and $p-1$ rounds per epoch. In the first reduction epoch, each process i sends segment i to process $i+1 \mod p$ and receives a segment from process $i-1 \mod p$. The received segment is added to the local data and sent on to process $i+1 \mod p$ in the next round. After $p-1$ such rounds, each process has the full sum of one segment. The second epoch is simply sending the summed segments along the pipeline. The overall time $T_p \approx 2p\alpha + 2S\beta$ is bandwidth optimal because each process only sends and receives each segment twice [48].

We propose *bidirectional pipelined rings* to utilize two network interfaces by splitting the data size in half and sending each half along a different direction. The latency stays unchanged because each segment travels twice through the whole ring but the data is half in each direction, leading to a runtime of $T_{bp} \approx 2p\alpha + S\beta$. Here and in the following, β is the time per Byte of each interface, i.e., a system with k network interfaces can inject k/β Bytes per second.

We now extend this idea to four network interfaces per HxMesh plane: we use two bidirectional rings, each reducing a quarter of the data across *all* accelerators. The two rings are mapped to two disjoint Hamiltonian cycles covering all accelerators of the HxMesh [49]. The overall time for this scheme is $T_{rings} \approx 2p\alpha + \frac{S}{2}\beta$.

c) Two-dimensional torus: Pipelined rings are bandwidth-optimal if they can be mapped to Hamiltonian cycles on the topology. However, we find that for large HxMeshes and moderate message sizes, the latency component can become a bottleneck. We thus define another algorithm that uses a 2D toroidal communication pattern with $\sqrt{p}$ latency and good bandwidth usage: each process executes first a reduce-scatter with the other processes on the same row (cost $\sqrt{p}\alpha + \frac{S}{2}\beta$) [50]. Then each process runs an allreduce with the other processes on the same column, on the previously reduced chunk of size $\frac{S}{\sqrt{p}}$ (cost $2(\sqrt{p}\alpha + \frac{S}{2\sqrt{p}}\beta)$) and, eventually, an allgather with the other processes on the same row (cost $\sqrt{p}\alpha + \frac{S}{2}\beta$). To use all four network interfaces at the same time, two of these allreduce can be executed in parallel, each on half of the data each (one reduce consider a transposed network). Thus, the overall time for this scheme is $T \approx 2 \cdot 2\sqrt{p}\alpha + S\beta(\frac{1+2\sqrt{p}}{4\sqrt{p}})$.

d) Summary: The pipeline ring and 2D torus algorithms have sparse communication patterns: each process only communicates with two or four direct neighbors that can be mapped perfectly to HxMesh. Broadcast and other collectives can be implemented similarly (e.g., as the second part of our allreduce) and follow similar tradeoffs. Furthermore, each dimension of a logical job topology is typically small as the total number of accelerators is the product of all dimensions. For example, even for a very large system with 32,768 accelerators, each of the dimensions could only be of size 32 if we decompose the problem along all dimensions. This means that the largest allreduce or broadcast would only be on 32 processes where ring algorithms would perform efficiently.

e) Full system allreduce job: This experiment shows a single job using the last two allreduce algorithms on various topologies. In Dragonfly and fat tree, each accelerator connects with a single NIC to each of the four planes and we use the standard "ring" algorithm. For the single allreduce on the large HxMesh clusters, we use both the two bidirectional rings ("rings") as well as the two-dimensional torus ("torus") algorithm. Figure 13 shows the achieved bandwidths.

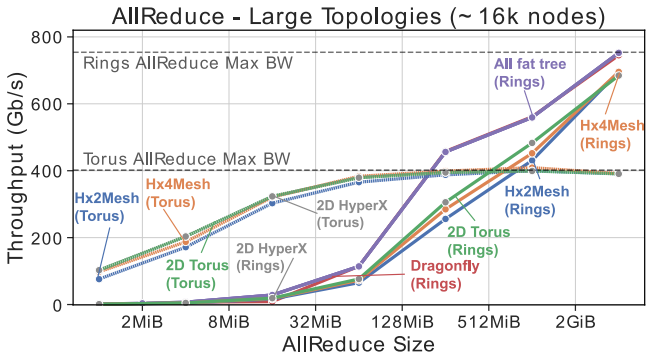


Fig. 13: Global allreduce using different algorithms.

We see that all topologies deliver nearly full bandwidth

[4]with $\approx$, we omit additive constants and minor lower-order terms for clarity

for the ring algorithms. For large messages, HxMesh is 2.8x to 14.5x cheaper per bandwidth than a nonblocking fat tree (Table II). The torus algorithm, which is 2x less bandwidth-efficient, achieves higher throughput at smaller message sizes. This illustrates that multi-algorithms should be used to optimize performance, similar to established practice in MPI [51].

B. DNN Workloads

We now proceed to define accurate communication patterns including computation times for real DNN models. For this, we choose three large representative models: ResNet-152, CosmoFlow, and Transformers (GPT-3) trained in FP32. We use NVIDIA's A100 GPU to benchmark runtimes of operators and we model communication times based on the data volumes.

