AFRL-RI-RS-TR-2019-070

MASSIVELY PARALLEL MODELING AND SIMULATION OF NEXT GENERATION HYBRID NEUROMORPHIC SUPERCOMPUTER SYSTEMS

RENSSELAER POLYTECHNIC INSTITUTE

MARCH 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-070 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
RYAN LULEY
Work Unit Manager

/ S /
QING WU
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2019	FINAL TECHNICAL REPORT	MAR 2015 – SEP 2018

4. TITLE AND SUBTITLE

MASSIVELY PARALLEL MODELING AND SIMULATION OF NEXT GENERATION HYBRID NEUROMORPHIC SUPERCOMPUTER SYSTEMS

5a. CONTRACT NUMBER

5b. GRANT NUMBER FA8750-15-2-0078

5c. PROGRAM ELEMENT NUMBER 62788F

6. AUTHOR(S)

Christopher Carothers
Prasanna Date
James Hendler
Mark Plagge
Noah Wolfe

5d. PROJECT NUMBER T2NS

5e. TASK NUMBER TR

5f. WORK UNIT NUMBER PI

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180-3522

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2019-070

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The major theme of research investigated here is how might neuromorphic computing impact future designs of supercomputer systems. This report provides both a summary and detailed experimental research results for the five core research thrusts (CRTs) covered in this research project.

15. SUBJECT TERMS

Neuromorphic computing architectures, artificial neural network models, inference models, level-based computing, spike-based computing, neuromorphic computing algorithms, neuromorphic system emulator, neuromorphic processor simulation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**RYAN LULEY**
U	U	U	UU	164	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Contents

List of Figures iii

List of Tables viii

1 SUMMARY **1**

2 INTRODUCTION **3**
2.1 Parallel Discrete-Event Simulation & ROSS . . . 3
2.2 Neuromorphic Computing, IBM TrueNorth & *NeMo* . . . 4
2.3 Classification of Supercomputer Failures Using TrueNorth: . . . 5
2.4 Durango – A Hybrid System Performance Modeling Framework . . . 6
2.5 HPC Network Models . . . 7
2.6 Classification of AFRL Data Using TrueNorth & *NeMo*: . . . 9

3 METHODS, ASSUMPTIONS AND PROCEDURES **9**
3.1 *NeMo* . . . 9
3.1.1 Background . . . 9
3.1.2 Design and Implementation of *NeMo* . . . 14
3.1.3 Design and Implementation of *NeMo* Super Synaspe (SS) . . . 18
3.2 Classification of Supercomputer Failures Using TrueNorth . . . 21
3.2.1 The RAS Data . . . 21
3.2.2 Using IBM TrueNorth . . . 23
3.3 Durango – A Hybrid System Performance Modeling Framework . . . 25
3.3.1 Aspen Overview . . . 26
3.3.2 Durango Approach . . . 27
3.3.3 Representing Communication in Aspen . . . 27
3.3.4 Synthetic Program Execution . . . 28
3.3.5 *Durango*-Instantiated Executable . . . 29
3.3.6 CODES: An Extreme-Scale Systems Modeling and Simulation Framework . 29
3.3.7 Durango Direct Integration: Aspen with CODES . . . 30
3.4 HPC Network Models . . . 34
3.4.1 CODES Framework . . . 34
3.4.2 HPC Network Models . . . 38
3.5 Classification of AFRL Data Using IBM TrueNorth & *NeMo* . . . 47

4 RESULTS AND DISCUSSION **49**
4.1 *NeMo* Results . . . 49
4.1.1 *NeMo* Validation . . . 49
4.1.2 *NeMo-ES* Performance Results . . . 51
4.1.3 *NeMo-SS* Performance Results . . . 55
4.2 Classification of Supercomputer Failures Using TrueNorth . . . 61
4.2.1 Exploratory Data Analysis (EDA) . . . 61
4.2.2 Classification Results and Comparative Analysis . . . 62

4.2.3 Speed and Power Consumption . . . 64
4.2.4 Discussion . . . 64
4.3 Durango – A Hybrid System Performance Modeling Framework . . . 65
4.3.1 Durango Generated vs. Real LULESH Results . . . 65
4.3.2 Evaluation of Durango Direct Integration . . . 68
4.4 HPC Network Models . . . 73
4.4.1 Model Validation . . . 73
4.4.2 Experimental Setup . . . 82
4.4.3 HPC Network Model Evaluations . . . 95
4.5 Classification of AFRL Data Using IBM TrueNorth & *NeMo* . . . 130

5 CONCLUSIONS **133**
5.1 NeMo: A Massively Parallel Neuromorphic Simulator . . . 133
5.2 Classification of Supercomputer Failures Using TrueNorth . . . 133
5.3 *Durango* – A Hybrid System Performance Modeling Framework . . . 134
5.4 HPC Network Models . . . 134
5.5 Classification of AFRL Data Using TrueNorth & *NeMo* . . . 136

A Publications **151**

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS **152**

List of Figures

1 Leaky Integrate and Fire (LIF) General Neuron Model. 12
2 TrueNorth leaky integrate and fire neuron model (TNLIF) 13
3 The *NeMo* neuron event flow. Details of each block, numbered 1 through 18, are discussed in the text below. 15
4 Example event chain in NeMo with 3 neurons per neurosynaptic core. In this diagram, an event is received at Axon 0 within a core at time t. At $t+0.0001+\varepsilon$ Axon 0 sends a message to Synapse 0,0. Synapse 0,0 then sends a message at $t+0.0002+\varepsilon$ to Neuron 0 and Synapse 0,1. Synapse 0,1 sends messages to Neuron 1 and Synapse 0,2 at $t+0.0003+\varepsilon$. Synapse 0,2 then sends a message to Neuron 2 at $t+0.0004+\varepsilon$. If no messages are received on Axon 1 and 2, no messages are sent. Neurons will send outgoing spike messages, if applicable, at $t+1.0+\varepsilon$. 19
5 Figure (a) shows the *NeMo-SS* core layout with the super synapse virtual synaptic grid. Figure (b) shows a trivial neurosynaptic core with 2 neurons. Input spikes arrive at the start of the current big tick, t_1 with a jitter value added. Jitter, represented by ϵ, is added to every communication to prevent message collision. Some messages have both a counter and jitter applied to the time stamp, represented by ν 20
6 Network configuration of TrueNorth SNN 24
7 Visualization of TrueNorth SNN 25
8 *Durango* overview. 25
9 CODES network simulations. 30
10 Durango hybrid runtime. 32
11 Diagram showing the general execution path of events in the Slim Fly specific parallel discrete-event simulation. 36
12 A full 3-level fat-tree network using eighty 8-port switches results in a total of eight pods with four switches per level per pod and a total of $k^3/4 = 128$ compute nodes. 39
13 Example configuration for a pruned 3-level fat-tree using 32 8-port switches in three pods and spine level yields a total of $3 \cdot (k/2)^2 = 48$ compute nodes (darker colored lines between L1 and L0 indicate a bundle of two links). 40
14 Full Five group dragonfly network. 42
15 General structure and layout of MMS slim fly graphs. Global connections between subgraphs have been generalized for clarity. There are no intergroup connections within the same subgraph. Each router contains one global connection to one router in each of the q-many router groups in the opposing subgraph. 45
16 Example MMS graph with $q = 5$ illustrating the connection of routers within groups and between subgraphs. 46
17 Two Izhikevich Validation Run TNLIF Neuron Parameters and Results. 50
18 Weak scaling performance experiments. 51
19 Strong scaling performance experiments. 52
20 Comparison of the efficiency and the running time of the strong scaling experiments. 53
21 Breakdown of the primary and secondary rollbacks for the strong scaling simulation. Note that the net event population for these experiments is 13 billion events, but the maximum number of rollbacks observed is only 7.6 million. 54

22 Calculating *Compass*'s events per second per Blue Gene/Q Rack 55
23 Neuron activity in the identity matrix benchmark simulation. This chart represents the number of spikes sent by each neruon in a 4096 neuron simulation across 512 ticks, demonstrating the workload generated by the benchmark model run. We observed an even distribution of neuron activity across the simulation running this benchmark. 57
24 Blue Gene/Q weak scaling performance experiments. Figure (a) shows the results of running *NeMo-ES* with 256 neurons per core with 16 cores per rank. Figure (b) shows the results of running *NeMo-SS* with 512 neurons per core with 64 cores per rank. Figure (c) shows the results of *NeMo-SS* with 256 neurons per core with 64 cores per rank. Figure (d) shows the wall clock time taken for each run. 58
25 *NeMo-SS* BG/Q Weak Scaling Time Detail. 59
26 *NeMo-SS* Blue Gene/Q Strong Scaling Experiment Results. 60
27 Histograms of Categorical Variables. 61
28 Plot of Design Index for All Techniques. Accuracy Index is computed as $\log_{10}\left(\frac{e_p}{e_t}\right)$, where e_p is the threshold error and e_t is the training error [52]. Overfitting Index is computed as $\log_{10}\left(\frac{e_v}{e_t}\right)$, where e_v is the validation error and e_t is the training error [52]. 64
29 LULESH vs. Durango: Average torus network packet hop count as a function of the different LULESH phases. 66
30 LULESH vs. Durango: Finished torus network packet count as a function of the different LULESH phases. 67
31 LULESH vs. Durango: Finished torus network packet hop count as a function of the different LULESH phases. 68
32 LULESH vs. Durango: Total torus network bytes sent as a function of the different LULESH phases. 69
33 Durango in direct integration mode with 32K node torus network and Aspen compute node generator for 32 to 2,048 MPI ranks. 72
34 Durango in direct integration mode with 32K node torus network and Aspen compute node generator for 1K to 16K MPI ranks. 73
35 Durango in direct integration mode with 1.3M node dragonfly network and Aspen compute node generator for 1K to 32K MPI ranks. 74
36 Worst-case traffic layout for the slim fly topology. 75
37 Throughput comparison of minimal routing for uniform random (UR) and worst-case (WC) traffic workloads. Figures are best viewed in color. 76
38 Throughput comparison of non-minimal routing for uniform random (UR) and worst-case (WC) traffic workloads. Figures are best viewed in color. 77
39 Throughput comparison of adaptive routing for Uniform Random (UR) and worst-case (WC) traffic workloads. Figures are best viewed in color. 78
40 Router occupancy comparison for simulations as a function of the Router ID and percentage of simulation time across different offered loads for virtual channels (VCs) 0 and 1 using UR traffic and minimal routing with increasing injection load. Figures are best viewed in color. 79

41 Messages sent and received over time for the simulation across different router IDs using UR traffic and minimal routing using 100% load. Figures 41a and 41b show the number of sends and receives sampled over the simulation run time for all the compute nodes. Figures 41c and 41d show the same for all routers in the simulation. 80
42 Distribution of simulation time for the 74K-node Slim Fly model with minimal routing, UR traffic, and 10% load. 81
43 Verification experiments showing observed network performance while increasing the number of rails under a synthetic bisection-pairing workload. Both axes show the relative injection in percent of the link speed (12.5 GB/s). 82
44 Validation results comparing network performance between the physical DRP system and a similarly configured simulation system using the CODES Fat-Tree model. Both simulation and hardware perform the ping-pong benchmark pattern between two MPI processes on separate compute nodes. Subfigure 44a shows the packet end-to-end latencies and Subfigure 44b presents the error between hardware and simulation. . . 83
45 Visualization of the networks utilized in this paper. Green and blue nodes are routers (green being routers in the first plane/rail and blue routers being in the second plane/rail, if one exists). Red nodes are compute nodes. 84
46 Communication pairing diagrams showing the level of communication between all MPI processes in uniform random, and bisection workloads. Sending process IDs are on the left axis and receiving process IDs are on the bottom axis. *Note, the bisection heapmap lines are faint due to resolution limitations of the image coupled with the very high number of MPI ranks used in this graphic.* 88
47 Communication pairing diagrams showing the level of communication between all MPI processes in one, two, and three dimensional nearest neighbor workloads. Sending process IDs are on the left axis and receiving process IDs are on the bottom axis. *Note, the heatmap lines are faint due to resolution limitations of the image coupled with the very high number of MPI ranks used in this graphic.* 89
48 Communication pairing diagrams showing the level of communication between all CPU MPI processes in each CPU application. Sending process IDs are on the left axis and receiving process IDs are on the bottom axis. 90
49 Integration of IBM TrueNorth, NeMo, and CODES ecosystems to form the large-scale neuromorphic systems simulation workflow. The red dashed boxes indicate the various entry points for users to setup a large-scale neuromorphic simulation. 92
50 Communication pairing diagrams showing the level of communication between all neuromorphic chips in each neural network application. Sending chip IDs are on the left axis and receiving chip IDs are on the bottom axis. 94
51 Million-node compute scaling analysis simulating 100 μs using minimal Routing, UR traffic, and 10% network load. Figures are best viewed in color. Added result for sequential execution. 97
52 Million-node memory and time scaling analysis simulating 100 μs using minimal Routing, UR traffic, and 10% network load. The time scaling shows the slowdown factor of simulation compared to real hardware as the number of MPI ranks increases. Figures are best viewed in color. 98

53 Distribution of simulation time for the 74K-node Slim Fly model with minimal routing, UR traffic, and 10% load. Per node time is the average time each node spent in each task while the total time is the average multiplied by the number of MPI ranks in the simulation. Figures are best viewed in color. 100
54 Performance results for the Crystal Router application running with 1K MPI ranks on the 3,042-node Slim Fly network. Black 'X's indicate the mean value across all compute nodes. Figures are best viewed in color. 102
55 Performance results for the Multigrid application running with 10K MPI ranks on the 74K-node Slim Fly network. Black 'X's indicate the mean value across all compute nodes. Figures are best viewed in color. 103
56 Performance results for the Multigrid application running with 110K MPI ranks on the 74K-node Slim Fly network. Figures are best viewed in color. 104
57 Network statistics per compute node comparing AMG, Crystal Router, and Multigrid application traces running in parallel on the 3,564-node dual-rail Fat-Tree system. . 107
58 Network performance comparison of a single rail 100 Gb/s network and a dual rail 56 Gb/s network. 108
59 Observed network bandwidth for one, two, four, and eight rails. 110
60 Multi-job allocation study on the dual-rail Fat-Tree network using contiguous, random, and clustered allocation policies. 111
61 Dual-rail network performance when increasing the number of trace processes mapped to a node from 8 to 48. 113
62 Observed load of the networks in response to increasing offered load (measure as a percentage of link speed) for the uniform random synthetic traffic workload. 115
63 Observed load of the networks in response to increasing offered load (measure as a percentage of link speed) for the bisection synthetic traffic workload. 116
64 End time performance for all network configurations running uniform random synthetic workload with increasing offered load. 117
65 End time performance for all network configurations running bisection synthetic workload with increasing offered load. 118
66 End time performance for all network configurations running 1D, 2D, and 3D nearest neighbor synthetic workloads with increasing offered load. 119
67 Simulated end time results for CPU applications running alone on the Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly HPC systems. End times are normalized within a workload to the slowest performing topology result. Lower is better. 121
68 Aggregate global link traffic for Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly running the single-job AMG execution. 121
69 Simulated end time results for Neuromorphic applications running alone on the Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly HPC systems. End times are normalized within a workload to the slowest performing topology result. Lower is better. 122
70 Aggregate global link traffic for Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly running the single-job MNIST execution. 123

71 The top subfigure presents simulated end time results for the CPU applications when running in the presence of neuromorphic applications on the Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly HPC systems. End times are normalized within each workload pairing to the slowest performing topology result. The bottom subfigure presents the net slowdown in end time performance for CPU workloads when running in the presence of neuromorphic workloads. In both subfigures, lower is better. 125
72 Aggregate global link traffic for Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly running the hybrid AMG-MNIST execution. 125
73 The top subfigure presents simulated end time results for neuromorphic applications when running in the presence of CPU applications on the Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly HPC systems. End times are normalized within each workload pairing to the slowest performing topology result. The bottom subfigures present the net slowdown in end time performance for neuromorphic workloads when running in the presence of CPU workloads. In all subfigures, lower is better. 126
74 Normalized end time results for each topology averaged over each workload category including synthetic, CPU and neuromorphic workloads. Additionally a fourth summary is provided averaging normalized end time across "all" workloads. 128

List of Tables

1 A matrix representation of a neurosynaptic core. 16
2 Description of Variables in a Typical RAS Log Entry. 22
3 Descriptions of symbols used. 44
4 Neuron Validation Parameters. 50
5 Breakdown of time spent during the simulation of 65,536 neurosynaptic cores on 1024 Blue Gene/Q nodes each with 64 MPI ranks. 52
6 Experimental Run Configurations. 56
7 Breakdown of time spent during the simulations on 2048 Blue Gene/Q nodes each with 64 MPI ranks. The columns labeled with "Real-Time" text indicate the *NeMo-SS* runs that use the Real-Time GVT synchronization protocol, in contrast to the event count based GVT method. 60
8 Performance Comparison of All Techniques. 62
9 Design Index Computation. 63
10 Network configurations (Fat-Tree & Dragonfly). 85
11 Network configurations (Slim Fly). 86
12 CPU workload comparison. 90
13 Neuromorphic workload comparison. 94
14 Network cost comparison. 130
15 Speedup per $1M. 130
16 MNIST Data Set: *NeMo* vs. NSCS Spike Counts 131
17 AFRL MSTAR Data Set: *NeMo* vs. NSCS neuromorphic Output Core Spike Counts. 132

1 SUMMARY

The central task of the proposed research here is to explore and analyze how might a neuromorphic accelerator type processor be used to improve the HPC application performance and overall system reliability of future extreme-scale systems. To address this problem, we created a platform prototype system modeling capability that leverages current supercomputer systems available at AFRL and Rensselaer complete with the ROSS massively parallel discrete-event simulator. From this 3 year research program, we completed the following five major core research thrusts (CRTs):

1. **NeMo: A Massively Parallel Neuromorphic Simulator:** Neuromorphic computing is a broad category of nonvon Neumann architectures that mimic biological nervous systems using hardware. Current research shows that this class of computing can execute data classification algorithms using only a tiny fraction of the power conventional CPUs require. This raises the larger research question: How might neuromorphic computing be used to improve application performance, power consumption, and overall system reliability of future supercomputers? To address this question, an open-source neuromorphic processor architecture simulator called NeMo is being developed. This effort will enable the design space exploration of potential heterogeneous compute systems that combine traditional CPUs, GPUs, and neuromorphic hardware. This work examines the design, implementation, and performance of NeMo. Demonstration of NeMo's efficient execution using 2,048 nodes of an IBM Blue Gene/Q system, modeling 8,388,608 neuromorphic processing cores is reported. The peak performance of NeMo is just over ten billion events-per-second when operating at this scale.

2. **Classification of Supercomputer Failures Using TrueNorth:** Today's petascale supercomputers are comprised of ten's of thousands of compute nodes. Failures on these massive machines are a growing problem as the time for a single compute node to fail is shrinking. Ideally, the job scheduler would like the capability to predict node failures ahead of time in order to minimize the impact of node failures on overall job throughput. However, due to the tight power constraints of future systems, the online modeling of real-time error data must be accomplished using as little power as possible. To this end, the IBM TrueNorth Neurosynaptic System is used to create a Spiking Neural Network (SNN) model of supercomputer failure data and the classification accuracy of this model is compared to other Machine Learning (ML) and Deep Learning (DL) techniques. It is observed that the IBM TrueNorth failure classification model yields a training accuracy of 99.41%, validation accuracy of 98.12% and testing accuracy of 99.80% and outperforms other machine learning and deep learning approaches. Moreover, the TrueNorth SNN consumes five orders of magnitude less power than the other ML/DL approaches during the testing phase. Additionally, it is observed that all ML/DL approaches investigated as part of this study are able to produce accurate models of the supercomputer system failure data.

3. ***Durango* – A Hybrid System Performance Modeling Framework:** Performance modeling of extreme-scale applications on accurate representations of potential architectures is critical for designing next generation supercomputing systems because it is impractical to construct prototype systems at scale with new network hardware in order to explore designs

and policies. However, these simulations often rely on static application traces that can be difficult to work with because of their size and lack of flexibility to extend or scale up without rerunning the original application. To address this problem, we have created a new technique for generating scalable, flexible workloads from real applications, we have implemented a prototype, called Durango, that combines a proven analytical performance modeling language, Aspen, with the massively parallel HPC network modeling capabilities of the CODES framework. Our models are compact, parameterized and representative of real applications with computation events. They are not resource intensive to create and are portable across simulator environments. We demonstrate the utility of Durango by simulating the LULESH application in the CODES simulation environment on several topologies and show that Durango is practical to use for simulation without loss of fidelity, as quantified by simulation metrics. During our validation of Durango's generated communication model of LULESH, we found that the original LULESH miniapp code had a latent bug where the `MPI_Waitall` operation was used incorrectly. This finding underscores the potential need for a tool such as Durango, beyond its benefits for flexible workload generation and modeling. Additionally, we demonstrate the efficacy of Durango's direct integration approach, which links Aspen into CODES as part of the running network simulation model. Here, Aspen generates the application-level computation timing events, which in turn drive the start of a network communication phase. Results show that Durango's performance scales well when executing both torus and dragonfly network models on up to 4K Blue Gene/Q nodes using 32K MPI ranks, Durango also avoids the overheads and complexities associated with extreme-scale trace files.