1) Communication traffic characterization: All example models are constructed of a sequence of identical layers containing multiple operators. Each parallel dimension carries a different volume, depending on the details of the model, training hyperparameters, and the other dimensions. We assume the most general case where the network can utilize all three forms of parallelism running on $D \times P \times O$ accelerators.

a) Data dimension: If we only have data parallelism ($O = P = 1$), then each process needs to reduce all gradients. If we distribute the model between O or P dimension processes, then the total allreduce size is $V_D = \frac{WN_{\mathcal{P}}}{OP}$. The reduction happens once at the end of each iteration after processing a full minibatch and draining the pipeline. It can be overlapped per layer using nonblocking allreduce [52]–[54].

b) Pipeline dimension: If we only have pipeline parallelism ($D = O = 1$) and $N_{\mathcal{A}}$ output activations at the "cut" layer then each process sends all $\frac{M}{P}N_{\mathcal{A}}$ output values to the next process in the forward pass and the same volume of errors during the backward pass. If the layer and its inputs and outputs are distributed to O PEs, then the total send volume in this dimension is $V_P = \frac{MWN_{\mathcal{A}}}{DPO}$. This communication can be hidden at each accelerator as shown in Figure 14 by overlapping nonblocking send/receive operations (bottom, blue) with operator computation (top, green).

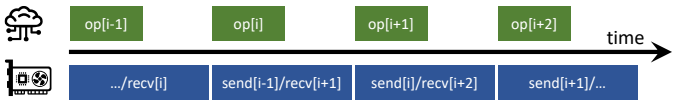


Fig. 14: Overlap in pipelined-parallel execution

c) Operator dimension: For operator parallelism, each process' send volume depends only on the operator parallelization itself and is not influenced by either D or P. The operator can be seen as the "innermost loop" in this sense. Each operator distribution scheme will have its own characteristics that we capture by $V_O = WN_{\mathcal{O}}$. The operator communication volume during each forward and backward pass is a function of the local minibatch size M/DP per process.

2) ResNets: ResNets achieve state of the art in many vision tasks. The standard ResNet on the ImageNet dataset has relatively small operators acting on 224×224×3 input data that does not warrant parallelization. Thus, we use only data parallelism (P=1, O=1). We simulate three sizes $D = \{256, 512, 1024\}$ with most-square jobs (e.g., 16×16) with a minibatch size of $M = 32,768$ [55] and M/D examples at each accelerator. To reduce the latency overhead and overlap communication with computation, we divide the gradients for the 60.2M parameters of our ResNet-152 into 10 equal-sized groups. Once a group is ready, we use a nonblocking *allreduce* to perform the reduction asynchronously.

The compute time in one training iteration of ResNet-152 is 108 ms on 1,024 A100s. The communication can be overlapped nearly completely on all topologies. On HxMeshes and torus, the complete iteration (including overlapped communication) finishes in 110.1 ms and on the other topologies in 109.7 ms, i.e., less than 2.5% communication overhead in the worst case. Other sizes had even less communication overhead.

Thus, the effective network cost savings of HxMesh compared to other topology options can be calculated easily from Table II. For example, an Hx4Mesh is more than 4.1x less expensive than an 75% tapered fat tree and more than 7.8x less expensive than a nonblocking fat tree.

3) CosmoFlow: CosmoFlow [56] is a convolutional network with large input data, one input sample of CosmoFlow is of size 128×128×128×4. Thus, we model a hybrid of operator and data parallelism for CosmoFlow. We execute with $D = 256$, $P = 1$, $O = 4$ on 1,024 accelerators. We assume a minibatch size of $M = 8,192$ [56] with a local batch size of 32. CosmoFlow mainly consists of convolutional layers and fully-connected layers, with $8.9M$ trainable parameters total. In the forward pass, each convolutional layer has to exchange a halo region of the input data with its neighbors using send/recv communication; each fully-connected layer uses allgather to collect the input data. The backwards pass is similar using send/recv, reducescatter, and allreduce communications.

CosmoFlow is more complex with two levels of parallelism and different communication operations. The compute time of CosmoFlow is 44.3 ms on A100. Nearly all communication time can be overlapped leading to less than 2% overhead on all topologies but Hx4Mesh and torus (3.4% and 4.4%).

4) DLRM: DLRM [57]–[59] uses a combination of model parallelism and data parallelism for its embedding and MLP layers, respectively. Two alltoall operations aggregate sparse embedding lookups in the forward pass, and their corresponding gradients in backward pass. Allreduce is required to synchronize the gradients of the data-parallel MLP layers. The parallelism of DLRM is limited by both the mini-batch size and the embedding dimension. DLRM is trained with up to 128 GPU nodes [57], [60]. The total runtimes on the fat tree variants are 2.96 ms, 2.97 ms, and 2.99 ms, respectively. On torus, the code executes for 3.12 ms. HyperX is at 2.94 ms. Hx2Mesh and Hx4Mesh are at 2.97 ms and 3.00 ms, respectively. On A100, DLRM computes around 95 us, 209 us, and 796 us for the embedding, feature interaction, and MLP layers respectively, and communicates 1 MB per alltoall and 2.96 MB per allreduce. All simulation results are shown in Figure 15.

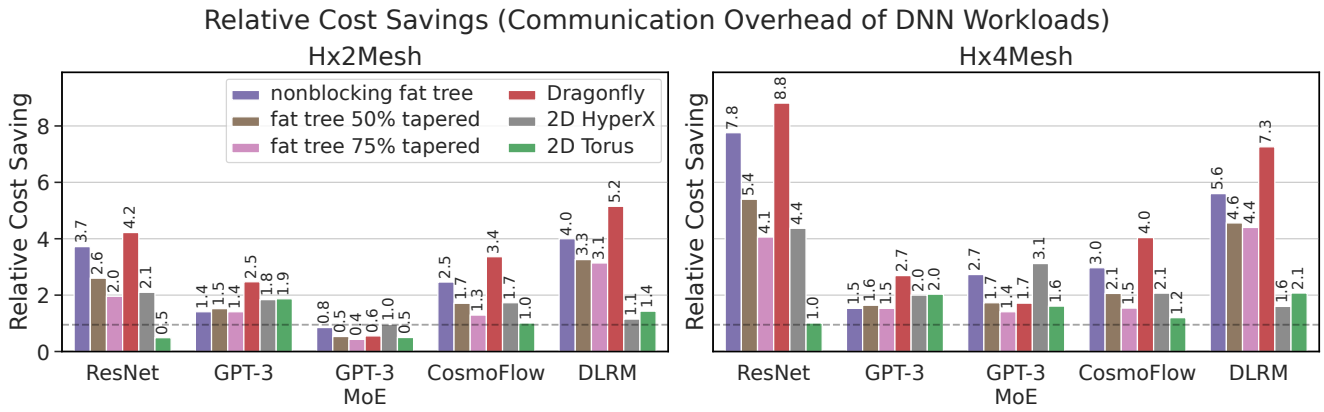


Fig. 15: HxMesh cost savings relative to other topologies.

5) Transformers: Transformers are the most communication intensive [4]. A transformer block consists of multi-head attention (MHA) and two feed-forward (FF) layers. The MHA and FF input/outputs are of size (embedding dimension×batch×sequence length). For example, GPT-3's [61] feed forward layers multiply 49,152×12,288 with 12,288×2,048 matrices per example in each layer.

GPT-3 has a total of 96 layers and each layer has activations of size $N_{\mathcal{A}} = 4 \cdot 2,048 \times 12,288 \approx 100$MB per example as input and output. We choose $P = 96$, such that each pipeline stage processes one layer, and no data parallelism ($D = 1$). For operator parallelism, we use $O = 4$ and the scheme outlined by Megatron-LM [62], which performs one allreduce for FF and one for MHA in both the forward and backward passes.

All operations are the same size as the layer input/output. Thus, the volume for both pipeline communication and operator-dimension allreduce is $N_{\mathcal{A}}$ per example for forward and backward passes. One iteration of GPT-3 computes for 31.8 ms. The total runtimes on the three fat tree variants are 34.8 ms, 36.4 ms, and 37.5 ms, respectively. On torus, the code executes for 72.2 ms per iteration. HyperX is at 40.9 ms. Hx2 and Hx4Mesh are at 41.7 ms and 49.9 ms, respectively.

For GPT-3 with Mixture-of-Experts (MoEs) [63], we use 16 experts. In GPT-3, the FFs have $1.8B$ parameters. Therefore, each expert has $1.8B/16 \approx 113M$ parameters. MoEs perform two alltoalls for FF in both the forward and backward passes, and all operations are the same size as the input/output. The computation time on an A100 is 49.9 ms. The total runtime on the fat trees varies from 52.2 ms to 52.9 ms depending on tapering. On torus, the code executes for 73.8 ms per iteration. HyperX takes 53.9 ms while Hx2 and Hx4Mesh are at 58.3 ms and 63.3 ms, respectively.

Figure 15 shows the relative cost savings of HxMesh compared to other topologies. These are calculated as the ratio of the network costs in Section II times the inverse of the ratio of communication overheads presented in this section.

We conclude that both Hx2 and Hx4Mesh significantly reduce network costs for DNN workloads. While some torus network configurations can be cheaper than Hx2Mesh, they provide significantly less allocation and management flexibility, especially in the presence of failures. Moreover, we also conclude that even in the presence of alltoall communications patterns in GPT-3 MoE and DLRM HxMesh topologies still offer a significant cost advantage compared to traditional topologies. As the scale of the network increases, Hx4Mesh becomes significantly more cost efficient than Hx2Mesh especially in the presence of alltoall traffic.

VI. Related Work

Dragonfly [36] and PERCS [64] are global bandwidth topologies designed for lowest cost given electrical and optical cables and highest global bandwidth. Slim Fly [34] and other diameter-2 topologies [37] provide the maximum number of nodes at diameter two (number of switch hops) and full global bandwidth. While they minimize the number of switches and cables, their structure does not take advantage of technology parameters such as PCB traces or DAC vs. AoC cables.

Other closely related two-dimensional topologies are Flattened Butterfly [65] and 2D Hyper-X networks [66]. Kim et al. [36] show that Dragonfly topologies are 20% more cost effective than Flattened Butterflies and HxMesh improves over both. 2D Hyper-X topologies are isomorphic to 2D Hamming graphs, i.e., Hx1Meshes with a 1x1 board layout. The Cube Collective topology [67] uses on-board links in combination with a fully-connected global topology similar to a Dragonfly.

Large-scale 3D torus networks have been used in supercomputers [68], [69]. Google's TPUs also utilize torui for cost-efficiency [70]. Higher-dimensional torus networks provide higher global bandwidth [71]. Yet, torus networks remain relatively inflexible for allocating jobs and dealing with failures.