4. **HPC Network Models:** In the first part of this research thrust, we describe a subset of HPC network topologies chosen for evaluation. The networks are chosen because they are either currently used in a deployed HPC system or they posses characteristics such as a low-diameter that make them a promising option as the interconnection network in a next generation supercomputer. We describe the Fat-Tree network and extensions made to represent pruned multi-rail configurations. Additionally we discuss two approaches to Dragonfly networks selected for comparison that leverage all-to-all connections and 2D grid connectivity within router groups. We also cover a recently proposed theoretical network topology called the Slim Fly. The topology layouts, connectivity and routing algorithms, as well as model validation are discussed to provide a clear picture of each networks theoretical capabilities and simulator accuracy. In the second part of this research thrust, we perform numerous evaluations analyzing the scaling performance of the simulation framework as well as the performance of these networks at large-scale in response to various workloads and HPC environment conditions. The back-end discrete-event simulator is analyzed showing the effectiveness of the approach in speeding up the simulation run times by running in parallel. The discrete-event based network models are then used to perform a number of studies to predict and quantify performance of the networks. We test the Slim Fly at large-scale under CPU workloads to observe the effect of routing on end time performance. We study the performance benefits of additional rails in the Fat-Tree network by analyzing rail-scaling, job placements, multi-job execution, and increased computational power per compute node. Finally, we test equally provisioned Dragonfly, Fat-Tree and Slim Fly networks under synthetically generated workloads as well as real CPU application and novel neuromorphic application trace workloads taken from *NeMo* to provide a fair comparison across a wide range of traffic workloads. Lastly, the results of

Approved for Public Release; Distribution Unlimited.

the comparisons are summarized and compared with physical system costs in an attempt to provide a single figure of merit in comparing each network's performance as an HPC system interconnect.

5. **Classification of AFRL Data Using TrueNorth & *NeMo*:** To determine the accuracy of the *NeMo* model, we used the MNIST benchmark and AFRL MSTAR SAR imagery data-sets and compared *NeMo*'s output accuracy for each of those models against the IBM TrueNorth neurosynaptic processor system. Here, each model was trained using the EEDN convolutional neural network toolkit, a machine learning software toolkit provided by IBM for use with the TrueNorth hardware. EEDN generates trained spiking neural networks in a format that can be used for the TrueNorth hardware system. These trained networks were run using the IBM TrueNorth software simulation tool, where spike activity between neurons and cores was recorded. We then imported these models into *NeMo*, and compared the results against those obtained on IBM TrueNorth.

2 INTRODUCTION

2.1 Parallel Discrete-Event Simulation & ROSS

Capturing performance measurements of extreme-scale networks having millions of nodes requires an approach capable of decomposing a large problem domain. One such method, used in this work, is parallel discrete-event simulation (PDES). PDES decomposes the problem into distinct functional components called logical processes (LPs), each with its own maintained state. These LPs encapsulate the specific functionality of computing components in the simulation such as routers, nodes, and workload processes. LPs interact with one another and represent the system dynamics by passing time-stamped event messages to one another. These LPs are mapped to physical MPI rank processing elements (PEs), that compute their corresponding LPs' events in time-stamped order.

The Rensselaer Optimistic Simulation System (ROSS) provides the PDES platform with support for both conservative and optimistic parallel execution of events [36]. Conservative execution uses the YAWNS protocol [132] to keep all LPs from computing events out of order. The optimistic event scheduler allows each LP to keep its own local time and therefore compute events out of order with respect to other LPs. Optimistic event scheduling enables highly scalable simulations and has demonstrated super-linear speedup capable of processing 500 billion events per second with over 250 million LPs on 120 racks of the Sequoia supercomputer at LLNL [24]. The speedup associated with optimistic execution comes at the cost of out-of-order event execution, which is handled by reverse computation [37]. When a temporal anomaly occurs and an event is processed out of timestamp order, all events must be incrementally rolled back to restore the state of the LP to just before the incorrect event occurred.

The rollback process uses a reverse event handler to undo the events. The reverse handlers for the model must be provided by the model programmers. The reverse event handler is a negation of the forward event handler performing inverse operations on all state changing actions. For example, in the Slim Fly model using non-minimal routing, when a message packet arrives at its first router from a node, that router LP performs forward operations in the router-receive forward event handler. The router LP (1) increments the number of received packets, (2) sends a credit event

to the sending node LP, (3) computes the next destination by sampling a random number for the random intermediate destination, and (4) creates a new router-send event to relay the packet to the next hop router LP. The reverse event handler needs to undo these operations by (1) decrementing the received packets state variable, (2) sending an anti-message to the sending node to reverse the credit sent, (3) unrolling the random number generator by one, and (4) creating an anti-message (a message indicating an event was issued out of timestamp order and needs to be rolled back in optimistic execution) to cancel the router-send event.

2.2 Neuromorphic Computing, IBM TrueNorth & *NeMo*

In recent years, a new type of processor technology has emerged called *neuromorphic computing*. This new class of processor provides a brain-like computational model that enables complex neural network computations (e.g., data classification) to be done using significantly less power than current processors. For example, IBM has created an instance of the *TrueNorth* architecture [15, 21, 40, 41] that has 5.4 billion transistors arranged into 4,096 neurosynaptic cores with a total of 1 million spiking neurons and 256 million reconfigurable synapses. This architecture consumes only 65 mW of power when executing a multi-object detection and classification program using real-time video input (30 fps) for 400×240 pixel images. Using that little power, TrueNorth could run for over one week on a single charge inside today's smartphones. For a list of TrueNorth-capable algorithms and applications, see Esser et al. [59].

This extreme, low-power data analytics capability is particularly interesting as next generation HPC systems are about to experience a radical shift in their design and implementation. The current configuration of leadership class supercomputers provides much greater off-node parallelism than on-node parallelism. For example, the 20 PFLOP "Sequoia" Blue Gene/Q supercomputer located at LLNL has over 98 thousand compute nodes but each compute node provides at most 64 threads of execution. However, in order to reach exascale compute capabilities, a next generation system must be 50 times more power efficient. This dominating demand for power efficiency is resulting in future designs that dramatically decrease the number of compute nodes while increasing the computational power and number of processing cores. Case in point, a recent NASA vision report [151] predicts that exascale class supercomputers in the 2030 time frame will have only 20,000 compute nodes and the number of parallel streams per node will rise to nearly 16,000.

To meet the computational demands of these future designs, it has become a widely held view that on-node *accelerator* processors, in close coordination with multi-core CPUs, will play an important role in compute-node designs [151]. These accelerators are currently used in two forms. The first are graphical processing units (GPUs) that offer a single-instruction-multiple-data approach to parallelism, which matches the execution paradigm of graphics applications. GPUs offer a massive amount of numerical compute power at a very affordable price. The second form of compute node accelerators is a mesh processor architecture such as the Intel Phi [46] which has now been discontinued. Here, a collection of lower clock-rate x86 cores are interconnected over an on-chip mesh network. While GPUs promise increased FLOPs at a lower cost, the Intel Phi provides a simpler programming paradigm, one capable of executing legacy HPC codes. The current marketplace appears to be driven by performance over ease of programming.

A specific example neuromorphic computing device is the IBM TrueNorth Neurosynaptic System [42]. It is a flexible, programmable substrate for neural algorithms, suited to sensory processing and spatio-temporal pattern recognition. It features a many-core processor network on a chip de-

Approved for Public Release; Distribution Unlimited.

sign. Each TrueNorth chip contains 4096 neurosynaptic cores, each composed of 256 programmable neurons, which is over a million neurons per chip. It typically uses about 65 mW of power [16]. Research is currently being pursued to explore the possibility of having neuromorphic chips on the next generation supercomputing systems [80]. So, the setting that this work lies is one where the next generation supercomputers have a Neuromorphic Processing Unit (NPU) on each node and these NPUs are able to perform ML/DL tasks extremely fast and in an energy efficient way.

Given the advent of neuromorphic computing, the question we will address in this work is how might a neuromorphic processor be used as an accelerator to improve the application performance, power consumption, and overall system reliability of future exascale systems. This report highlights the neuromorphic architecture as a key technology (especially in the next generation of supercomputing systems) for large-scale data processing.

To address this larger research question, an open-source neuromorphic processor architecture simulator called *NeMo* was created. This effort will enable the design space exploration of potential hybrid compute and neuromorphic systems.

2.3 Classification of Supercomputer Failures Using TrueNorth:

In addition to the *NeMo* simulator, we look at a particular application that would potentially be running on the IBM TrueNorth processor that is available to us as part of an IBM Loan agreement. The application is an operating system tool that can predict when a node on the supercomputer is about to fail – a node failure predictor. Failures in supercomputers, especially node failures, are an unfortunate and unpleasant reality that the modern day supercomputing systems have due to their size and overall system complexity [171]. They not only result in extensive downtimes, lower reliability and component loss, but also consume significant amount of human resources. Therefore it would be ideal to have the ability to predict such failures ahead of time so that the jobs running on a potentially failing node can be rerouted to other nodes, new jobs can be deferred to healthier nodes and the node itself can be repaired during a scheduled maintenance period.

The main contributions of this work are as follows:

- We demonstrate that node failures can be modeled and classified using a neuromorphic computing approach by leveraging the IBM TrueNorth chip.
- We show that the Spiking Neural Network (SNN) of TrueNorth outperforms other Machine Learning (ML) and Deep Learning (DL) approaches for our application.
- We show that all ML, DL and neuromorphic computing approaches yield accurate results for our application.

Now, the majority of research in modeling and predicting node failures has been domain-specific. Gainaru and Cappello provide an overview of failures observed in large-scale High Performance Computing (HPC) systems and also point out their characteristics with emphasis on modeling, detection and prediction [69]. Schroeder and Gibson review the sources of failures, corresponding decrease in application effectiveness, and also the coping strategies like application-level checkpoint compression and system level process-pairs fault-tolerance for supercomputing [148]. Fiala et al. study the use of redundancy to detect and correct soft errors in MPI applications that lead to silent data corruption and eventually node failures [65]. Abawajy proposes a fault-tolerant scheduling

policy which couples job scheduling with job replication so that jobs are efficiently and reliably executed for grid computing systems [11]. Chen et al. explore the use of a floating-point arithmetic coding approach to build survivable HPC applications that can adapt to node failures without being aborted [45]. Fagg et al. emphasize the importance of application-level fault-tolerant systems and present a fault-tolerant version of the Message Passing Interface (MPI), which lets applications recover from a node or link error and as a result continue execution normally [63]. These papers focus on fault-tolerance from the point of view of applications being run.

When it comes to modeling failures, Zheng and Lan extend the Amdahl's Law and Gustafson's Law in order to consider the impact of failures and the effect of fault-tolerance techniques on applications [172]. Their reliability-aware models can predict application scalability in failure prone environments and also evaluate various fault-tolerant techniques. When it comes to tackling node failures, Bautista-Gomez et al. propose a low-overhead high-frequency multi-level checkpoint technique to tackle node failures by integrating a highly reliable topology-aware Reed-Solomon encoding in a three-level checkpointing scheme [27]. Egwutuoha et al. review the failure rates of HPC systems, survey fault-tolerance approaches, discuss the feature requirements of rollback-recovery techniques and develop a taxonomy for twenty checkpoint-restart solutions [57]. Brown and Patterson propose Recovery-Oriented Computing (ROC), which is an approach that recognizes the inevitability of unanticipated failure and focuses on recovery and repair as opposed to simple fault-tolerance [34].

Apart from the domain-specific approaches mentioned above, researchers have also used statistical methods as to analyze node failures. Schroeder and Gibson analyze failure data from two large HPC sites, and use a statistical approach to study the root cause of failures, the mean time between failures and mean time to repair [147]. Hacker et al. analyze the event logs of two IBM Blue Gene systems, characterize system failures statistically and propose a semi-Markov model for predicting the probability of a node failure [77]. This work found that components within the supercomputer often become "noisy" by issuing four or more warning events prior to complete failure and is a driver for our research here. Liang et al. have used RIPPER (a rule-based classifier), Support Vector Machines (SVM) and a Nearest Neighbor method to classify failures using error logs from the IBM Blue Gene/L machine [112]. SVM (recall: 80%, precision: 50%) and Bi-Modal Nearest Neighbor (BMNN) (recall: 85%, precision: 35%) were seen to perform best for 12-hour and 6-hour prediction windows respectively. Yu et al. have used event-based Bayesian model to classify failures using error logs from the IBM Blue Gene/P machine achieving 83.8% accuracy [169].

2.4 Durango – A Hybrid System Performance Modeling Framework

Performance modeling of extreme-scale applications using accurate representations of potential architectures is critical for designing next generation supercomputing systems because it is impractical to construct prototype systems at scale with new network hardware in order to explore designs and policies. However, simulations often rely on static application traces that can be difficult to work with because of their size and lack of flexibility to extend or scale up without rerunning the original application. For example, the application traces available as part of the DOE Design Forward program (see: `http://www.exascaleinitiative.org/design-forward`) can be hundreds of gigabytes. Moreover, once traces are created, they are fixed and cannot be changed Also, traces require a system and time where the trace can be created.

On the other hand, well-known patterns [49] coded in a simulator-specific language also have shortcomings. First, these patterns typically are synthetic and not often representative of real

application behaviors, thus driving the need for real application traces. Second, these patterns often do not include any computation for the processors, so there is limited ability to inject realistic processor behaviors between communication events.

To address this problem, we have created a new technique for generating scalable workloads from real applications, and we have implemented a prototype, named *Durango*, that combines a proven analytical performance modeling language, *Aspen*, with the massively parallel HPC network modeling capabilities of the CODES framework. Our models are compact, parameterized and representative of real applications with computation events. They are not resource intensive to create and are portable across simulator environments. Specifically, we make the following two contributions in this research thrust:

- Comparison of the Aspen-generated network communication patterns for the LULESH miniapp with real LULESH application network communications via traces that are run through the CODES packet-level network simulation framework. Durango shows identical agreement with the real application trace data for key network performance statistics. During our validation of Durango's generated communication model of LULESH, we found that the original LULESH miniapp code had a latent bug where the `MPI_Waitall` operation was used incorrectly. This finding underscores the potential need for a tool such as Durango, beyond its benefits for flexible workload generation and modeling.
- A scaling study of Durango's direct integration approach, which links Aspen into CODES as part of the running network simulation model. Here, Aspen generates the application-level computation timing events as part of an overall discrete-event system model, which in turn drive the start of a network communication phase. Performance results show that Durango's performance scales well when executing both torus and dragonfly network models on upto 4K Blue Gene/Q nodes using 32K MPI ranks and avoids the overheads and complexities associated with extreme-scale trace files.

2.5 HPC Network Models

As we move to the era of exascale computing, increasing computational power of supercomputing systems will continue to increase demand on the underlying interconnection network to handle inter-process communication. HPC systems today come in many sizes, compute complexities, compute densities, and cost constraints making it hard to select one particular network topology as a "one size fits all." Furthermore, the increased adoption of hybrid architecture systems opens the door for the consideration of new computing architectures like the biologically inspired neuromorphic processor, further straining the network to handle complex new traffic workloads. To keep up with the increasing amount of compute power and compute heterogeneity, HPC system designers must weigh the many options for improving network performance to match. This research uses high-fidelity end-to-end modeling and simulation using the CODES framework to analyze the ability of current interconnection network topologies as well as theoretical topologies to handle such a wide set of communication traffic workloads.

In recent years, new types of processor and network technologies have emerged that may be deployed in future supercomputer system designs. First, neuromorphic computing is a new class of processor that provides a brain-like computational model that enables complex neural network computations to be done using significantly less power than current processors. IBM has created an

instance of a spiking neuromorphic processor, called TrueNorth, that has 4096 neurosynaptic cores with a total of 1 million spiking neurons and 256 million re-configurable synapses that consume only 63 milliwatts when executing a multi-object detection and classification program using real-time video input [122]. Several other advanced neuromorphic processor hardware architectures have been developed as well, featuring various hardware designs and features [97], [120], [29]. More recently, Intel has created the *Loihi* spiking neuromorphic processor that is able to perform on-chip learning [53].

The ability of the interconnection network to transfer data efficiently remains essential to the successful implementation and deployment of large-scale HPC systems even in the face of architectural changes like the neuromorphic processor. Now, supercomputer network designs are moving away from multidimensional torus networks used in Cray XT and IBM Blue Gene systems with the advent of high-radix routers enabling lower diameter interconnect topologies such as Dragonfly [98], [62], Fat-Tree [137] and Slim Fly [32]. There is a trade-off of latency, bandwidth, network diameter, and cost among the potential network topologies. These networks promise either full bisection bandwidth as is the case with the Fat-Tree or strike a balance between global and nearest neighbor traffic patterns using fewer network links and lower overall costs as with the Dragonfly and Slim Fly topologies. Each network topology provides a compelling option as interconnection networks for large-scale computing systems.

In this research thrust, traditional synthetic workloads as well as representative workloads from leadership class supercomputer systems and neuromorphic applications are used to analyze the performance of homogeneous and hybrid compute systems where a compute node has both CPU and neuromorphic chips. To perform the detailed design space exploration, we use an end-to-end modeling and simulation workflow that allows asking metric-driven questions about the capabilities of potential hybrid CPU-neuromorphic supercomputer system designs. Traditional, cycle-accurate simulations provide high accuracy and are very detailed. Yet when it comes to replaying real application traces using large node and process counts, the added detail limits the runtime performance scalability resulting in the use of synthetic workloads [32], [98], [128]. Analytical modeling, on the other hand, can provide an accurate picture of the application, but it lacks in modeling critical interconnect details such as congestion and flow control. In this work, we leverage the CODES simulation framework that uses the optimistic discrete-event scheduling capability of ROSS to enable efficient, scalable, and detailed packet-level network simulations [168], [130].

The focus of this research thrust is on predicting and analyzing the performance of different networks under various potential configurations and traffic load conditions of next generation supercomputing systems. To reach that goal, the parallel discrete-event simulation (PDES) based HPC system modeling package of ROSS and CODES are used as the foundation simulation framework. The work in this thesis builds on that framework to include the network construction and packet-level simulation capabilities of a new bandwidth optimized and cost effective theoretical Slim Fly network topology. Additionally, the previously supported CODES Fat-Tree network model is also expanded to provide the ability to simulate pruned Fat-Tree configurations as well as Fat-Tree networks deployed with multiple network rails. Each of the new network models and model extensions has been validated with published results, controlled workload analysis, and/or a controlled hardware system comparison to ensure expected and accurate prediction of simulation results.

Finally, to perform detailed design space exploration of next generation hybrid compute systems, the MPI layer in the CODES framework, which is responsible for replaying CPU application trace workloads, has been expanded to read and replay real neuromorphic computing application

traces through all supported HPC network models and configurations. This new feature provides the capability to simultaneously replay CPU and neuromorphic workloads and allows the study of metric-driven questions regarding the capabilities of potential hybrid CPU-neuromorphic supercomputer system designs. The HPC simulation contributions made to the CODES framework are then used to predict and analyze performance of the available CODES network topologies under synthetic, CPU trace, and neuromorphic trace workloads.

2.6 Classification of AFRL Data Using TrueNorth & *NeMo*:

To determine the accuracy of the *NeMo* model, we used the MNIST benchmark and AFRL MSTAR SAR imagery data-sets and compared *NeMo*'s output accuracy for each of those models against the IBM TrueNorth neurosynaptic processor system. Here, each model was trained using the Energy-Efficient Deep neuromorphic Networks (EEDN) toolkit, a machine learning software toolkit provided by IBM for use with the TrueNorth hardware. Here, EEDN generates trained spiking neural networks in a format that can be used for the TrueNorth hardware system. These trained networks were run using the IBM TrueNorth software simulation tool, where spike activity between neurons and cores was recorded. We then imported these models into *NeMo*, and compared the results against those obtained on IBM TrueNorth.

3 METHODS, ASSUMPTIONS AND PROCEDURES

3.1 *NeMo*

3.1.1 Background

Neuromorphic computing refers to hardware implementations of cognitive computing techniques. More specifically, the goal of neuromorphic computation is the design and development of neuron inspired hardware in an energy efficient package [121].