A specific co-design between the network architecture and the allreduce algorithm has been proposed [72] but it lacks the cost-effectiveness of HammingMesh, especially on a more general scenario involving also alltoall operations. Moreover it is relatively more complex and it also lacks a robust job failures analysis.

BML [73] uses the BCube topology to take advantage of multiple NICs per server and implement only the data-parallel dimension (allreduce). It achieves about twice the performance for a ring algorithm compared to a parameter server on a nonblocking fat tree and it can be built with 40% of the switches of the fat tree. We show that an Hx4Mesh is more cost efficient by achieving the optimal ring bandwidth at 10.8% (1/9.3x) of the cost. We also show how model (pipeline and

operator) parallelism can be mapped efficiently for complex parallel models.

HammingMesh explicitly combines on-board inexpensive PCB-based mesh topologies with global AoC and DAC cabled topologies. This general idea opens many avenues for future designs such as combining different board and global topologies to accommodate technology trends such as larger packages with chiplets or co-packaged optics.

VII. Discussion and extensions

The design of HammingMesh is based on two fundamental insights: (1) deep learning workloads do not require full global bandwidth and exhibit torus-like communication patterns and (2) combining local low-cost meshes with a global full-bandwidth topology unlocks a design space with intriguing tradeoffs between cost, flexibility, and local/global bandwidth. We show examples from deep learning but we expect that other similar workloads such as tensor contractions, relevant in quantum system simulation [74] or (multi)linear algebra [75] can also be supported efficiently.

We could only show a small part of the design space, combining local 2D meshes with fat trees. Straight-forward extensions are a mix of different local torus topologies and other global layouts (e.g., 1D HxMeshes). Furthermore, one could consider different global-bandwidth topologies, such as Dragonfly or Slim Fly to connect the rows and columns. Such topologies enable connecting more nodes cheaply and could even span multiple (potentially all) rows and/or columns.

Much of HxMesh's cost benefit comes from cheaper global topologies. One could change the arrangement of endpoints in racks to enable more aggressive use of cheaper copper cables.

Several works shown the performance advantages of offloading the allreduce operation to switches. The idea behind these works is to build a tree where the leaves are the compute nodes, and the intermediate nodes are a subset of switches in the network. Data packets get reduced by the switch and, once they reach the root of the tree, the reduced data is multicasted back to the nodes down the tree. This is orthogonal to the topology design and similar solutions can be applied to HammingMesh topologies, since it is enough to build such a reduction tree. Switches in the fat tree can use SHARP [76], [77] rely on programmable switches [17], [78], whereas a similar technology can also be adopted within the boards, as recently shown for systems similar to DGX-2 [79].

Existing state distribution strategies such as ZeRO [80] or parallelizing frameworks such as FlexFlow [81] can take advantage of HammingMesh topologies and achieve highest performance when the traffic is mapped to rings or the collective primitives discussed above.

VIII. Conclusions

HammingMesh is optimized specifically for machine learning workloads and their communication patterns. It relies on the observation that deep learning training uses three-dimensional communication patterns and rarely needs global bandwidth. It supports extreme local bandwidth while controlling the cost of global bandwidth. It banks on an inexpensive local PCB-mesh interconnect together with a workload-optimized global connectivity forming virtual torus networks at adjustable global bandwidth.

Due to the lower number of switches and external cables, it can be nearly always more cost effective than torus networks while also offering higher global bandwidth and significantly higher flexibility in job allocation and dealing with failures.

All-in-all, we believe that HammingMesh will drive future deep learning systems and will also support adjacent workloads, such as (multi)linear algebra, quantum simulation, or parallel solvers, that have Cartesian communication patterns.

Acknowledgment

We thank Microsoft for hosting TH's sabbatical where much of the idea was developed [82], [83]. We thank the whole Azure Hardware Architecture team and especially Doug Burger for their continued support and deep technical discussions. We thank the Swiss National Supercomputing Center (CSCS) for the compute resources on Piz Daint and the Slim Fly cluster (thanks to Hussein Harake) to run the simulations. This project has received funding from the European Research Council under grant agreement PSAP, No. 101002047. Daniele De Sensi is supported by an ETH Postdoctoral Fellowship (19-2 FEL-50).

References

[1] A. Karpathy, "Software 2.0," November 2017, [Online; posted 11-Nov-2017]. [Online]. Available: https://karpathy.medium.com/software-2-0-a64152b37c35

[2] D. H. Dario Amodei, "Ai and compute," online https://openai.com/blog/ai-and-compute/, 05 2018.

[3] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[4] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data Movement Is All You Need: A Case Study on Optimizing Transformers," in *Proceedings of Machine Learning and Systems 3 (MLSys 2021)*, Apr. 2021.

[5] NVIDIA Corporation, "NVIDIA Tesla V100 GPU Architecture," Tech. Rep. WP-08608-001_v1.1, 08 2017.

[6] Xilinx Corporation, "Xilinx AI Engines and Their Applications," Tech. Rep. WP506 (v1.1) July 10, 2020, 07 2020.

[7] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner, A. Forin, H. Zhu, T. Na, P. Patel, S. Che, L. Chand Koppaka, X. SONG, S. Som, K. Das, S. T, S. Reinhardt, S. Lanka, E. Chung, and D. Burger, "Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 10 271–10 281.