Several viable designs have already been produced, including offerings from IBM, Intel, and several startup companies. Intel's processor, "Loihi", features a spiking neural network design along with on-chip learning[53]. IBM's offering, the "TrueNorth" processor [15, 21, 40, 41], features 1 million spiking neurons and 256 million reconfigurable synapses, consuming only 65 mW of power when executing a multi-object detection and classification program using real-time video input (30 fps) for 400×240 pixel images. TrueNorth could run for over one week on a single charge inside today's smart-phones.

Neuromorphic hardware development has significantly progressed, giving rise to new processor designs. These hardware designs often implement spiking neural networks in hardware. Spiking neural networks are a type of artificial neural network, similar in concept to a multilayer Perceptron model. However, spiking neural network models have significant design differences from the more traditional machine learning artificial neural networks used in many ML problems today.

In general, neuromorphic hardware implements a type of neuron model in specialized hardware. There are several categories of neuron models used in modern applications, with each model providing tradeoffs between complexity, flexibility, and power.

One of the most prevalent category of neuron model are those based on the McCulloch-Pitts style neuron[118]. These neurons form the basis of what can be considered the "traditional" artificial

Approved for Public Release; Distribution Unlimited.

neural network. These neuron models generally feature complex, differentiable activation functions, allowing for well understood training algorithms [143]. Implementations of this neuron model in hardware are complex[96, 18, 92], but this complexity can provide advanced capabilities, including hardware based derivatives to facilitate on-chip training [26].

The other major category of neuron model contains neurons that behave in a manner inspired by or directly modeling biological behavior. The original intent of these models were to reproduce observed behavior in biological neurons, however, they are capable of solving machine learning tasks [116]. Several different models have been developed, each with variations in complexity and accuracy when compared to a biological neuron [87, 72]. The Hodgkin-Huxley model, for example, uses four dimensional nonlinear differential equations to represent detailed behaviors observed in a biological neuron [118]. Since this model provides accurate neuronal behaviors, it is a popular choice for neuromorphic hardware that attempts to model biological systems [33, 73, 115, 83].

Other spiking models make a trade-off between computational complexity and model accuracy. The simplest model, "Integrate-and-Fire", takes the sum of weighted inputs, and produces an output if the sum is greater than some set threshold. The Integrate-and-Fire model is relatively simple to implement in hardware, but can still provide enough complexity for simple biological simulation tasks, [71] as well as machine learning tasks [141, 86, 156].

These models differ from a traditional machine learning neuron model in a few key areas. The output of a traditional artificial neural network is generally a function based on its total input:

$$y_i = f(\sum_{j=0}^{n} w_{i,j} x_j)$$

[101]. Here i is a particular neuron in the ANN. The weight of a given input$j$, is given as w_{ij}. The input values to this neuron are represented by j, and y_i is neuron i's output value. From this simplified version of a neuron, we can see that when inputs are sent to the artificial neural network, outputs are calculated immediately, and there are no constraints on the type of values used for inputs and outputs. This neuron model is the McCulloch-Pitts model, and is widely used in machine learning tasks in software. For example, one commonly used Perceptron model is a McCulloch-Pitts neuron with a simple linear threshold function as:$f(y_i) = \text{sign}(\sum_{j=0}^{n} w_{i,j} x_i)$ [12].

Spiking neural networks, on the other hand, have significantly different features. One of the more common models of spiking neurons, commonly used for machine learning tasks, is the leaky itegrate and fire (LIF) model. An LIF neuron's activation function is defined as [88]:

$$Cv = g_{\text{leak}}(E_{\text{leak}} - v) + I(t), \text{if } v \geq v_{\text{threshold}} \text{ then } v \leftarrow c$$

Here, v is the "membrane potential" of the neuron, g_{leak} and E_{leak} are "leak" factors, and $I(t)$ is the input value, at a particular time, t. This model has significant differences over the traditional neural network model. Firstly, the spiking model has the concept of time and state changes. Spiking neurons are modeled over a period of time, and have internal state that changes depending on the model and input values. Secondly, the neuron outputs values as "spikes", which are generally considered to be equivalent to outputting `True` to the next connected neuron. This model is discussed in more detail below.

Even though traditional machine learning neural networks used in deep learning are significantly different than spiking models, the current generation of neuromorphic hardware generally implements spiking neuron models with binary outputs [149]. Spiking neural networks in hardware allow

for higher efficiency, providing excellent power per synaptic operation. Signals enter an axon, are passed to a synapse, and are processed by a neuron. The neurons in these models currently manage all of the computation—axons and synapses merely act as a signal transfer service to the neuron [81].

The concept of using neurons to facilitate computation has been an area of research since even before the introduction of the perceptron model [12]. Conceptually in the machine learning field, using multiple neurons chained together in an artificial neural network is a complex linear model [12]. Variations on these models have been used to great effect in many problem domains [23, 146, 52], showing just how versatile these models can be.

Implementations of neural networks in hardware has significantly progressed, giving rise to many new processor designs. These hardware designs implement a wide variety of neuron models, including variations of biologically inspired models [87], integrate and fire models, and even models based on the McCulloch-Pitts neuron model [18, 22]. This work focuses on simulations of neuromorphic hardware implementing spiking neural network models, primarily based on the leaky integrate-and-fire neuron model.

There are numerous implementations of this type of hardware [15, 134, 121, 122], and new designs are currently being developed. Spiking neural networks have input, internal connections, and output components. Input elements are referred to as *axons*, output elements are called *neurons*, and the connectors between axons and neurons are called *synapses*. Signals sent from a neuron are generally referred to as "spikes," as they are treated as binary signals. These terms stem from more general ANN models, which in turn are borrowed from neuroscience [146]. Neuromorphic processors generally operate on a synchronized clock, allowing them to receive, process, and send new messages in between external clock cycles. For example, the TrueNorth hardware architecture has an external clock rate of 1 kHz, allowing each neuron to receive and send a spike 1,000 times per second [15].

Leaky Integrate and Fire (LIF) Neuron Model: This is a simple neuron model that is able to emulate some biological neuron functions [87]. Because it is so straightforward (and does not rely on partial differential equations), the LIF model (and variations of it) is used in many neuromorphic hardware implementations.

Neurons that implement the LIF model follow a simple pattern of execution, shown in 1 [155]. Execution consists of an integration period, leak calculations, threshold checking, firing, and then reset. In 1, the general form of the neuron equations are presented. During integration, shown in 1, the neuron updates its internal voltage based on each synapse i's synaptic weight, s_i. This is calculated based on the synapse's activity at time t, shown as $x_i(t)$ in 1. Next, the LIF neuron calculates leak, by subtracting the set leak value, λ, from the current membrane potential, shown in 2. Next, in 3, the neuron checks the threshold value, α, against the current membrane potential. If the current membrane potential is greater than α, the neuron spikes. If the neuron spikes, 4 executes, setting the neuron membrane potential to the reset voltage, R_j. This model forms the basis of the TNLIF neuron model, and thus the basis of the *NeMo* simulation model.

IBM TrueNorth Neuron Model: This neuron model is a significantly enhanced version of the simple LIF model. In 2, the full TrueNorth neuron model is presented. Functions used in this model include the signum (sgn):

$$sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

Integration:

$$V_j(t) = V_j(t-1) \sum_{i=0}^{n-1} [x_i(t)\, s_i] \tag{1}$$

Leak:

$$V_j(t) = V_j(t) - \lambda_j \tag{2}$$

Threshold Check and Spike:

$$\textbf{if} \quad V_j(t) \geq \alpha_j \tag{3}$$
Spike
$$V_j(t) = R_j \tag{4}$$
$$\textbf{end if} \tag{5}$$

Figure 1: Leaky Integrate and Fire (LIF) General Neuron Model.

a comparison function for stochastic operations:

$$F(s,p) = \begin{cases} 1, & |s| \geq p \\ 0, & |s| < p \end{cases}$$

and the Kroneker delta function: $\delta(x)$. The TrueNorth LIF or TNLIF neuron model features a fully connected "neurosynaptic crossbar." This crossbar connects each input axon with all neurons. When an axon receives a spike, it sends signals to all connected synapses. The neuron integration equation is presented in 6. At time t, if an axon ,i, is active, the synaptic activity, $A_i(t)$, is 1, otherwise it is 0. In the equation, $w_{i,j}$ represents connectivity between axons and neurons. If $w_{i,j}$ is 1, there is a connection between axon i and neuron j. If the value is 0, there is no connection.

Each neuron assigns a type, represented by G_i, to each axon. Weights are then assigned to each axon type. G_i is limited to four types, therefore each axon may be assigned one of four different weights by each neuron. Neuron weights are stored as signed integers, shown in the equation as $s_j^{G_i}$. $b_j^{G_i}$ sets deterministic or stochastic integration mode. If the value is 0, neurons update their membrane potential by taking the sum of each axon multiplied by each axon's weight: $\sum_{i=0}^{n-1}[s_j^{G_i}](A_i(t)w_{i,j})$

Neurons can be configured to use stochastic synaptic and leak integration. Setting $b_j^{G_i} = 1$ enables stochastic synaptic integration and setting $c_j^{\lambda} = 1$ enables stochastic leak integration. Stochastic integration functions similarly for both leak and synaptic weight. For each integration event (either a synaptic weight or a leak computation), a random number is drawn and stored as p_j. If the drawn random number is higher than the relevant weight (synaptic weight $s_j^{G_i}$ or leak weight λ_j), then the neuron adds $sgn(\lambda)$ or $sgn(s_j^{G_i})$ to its membrane potential. Synapse integration is shown in Equation 6, and leak integration is shown in Equations 8 and 7.

The TNLIF neuron model enhances the leak functionality of the LIF model by adding positive or negative leak values, and a "leak-reversal" ability. Normal leak operation calculates the sign of

Integration:

$$V_j(t) = V_j(t-1) + \sum_{i=0}^{n-1} \left[A_i(t)\, w_{i,j} \left[(1 - b_j^{G_i})\, s_j^{G_i} + b_j^{G_i}\, F(s_j^{G_i}, p_{i,j})\, \mathrm{sgn}(s_j^{G_i}) \right] \right] \tag{6}$$

Leak Integration:

$$\Omega = (1 - \epsilon_j) + \epsilon_j\, \mathrm{sgn}(V_j(t)) \tag{7}$$

$$V_j(t) = V_j(t) + \Omega \left[(1 - c_j^{\lambda})\, \lambda + c_j^{\lambda}\, F(\lambda_j, p_j^{\lambda})\, \mathrm{sgn}(\lambda_j) \right] \tag{8}$$

Threshold, Fire, Reset:

$$eta_j = p_t^T \,\&\, M_j \tag{9}$$

if $\quad V_j(t) \geq \alpha + \eta_j$ (10)

Spike

$$V_j(t) = R_j + \delta(\gamma_j - 1)\,(V_j(t) - (\alpha + \eta_j)) + \delta(\gamma_j - 2)\, V_j(t) \tag{11}$$

else if $\quad V_j(t) < -\left[\beta_j \kappa_j + (\beta_j + \eta_j)(1 - \kappa_j) \right]$ (12)

$$V_j(t) = -\beta_j\, \kappa_j + \left[-\delta(\gamma_j)\, R_j + \delta(\gamma_j - 1)\,(V_j(t) + (\beta_j + \eta_j)) + \delta(\gamma_j - 2)\, V_j(t) \right] (1 - \kappa_j) \tag{13}$$

end if

Figure 2: TrueNorth leaky integrate and fire neuron model (TNLIF)

the leak value λ, stores this value as Ω, and then integrates this value into the neuron's membrane potential. Leak sign calculation is shown in Equation 7, and integration is shown in Equation 8. Leak-reversal mode changes the behavior of the leak function such that if the neuron has a positive membrane potential, λ is integrated directly, but if the neuron has a negative membrane potential, $-\lambda$ is integrated. In addition, if the membrane potential of a neuron is 0, then no leak is applied.

In addition to the deterministic threshold modes available, the TNLIF neuron model provides a stochastic threshold mode. Here, η_j is added to α_j and β_j. Then, η_j is calculated every cycle by first generating a random number value, p_j^T, then taking the bitwise AND of M_j and p_j^T, as seen in 9. In Equations 12 and 10, η_j is added to the threshold values before they are checked against the neuron membrane potentials.

The TNLIF model also adds two new reset modes to the standard LIF model. TNLIF supports normal reset mode, a linear reset mode, and a non-reset mode. These values are chosen through the variable γ_j, and used in Equations 12 and 11. Normal mode follows the standard LIF model. Linear reset mode subtracts the threshold value from the membrane potential. In non-reset mode, the membrane potential is not changed after a spike. These reset modes add additional functionality to the standard LIF neuron model.

TNLIF adds a negative threshold feature to the LIF. This negative threshold value is represented by β_j, an unsigned integer. This gives neurons the ability to have a membrane potential floor or a "bounce" feature. In the case of a floor setting, neurons with membrane potentials below $-\beta_j$ will set their values at $-\beta_j$. If the setting is set to a "bounce" value,, the neuron's membrane potential is reset to $-\beta_j$. The mode is set by changing the value of κ_j. Equation 12 shows the negative

threshold check, and Equation 13 shows negative threshold reset and saturation.

The enhancements to the LIF model provided by TNLIF improves its flexibility and power. The additional stochastic integration and threshold features allow the TNLIF model to emulate continuous weight functions. Furthermore, the stochastic features allow neural networks trained with traditional back-propagation techniques to run directly on the hardware [60]. Cassidy et al. demonstrated the flexibility and power of this neuron model in [40] and Akopyan et al. implemented this model in hardware in [15].

3.1.2 Design and Implementation of *NeMo*

NeMo simulates neuromorphic hardware using speculative simulation techniques. The properties of neuromorphic hardware are such that discrete event simulation may provide excellent performance. Spiking neural networks, as implemented in hardware, generally have a low rate of neuron activity at any given time. Furthermore, spikes do not carry more than a binary piece of information, making all spikes homogeneous across the network. This discrete output from the neurons, coupled with the low average network activity, produces a connection dense network with relatively low message activity. Based on promising results published by [114], *NeMo* implements a discrete-event simulation of neuromorphic hardware, using optimistic event scheduling.

Given the properties of neuromorphic hardware activity rates, we chose a parallel discrete event simulation that uses the Time Warp optimistic synchronization algorithm. Using an optimistic algorithm can lead to performance gains over a conservative time-stepped simulation when synchronization does not need to occur at every time-step. When simulating neuromorphic hardware, this speedup will be limited by how active the neurons are over time. The worst case scenario for this type of simulation would be a model where all neurons are actively sending spikes to each other. In this situation, the optimistic synchronization method will fall behind the performance of a conservative algorithm, due to the overhead induced by the optimistic synchronization algorithm.

The TNLIF model has specific limitations due to its implementation in hardware. *NeMo*, however, is not designed as a simulation of solely the TrueNorth processor hardware, rather it is a more generic neuromorphic processor simulation model. Given the constraints of the TNLIF model, *NeMo* implements all documented features of the hardware. In addition, *NeMo* supports significantly more features than the TrueNorth hardware. *NeMo* does not have the bit length constraints that are part of TrueNorth. *NeMo* may have a 64-bit signed integer value for weights, thresholds, and pseudo-random numbers. Furthermore, while *NeMo* operates with the same conceptual neurosynaptic crossbar that TrueNorth uses, the crossbar can be set to an arbitrary size, constrained only by memory of the system. This allows *NeMo* to simulate neurosynaptic cores of any size, across one or more MPI ranks. *NeMo* adds to these features by allowing the removal of the neurosynaptic crossbar completely, collapsing the model into a more traditional spiking neural network.

NeMo is also capable of simulating compute-on-synapse and compute-on-axon event models. This features gives *NeMo* the ability to execute operations at the synapse or axon level, allowing for more complex neurosynaptic chip designs to be simulated.

NeMo partitions the model of a neuromorphic processor into individual components. Each axon, synapse, and neuron are modeled as a unique LP type. By having individual elements of the neuromorphic chip running as individual LP types, *NeMo* is able to add processing features to the synapses and axons. Furthermore, advanced axon $\rightarrow$ synapse $\rightarrow$ neuron connections could possibly be modeled. A collection of axons, synapses, and neurons are contained within a logical container,

Approved for Public Release; Distribution Unlimited.

referred to as a neurosynaptic core. *NeMo* can model thousands of neurosynaptic cores with each core containing hundreds of neurons and axons and tens of thousands of synapses.

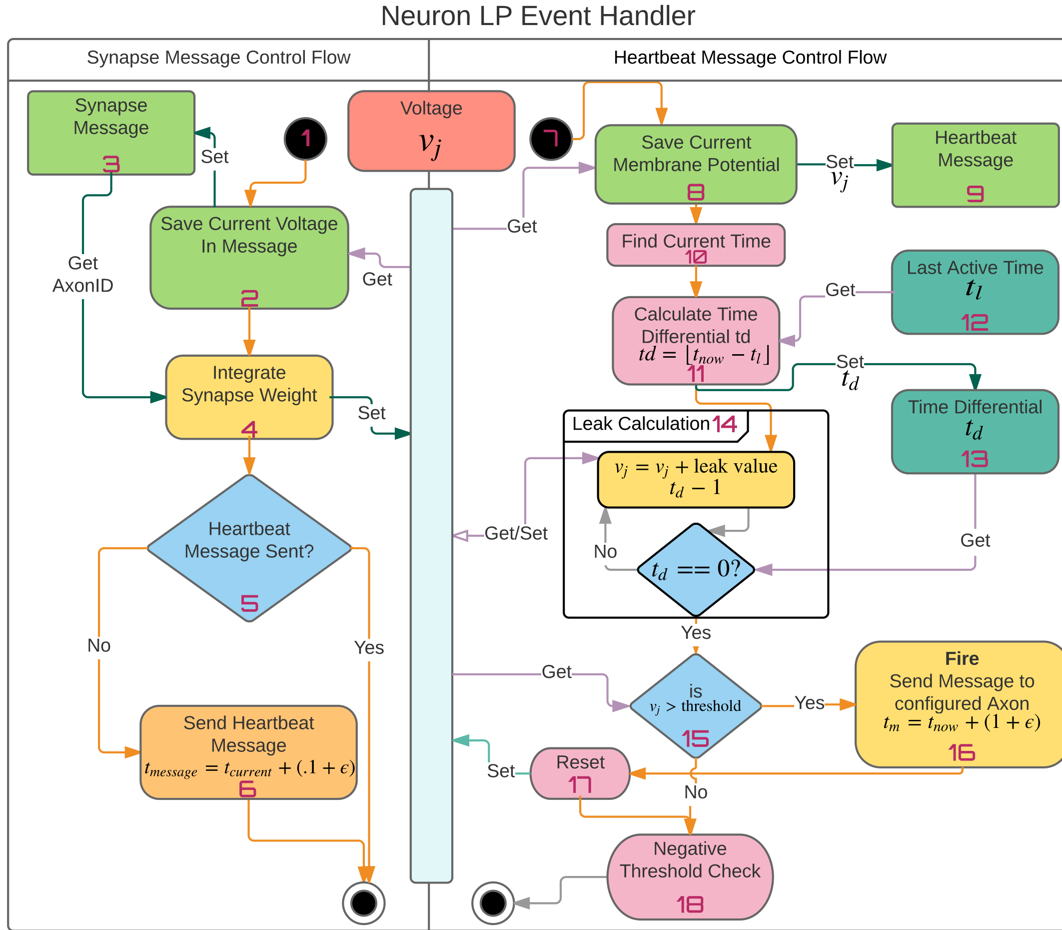


Figure 3: The *NeMo* neuron event flow. Details of each block, numbered 1 through 18, are discussed in the text below.

Forward Event Computation: For the benchmarking and testing of *NeMo*, we implement a model with similar capabilities as the TNLIF model. Therefore, we do not add any computation to the axon and synapse logical processes or LPs. Table 1 shows the logical layout of neurons, axons, and synapses on a neurosynaptic core. When an axon receives a message it relays the message to each synapse in its row. In this model, the synapses simply relay any received message to the neuron in their column. Like the TNLIF model, there are no computations that occur when axons and synapses receive events; they simply relay their messages to the next element in the model.

In 3, Blocks 1–18, we show the *NeMo* neuron model control flow for the forward event handler. The flow starts at the current simulation time, t, where t is measured in microseconds. If $t > 1$, there has been at least one neurosynaptic tick since the simulation has started. There are two event types that neurons receive: synapse messages and heartbeat messages. Synapse messages are set at a nanosecond resolution, with events occurring at $t + 0.0001 + \epsilon$ where ϵ is an extremely small random "jitter" value used to prevent ties in the scheduling of events to ensure a deterministic ordering of events.

ϵ is a small random value chosen to prevent event timestamp collisions. While the TimeWarp algorithm can theoretically return correct results when events occur simultaneously, the TimeWarp implementation in ROSS does not guarantee that multiple runs of the same model will produce the same output when event collisions occur. When ROSS manages rollback messages, these messages are organized in a priority queue. If two events have the same timestamp, there is no way to guarantee what order they will appear in the queue of events. Despite this, *NeMo* may produce correct results even with event collisions, however, making *NeMo* deterministic and forcing *NeMo* to generate exactly reproducible output with every run required adding a random jitter factor to each event.