[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[9] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, sep 2020. [Online]. Available: https://doi.org/10.1109%2Fhpec43674.2020.9286149

[10] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks," *Journal of Machine Learning Research*, vol. 22, no. 241, pp. 1–124, Sep. 2021.

[11] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the "new normal"' for computer architecture," *Computing in Science Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[12] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. P. Jouppi, and D. A. Patterson, "Google's Training Chips Revealed: TPUv2 and TPUv3." 08 2020, Hot Chips Symposium, pp. 1-70. 2020.

[13] P. DeSantis, "Keynote at AWS re:Invent 2021," online https://www.youtube.com/watch?v=9NEQbFLtDmg&t=4105s, 12 2021.

[14] NVIDIA Corporation, "NVIDIA DGX A100 System Architecture," Tech. Rep. WP-10083-001_v01, 07 2020.

[15] ——, "NVIDIA H100 Tensor Core GPU Architecture," Tech. Rep. V1.01, 03 2022.

[16] G. Venkataramanan, "Talk at tesla ai day," online https://www.youtube.com/watch?v=j0z4FweCy4M&t=6775s, 08 2021.

[17] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik, "Scaling Distributed Machine Learning with In-Network Aggregation," in *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, Apr 2021.

[18] C. Renggli, D. Alistarh, M. Aghagolzadeh, and T. Hoefler, "SparCML: High-Performance Sparse Communication for Machine Learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, Nov. 2019.

[19] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 65:1–65:43, Aug. 2019.

[20] D. Alistarh, T. Hoefler, M. Johansson, S. Khirirat, N. Konstantinov, and C. Renggli, "The Convergence of Sparsified Gradient Methods," in *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., Dec. 2018.

[21] A. Dieuleveut and K. K. Patel, "Communication trade-offs for local-sgd with large step size," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[22] S. U. Stich, "Local sgd converges fast and communicates little," 2019.

[23] E. Gorbunov, F. Hanzely, and P. Richtárik, "Local sgd: Unified theory and new efficient methods," 2020.

[24] A. Khaled, K. Mishchenko, and P. Richtárik, "Tighter theory for local sgd on identical and heterogeneous data," 2020.

[25] N. Dryden, S. A. Jacobs, T. Moon, and B. Van Essen, "Communication quantization for data-parallel training of deep neural networks," in *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, ser. MLHPC '16. IEEE Press, 2016, p. 18.

[26] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018.

[27] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli, "The convergence of sparsified gradient methods," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018.

[28] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," in *International Conference on Learning Representations Workshop Track*, 2016. [Online]. Available: https://arxiv.org/abs/1604.00981

[29] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," 2019.

[30] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," 2018.

[31] S. Li and T. Hoefler, "Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC21)*. ACM, Nov. 2021.

[32] B. Yang, J. Zhang, J. Li, C. Re, C. Aberger, and C. De Sa, "Pipemare: Asynchronous pipeline parallel dnn training," in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 269–296.

[33] B. Prisacari, G. Rodriguez, P. Heidelberger, D. Chen, C. Minkenberg, and T. Hoefler, "Efficient Task Placement and Routing in Dragonfly Networks ," in *Proceedings of the 23rd ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*. ACM, Jun. 2014.

[34] M. Besta and T. Hoefler, "Slim Fly: A Cost Effective Low-Diameter Network Topology," Nov. 2014, proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC14).

[35] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler, "Bandwidth-optimal All-to-all Exchanges in Fat Tree Networks," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ACM, Jun. 2013, pp. 139–148.

[36] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *2008 International Symposium on Computer Architecture*, 2008, pp. 77–88.

[37] G. Kathareios, C. Minkenberg, B. Prisacari, G. Rodriguez, and T. Hoefler, "Cost-Effective Diameter-Two Topologies: Analysis and Evaluation." ACM, Nov. 2015, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).

[38] F. T. Leighton, *Introduction to parallel algorithms and architectures: Arrays, trees, hypercubes*. Elsevier, 1991.

[39] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990.

[40] "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/weng

[41] Alibaba, "Alibaba cluster trace program," 2020, [Online; accessed 04-Mar-2022]. [Online]. Available: \url{https://github.com/alibaba/clusterdata/blob/master/cluster-trace-gpu-v2020/README.md}

[42] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An In-Depth Analysis of the Slingshot Interconnect," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC20)*, Nov. 2020.

[43] NVIDIA Corporation, "NVIDIA InfiniBand Adaptive Routing Technology," Tech. Rep. WP-10326-001_v01, 07 2020.

[44] S. Sinha, S. Kandula, and D. Katabi, "Harnessing TCPs Burstiness using Flowlet Switching," in *3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, San Diego, CA, November 2004.

[45] T. L. Rodeheffer, C. Thacker, A. Birrell, T. Rodeheffer, H. Murray, M. Schroeder, E. Satterthwaite, R. Needham, M. Burrows, M. D. Schroeder, and M. Schroeder, "Autonet: A high-speed, self-configuring local area network using point-to-point links," *IEEE Journal on Select Areas of Communication*, vol. 9, October 1991.

[46] C. Glass and L. Ni, "The turn model for adaptive routing," in *Proceedings the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 278–287.

[47] H. Adalsteinsson, S. Cranford, D. A. Evensky, J. P. Kenny, J. Mayo, A. Pinar, and C. L. Janssen, "A simulator for large-scale parallel computer architectures," *Int. J. Distrib. Syst. Technol.*, vol. 1, no. 2, p. 5773, apr 2010. [Online]. Available: https://doi.org/10.4018/jdst.2010040104

[48] M. Barnett, R. Littlefield, D. Payne, and R. Vandegeijn, "Global combine algorithms for 2-d meshes with wormhole routing," *J. Parallel Distrib. Comput.*, vol. 24, no. 2, p. 191201, feb 1995. [Online]. Available: https://doi.org/10.1006/jpdc.1995.1018

[49] M. M. Bae, B. F. AlBdaiwi, and B. Bose, "Edge-disjoint hamiltonian cycles in two-dimensional torus," *Int. J. Math. Math. Sci.*, vol. 2004, no. 25, pp. 1299–1308, 2004. [Online]. Available: https://doi.org/10.1155/S0161171204307325

[50] M. Cho, U. Finkler, M. Serrano, D. Kung, and H. Hunter, "Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 1:1–1:11, 2019.

[51] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, p. 4966, feb 2005. [Online]. Available: https://doi.org/10.1177/1094342005051521

[52] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI," in *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[53] S. Rashidi, M. Denton, S. Sridharan, S. Srinivasan, A. Suresh, J. Nie, and T. Krishna, *Enabling Compute-Communication Overlap in Distributed Deep Learning Training Platforms*. IEEE Press, 2021, p. 540–553. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00049

[54] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen, "Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems," in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, 2018, pp. 1–13.

[55] Y. You, I. Gitman, and B. Ginsburg, "Large batch training of convolutional networks," 2017.

[56] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Karna, D. Moise, S. J. Pennycook, K. Maschoff, J. Sewall, N. Kumar, S. Ho, M. Ringenburg, Prabhat, and V. Lee, "Cosmoflow: Using deep learning to learn the universe at scale," 2018.

[57] J. A. Yang, J. Park, S. Sridharan, and P. T. P. Tang, "Training deep learning recommendation model with quantized collective communications," in *Conference on Knowledge Discovery and Data Mining (KDD)*, 2020.

[58] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[59] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia *et al.*, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 488–501.

[60] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park *et al.*, "Software-hardware co-design for fast and scalable training of deep learning recommendation models," *arXiv preprint arXiv:2104.05158*, 2021.

[61] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

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

[63] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," 2020. [Online]. Available: https://arxiv.org/abs/2006.16668

[64] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, "The PERCS High-Performance Interconnect," in *Proceedings of 18th Symposium on High-Performance Interconnects (Hot Interconnects 2010)*. IEEE, Aug. 2010.

[65] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: A cost-efficient topology for high-radix networks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 126–137. [Online]. Available: https://doi.org/10.1145/1250662.1250679

[66] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "Hyperx: topology, routing, and packaging of efficient large-scale networks," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–11.

[67] K. D. Underwood and E. Borch, "Exploiting communication and packaging locality for cost-effective large scale networks," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 291300. [Online]. Available: https://doi.org/10.1145/2304576.2304616

[68] B. Bode, M. Butler, T. Dunning, T. Hoefler, W. Kramer, W. Gropp, and H. Wen-Mei, *The blue waters super-system for super-science*. CRC Press, Jan. 2013, pp. 339–366.

[69] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates, "An overview of the bluegene/l supercomputer," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002, pp. 60–60.

[70] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, p. 67–78, jun 2020. [Online]. Available: https://doi.org/10.1145/3360307

[71] Yuichiro Ajima, "High-dimensional Interconnect Technology for the K Computer and the Supercomputer Fugaku," Tech. Rep., 06 2019, fujitsu Technical Review.

[72] T. T. Nguyen and M. Wahib, "An allreduce algorithm and network co-design for large-scale training of distributed deep learning," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 396–405.

[73] S. Wang, D. Li, Y. Cheng, J. Geng, Y. Wang, S. Wang, S. Xia, and J. Wu, "A scalable, high-performance, and fault-tolerant network architecture for distributed machine learning," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1752–1764, 2020.

[74] T. Häner and D. S. Steiger, "0.5 petabyte simulation of a 45-qubit quantum circuit," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126947

[75] G. Kwasniewski, M. Kabic, T. Ben-Nun, A. N. Ziogas, J. E. Saethre, A. Gaillard, T. Schneider, M. Besta, A. Kozhevnikov, J. VandeVondele, and T. Hoefler, "On the parallel i/o optimality of linear algebra kernels: Near-optimal matrix factorizations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476167

[76] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi, "Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction," in *Proceedings of COM-HPC 2016: 1st Workshop on Optimization of Communication in HPC Runtime Systems - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis*. Institute of Electrical and Electronics Engineers Inc., jan 2017, pp. 1–10.

[77] R. L. Graham, L. Levi, D. Burredy, G. Bloch, G. Shainer, D. Cho, G. Elias, D. Klein, J. Ladd, O. Maor, A. Marelli, V. Petrov, E. Romlet, Y. Qin, and I. Zemah, "Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)TM Streaming-Aggregation Hardware Design and Evaluation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12151 LNCS. Springer, jun 2020, pp. 41–59. [Online]. Available: https://doi.org/10.1007/978-3-030-50743-5_3

[78] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, and T. Hoefler, "Flare: Flexible in-network allreduce," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21, 2021.