To wit, we use a random number generator that provides enough entropy to prevent event collision. Since the primary goal of *NeMo* is to simulate a neurosynaptic tick, we store the current tick value as the whole number in a 64 bit floating point number. Jitter is added as a small component of the floating point number. We assign 8 digits to the jitter value in the time variable, using the integrated ROSS random number generator. ROSS provides warnings that alert model developers to event collisions, allowing us to determine if any collisions have occurred. When combined with the incrementing values in the time-step, we found that this technique prevents event collisions during our tests.

The synapse message process begins in Block 1, when the neuron receives a synapse message. The neuron first saves the current voltage value, Block 2, a double precision floating point value V_j, in the synapse message, Block 3. This is to facilitate reverse computation, by saving V_j in the message, when rolling back messages neurons are able to revert changes made during forward computation.

The neuron then performs the integration function, shown in 3 as Block 4. This updates V_j with a new value, computed by the integration function defined in 6.

Neuron heartbeat messages are *NeMo*'s technique to synchronize neuron firing. In a LIF model, neurons integrate, leak, fire, and reset at specific intervals. We use heartbeat messages as way to ensure that neurons will perform the leak,fire, and reset functions only after receiving a spike event. This technique was chosen as a way to ensure that a neuron will leak and fire only after receiving a spike in a time-stepped method, while still providing performance gains that come from inactive neurons not computing at every time-step, as in a synchronized parallel model.

To increase performance, a heartbeat message is sent only when a neuron activates. In Block 5, the neuron checks if it has already sent a heartbeat message. If it has not, it schedules a heartbeat message at $t + 0.1 + \epsilon$, in Block 6. This action completes the neuron's integration function for a particular axon. By executing this flow every time an axon message is received, *NeMo* recreates the

Table 1: A matrix representation of a neurosynaptic core.

Axons	**Synapses**			
0	0,0	0,1	...	$0, n$
1	1,0	1,1	...	$1, n$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	$n, 1$	$n, 2$	$n, 3$	n, n
Neurons	0	1	...	n

Approved for Public Release; Distribution Unlimited.

integration formula in Equations 2 and 6.

When a heartbeat message is received, as shown in Block 7, the neuron begins its leak, fire and reset function. The neuron also saves its current membrane potential in the received message(Blocks 8 and 9).

The neuron then finds the current neurosynaptic time in Block 10. This is computed as $\lfloor t \rfloor$. In Block 11, the neuron calculates a time differential, t_d. This value represents how many neurosynaptic clock cycles have passed since this neuron has been active. By taking the last active time value, Block 12, and subtracting the current time, the neuron is able to determine how many times it needs to run the leak calculation, shown in Block 13. The neuron uses this time differential value to compute leak. By using a loop, the neuron is able to run the leak function, shown in Equations 7 and 8, t_d times, bringing its voltage to where it would have been if the neuron had been calculating the leak function in a synchronous fashion. This loop is shown in Block 14.

Once the neuron has computed the leak function, it proceeds to check the positive threshold (Block 15), and either fires and resets, or moves on to the negative threshold check. If v_j is greater than the threshold, the neuron will fire(Block 16) and reset(Block 17). A fire operation schedules a new message with ROSS at the next neurosynaptic clock time. Since the neurosynaptic clock operates at the integer scale, simply adding $1 + \varepsilon$ to the current time will schedule the fire event at the proper future time.

After the neuron completes the fire/reset functions, it then checks for negative threshold overflows(Block 18). If the neuron's voltage is beyond the negative threshold, the neuron performs the negative threshold integration functions specified in Equation 13. With some neuron configurations, the reset voltage may be configured to subtract a value from the current membrane potential, rather than reseting the value to zero. As a final check after both reset functions have completed, if there is a remainder voltage that is past the threshold the neuron will schedule a heartbeat message for the next neurosynaptic tick.

The neuron has now completed one neurosynaptic tick. This example is that of a neuron as it receives spike events from connected axons. Here, the event flow assumes that the neuron is not self-firing. Certain configurations of neurons can create a situation where neurons are able to self-fire; setting the leak to a negative value, for example, will result in a neuron that spikes without receiving any axon inputs. In *NeMo*, this situation is handled by first detecting the presence of a self-firing neuron, and then managing the neuron.

If a neuron has a positive leak (a leak which increases the membrane potential of the neuron) or a negative leak value with a corresponding negative reset value above the threshold value, *NeMo* will detect this and apply a self-firing tag. This tag may be manually set when implementing neurons, allowing for other, non-detected self-firing neurons to be handled properly.

If the self-firing flag is set, *NeMo* will detect this at the beginning of the simulation, and schedule a heartbeat message for the next neurosynaptic tick. For all new ticks, this neuron will schedule heartbeat messages at each neurosynaptic tick to ensure that all self-firing events are simulated.

Reverse Computation: Reverse computation is handled through swapping states at key points in the neuron process and using bitfields to manage secondary state changes. The primary state change that occurs is V_j, the neuron's voltage. Neurons also contain a flag, marking when a neuron has sent itself a heartbeat message. When performing reverse computation, neurons must revert changes to both of these state elements.

Whenever a neuron receives a message from a synapse or receives a heartbeat message, before any changes are made to V_j, it saves the current current voltage in the incoming message. During reverse

computation, neurons restore the saved voltage from the message. This reverts all integration, leak, and reset functions that changed V_j.

When a neuron receives a synapse message for the first time, it checks to see if it has sent a heartbeat message. If it has not, it changes an internal flag, and sends the message. The neuron also changes the flag when receiving a heartbeat message. Neurons record boolean flag changes in a bitfield in the incoming message. If there is a non-zero entry in the bitfield during reverse computation, the flag state is toggled.

Fanout: Since *NeMo* has individual LPs configured for each component, simulations have a large number of LPs running simultaneously. There are 2,164,260,864 LPs in our largest simulation experiment. If *NeMo* sent messages at every time stamp, it would send 66,048 messages per neurosynaptic core per tick. This large event population quickly becomes unmanageable due to memory constraints. To counter this, *NeMo* implements a fanout technique for message transmission based on work done in [106].

In Figure 4, an example of the fanout message technique is shown. Here we see a neurosynaptic core with three axons, nine synapses, and three neurons. When a message is received by an axon, it sends an axon message to the first synapse in the neurosynaptic core at time $T + 0.0001 + \epsilon$. The synapse then sends two messages: first to the neuron attached to it, second to the next synapse in the row at $T + 0.0002 + \epsilon$. The next synapse does the same, until the final synapse has been reached. This technique generates far fewer messages, preventing memory usage issues.

3.1.3 Design and Implementation of *NeMo* Super Synaspe (SS)

NeMo-SS builds on original *NeMo* or *NeMo-ES*, implementing all of the features, while making some significant changes to the simulation design. The biggest change implemented is the introduction of a super-synapse, a single LP that manages axon $\rightarrow$ neuron communication. This design reduces the number of LPs by n^2, where n is the number of neurons in a neurosynaptic core. For example, using the standard TrueNorth core size of 256 neurons, *NeMo-ES* creates 65,536 synapse LPs per core, while *NeMo-SS* creates 1 synapse LP per core. This reduction in the number of LPs reduces memory constraints, and allows for larger simulations, as demonstrated later in the results section.

Further enhancements were made to the implementation of *NeMo-ES*, including optimizing of the in-memory representation of neurons and providing enhanced I/O options for simulation results.

The major design change in *NeMo-SS* versus *NeMo-ES* is the synapse grid design. *NeMo-SS* replaces individual synapse LPs with a single LP, here called a super-synapse. In Figure 5(a), *NeMo-SS*'s super-synapse layout is shown. The super-synapse represents the grid of synapse LPs used in *NeMo-ES*.

This super-synapse folds the synapse event fanout in *NeMo-SS* into a series of "heartbeat" messages. Compared with the event flow of *NeMo-ES*, shown in Figure 3, *NeMo-SS*'s event flow uses a heartbeat style message to manage events sent to the neurons. Figure 5(b) shows an example event chain in a trivial neurosynaptic core. For the purposes of illustration, Figure 3 shows a two neuron core receiving two outside messages.

To manage the *"super-synapse's"* heartbeat and neuron messages, we designed two time-steps: ν, the internal clock time of a neurosynaptic core and t, the simulated hardware time. As with *NeMo*'s fanout messages, ϵ is a small value added to scheduled event times that prevents collisions. These values are defined as:

t_n is the current simulated neurosynaptic hardware tick n. Each complete cycle of the hardware

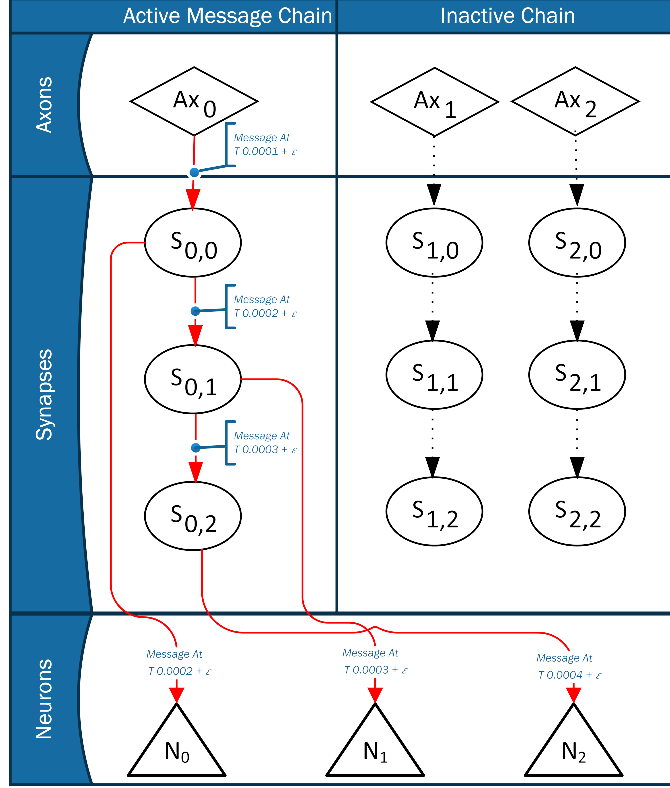


Figure 4: Example event chain in NeMo with 3 neurons per neurosynaptic core. In this diagram, an event is received at Axon 0 within a core at time t. At $t + 0.0001 + \varepsilon$ Axon 0 sends a message to Synapse 0,0. Synapse 0,0 then sends a message at $t + 0.0002 + \varepsilon$ to Neuron 0 and Synapse 0,1. Synapse 0,1 sends messages to Neuron 1 and Synapse 0,2 at $t + 0.0003 + \varepsilon$. Synapse 0,2 then sends a message to Neuron 2 at $t + 0.0004 + \varepsilon$. If no messages are received on Axon 1 and 2, no messages are sent. Neurons will send outgoing spike messages, if applicable, at $t + 1.0 + \varepsilon$.

is considered one tick. Within ROSS, each event has a simulation time value, represented by a floating point number, associated with it. *NeMo* and *NeMo-SS* use this to define the hardware time as:

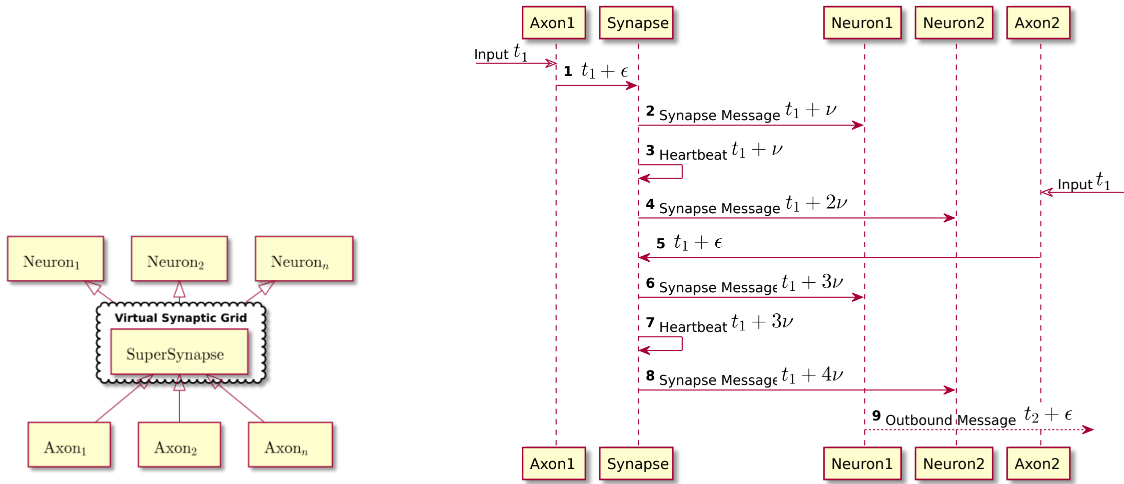
$$t_n = \lfloor t \rfloor$$

ϵ represents a small "jitter" value. This is a similar value to the one described in the implementation of *NeMo-ES*.

ν is a small value that represents simulation steps that must be done within each hardware tick, t_n. This value is based on the number of neurons in each core. *NeMo-SS* calculates ν as:

$$\nu = \frac{1}{2 * \text{Neurons Per Core}} + \epsilon$$

Figure 5(a) shows the event flow of a two neuron core inside *NeMo-SS*. For illustration, the core receives two spike events, both occurring at $t_1 + \epsilon$. We define our jitter value, ϵ, such that $\epsilon \ll \nu$. Upon receiving the first axon message, the synapse LP sends a message to neuron 1 at time $t_1 + \nu$ and a heartbeat message at time $t_1 + \nu$. When the synapse receives this heartbeat message, it sends



(a) The *NeMo-SS* Core Layout Showing a Virtual Synapse Grid.

(b) *NeMo-SS* Event Implementation.

Figure 5: Figure (a) shows the *NeMo-SS* core layout with the super synapse virtual synaptic grid. Figure (b) shows a trivial neurosynaptic core with 2 neurons. Input spikes arrive at the start of the current big tick, t_1 with a jitter value added. Jitter, represented by ϵ, is added to every communication to prevent message collision. Some messages have both a counter and jitter applied to the time stamp, represented by ν

a message to neuron 2, scheduled for time $t_1 + 2\nu$. At this point, axon 2's message is received by the synapse, generating a new neuron message at $t_1 + 3\nu$ and a new heartbeat message at $t_1 + 3\nu$. When the synapse receives this second heartbeat message, a new neuron event is scheduled at $t_1 + 4\nu$. In this example, we assume that neuron 1 has a high enough voltage to spike, so it schedules an outbound neuron message at $t_2 + \epsilon$.

By separating the timestamps used by the synapse in this way, we observe that the order in which events are processed by the super-synapse do not affect the outcome of the simulation. Neuron integration can occur in any order in between the simulation ticks without changing the determinism of the model.

With *NeMo-SS*, we implemented a reverse computation technique for managing rollback events that affect neuron state. When a neuron in *NeMo-SS* receives a reverse message, the neuron applies a reverse integration function, along with a reverse leak function. These reverse computation methods were developed for both the LIF and TNLIF models, even though only the TNLIF model was used for benchmarking purposes. In general, spiking neuron integration, leak, and reset functions tend to be good candidates for reverse computation. Integration and leak functions are generally linear, and reset functions are almost always deterministic. *NeMo-SS* maintains the ability to perform incremental state swaps as well. Using state swapping will allow the quick addition of new models without devising reverse computation methods, as well as the addition of non-linear or other complex neuron functions.

An important feature of *NeMo* is the ability to not only simulate the TNLIF neuron model but virtually any neuromorphic model. This could be complex compute-on-synapse or compute-on-axon models where synapse and axon LPs act as more than just message carriers but also play parts on

computation. It can also be as simple as a basic feed forward ANN.

In order to show this feature, the more simple Leaky Integrate and Fire (LIF) model described in [87] was implemented to run on *NeMo*. The implementation of this model, while rudimentary, shows the capability of *NeMo*: that it is general and flexible.

The model is generally simpler than the TNLIF model in most respects but it has distinctive differences and flexibility. The TNLIF model's behavior is defined and restricted by hardware that the general LIF model (or any other conceivable model) is not bound to. For example, the data types that define the synaptic weights can be 64-bit signed floating point values instead of integers. In addition, there is no specified limit on the number of unique axon types that each neuron can store as potential inputs whereas the TNLIF model is limited to four types.

The significance of these differences is that *NeMo* is able to show scalable performance of different models ranging from simple to complex on arbitrary neuromorphic systems – including those that have not yet been implemented in hardware.

3.2 Classification of Supercomputer Failures Using TrueNorth

The ability of machines to *learn from data* forms the crux of the field of Machine Learning (ML) [102] Specifically, ML targets three problems: (i) Supervised Learning or Classification; (ii) Unsupervised Learning or Clustering; and (iii) Reinforcement Learning. Supervised Learning refers to learning from data when it is already available in a labeled format. Unsupervised Learning or Clustering refers to learning from data when it is not available in labeled format. Reinforcement Learning refers to learning by doing tasks, where a reward is awarded to the machine if it performs the task correctly and the aim of the machine is to maximize the reward.

In the field of machine learning, there is a plethora of learning algorithms and techniques to choose from: decision trees, rule-based learning, support vector machines, bayesian learning, evolutionary algorithms [105, 165] etc. Deep Learning (DL) also is one of the techniques that enable machine learning. While most of ML techniques target a specific problem (i.e. Classification, Clustering or Reinforcement Learning), DL can tackle all three problems. ML can be thought of as a class of problems in Artificial Intelligence (AI), and DL can be thought of as a technique that enables ML, and in turn, also enables the larger field of AI [28].

3.2.1 The RAS Data

In our study, we used the Reliability, Availability and Serviceability (RAS) logs from the IBM Blue Gene/L supercomputer that was stationed at the Center for Computational Innovation (CCI) at the Rensselaer Polytechnic Institute (RPI). RAS logs from Blue Gene systems have been used to model, classify and predict supercomputer failures in the literature [173]. Our Blue Gene/L system consisted of 16,384 compute nodes, each node having two IBM 440 PowerPC processors. Each rack is organized into two midplanes each with 512 compute nodes or 1024 compute nodes total per rack. The network was a 3-D torus topology. The RAS log files span over one year worth of the machine's operational life. After cleaning and preprocessing the data, a typical log entry looks like the one shown in Table 2. It documents data corresponding to sixteen variables. Table 2 also describes what each variable means, whether it is categorical or not and what a sample value for the variable looks like. The Severity feature in Table 2 denotes the severity of the log messages. It can contain one of five values: `INFORMATION`, `WARNING`, `ERROR`, `SEVERE` or `FATAL`. A `FATAL` log entry corresponds

Table 2: Description of Variables in a Typical RAS Log Entry.

Variable Name	**Variable Description**	**Categorical?**	**Sample Value**
RECID	Record identification number of log entry	No	1
FACILITY	Facility in the machine that registered the log entry	Yes	APP
NODE	Virtual location of the node	Yes	236
SERIAL-NUMBER	Serial number of the component	No	7.8E+56
YEAR	Time variable: year	No	2006
MONTH	Time variable: month	No	12
DAY	Time variable: day	No	6
HOUR	Time variable: hour	No	16
MINUTE	Time variable: min.	No	48
SECOND	Time variable: sec.	No	55
MICRO-SECOND	Time variable: microsecond	No	7849
LOC_RACK	Rack location	Yes	0
LOC_MID-PLANE	Midplane location	Yes	1
LOC_NODE	Node location	Yes	8
LOC_LINE	Line location	Yes	0
SEVERITY	Severity of log entry	Yes	1

to occurrence of a node failure. We want to classify the RAS logs into these five severity levels using the remaining fifteen features.

Problem Description: Use neuromorphic computing, machine learning and deep learning techniques to classify RAS logs into five Severity levels using fifteen features, which contain information pertaining to origin of the log entry (Record ID, Facility, Virtual Node Address), serial number of the component that initiated the log entry, temporal information (Year, Month, Day, Hour, Minute, Second and Microsecond) and spatial information (Rack, Midplane, Node and Line).

Note that this is a time series dataset in the sense that the decision to label a node as failing or not at a certain point of time may depend on the log entries that the node has registered till that time. If it has registered a large number of `ERROR` or `SEVERE` messages, it is likely to fail soon and it would be advisable to raise a red flag saying the node is not healthy for computation. This finding was determined in a prior study by Hacker et al [77] using the same RAS log data. At no point do we want the node to register a `FATAL` log entry because that would mean that a node failure has occurred.

The raw data that we used consisted of 151, 766 log entries. The first task after obtaining the data was to clean it. This comprised of removing data entries that contained values that were not relevant to the study. For instance, some of the data entries contained 'NaN' entries and irrelevant strings like "????????" which were removed. We ended up discarding only a small portion (1939 out of 151, 766 data entries or 1.28%) of the data during this process. After cleaning the data, we were left with 149,827 log entries. We converted the data into a numeric format by assigning numeric values to categorical variables like Facility, Rack Location, Midplane Location etc. Next, we only kept the variables that were relevant to the task of node failure classification and discarded

the rest. Finally, we were left with fifteen features and 149,827 data points.

The next task was to split the dataset into Exploratory Data Analysis (EDA) set, training set and test set. For this, we kept 10% of the data in the EDA set, 70% of the data in the training set and 20% of the data in the test set. Although our dataset is a time-series dataset, we did not want to be restricted to sequential classification techniques, which are generally used to analyze such data (for e.g. Recurrent Neural Networks), but also wanted to leverage non-sequential techniques (for e.g. Deep Neural Networks, K-Nearest Neighbors, Support Vector Machines). Since the non-sequential classification techniques are chronologically independent we picked the data points in each set uniformly at random. After this, each dataset was chronologically sorted in order to preserve time-dependence for sequential classification techniques.

3.2.2 Using IBM TrueNorth

The IBM TrueNorth Neurosynaptic System is used to execute a Spiking Neural Network (SNN) model which classifies the previously described failure data. A Spiking Neural Network (SNN) is a type of neural network in the discrete domain, where each neuron in a layer may send discrete signal spikes (for e.g. voltage in analog SNN, or a digital packet in digital SNN like TrueNorth) to neurons in the next layer. This is unlike traditional Deep Neural Networks (DNNs) where all neurons send all data at all times to the next layer, regardless of their input, or stimuli.

The software environment used for SNNs is EEDN, which is a framework built in conjunction with MatConvNet (MATLAB) [160]. To compare the performance of SNN with other techniques, we chose three machine learning (ML) techniques – Logistic Regression, K-Nearest Neighbors (KNN) and Support Vector Machines (SVM) – and two deep learning (DL) techniques – Deep Neural Networks (DNN) and Recurrent Neural Networks (RNN). We used the Scikit-Learn [135] and TensorFlow [10] libraries in Python for ML and DL techniques respectively. All the above models were run on a machine that had 32 cores of two-way multi-threaded Intel Xeon CPUs running at 2.60 GHz, three NVIDIA GPUs (GeForce GTX 1080 Titan, GeForce GTX 950 and GeForce GTX 670), 112 GB DIMM Synchronous RAM, 32 KB L1 cache, 256 KB L2 cache and 20 MB L3 cache.

The IBM TrueNorth Neurosynaptic System can cater to any application having streaming unstructured data that needs to be processed. It accepts input values that are 8-bit integers ($[0, \ldots, 255]$). For a computer vision application, these correspond to the red-green-blue (RGB) values that would be seen in an image. Furthermore, it supports Convolutional Neural Networks (CNN), which are the *de facto* deep learning models used in computer vision tasks. Each input data point must be reshaped and presented to the TrueNorth SNN as a 3-D array. This is because, the TrueNorth system lends itself very naturally to image data, as compared to any other data. Continuing the parallel to computer vision tasks, this corresponds to an image being read as a 3-D array (Image Width $\times$ Image Height $\times$ Number of Channels) in a CNN. Since images are 3-D arrays usually, TrueNorth expects its input data to be a 3-D array as well.

Our input data points were 15-dimensional vectors. So, in order to make them TrueNorth compatible, we first reshaped them into 'images' of shape $1 \times 1 \times 15$ and later on added two layers of zero padding, which were seen to work best for our data. Thus, the shape of each of our data points was $5 \times 5 \times 15$. This data reshaping, was the only way to run numeric datasets on the TrueNorth chip. It must be noted that this reshaping of data is simply a workaround, and to the best of our knowledge, does not have any learning advantages.

Developing neural network applications with TrueNorth follows a standardized workflow. Devel-

Approved for Public Release; Distribution Unlimited.

```
LAYER TYPES
I: Image
P: Preprocessing layer
C: Convolution layer
D: Dropout layer

       Layer Size                 Patch                    TN Cores   Patch-in-Image
Lyr    Row x Col x Ftr  (Grp)     Str  Row Col Ffr         Base+Splt  Str   Row x Col
I      1   x 1   x 15

P1     5   x 5   x 64   (1  )     1   [1 x 1 x 15 ]        0   +0     1    [1   x 1  ]
C2     5   x 5   x 32   (1  )     1   [1 x 1 x 64 ]        15  +12    1    [1   x 1  ]
C3     5   x 5   x 32   (1  )     1   [1 x 1 x 32 ]        9   +0     1    [1   x 1  ]
C4     5   x 5   x 16   (1  )     1   [1 x 1 x 32 ]        9   +0     1    [1   x 1  ]
```

Figure 6: Network configuration of TrueNorth SNN

opment is done over six phases: Dataset, Preprocess, Train, Build, Test and Application. In the Dataset phase, we converted the data from the raw Comma Separated Values (CSV) format into the Lightening Memory-Mapped Database (LMDB) format. In the Preprocess phase, we scaled the data down to 8-bit range ($0 - 255$) by normalizing and then rounding to the nearest integer. In doing so, we did not incur a significant loss of information because almost all of our features had values in the 8-bit integer range to begin with. The features that did not have values in this range were scaled down. However, as can be seen from Section 4, any potential loss of information arising from this operation did not seem to compromise our results.

The Train phase comprised of designing a SNN in EEDN and training it on the data obtained from the Preprocess phase. Figure 6 shows the configuration of our SNN in a terminal window output format and Figure 7 presents it in a pictorial format. Our network comprised of five layers, of which the first was the input layer (`I`), followed by the transduction layer (`P1`) and subsequently three convolutional hidden layers (`C2`–`C4`). The $5 \times 5 \times 15$ input data was fed to the input layer, and was encoded into spikes in the transduction layer. All in all, our SNN used 45 TrueNorth cores. With 256 neurons per core, this gives an upper bound of $11,520$ TrueNorth neurons in total. Note that $11,520$ is just an *upper bound* – it does not mean that our model used those many neurons. It was not possible to get the exact neuron count using the TrueNorth development kit at our disposal. Furthermore, note that spiking neurons in TrueNorth are hardware neurons and are different from the software neurons used in deep learning frameworks like TensorFlow. In general, a software neuron corresponds to multiple hardware neurons. The exact mapping is defined by the encoding algorithm, which in this case was not available in the TrueNorth development kit.

The Build phase comprised of generating the binaries that would be deployed on the TrueNorth chip. Finally, during the Test phase, we deployed our SNN model on the TrueNorth chip and obtained the test results. The Application phase runs an application on the TrueNorth chip in real time along with visualizations, analysis etc. In context of this work, it could potentially mean that the model running on the TrueNorth chip accepts a live stream of error data generated by the supercomputer and can predict failures ahead of time.

We now briefly describe all the machine learning and deep learning techniques that we have used. For any classification problem, Logistic Regression serves as the first step of analysis as it is a simple linear model and does not overfit the data easily unlike a complex model. We ran Logistic Regression in Python using the TensorFlow library. The second technique that we applied was K-Nearest Neighbors. We chose two values of K: $K = 3$, and $K = 289$. The former value has been known to perform well empirically and the latter value, which is the square root of number of data points in the training set (i.e. $\sqrt{83,903} \approx 289$), is known to perform well theoretically. The third technique that we used was Support Vector Machine (SVM) as it produces robust classifiers and can model nonlinear data as well using the kernel trick. To run SVM and KNN, we used the

Approved for Public Release; Distribution Unlimited.

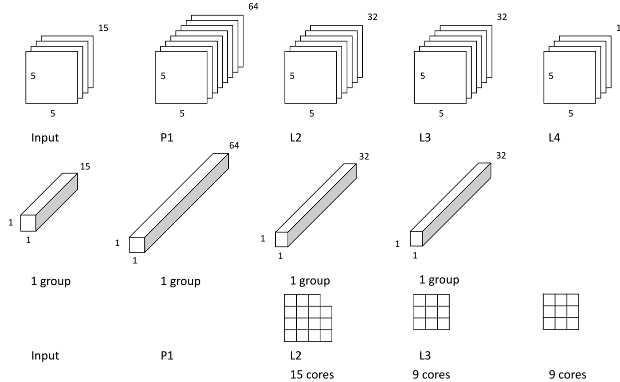


Figure 7: Visualization of TrueNorth SNN

Scikit-Learn library in Python.

The two deep learning techniques that we used were Deep Neural Networks (DNN) and Recurrent Neural Networks (RNN). These were run using the TensorFlow library in Python. Deep Neural Networks (DNN) are the traditional deep learning model where each neuron is a perceptron. Recurrent Neural Networks (RNN) are known to perform well on sequential data. So, RNNs are the go to model for speech recognition and natural language processing. Since our data is a time series data, it is also sequential in nature and that was our motivation to pick RNN as one of our techniques.

3.3 Durango – A Hybrid System Performance Modeling Framework

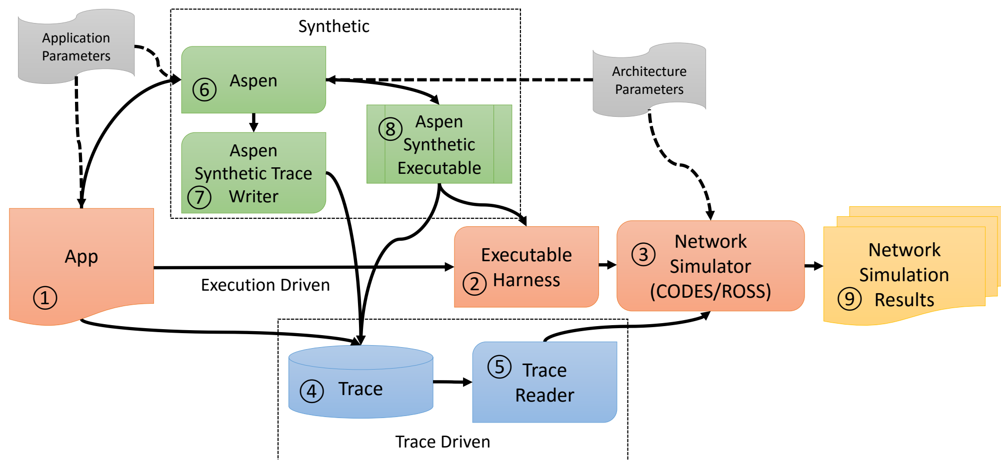


Figure 8: *Durango* overview.

To motivate *Durango*, we refer to Figure 8, which illustrates the multiple workflows for generating a simulation workload. In this paper, we consider four scenarios when capturing the characteristics of the application to be simulated. In the first scenario, we simply execute the application, capturing the communication events, and transferring execution control between the application and simulation as appropriate (path 1-2-3-9 in Figure 8). The application is built as it normally would be, but the communication events are intercepted by the simulator and then simulated on

the theoretical network. This method is relatively straightforward and easy to perform, but it has the disadvantage that it uses considerable resources in terms of execution time and memory.

In the second scenario, we use a tracing tool, such as DUMPI, to capture an event trace of all the communicating tasks and communication events in the application. This trace is digested by the simulator (path 1-4-5-3-9 in Figure 8) and the trace information is typically stored in a huge file. Moreover, all the parameters of the trace are fixed at capture time; that is, the architect cannot change the problem size or the number of processors after the trace has been created.

In the third scenario, we use synthetic communication patterns as a proxy for the application communication patterns (path 1-6-7-4-5-3-9 in Figure 8). In the fourth scenario, the proxy trace generator is glued directly into the simulator (path 1-6-8-2-3-9 in Figure 8). These last two approaches are the focus of this paper and are combined into a system we call *Durango*.

3.3.1 Aspen Overview

Aspen is a domain-specific language designed for analytical performance modeling [154, 161]. The models represented in Aspen comprise application models and abstract machine models. The machine models describe the hierarchy of a machine as well as speeds and feeds of its components for processing computation and communication. Application models contain descriptions of resource usage of an algorithm (such as the computation and communication requirements) and control flow (iteration, sequential and parallel dependencies, and kernel nesting).

The COMPASS framework described in [109] generates a parameterizable performance model from a target application's source code using OpenARC [110] for automated static analysis and then evaluates this model using various performance prediction techniques available in Aspen. Prior to using OpenARC, a small amount of manual annotation is required in order to specify regions of interest and declare parameters to be directly exposed in the generated Aspen application model. The Aspen control flow walker can dynamically instantiate values of parameters that cannot be determined statically. Generation of applications models for the LULESH proxy application and the matrix multiplication kernel used in the Durango research is described and validation of the Aspen resource usage and runtime predictions with measurements on a real system is given in [110]. Aspen currently combines a the throughput-based node performance model based on the Roofline model [164] with a simple latency plus bandwidth communication model, where computation and communication can be overlapped to a specified degree. Part of the motivation of the Durango research is to achieve more accurate communication modeling than is possible using Aspen's analytical methods.

Aspen, as an analytical tool, was initially designed to compute symbolic results for analysis queries, such as the number of floating-point operations at a given problem size or the performance of a kernel as a function of bus bandwidth. However, the expansion of Aspen's capabilities has allowed for more complex uses; while an Aspen model is not source code and cannot be executed per se, the representation of a control flow is sufficiently rich to allow Aspen-based tools to traverse an application model with the same control flow as with the original application. We build this new capability to provide synthetic communication trace generation along with direct integration in *Durango*.

3.3.2 Durango Approach

In contrast to the first two scenarios described above, our new approach allows the architect to create a parameterized model of an application, and then instantiate the application with specific parameters at simulation time. The potential benefit of this approach is multifaceted. First, the model is malleable: the user can easily change the application and architecture parameters. In our approach, a model can be instantiated for 16 MPI tasks as well as 1M MPI tasks. Second, the models are compact, typically being only a dozen lines for miniapplications and up to a few hundred lines for real applications. Third, the *Durango* models include computational and I/O events in addition to the communication events so that computation and I/O demands can change in concert with communication parameters. This approach has multiple benefits: the synthetic workload can be generated dynamically as necessary with the required architecture and application parameters; the description is compact; and the Aspen model can be as detailed or as sparse as required, including computation, communication, I/O, and other important events.

More specifically, Durango requires that the user create a parameterized Aspen model (6) of the application (1) and then use the Aspen model to create a synthetic workload, either by generating a compatible trace directly (7) or by generating a synthetic MPI application (8) that can then be traced (4-5-3-9 in Figure 8) or directly included in the simulation executable that avoids the need to perform expensive read operations of large trace datasets.

3.3.3 Representing Communication in Aspen

Resource descriptions in Aspen are user-defined. There are common conventions such as `flops`, `loads`, and `stores` for floating point operations and memory traffic, but these must merely match what is defined in the abstract machine model for Aspen to be able to return predictive costs such as runtimes and power consumption.

A commonly used resource for MPI communication is `messages`, but this results in a simplistic mapping. Extensions via traits enable more expressive message passing patterns. For example, a trait description such as "as positive_x" can direct an Aspen-based tool to generate MPI calls if given a regular processor decomposition. However, these trait descriptions are low-level, however, and require significant effort from modelers to achieve communication patterns like 3D nearest-neighbor; for example, requiring up to 26 separate `message` resource calls to generate message traffic for each neighboring processor.

Listing 1: Example Aspen model with communication

```
model example
{
  param nelem = 23
  param wordsize = 8

  kernel main
  {
    execute{flops [8*nelem^3] from Domain as
            simd}
    execute{comm [1] of size [word] as allreduce,
            min}
```

```
    execute{comm [nelem] of size [3*word] as nn3d,
            face}
  }
}
```

Instead, we created a new resource convention, `comm`, that represents communication patterns at a more semantic level. An abstract machine model can still easily interpret these in the same manner as the simpler `message` construct for the purposes of determining costs, but we are now free to interpret this new `comm` resource at a higher level in an Aspen tool in order to generate synthetic patterns for executables and traces.

Listing 1 shows an example Aspen application model using this new `comm` resource. Note how our new Aspen `comm` resource corresponds to popular synthetic communication patterns such as nearest-neighbor but can be customized for application kernels as well. In particular, the pattern is listed in the traits for the resource, and any options are listed as additional traits. For example, line 10 has the *allreduce* collective pattern with a *min* operation, and these are represented in the traits as `as allreduce, min`. Combined with the quantity and size, that resource description is sufficient to generate a synthetic MPI call. Line 12 has the nearest-neighbor 3D point-to-point pattern (`nn3d`) for a domain size of `nelelm` with three double-precision fields (`3*word`) and communication only along the six faces of each domain (`face`).

3.3.4 Synthetic Program Execution

Aspen models are not designed to be "executed" per se but have control flow such as iteration loops, parallel maps, and kernel invocations. This control flow information was sufficient for us to create a "walker" tool that uses the Aspen library to traverse application models in control flow order, as if they are executable programs.

The tool we created supports two major types of traversal: implicit and explicit. The difference typically manifests in control structures such as iteration. For example, if kernel K is called from within an iteration control with count 7, the explicit traversal will descend into the kernel call seven times, while the implicit traversal will descend into the kernel call only once but knows that it is executing with multiplicity seven with a sequential dependence.

Some types of analyses are amenable to implicit traversal. For counting floating-point operations, we can typically count how many operations are in kernel K in our example above and multiply by seven. This is not possible in all cases, however, for example, if floating point operation counts vary stochastically, we must use explicit traversal and sample values at each iteration. In the case of synthetic trace generation, both implicit and explicit traversal have their place.

In typical analyses, implicit traveral will be faster because it performs less analysis and accumulates results in bulk to provide the correct answer. In the context of generating a synthetic MPI executable, in the implicit mode we can output a C `for` loop containing the appropriate MPI calls, while the explicit mode would require generating many C copies of the same function calls. If Aspen were to hook up directly to a simulator, we would, by necessity, switch to explicit traversal because Aspen would need to feed the simulator every MPI call generated by the synthetic program.

Listing 2: Aspen model with control flow and communication.

```
model mpitest {
  kernel main {
```

```
    iterate [10] {
      execute { comm [2] of size [word] as
                 allreduce , min }
    }
  }
}
```

Listing 3: Source generated by model in Listing 2 using implicit traversal.

```
for (int loopctr=0; loopctr<10; loopctr++)
  {
    int nwords=2;
    std::vector<float> sendvec(nwords, rank*1.0f);
    std::vector<float> recvvec(nwords);
    MPI_Allreduce(&(sendvec[0]), &(recvvec[0]),
                  nwords, MPI_FLOAT,MPI_MIN,
                  MPI_COMM_WORLD);
  }
```

3.3.5 *Durango*-Instantiated Executable

Our first option for generating a synthetic workload is to instantiate an MPI-based source code file that captures the parameters and patterns of the application via the Aspen model. Listing 2 shows an example of a simple Aspen model with an iteration loop along with a single communication pattern — in this case, an Allreduce. Listing 3 shows the output code (minus boilerplate) for this small Aspen model. Note that this output was captured in implicit mode; in explicit mode we would get ten copies of the body of the `for` loop, instead of a `for` loop with a count of ten.

After generating the source code instantiating the model's control flow and communication patterns, we can compile and execute it, optionally capturing a trace for study as output from the *Durango* tool.

3.3.6 CODES: An Extreme-Scale Systems Modeling and Simulation Framework

To demonstrate our new *Durango*'s methodology and functionality, we have integrated *Durango* with a popular massively parallel simulation system for interconnection networks, CODES/ROSS.

CODES enables the design-space exploration of HPC networks and storage systems with the help of scalable discrete-event simulations of interconnection networks and storage systems. CODES uses ROSS as its underlying parallel discrete-event simulation framework, which enables efficient and scalable network models thanks to the optimistic event scheduling capability of ROSS [127]. The core object within a ROSS model is a *logical process (LP)*, which models some distinct component of a network such as a terminal or router. Simulation time is advanced by LPs exchanging time-stamped event messages. The optimistic parallel synchronization approach used by ROSS guarantees that events are processed in time stamp order.

CODES supports high-fidelity network models for dragonfly, torus, and SlimFly interconnect topologies. It uses an abstraction layer on top of the network models that allows users to conveniently plugin multiple network topologies while making minimal changes to their simulation

Approved for Public Release; Distribution Unlimited.

code. The network topologies are simulated at a packet-level detail with congestion control being modeled through a credit-based flow control methodology on the virtual channels. The dragonfly network model in CODES is built on the high-radix, low-cost network configuration proposed by Kim et al. [99]. It models four forms of routing algorithms: minimal, non-minimal, adaptive, and progressive adaptive routings. Multiple virtual channels are used for deadlock avoidance with different routing algorithms. The dragonfly model has been validated against the *Booksim* interconnect simulator using synthetic traffic patterns [125]. The torus network model is inspired by the Blue Gene architecture. It uses a bubble-escape virtual channel for deadlock avoidance with deterministic dimension-order routing. Validation of the torus model has been carried out against the Blue Gene/P and Blue Gene/Q architectures [126]. The SlimFly network model is based on another high-radix network topology proposed by Besta and Hoefler to reduce the network cost and diameter [30]. The CODES SlimFly simulation results were validated against the simulator by Besta and Hoefler.