[79] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives," 2020.

[80] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," 2019. [Online]. Available: https://arxiv.org/abs/1910.02054

[81] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," 2018. [Online]. Available: https://arxiv.org/abs/1807.05358

[82] T. Hoefler, M. C. Heddes, and J. R. Belk, "Distributed processing architecture," U.S. Patent US11 076 210B1, Jul., 2021.
[83] T. Hoefler, M. C. Heddes, D. Goel, and J. R. Belk, "Distributed processing architecture," U.S. Patent US20 210 209 460A1, Jul., 2021.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

1 ABSTRACT

Numerous microarchitectural optimizations unlocked tremendous processing power for deep neural networks that in turn fueled the AI revolution. With the exhaustion of such optimizations, the growth of modern AI is now gated by the performance of training systems, especially their data movement. Instead of focusing on single accelerators, we investigate data-movement characteristics of large-scale training at full system scale. Based on our workload analysis, we design HammingMesh, a novel network topology that provides high bandwidth at low cost with high job scheduling flexibility. Specifically, HammingMesh can support full bandwidth and isolation to deep learning training jobs with two dimensions of parallelism. Furthermore, it also supports high global bandwidth for generic traffic. Thus, HammingMesh will power future large-scale deep learning systems with extreme bandwidth requirements.

2 INTRODUCTION

This document contains the basic requirements and instructions needed to reproduce the results of the paper. In the first part we will present the general organization of the project and how it is possible to reproduce the results. In the second part we will go more into details about how to install and use the two containers . The file structure of the project is organized in the following directories:

- `benchmarks/` contains all the scripts and supporting code needed to run the various benchmarks (both locally or on a cluster with sbatch). It also contains the code needed to generate the final plots and results. Each subfolder of the `benchmarks/` folder has a more detailed README based on the chosen benchmark.
- `results/` contains most of the actual results obtained from the benchmarks run. It is also possible to use this folder to generate the final plots and results. Note that few results are not included directly in this repository due to their large size (100+ GB).
- allocation_simulations/ contains the code and files used to simulate and test our job allocation algorithm. It also contains the code to generate the respective plots.
- sstcore-11.1.0/ contains the core functions of the SST Simulator (version 11.1.0). These files have not been changed for the purpose of this paper compared to the original ones.
- sst-elements-library-11.1.0/ contains the elements part of the SST Simulator and has been heavily modified to support our topology, the benchmarks that we used and all the supporting code.
- `sst_test_outputs` contains some test file used by SST to check its correct behaviour.

We provide two methods to run the project: an easy to install Docker container with everything ready to go (`hx_container` artifact) and the source code which can be installed and tuned to run on any machine including clusters supporting sbatch (`hx_source_code` artifact). While the container is easy to use and test, it is not always ideal to be used on a general purpose cluster and for this reason we also provide the source code and the relative instructions about how to install it. Having said that, it is theoretically also possible to run all the benchmarks locally although the completion time will be very long in some cases. In the second part of this document we will explain more in detail how to work with the two containers.

3 GENERAL USAGE

The benchmark folder is used to run all the simulations used in this paper. Inside the `benchmark/` folder we can find a series of topologies. The ones used in the final results of this paper are:

- `smallTopology/`
- `largeTopology/`
- `1536nodes/`

Inside each one of these subfolder there are more subfolders, one for each benchmark. As a general guideline (with few exceptions), each specific subfolder contains two Python files: one to run the benchmark itself and to generate the data and one to parse the data and create the plots in PDF and PNG format. For example, from inside the `benchmarks/smallTopology/DLRM` folder we can run the benchmark by using `python3 launchDLRM.py` and then parse it by using `python3 parseAndPlotDLRM.py`. Having said that, we refer to each individual README for more in-depth details and to the different command line parameters available (can also be shown using the `-help` command for each individual launch script). We also provide the final data used for the paper inside the `results/` folder. It is possible to use it to generate the plots without having to re-run all the simulations. In that case, simply copy one `output/` folder from `results/` to its corresponding `benchmarks/` folder (for example copy `results/DLRMSmall/output/` to `benchmarks/smallTopology/DLRM`. We will have more details inside each individual README for the corresponding benchmark. Here is the list of benchmarks that are used for the paper and their locations:

- `benchmarks/smallTopology/AllReduce`
- `benchmarks/smallTopology/AllToAll`
- `benchmarks/smallTopology/RandomPermutation`
- `benchmarks/smallTopology/Cosmo`
- `benchmarks/smallTopology/DLRM`
- `benchmarks/smallTopology/GPT3`
- `benchmarks/smallTopology/ResNet`
- `benchmarks/largeTopology/GPT3MOE` or `benchmarks/1536nodes/GPT3MOE` for faster execution.
- `benchmarks/largeTopology/AllReduce`
- `benchmarks/largeTopology/AllToAll`

3.1 Plots reproducibility

This section will explain how to generate the plots which are used in the paper. We assume that the data has already been generated using

the instructions above or that we are using the already generated and saved results.