The CODES network models report detailed statistics about network performance for each simulated network node and router. Using metrics such as the average number of hops traversed, average packet latency, data transmitted, and number of packets completed. Detailed statistics are reported at the network link level, including the amount of data transmitted at each network link and the time that the link gets saturated during the simulation. These metrics can be used to get detailed insight into the network performance with different workloads.

To replay the MPI operations on CODES network models, one needs a mechanism that avoids transmitting back to back MPI send messages on the network. Figure 9 shows how the MPI simulation layer interacts with the CODES network abstraction layer to replay the workload operations on top of the simulated networks. The MPI simulation layer in CODES digests the MPI operations from the workloads and simulates them on top of the network models. It tracks the queues of MPI sends and receives, matches sends with the receives, and simulates MPI wait and ait-all operations. This functionality is vital for maintaining the correct causality order of MPI operations coming from the traces.

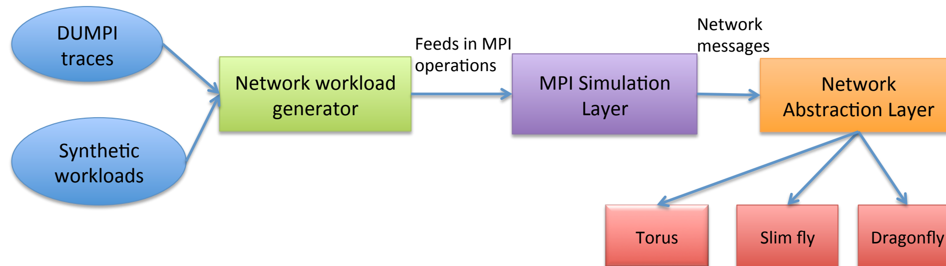


Figure 9: CODES network simulations.

3.3.7 Durango Direct Integration: Aspen with CODES

At the core of the Durango direct integration approach is a CODES discrete-event model that drives a network simulation component, coupled to an Aspen-based runtime estimator for parallel applications. The model defines Aspen Server Logical Processes (Aspen LPs), which are the entities

in the simulation responsible for driving the creation of network traffic and for performing Aspen-related computations. Each Aspen LP is paired to a corresponding CODES network terminal LP to facilitate communication with the network layer within the CODES model. The CODES network LPs are generated and organized based on the network topology chosen for the current runtime estimation. When Aspen is called through the Aspen Server LPs, the estimation parameters are passed from the primary configuration file for Durango. In return, Aspen returns the runtime estimate based on the application and the machine details specified by the configuration file.

Listing 4: Aspen LP kickoff event handler.

```
static void handle_kickoff_event(
  aspen_svr_state * ns,
  tw_bf * b,
  aspen_svr_msg * m,
  tw_lp * lp)
{
    int dest_id;
    aspen_svr_msg m_local;
    aspen_svr_msg m_remote;

    m_local.aspen_svr_event_type = LOCAL;
    m_local.src = lp->gid;
    m_remote.aspen_svr_event_type = REQ;
    m_remote.src = lp->gid;

    /* record when transfers started
    // on this server */
    ns->start_ts = tw_now(lp);

    dest_id = get_next_server(lp);

    model_net_event(net_id, "test",
        dest_id, payload_sz, 0,
        sizeof(aspen_svr_msg),
        (const void*)&m_remote,
        sizeof(aspen_svr_msg),
        (const void*)&m_local, lp);
    ns->msg_sent_count++;
}
```

Under the current direct integration mode, *Durango* simulates a given application in two-step rounds of network simulation and computation runtime estimation, handled by the internal CODES model and Aspen respectively.

Figure 10 illustrates the runtime of a network-computation round in greater detail. First a CODES network simulation is executed, and then the 0th Aspen Server LP, labeled "Aspen Master," reduces and processes all network data. Once the data have been reduced, the Master LP passes control of the simulation to the Aspen runtime in order to estimate the computation cost for the

Approved for Public Release; Distribution Unlimited.

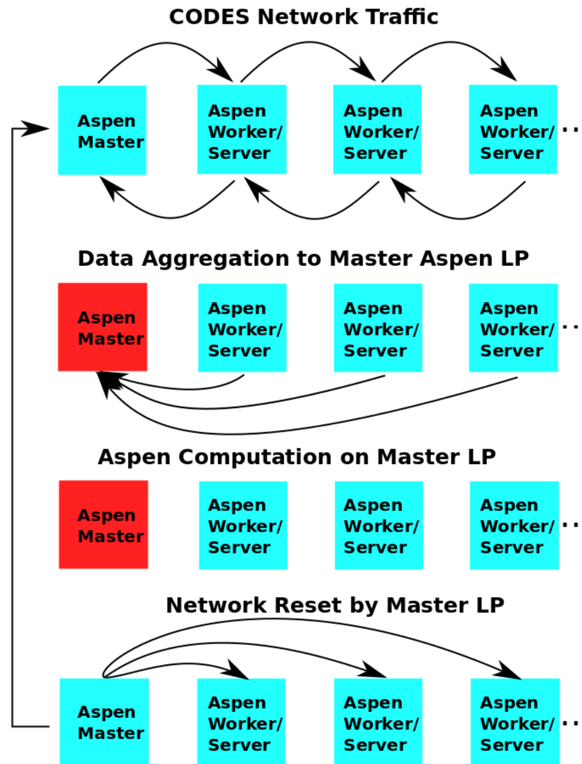


Figure 10: Durango hybrid runtime.

current round. Aspen returns a runtime estimate, and then returns control to the Master LP, which resumes subsequent simulator rounds by sending network restart events to all other Aspen LPs.

Listing 5: Aspen computation event handler.

```
static void handle_computation_event(
    aspen_svr_state * ns,
    tw_bf * b,
    aspen_svr_msg * m,
    tw_lp * lp)
{
  // Compute overall network time elapsed:
  delta += end_global - start_global;

  /* Call ASPEN framework to estimate
  // computation cost: */
  delta += runtimeCalc(Aspen_App_Path[
    roundsExecuted - computationRollbacks],
    Aspen_Mach_Path,
    Aspen_Socket[roundsExecuted -
```

```
    computationRollbacks]);

  // Save totalRuntime and then update it:
  m->end_ts = totalRuntime;
  totalRuntime += delta;

  // Increment number of rounds executed:
  roundsExecuted ++;
  .
  .
  .
}
```

Each CODES network simulation phase begins with "kickoff" events sent by the Master LP to all Aspen Server LPs, itself included. The code for handling kickoff events is shown in Listing 4. In response to kickoff events, Aspen LPs record the current simulation time and use `get_next_server(lp)`, a mapping function that uses the underlying CODES API for organizing the logical processes to retrieve the identity of the LP that they will communicate with for the duration of the network-computation round. Depending on the network traffic type specified in the configuration file, `get_next_ser-ver(lp)` returns a different ID. If nearest neighbor is specified, for example, then the next logical process ID that corresponds to an Aspen LP is returned. This is an important distinction, since some of the logical processes in the simulation serve other purposes, such as network terminals, rather than Aspen Servers. In the case of a random network traffic pattern, a random identity corresponding to any other Aspen LP is returned. This means that multiple Aspen LPs may communicate with one Aspen LP for the duration of the round, but it does not mean that they may communicate with themselves. The kickoff handler ends after the CODES API is used with `model_net_event(...)` to send a "ping" to the LP whose ID was returned by `get_next_server(lp)`.

Upon receipt of a "ping" request, Aspen LPs respond with an acknowledgment event message to the sender. For every "ping" and "ack" event that an Aspen LP receives, counters are incremented and saved in the logical process state. The request-acknowledgment interchange continues until the Aspen LP that initiated the exchange has sent the configured number of requests and received the correct number of acknowledgments. The Aspen Server that initiated the exchange records the simulation timestamp at which network communication ended with its counterpart.

With the first half of the network-computation round completed, the Aspen LPs cease driving network communications and send their start and end timestamps to the 0th Aspen LP. Note that the Aspen LPs could simply send their own total elapsed network communication time instead of the separate start and end values; however, then the overall longest communication time might not be accurately measured if the LP with the longest communication time begins simulation before some other LPs but also ends before other LPs that began their network conversations later. In all cases, the total network time elapsed is the difference between the globally earliest and globally latest start and end timestamps, respectively. As a result, the 0th LP is responsible for finding the longest time spent in the network phase overall and therefore keeps track only of the earliest and latest values it receives from all of the Aspen LPs, itself included. This process is handled by the `handle_data_event(...)` handler. Once all timestamps have been received, the Master LP

Approved for Public Release; Distribution Unlimited.

sends an "Aspen Computation" event to itself. The "Aspen Computation" event handler performs a function call to the Aspen simulation engine with paths to the application model and machine model sourced from the Durango configuration file. In Listing 5, an excerpt of the computation event handler, the total network runtime, is calculated. Then a call to Aspen is made, passing the parameters of the application and machine model as well as which compute socket to run on. The computation runtime is returned and added to the global running counter, marking the end of the network-computation round.

If only one round has been configured in the Durango configuration file, then the simulation will terminate with only one iteration of CODES network simulation and Aspen runtime computation estimation. Otherwise, the Master LP re-sends kickoff messages to each Aspen Server LP, signaling the start of a fresh network-computation round. Upon receiving the re-kickoff message, all Aspen Server LPs then find a new LP with which to communicate and start the next network simulation phase.

3.4 HPC Network Models

3.4.1 CODES Framework

Built on top of ROSS, the **CO-D**esign of multi-layer **E**xascale **S**torage and data-intensive systems (CODES) framework can be used to simulate storage [152] and HPC network systems. CODES helps to facilitate the use of HPC network workloads and simulate network communication in the context of discrete event simulations. CODES also provides a range of high-fidelity packet-level network models including Dragonfly [128], [131], Torus [129], Slim Fly [167], Fat-Tree [166], and analytical LogGP [20]. CODES also supports a variety of network and I/O workloads that can drive these high-fidelity models [130], [153]. Network workloads can be either synthetic traffic injection or application communication traces. CODES supports traditional CPU application traces collected using the DUMPI MPI tracing tool that is part of the structural simulation toolkit [142]. For this work, we focus on DUMPI application communication traces from the Design Forward program [55].

The DUMPI MPI tracing tool provides a utility for collecting all MPI point-to-point and collective operations executed by processes in the application. CODES' MPI simulation layer ingests these operations and replays them through the network models. This layer acts as a bridge between the network workload and interconnect model, and is responsible for maintaining the correct causality order between messages/events of the trace [130]. The traces in this work are run with compute times disabled. In this case, the collected compute times within each trace are ignored and packets are sent using their collected start times.

In addition to traditional CPU workloads, CODES now supports neuromorphic application traces. Details on the new neuromorphic workload replay capability as well as the available workloads are discussed later in this report.

Model Implementations: All network models discussed in this work follow the same overall design and implementation within the CODES framework. Each topology is integrated into a high level layer of abstraction called *model-net*. The model-net layer provides the convenience of selecting and configuring different network models at runtime as well as consistent integration of all network models with both synthetic and application trace workload generators. Since the simulation framework is built on the discrete-event method of computing, each topology implementation follows the same logical flow of function calls. Within the functions exist details specific to each topology

such as routing algorithms and number of virtual channels but the general outline of sending and receiving packets remains consistent. Furthermore, each of the models have been implemented to support reversible computation and optimistic execution of events. Each network model also incorporates similar techniques for flow control and deadlock avoidance.

Flow Control: To control network congestion and avoid dropping packets, credit-based flow control [104] methods are implemented at two different levels. Within each network model, each compute node and router maintains a buffer space to store packets needing to be injected into the network. The upstream router or node increments its credit counter when a packet is forwarded and decrements the counter when credits are received. The Downstream router or node sends a credit to the upstream router/node when a packet is forwarded and buffer space is available. At the model-net layer, we utilize an additional FIFO queue for flow control which represents the queuing of MPI and many higher-layer protocols.

To remove the possibility of head of line blocking associated with certain routing approaches, physical channels (ports on a router/switch) are broken into virtual channels [51]. Generally, we discretize our selection of virtual channels to the number of hops a message packet has taken. Credit messages are also transmitted to the sender using the same virtual channel used in the forward direction. In terms of the implementation, an *output_vc* variable is added to the compute node message state structure and initialized to 0 when a message is created. Each time a router sends a message, it sends the message on the *output_vc* virtual channel and increments *output_vc* so that the next router on the path will use the next corresponding VC.

Deadlock avoidance is obtained in our Slim Fly model with the use of virtual channels [51]. Following the approach in [31], we discretize our selection of virtual channels to the number of hops a message packet has taken. In other words, for every hop i that a message packet takes, when leaving a router, that packet uses the ith virtual channel. Packets that take a local route and have only one hop will always use $VC0$. Packets that take a global path (assuming minimal routing) will use $VC0$ for the first hop and then $VC1$ for the second hop. Thus, two VCs are needed for minimal routing. In the case of non-minimal routing such as valiant and adaptive routing, the number of virtual channels used is four, because the maximum possible number of hops in a packet's route is four. Credit messages are also transmitted to the sender using the same virtual channel used in the forward direction.

Discrete-Event Simulation: Each of the CODES network simulations follows the same overall computational procedure. After constructing and configuring the LPs that make up the compute nodes, routers, and MPI processes in the selected network, the dedicated MPI workload LPs (representing MPI processes allocated to compute nodes) generate and receive messages. In the case of synthetic workloads, the MPI LPs generate messages of a preset size at a rate determined by an input load variable. For application trace workloads, the MPI operations are read from file and placed in a queue allowing MPI LP processes to generate and replay the corresponding messages on the network. Messages generated by MPI workload LPs are then sent to their corresponding attached compute node LPs for injection into the network [127].

In the CODES network models, each LP represents one router, compute node, or simulated MPI workload process in the network. Each timestamped event represents either a network packet transferring through the network, or a message from an MPI workload process needing to be broken down into packets. Figure 11 shows the general structure and event-driven procedure for the Slim Fly network simulation, but the same general process is observed on each of the other models. In this figure, we are running the simulation on two physical cores with one MPI rank per core, resulting

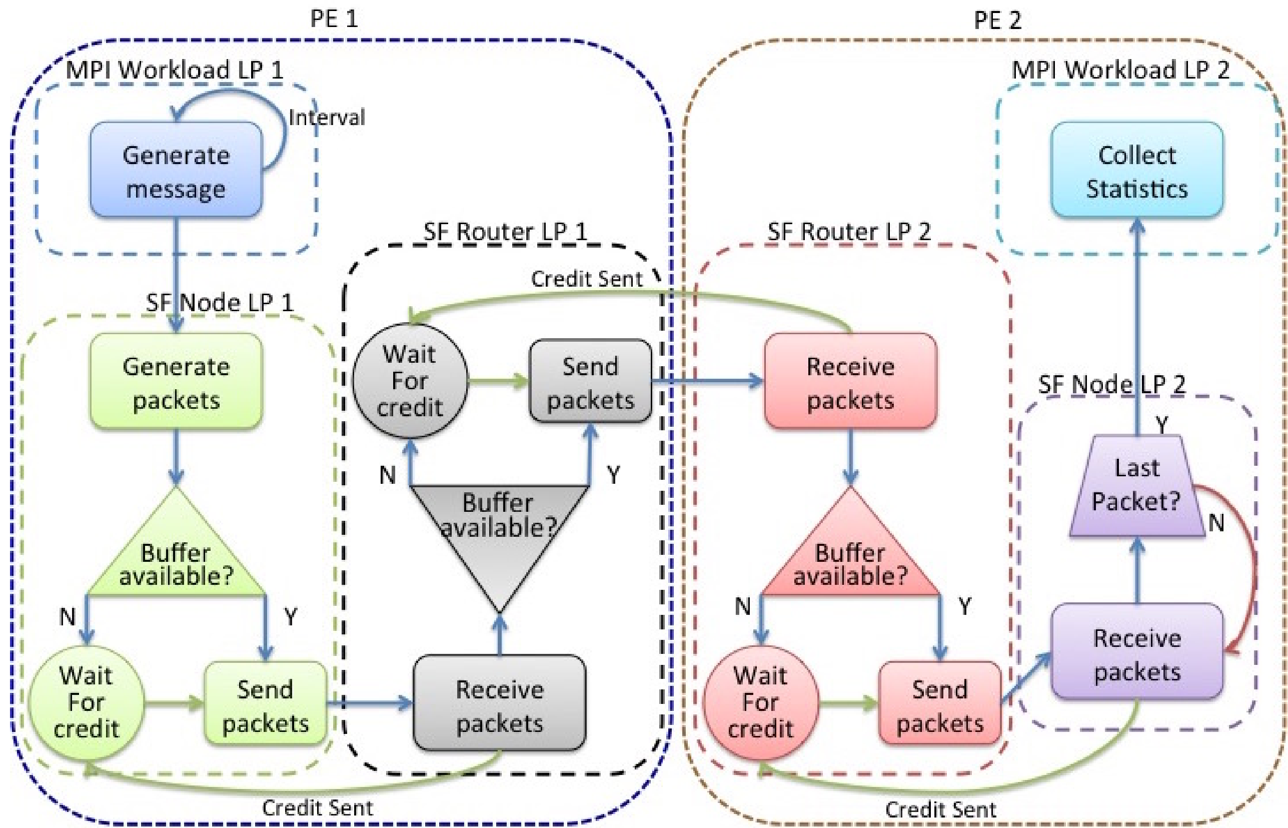


Figure 11: Diagram showing the general execution path of events in the Slim Fly specific parallel discrete-event simulation.

in two PEs. All LPs are distributed equally among the two PEs. Events/messages, represented by the arrows between LPs, are transferred between the LPs. For simplicity, only the LPs involved in the example are illustrated.

Upon receiving a message event, the compute node LP decomposes the message into packets and extracts the message destination. The compute node LP computes the next hop and corresponding output port for each packet using the selected routing algorithm (specific to each network). Prior to sending a packet, the sending node LP checks the occupancy of the selected port and virtual channel. If space exists, the packet is allocated, and a receive event is scheduled on the destination router with a time delay. This time delay incorporates the bandwidth and latency of the corresponding network link. If the buffer is full, the node LP follows credit-based flow control and must wait for a credit from the destination router to open up a space on the corresponding link.

In order to accurately analyze the network, various parameters and statistics are collected and stored in both the LPs and the event messages. These statistics include start and end times of packets on the network, average hops traversed by the packets, and the port occupancy of all routers.

Once a packet arrives at the router LP, a credit event is sent back to the sending LP to free up space in the sending LP's output buffer. The LP then extracts the destination node ID. The router LP determines the next hop and corresponding output port, once again using the routing algorithm specified. The router also follows the same credit-based flow control scheme as the compute node

LP.

After the packet reaches its destination node LP, the node waits for all packets belonging to that message to arrive before issuing a message arrival event on the destination workload LP. At this point, we can collect the statistics stored in the messages, for example, packet latency and number of hops traversed.

Related Work: Significant research has been done in simulating large-scale network interconnects. The IBM-ETH collaborative Slim Fly model used in this work for validation leverages the Venus simulation framework [95]. Venus is an OMNET++ based discrete-event network simulator. The Venus framework provides packet-level granularity with support for multiple network topologies [124] but it has not been shown to achieve the ability to execute extreme-scale networks of the size presented in this work.

The Structural Simulation Toolkit (SST) combines multiple discrete-event components to provide network simulation capabilities. SST supports many different hardware components in addition to interconnects such as memory and processors [142]. Groves et al. [76] have used SST to study power and performance of Dragonfly networks at the scale of 110K nodes. However, SST only uses a conservative distance-based approach for event scheduling while ROSS also provides optimistic scheduling. Wen et al. [163] use the SST simulation framework to investigate a new Dragonfly-like network implementation using silicon photonics to provide reconfigurable inter-group connectivity.

A popular alternative to discrete-event simulation is a cycle-based approach that advances time by clock cycles and has no representation of time within a clock cycle [64]. Booksim, a cycle-accurate network simulation framework, was used by the developers of the Slim Fly topology for studying network performance in response to synthetic traffic workloads [95]. Booksim also has support for multiple network topologies and has been shown to scale up to large-scale networks [19]. However, Booksim is a serial execution framework and has shown to have a slow execution time in comparison to CODES [128]. A cycle-accurate network simulator was also used to analyze the HyperX network developed by Ahn et al. [14] and shown to provide strong network performance for extreme-scale networks on the order of 100,000 nodes. Grossman et al. [75] use a cycle-based simulation framework called Cascade to design and validate a specialized system for computing molecular dynamics computations.