- Figure 8-11 can be generated by first running `./run_all.sh` from inside the `allocation_simulations/` folder and then running `python3 plot_paper.py` from the same folder. The plots will be generated in the `allocation_simulations/plots/` folder.
- Figure 12 can be generated simply by running `python3 parseAndPlotAllToAll.py` from inside the `benchmarks/smallTopology/AllToAll` folder.
- Figure 13 can be generated simply by running `python3 parsePerm.py` from inside the `benchmarks/smallTopology/RandomPermutation` folder.
- Figure 14 can be generated simply by running `python3 parseAndPlotAllReduce.py` from inside the `benchmarks/largeTopology/AllReduce` folder.
- Figure 16 can be generated simply by running `python3 plot_cost_savings_diff_col.py` from inside the `benchmarks/smallTopology/Plot` folder.

Note that when each benchmark is parsed using the corresponding Python script, the standard output (stdout) will display some key information from the run. These values have been used to build Table II from the paper and also in section V.

4 CLUSTER USAGE

For the runs on the cluster we used the following compilers and software running on CentOS Linux 7:

- GCC 4.8.5
- Open MPI 1.10.7
- Python 2.7.5

All simulations are run by default locally if not specified. It is possible to run them on a cluster supporting sbatch by using the `–env=cluster` parameter when launching the various benchmarks. In this case it is likely necessary to modify the `general_bench.py` file (from line 70) to change the various sbatch details such as account name and possibly other limits (such as job duration or memory limits).

5 SIMULATION TIMES

The simulations used in this work can take a long time to complete since we are simulating large networks and billion of packets. We provide below (in Table 1) a short summary of the time needed to fully complete the simulations for each benchmark.

Note that in some benchmarks we also provide an option to run only a fraction of the final data points in order to have quickly access to at least some of the results. The details are reported in the README of each individual benchmark.

As a good starting point to get some initial validation, running DLRM or ResNet should be relatively quick even on a local machine.

Benchmark	Nodes	Runtime (H)	Node Hours
AllToAllSmall	64	38	2432
AllReduceSmall	64	10	640
AllToAllLarge	128	166	21248
AllReduceLarge	128	59	7552
RandomPerm.	32	<1	<32
ResNet	64	<1	<64
GPT3	64	4	256
GPT3-MOE	64	12	768
CosmoFlow	64	15	960
DLRM	16	<1	<16

Table 1: Simulations times for the different benchmarks. Runtime is is hours. Last column shows the total node hours per benchmark.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1
Persistent ID: `https://doi.org/10.5281/zenodo.6462395`
Artifact name: hx_container

Artifact 2
Persistent ID: `https://doi.org/10.5281/zenodo.6461582`
Artifact name: hx_source_code

Reproduction of the artifact with container:

6 INSTALLATION FROM DOCKER FILE

Docker version 20.10.7 was used to generate and test the Dockers images.

Please download the docker container from here or using the DOI link.

Load and run the docker image using the following command. Note this could take some time to run and might have to be run as root.

```
sudo docker load -i hxmesh_container.tar.gz
sudo docker run -it hxmesh:2.0
```

To go into the repository of the code simply

```
cd /hxnet
```

If viewing images such as plots inside the container is problematic, it is possible to transfer that file to the host machine using:

```
docker cp <containerId>:/file/path/within/container /host/path/target
```

It is possible to obtain the `<containerId>` by running

```
sudo docker ps
```

7 INSTALLATION FROM SOURCE CODE

The installation from source code should take about 10-15 minutes on a modern machine.

7.1 Requirements

C++11, python3, python-dev and OpenMPI 4.0.5 are necessary when installing the project from source code as these are the minimum requirements for SST to run. In particular, in our case, we run our code locally with the following compilers and software running on Ubuntu 20.04 LTS:

- gcc (Ubuntu 9.3.0-10ubuntu2) 9.3.0
- Open MPI 4.0.5
- Python 3.8.10

7.2 Installation

Please first download the repository using its relative DOI or directly from GitLab.

Once, the download has been finished, extract the archive and go inside the extracted folder. Once that has been done, it is possible to start installing SST.

To install SST 11.1.0 and OpenMPI 4.0.5 we refer to the official instructions from the SST website. Note that the source files for SST are already part of this repository and it is important to use them as they contains all our modifications compared to the original SST repository (keep this in mind also when setting the location of `SST_CORE_ROOT` and `SST_ELEMENTS_ROOT`) One common issue could be the `make all` of SST Core or SST Elements failing. In that case please try to run `automake` first. Once the installation has been done, it is possible to test its correctness by running:

`sst-test-core` `sst-test-elements -w "*simple*"`

The requirements.txt file should list all Python libraries that are used when parsing the data and generating the plots. These libraries can be installed running

`pip install -r requirements.txt`

Finally it is necessary to add this environmental variable to your `/.bashrc` file or equivalent for the correct execution of the scripts.

`export PYTHONPATH="{`$PYTHONPATH$`} :SST_ELEMENTS_ROOT /src/sst/elements/ember/test/"`

8 COMMON PROBLEMS

Q: The installationf fails when running `configure` or when trying to build it with `make`

A: In that case please make sure to run `automake` first.

Q: When trying to run the Python scrirpts for the benchmark, I get an error about a failed import.

A: Plese make sure that `export PYTHONPATH="$ PYTHONPATH}:$SST_ELEMENTS_ROOT/src/sst/elements/ember/ test/"` has been run. To make it permanent please consider adding this to your `~/.bashrc` file or equivalent.

Q: When parsing the data for the plots, I get an error

A: Please make sure the output files are not corrupted and that the simulations have fully completed.