Acun et al. [13] present TraceR, a tool that replays the BigSim application traces on top of CODES network models. TraceR provides the ability to test CODES network models under real-world production application workloads. In contrast, our work simulates synthetic uniform random and worst-case traffic workloads. Since the TraceR tool has been interfaced with the CODES and ROSS frameworks, it can be experimented with on the Slim Fly model.

Coll et al. [48] investigated the use of multiple independent network rails as a technique to overcome bandwidth limitations. The study applies the multi-rail concept to the Quadrics network (QsNet) which can be configured as Fat-Tree topology. The simulations cover round-robin and dynamic rail allocation policies. Multi-rail performance analysis is done using synthetic workloads with varying injection loads and message sizes.

Chen et al. [44] analyzed various topologies in comparison with a single-rail 3-level Fat-Tree. In addition to the DOE Design Forward miniapps of AMG and MiniAMR, the authors also tested the performance of these network topologies using uniform traffic, adversarial traffic, nearest neighbor, and all-to-all synthetic workloads, while assuming direct, valiant, or adaptive routing algorithms. However, the authors overlook the potential advantages or disadvantages of deploying multiple planes of the same topology.

Domke et al. [56] proposed a network simulation environment, which estimates the throughput of InfiniBand-based HPC interconnect topologies based on flit-level accurate simulations. The toolchain is capable of simulating two traffic injection patterns, uniform random all-to-all and an exchange pattern (similar to MPI_Alltoall), for various topologies and static, destination-based routing algorithms. However, the scalability of the flit-level simulation is limited and does not offer application performance prediction.

Liu et al. [113] demonstrate the effectiveness of applying the Fat-Tree interconnect to large data centers. The work focuses on the ability of Fat-Tree networks to perform well under data-center applications at large scale. Unlike our work that currently focuses on HPC workloads, their work focuses on workloads approximating the Hadoop MapReduce model and uses an equal-cost multi-paths (ECMP) routing algorithm focusing on minimal path.

Hatazaki [82] describes design considerations and early experiences for deploying a production HPC system with dual-rail Fat-Tree network connecting 1408 nodes using QDR InfiniBand. The 2.4 Pflop/s hybrid CPU/GPU system has a peak full-bisection bandwidth of 56.32 Tb/s and the dual-rail fabric is showing almost double the performance of a single-rail.

Work has also been done looking at non-traditional HPC network topologies. Fujiwara et al. [68] present a new HPC interconnection network called Skywalk that uses low-delay switches and random connections to achieve shorter hop counts and end-to-end packet latencies. The Skywalk topology is based on the random shortcut topology approach [100] which augments classical topologies with random links to generate reduced network diameter, shorter average path lengths, and shorter network cables.

3.4.2 HPC Network Models

The collection of networks chosen in this research for analysis range from the purely theoretical topologies such as the Slim Fly and 1D Dragonfly, to the currently realized Cray 2D Dragonfly and Fat-Tree. Here, we present the details of each topology to explain the configurations, routing algorithms and other features which make them attractive options as HPC system interconnects. We begin with the Fat-Tree network.

3.4.2.1 Fat-Tree

This network topology is a popular choice for modern HPC and data center networks[17]. Recently, with the release of the Summit [133] and Sierra [107] supercomputing systems at Oak Ridge National Laboratory and Lawrence Livermore National Laboratory, Fat-Tree networks have adopted a dual-rail configuration to provide additional network resources to match increasing computational power per compute node. In this section, the full fat-tree network model is presented, followed by the pruned tree configuration, and finally we describe the multirail topology.

Fat-tree networks, which provide full-bisection bandwidth, are a popular choice for HPC systems. DCS Summit supercomputer with its small number of computationally dense CPU-GPU nodes. This section first describes the full fat-tree network model, followed by its pruned configuration, and finally we describe the multi-rail topology.

Full Fat-Tree: The fat-tree graph layout is composed of typically two or three switch levels [136], where all switches have as many uplinks as downlinks. Hence, for a given radix of k, each switch will have $k/2$ link to switches in upper levels and $k/2$ link to switches or compute nodes in lower levels.

Approved for Public Release; Distribution Unlimited.

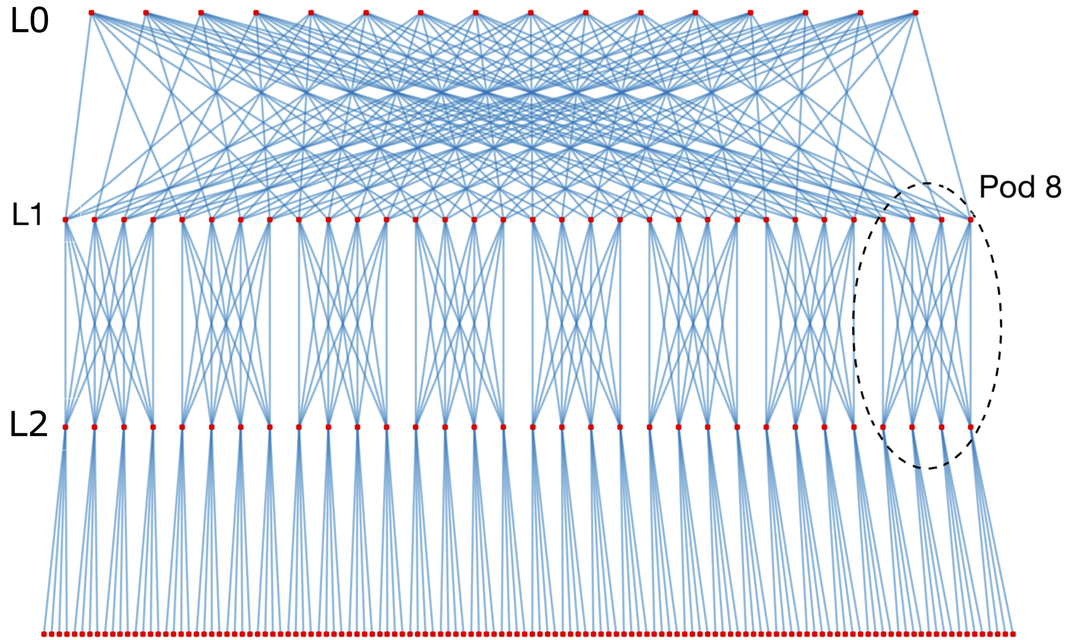


Figure 12: A full 3-level fat-tree network using eighty 8-port switches results in a total of eight pods with four switches per level per pod and a total of $k^3/4 = 128$ compute nodes.

For cost-saving purposes, the spine level L0, see Figure 12, uses half the number of switches with k downlinks each. Having an equal number of links in both directions allows each node to communicate via a unique path in the network. Theoretically, this results in full-bisection bandwidth, making the fat-tree a popular choice.

We assume 3-level fat-tree networks for the remainder of this paper. Given a switch radix of k, the structure for the 3-level fat-tree breaks down into k-many interconnected pods. Each pod contains $k/2$ switches in the first and second levels, labeled L1 and L2, respectively. The two levels of switches within each pod form a complete bipartite graph. Each switch of level L2 has $k/2$ connections down to compute nodes resulting in a total of $k \cdot (k/2) \cdot (k/2)$ compute nodes. Switches in the L1 level have $k/2$ connections to switches in the spine level L0. Furthermore, all links of L0 switches are connected downwards to provide the full interconnection of pods. An example of a small-scale full-bisection fat-tree, using $k = 8$ and totaling 128 nodes is shown in Figure 12.

Pruned Fat-Tree: Attempting to model a fat-tree network that matches the scale and performance of the future Summit HPC system requires modifications to the fat-tree. Summit's ≈3 400 nodes are interconnected via a dual-rail fat-tree using EDR InfiniBand (IB) technology [133]. Currently, Mellanox commodity EDR switches are available with up to $k = 36$ ports. Hence, using 36-port switches and the standard 3-level fat-tree configuration [136] results in too many compute nodes connected to the leaf switches, i.e., $36 \cdot (36/2) \cdot (36/2) = 11\,664$ nodes to be precise, to efficiently construct the 3 400-node Summit system.

An alternative design option, the pruned fat-tree, starts with the full 11 664-node 3-level fat-tree and then prunes/removes excess pods within the network. Full connectivity between pods is maintained by adjusting L1 and L0 connections as needed. This process continues until the desired node count for the Summit system is reached. Since, the number of compute nodes per pod $(k/2)$ is

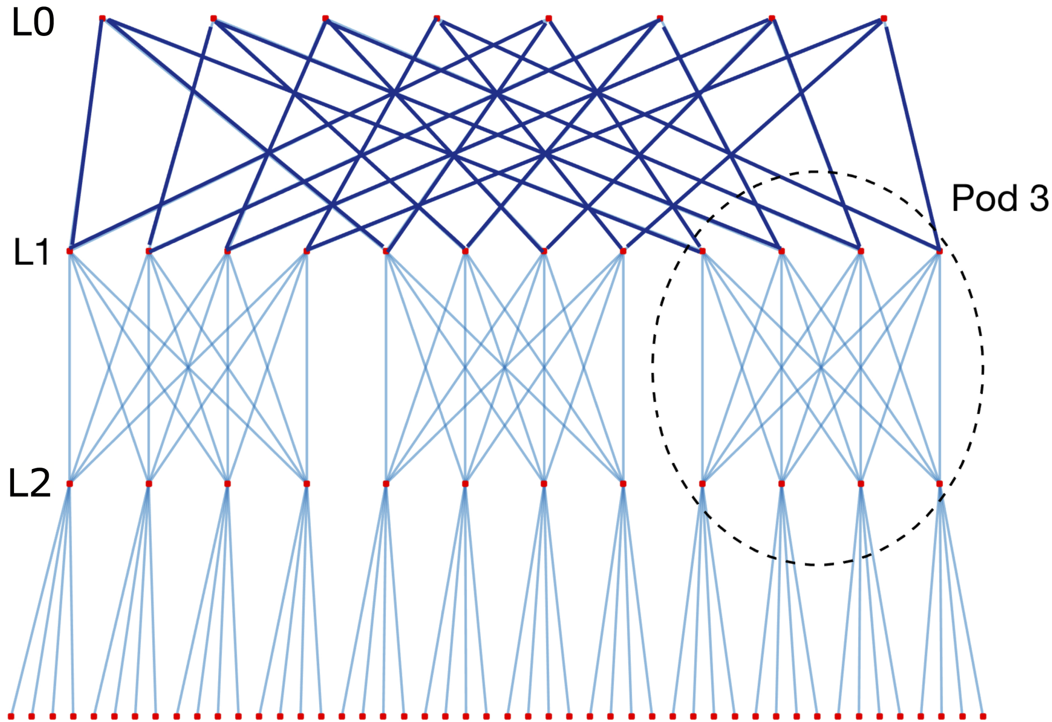


Figure 13: Example configuration for a pruned 3-level fat-tree using 32 8-port switches in three pods and spine level yields a total of $3 \cdot (k/2)^2 = 48$ compute nodes (darker colored lines between L1 and L0 indicate a bundle of two links).

not modified with this approach, we can calculate the number of pods, N_p, needed to get N-many nodes via the equation $N_p = \lceil \frac{N}{k/2 \cdot k/2} \rceil$. Hence, using the presumed 36-port switch radix, we need at least 11 pods to construct a pruned fat-tree able to accommodate at least 3 400 nodes, or a maximum of 3 564 nodes, respectively.

For visual reference, an example pruned fat-tree using 8-port switches is shown in Figure 13. Starting with the standard 128 node, eight pod full fat-tree network in Figure 12, five pods and eight L0 switches are removed to drop the total network node count to 48. Also, note the darker lines between L1 and L0 switches indicate a bundle of two links. These two link bundles are necessary to maintain the required $k/2$ uplinks to keep the full-bisection bandwidth.

Multi-Rail Fat-Tree: To mitigate the increased injection load of high-density compute nodes, as used in Summit, multi-rail networks can be deployed to utilize multiple network interface cards (NICs) to gain access to additional network planes. We assume that each network plane has the same topology. All rails, and their corresponding planes, are independent of one another, providing resiliency and additional relief from communication hot-spots. While the theoretical injection bandwidth increases linearly with the number of planes, e.g., up to 25 GB/s for a dual-rail EDR setup, so does the financial cost. Furthermore, an additional message scheduling layer is needed to distribute the traffic across the available rails.

Fat-Tree Routing Algorithms: Here, we describe our methods for both intra-rail message routing and inter-rail traffic injection used for the fat-tree network model. A key influencing factor for application performance on a fat-tree topology is the routing algorithm, which determines the

physical path traversed by packets in the network. These routing algorithms can be categorized and compared, such as static vs. dynamic, flow-oblivious vs. adaptive, and topology-aware vs. agnostic [66], [50]. For the remainder of the paper, we will focus on two routing approaches that we integrate into our fat-tree network model.

Fat-Tree Static Routing: One of the state-of-the-art routing algorithms for InfiniBand-based fat-tree topologies is the flow-oblivious and destination-based fat-tree routing [170]. Instead of implementing the fat-tree routing within the model, we adapt the approach of the fail-in-place simulation framework [56]. This allows us to make use of the highly optimized and well-tested fat-tree routing which is available as part of the InfiniBand (IB) subnet manager, called OpenSM [9]. Therefore, our simulations with static routing use the same forwarding tables as a IB-based production HPC system with a similar topology. The tool chain presented in [56] consists of four stages: topology loader/generator, routing engine, converter, and simulator. We make use of the first two stages and load the extracted forwarding tables (LFT) back into our fat-tree model. Therefore, our model writes out the topology specification in a hardware-neutral graph description language (DOT format). We update the topology loader to read the DOT format, which is then translated into an internal (IB-specific) network representation. The routing engine calculates and writes out the LFTs for the virtual IB network. Our fat-tree model incorporates these LFTs of all switches to accurately simulate the application traffic routed via the static fat-tree routing algorithm.

Fat-Tree Adaptive: This scheme aims at balancing traffic on different links by making locally optimal decisions for forwarding packets on a switch, similarly to the minimal adaptive routing of the CM-5 [111]. For each port (or virtual channel) on a switch, a token-count is maintained to estimate the load on the port. The initial value of token-count is set to the length of the virtual channel buffer used to store the packets being forwarded. When a packet is sent on a port, the token-count is decremented by the length of the packet. On receiving an *ack* or credit notification from the receiver, the value of the token-count is incremented. A port with higher value of token-count is considered less loaded, while no packets can be sent on a port whose token-count is zero. When a packet arrives at a switch, we compute all ports on which the packet can be forwarded assuming shortest path routing. Among these ports, the port with maximum token-count is selected to enqueue the packet.

Fat-Tree Multi-Rail Injection: Utilizing the additional networks in multi-rail configurations requires another scheduling layer. This additional layer can schedule packets at multiple rails deciding which NIC is selected to transmit the packet. After the packet is injected into the network on a rail, the selected intra-rail routing algorithm takes over to finish routing the packet to its final destination. Currently, random and adaptive policies have been implemented for rail selection as described below.

Fat-Tree Multi-Rail Random and Adaptive Routing: The random approach offers a uniform distribution of traffic over all network rails regardless of traffic load and network congestion. The adaptive approach samples the occupancy of all NICs on the compute node to select the one with the lowest occupancy. For light network traffic conditions (i.e. when the packet injection rate is less than link speed), all communication will traverse the default first rail as NIC occupancies are empty by the time the next packet arrives at the NIC. This leads to an unbalanced communication load among rails and increases the possibility of link congestion.

Approved for Public Release; Distribution Unlimited.

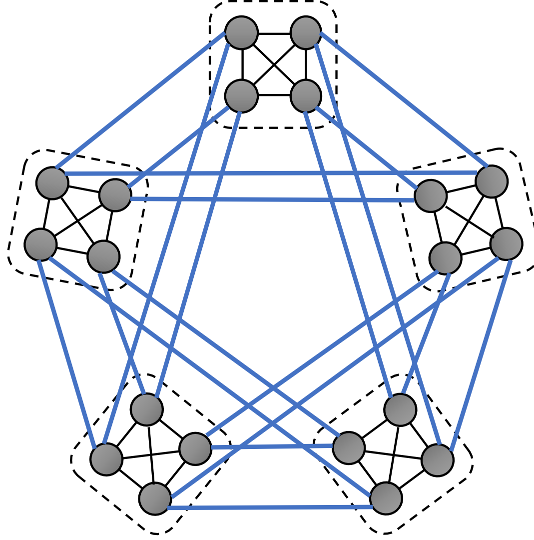


Figure 14: Full Five group dragonfly network.

3.4.2.2 Dragonfly

Next, we turn our attention to the Dragonfly network. In general, the Dragonfly network is a hierarchical graph structure leveraging high-radix routers to generate scalable networks maintaining a low diameter. The Dragonfly consists of many groups of routers following all-to-all global connections between the groups. An example dragonfly network is shown in Figure 14 consisting of five groups with four routers per group. Communication within groups traverses intra-group links (shown in black), while inter-group communication can utilize both intra-group and inter-group links (shown in blue), depending on the system connectivity and the location of the source and destination nodes. Within each router group there are many options for intra-group connectivity. In this work we study two different Dragonfly configurations based on the currently deployed two dimensional Cray Cascade system [62] and the originally proposed single dimension Dragonfly [99].

The first configuration, which we call Dragonfly-2D, adopts the approach incorporated into the "Edison" Cray XC30 [3] and "Cori" XC40 [2] systems deployed at Lawrence Berkeley National Laboratory and the "Theta" XC40 [8] system deployed at Argonne National Laboratory. Following the Cray Cascade architecture [62], the network nodes within a group are arranged in the form of a "2D" matrix where routers in the same row have all-to-all connections and routers in the same column have all-to-all connections. Furthermore, each router contains 8 global connections which are combined into bundles of 4 links connecting to a router in one of the other groups. The final distribution of connections for any given router is skewed toward local connections. In this work, we consider a Dragonfly-2D configuration with 16x6 router matrix groups and 48 ports per router. Routers can have up to 3.75x more local connections than global connections.

The second configuration, which we term Dragonfly-1D has much smaller groups with a better distribution of local and global connections per router. Dragonfly-1D groups consist of a "1D" array of routers with all-to-all connections. Depending on the radix of the switch, the remaining ports are used for compute node connections and global connections to routers in other groups. In this work we focus on a Dragonfly-1D network with arrays of 16x1 router array groups and 36 ports per router. This configuration has a good distribution of both intra-group and inter-group connectivity

with only 1.25x more local connections than global connections.

Dragonfly Routing Algorithms: The method for selecting message paths is especially important to efficiently utilize the hierarchical connectivity of the Dragonfly topology. In this work, we use adaptive routing with both Dragonfly configurations in order to offer an optimal dynamically selected balance between short message hop counts using minimal paths and reducing congestion by redistributing traffic across the network using non-minimal paths. Ideally, all workloads would have balanced traffic loads in which case minimal routing would offer optimal performance. In reality, workloads can have highly unbalanced traffic loads in which case a dynamic selection of minimal and non-minimal paths can provide the optimal performance.

Dragonfly Minimal Routing: Both the 1D and 2D Dragonfly networks follow the same general approach to minimal routing. Each network first routes locally following the shortest path within the source router group to a router containing a global connection to a router in the destination router group. Dragonfly-1D requires only one hop to accomplish the first step because of the all-to-all intra-group connectivity. The Dragonfly-2D requires at most 1 hop along each of the two dimensions to reach any intra-group router. After traversing the global link and reaching the destination router group, packets are again routed locally along the shortest path to the destination router.

Dragonfly Non-Minimal Routing: Similar to Slim Fly, the non-minimal routing approach for both Dragonfly configurations follows the Valiant randomized algorithm [158]. Instead of routing directly from source router to destination router, an intermediate router is randomly selected with the restriction that it is not in same group as the source or destination router. Packets are then routed minimally from the router to the intermediate router and then routed minimally again to the destination router. Non-minimal routing effectively distributes adversarial traffic across the entire network to make use of all links to minimize congestion latency at the cost of increased path lengths.

Dragonfly Adaptive Routing: Adaptive routing combines minimal and non-minimal approaches to achieve a dynamic selection of short paths and traffic load distribution based on router congestion. Both models again use a UGAL algorithm that monitors local router occupancy to influence the selection between minimal and non-minimal paths. The routing decision is made at the source router and the path with the least congestion is selected. Additionally, there exists an adaptive threshold value which allows the selection to be biased toward choosing minimal or non-minimal paths.

3.4.2.3 Slim Fly

The Slim Fly is a new network topology with low network diameter and low cost. While the Slim Fly is still a theoretical network with no known physical systems deployed to date, the low latency and highly efficient use of hardware lend itself to being considered for next generation systems. In this section, we describe the complex composition of the slim fly topology as well as the resulting characteristics that show promise in an HPC system setting.

Introduced by Besta and Hoefler [31], the slim fly consists of groups of routers with direct connections to other routers in the network, similar in nature to the dragonfly interconnect topology. Each router has a degree of local connectivity to other routers in its local group and a global degree of connectivity to routers in other groups. Unlike the dragonfly topology, however, the slim fly does not have fully connected router groups. Within each group, each router has only a subset of

Approved for Public Release; Distribution Unlimited.

Table 3: Descriptions of symbols used.

Topic	Symbol	Description
SF	p	Nodes connected to a router
	N_r	Total routers in network ($N_r = 2q^2$)
	N_n	Total nodes in network ($N_n = N_r * p$)
	k'	Router network radix
	k	Router radix ($k = k' + p$)
	q	Prime power
CODES/ ROSS	LP	Logical Process (simulated entity)
	PE	Processing element (MPI rank)

intragroup connections governed by one of two specific equations based on the router's subgraph membership. Furthermore, all router groups are split into two subgraphs. Each router possesses global intergroup connections only to routers within the opposite subgraph, forming a bipartite graph between the two subgraphs. These global connections are constructed according to a third equation [31]. Figure 15. shows a simple example of the described structure and layout of the slim fly topology.

An important feature of the slim fly topology is that its graphs are constructed to guarantee a given maximum network diameter. Network diameter describes the maximum shortest path length between all routers in the network. Decreasing a network's diameter shortens the path length (i.e. number of hops), resulting in packets that experience less router and link latency. One example set of graphs, which we use in this paper to construct router connections, is the collection of diameter 2 graphs introduced by McKay et al. [119], called MMS graphs. MMS graphs guarantee a maximum of 2 hops and because they approach the Moore bound [123], these graphs constitute some of the largest possible graphs that maintain full network bandwidth while maintaining a network diameter of two. Note, the MMS graphs are used to construct the slim fly router network layer. When the compute nodes are included, the system diameter becomes four with the addition of hops into and out of the network from compute nodes. The 2-hop property holds true while scaling to larger graphs because the router radix grows as well. For example, the 338 routers used to construct a 3K compute node slim fly system require a radix of 28, while a much larger 1M compute node system needs 53,138 routers with radix 367.

MMS graph Construction for Slim Fly: Following the methods derived in [78] and summarized and applied to the slim fly topology in [31], we developed a separate application to create the nontrivial MMS network topology graphs that govern the interconnection layout of nodes and routers in slim fly networks. The process requires (1) finding a prime power $q = 4w + \delta$ that yields a desired total number of routers $N_r = 2q^2$; (2) constructing the Galois field and the primitive element ξ that generates the Galois field; (3) using ξ, computing generator sets X and X' [78] and using them in conjunction with equations 14–16 to construct the interconnection of routers; and (4) connecting compute nodes to routers. It's important to note the importance of the variable q as it indicates the number of routers per group and the number of global connections.

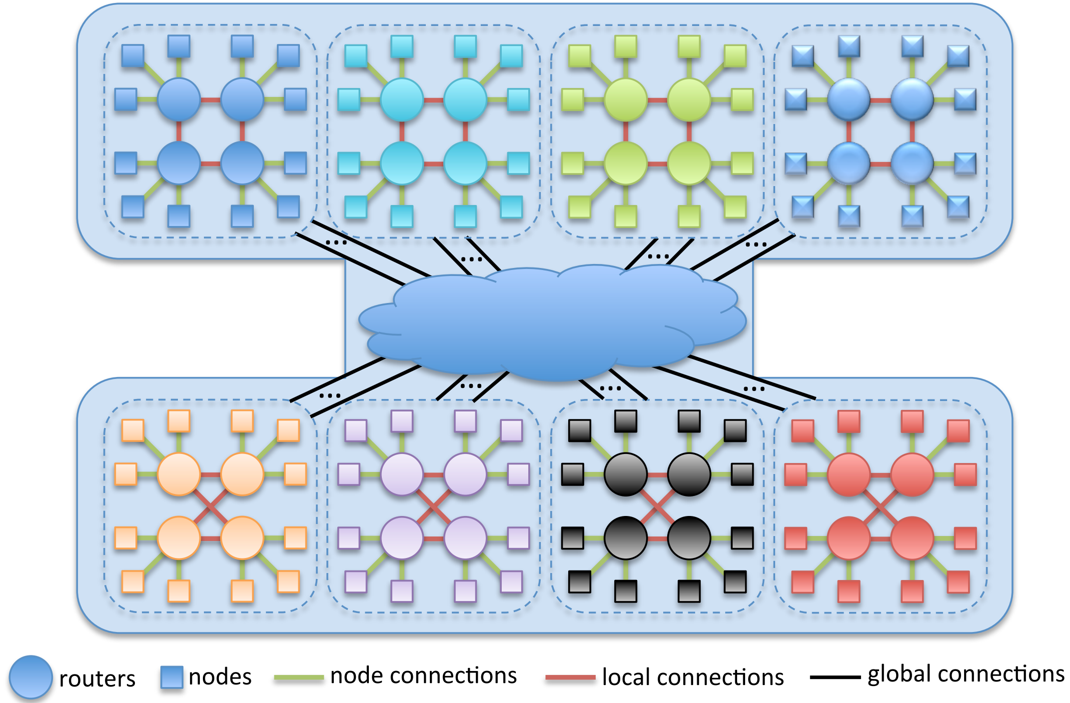


Figure 15: General structure and layout of MMS slim fly graphs. Global connections between subgraphs have been generalized for clarity. There are no intergroup connections within the same subgraph. Each router contains one global connection to one router in each of the q-many router groups in the opposing subgraph.

$$\text{router}(0, x, y) \text{ is connected to } (0, x, y') \text{ iff } y - y' \in X; \tag{14}$$

$$\text{router}(1, m, c) \text{ is connected to } (1, m, c') \text{ iff } c - c' \in X'; \tag{15}$$

$$\text{router}(0, x, y) \text{ is connected to } (1, m, c) \text{ iff } y = mx + c; \tag{16}$$

An example MMS graph is provided in Figure 16. As shown, all routers have three coordinates (s, x, y) indicating the location of the router in the network. The $s \in \{0, 1\}$ coordinate indicates the subgraph, while the $x \in \{0, ..., q-1\}$ and $y \in \{0, ..., q-1\}$ coordinates indicate the router's group and position within the group, respectively. Following the coordinate system, Equation 14 is used to compute the intragroup connections for all groups of subgraph 0 shown in Figure 16. Equation 15 performs the same computation for all groups in subgraph 1, shown in red. Equation 16 determines the connections between the two subgraphs, shown in blue. For simplicity, Equation 16 connections are displayed only for router$(1, 0, 0)$.

Slim Fly Routing Algorithms: Our slim fly model currently supports three routing algorithms for studying network performance: minimal, non-minimal, and adaptive routing. Overall, minimal routing focuses on maintaining the smallest hop count for all packets, whereas non-minimal routing sacrifices hop count to minimize congestion, and adaptive routing strives to find a balance between the two.

In other words, for every hop i that a message packet takes, when leaving a router, that packet uses the ith virtual channel. Packets that take a local route and have only one hop will always use

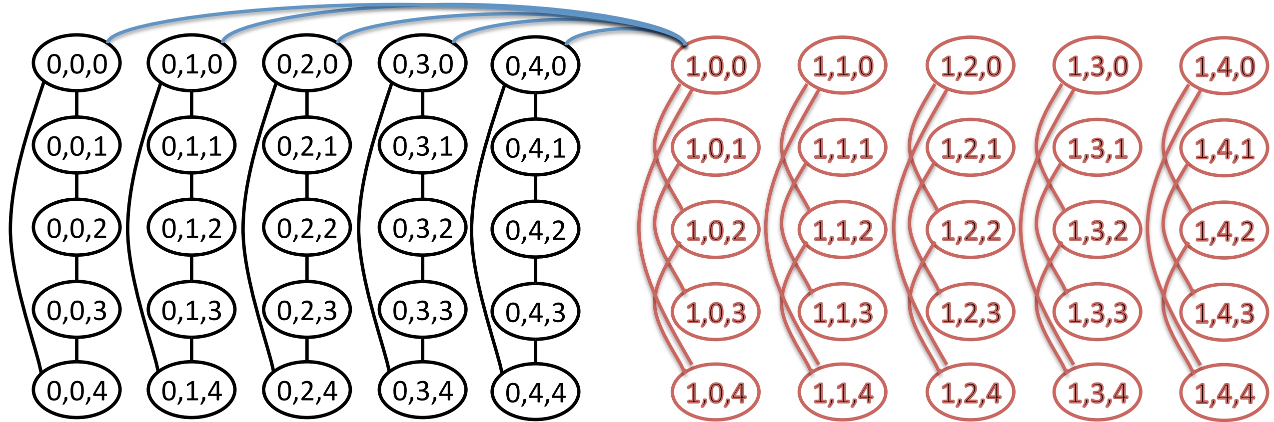


Figure 16: Example MMS graph with $q = 5$ illustrating the connection of routers within groups and between subgraphs.

$VC0$. Packets that take a global path (assuming minimal routing) will use $VC0$ for the first hop and then $VC1$ for the second hop. Thus, two VCs are needed for minimal routing. In the case of non-minimal routing such as valiant and adaptive routing, the number of virtual channels used is four, because the maximum possible number of hops in a packet's route is four.

Slim Fly Minimal Routing: The minimal, or direct, routing algorithm routes all network packets from source router to destination router using a maximum of two hops (property of MMS graphs guarantees router graph diameter of two). If the source router and destination router are directly connected, then the minimal path consists of only one hop between routers. If the source compute node is connected to the same router as the destination compute node, then there are zero hops between routers. In the third case, an intermediate router must exist that shares a connection to both source and destination router so the packet traverses a maximum of two hops between source and destination routers. The above numbers are all computed taking into account only the router interconnection network and not the connections to compute nodes. If we include the hops from source compute node to source router and from destination router to destination compute node, then the minimum and maximum number of hops a packet can take under minimal routing is two and four respectively. The number of virtual channels used is two because the maximum number of network hops in a packet's route is two.

Slim Fly Non-minimal Routing: Non-minimal routing for the slim fly topology follows the traditional Valiant randomized routing algorithm [158]. This approach selects a random intermediate router that is different from the source or destination router and routes minimally from source router to the randomly selected intermediate router. The packet is then routed minimally again from the intermediate router to the destination router. The number of hops traversed within the router network with non-minimal routing is double that of minimal routing. In the optimal case when all three routers (source, intermediate, destination) are directly connected, the path is two router hops. In the worst-case scenario, each minimal path to and from the intermediate router can have two router hops, bringing the maximum number of possible hops within the router network to four. Including the hops from source node into the router network and the hop from the network to the destination compute node results in minimum and maximum hop counts of four and six respectively. The number of virtual channels used for non-minimal routing is four.

Slim Fly Adaptive Routing: Adaptive routing mixes both minimal and non-minimal approaches by adaptively selecting between the minimal path and several valiant paths. To make direct comparisons for validating our model, we follow a slightly modified version of the Universal Globally-Adaptive Load-balanced (UGAL) algorithm [162] shown in [95]. After a packet reaches the first router, the minimal path and several non-minimal paths (n_I) are generated and their corresponding path lengths L_M and L_I^i, $i \in 1, 2, ...n_I$ are computed. Next, we compute the penalty $c = L_I^i / L_M * c_{SF}$, where c_{SF} is a constant chosen to balance the ratio between minimal and non-minimal paths. Next, the final cost of each non-minimal route $C_I^i = c * q_I^i$ is computed, where q_I^i is the occupancy of the first router's output port corresponding to the path of route i. The cost of the minimal path is simply the occupancy of the first router's port along the path q_M. Then, the route with the lowest cost is selected, and the packet is routed accordingly. With this method, each packet has a chance of getting routed between source and destination routers with anywhere from one to four hops. The minimum and maximum hop counts between any two compute nodes is two and six hops respectively.

3.5 Classification of AFRL Data Using IBM TrueNorth & *NeMo*

We first tested the neuron level accuracy of *NeMo* by re-creating results published by [40], with our results published in [138]. These results showed that *NeMo* was able to accurately model the IBM TrueNorth neuron when simulating synthetic benchmarks. To further validate the *NeMo* simulator, we tested it using trained large-scale real world models.

To determine the accuracy of the *NeMo* model, we tested multiple models against the IBM TrueNorth neurosynaptic processor system. Each model was trained using the EEDN convolutional neural network toolkit, a machine learning software toolkit provided by IBM for use with the TrueNorth hardware. EEDN generates trained spiking neural networks in a format that can be used for the TrueNorth hardware system. These trained networks were run using the IBM TrueNorth software simulation tool, where spike activity between neurons and cores was recorded. We then imported these models into *NeMo*, and compared the results.

The datasets and models we chose to use for this process were the MNIST handwritten digit dataset, and a model provided by the AFRL trained on the Moving and Stationary Target Acquisition and Recognition (MSTAR) data set.

MNIST is a classic benchmark dataset for machine learning models. It is extremely well understood, providing simple and regular data that allows for detailed analysis of machine learning algorithms. While MNIST is not a cutting edge or particularly challenging data set for machine learning algorithms, it is useful as a baseline test for the design and base analysis of these algorithms. MNIST consists of a series of images of handwritten numerical digits, size-normalized and centered. The full MNIST dataset contains 60,000 training samples and 10,000 test samples.

MSTAR data consists of synthetic aperture radar (SAR) imagery of military hardware. The data set contains SAR images of 15 targets (types of vehicles) along with various non-target images. The images were collected using X-band SAR, using 15 deg and 17 deg depression angles, with some targets containing 30 deg and 45 deg angles as well. These "images" are stored in a file format that encapsulates the SAR data along with descriptive metadata.

The conversion of a trained EEDN model into a form that *NeMo* can understand involved designing a intermediate representation of the model. EEDN generates a neural network file in a dense JSON file format. This format represents the network using prototypes. At the start of

the file, crossbar and core prototypes are defined, each with a unique key and information about connectivity. Next, neuron prototypes are defined. In the file, all neurons are given a default base configuration. Changes to this configuration, such as different weights, leak values, and other settings, are saved as neuron prototypes which are marked by a unique ID and placed in a list towards the beginning of the file. The file represents the network after these prototype definitions via a series of core entries that consist of:

- A Unique core identification information, including the core number.
- A list of dendrites, representing inputs
- A list of neuron types, with each value representing a neuron prototype.
- A list of destination core IDs and destination axon IDs for each neuron.
- The delays used for each neuron's spike timing A crossbar prototype ID, which must be previously defined in the top of the file.

The file resembles a flat-file relational database, with external keys defined in the list of cores connecting to the prototypes defined at the top of the file. To import this file into *NeMo* requires flattening the relations, so that *NeMo* neuron data structures can be created from the prototypes. This is necessary since *NeMo* considers neurons to the be the basis of the neurosynaptic simulation, whereas the EEDN file treats neurosynaptic cores as the base unit. A *NeMo* simulation would define all neurons as unique elements, with neurosynaptic core groups generated as part of the PDES simulation mapping. We attempted two techniques to do this conversion.

The first method uses a intermediary domain specific language. This technique uses a Python script to generate a new neural network definition file. This file is a Lua script. Neuron information is stored in a nested dictionary, with the neuron ID (consisting of the neurosynaptic core and local neuron ID) being the key, and the value being all parameters of that particular neuron.

We embedded a Lua interpreter into *NeMo*, which then reads this script file during the simulation initialization, and uses the dictionary to pull neuron parameters into the LP state as each LP is initialized. This technique could provide some advantages over using a binary file format, or a simpler text file format. Firstly, since the *NeMo* model file is a valid Lua script, if a different neuron model is required, no re-tooling is needed to read that particular neuron's parameters into *NeMo*. The Lua script and interpreter are dynamically typed, giving great flexibility for data input. For example, if we wished to simulate several TrueNorth neurons, followed by some custom neuron that uses 64-bit integers for weights, the Lua interpreter and configuration file would require only adding a new neuron type name to the file. The only other requirement would be adding the new neuron model to *NeMo*, and finding the different name.

Another advantage of using an embedded Lua interpreter is in model error checking. Lua supports introspection, and this is used inside *NeMo* to provide model level error checking. For example, if in a model file a neuron has an invalid value, the Lua interpreter will see this, and report the offending line number and neuron ID. This prevents both erroneous neuron models and provides insight into where the invalid values reside.

A major disadvantage to using a DSL with a Lua interpreter is performance. Lua itself is a tiny binary, and has minimal memory overhead when embedded into a C program. However, since the *NeMo* model file is a flat representation of the complete neural network, file size and

conversion time became an issue. For a modest neural network of 4,000 neurosynaptic cores, the *NeMo* configuration file would contain 1,024,000 lines, each with over 544 parameters, creating roughly 557,056,000 entries. This large size made both generation and loading of the file take significant time. To improve performance, we added a C++ library to *NeMo* which directly reads in the EEDN model file and parses it during the simulation initialization. This provides an order of magnitude reduction in initialization time, and eliminates the need for a large intermediary file.

We approached determining the accuracy of *NeMo* compared with TrueNorth from the following perspectives: First, we extracted core and neuron connectivity. This information was used to determine if *NeMo* was accurately representing the trained neural network that EEDN produced. To extract neuron and core connectivity from the Compass simulation too, we used the built-in spike activity monitoring system which records all spikes that take place in the simulation. This large file was squashed into a edge table, with each edge representing at least one spike between neurons. Using this data, we were able to compare *NeMo*'s interpretation of the EEDN model file with what the simulator produced.

Next, we added to *NeMo* the ability to save spike information. Since there can be a huge quantity of spikes occurring within a simulation, we added the option of saving all spike information, or just the output neurons. In TrueNorth models, and *NeMo*, the output layer is represented by cores with a value < 0. These spike activity files were used to gage the accuracy of *NeMo* by comparing the spike activity with what was reported by the NSCS simulation tool.

We used these spike activity files to find the accuracy of the *NeMo* simulator. We examined all spikes generated by *NeMo* and the NSCS simulator. For the MNIST data set and model, we examined all spike activity simulated in the network. This comparison shows not just output layer accuracy, but also internal spike activity. Using the pre-trained model provided by the AFRL, we examined the output layer of the network. We compared the recorded spikes in the output layer between what was reported by the NSCS simulation model and what *NeMo* produced, showing that *NeMo* is accurately generating classification or output spikes compared to the base line simulation. The results of this analysis are discussed in the next section.

4 RESULTS AND DISCUSSION

4.1 *NeMo* Results

4.1.1 *NeMo* Validation

Izhikevich implemented and reviewed 20 prominent features of biological neurons using a resonate-and-fire model [89]. The TNLIF model was used to recreate many of these behaviors, demonstrating the utility and validity of the TNLIF model [40]. *NeMo*, unlike Izhikevich's model and the *Compass* simulator used in the IBM reference paper, simulates TNLIF neurons using discrete events. Due to this difference, the discrete model can only approximate, although with a high degree of accuracy, Izhikevich's models. Neurons only update internal state when an input message is received or if they are a self-firing neuron (i.e., a neuron that can fire without first requiring input). However, we do recreate the neuron behavior observed in the TrueNorth neuron model. The result of this experimental run shows that while the inter-spike state of a simulated neuron may not be accurate, the spike times match what is observed when running these biological models using the Compass simulator.

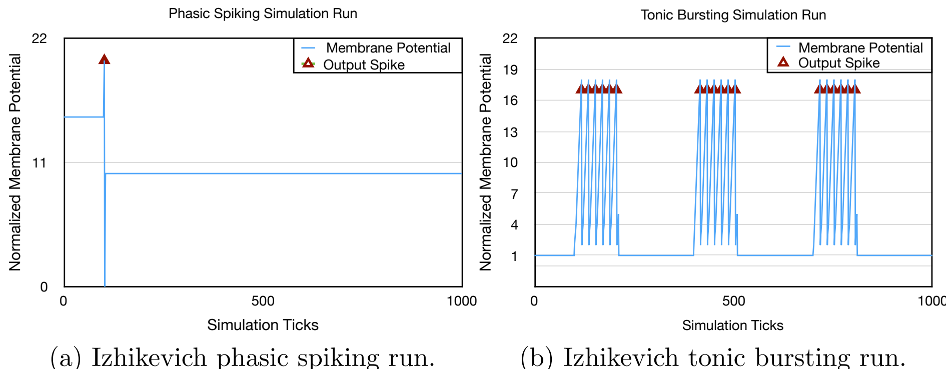
Parameter	Neuron 0 Value
Synaptic Weights $(s_j^{G_i})$	0,20,0,0
Leak Value (λ)	2
Positive Threshold (α)	2
Negative Threshold (β)	-10
Reset Voltage (R_j)	-15
Reset Mode	Normal Negative Saturation

(a) Phasic Spiking Parameters

Neuron 0 Value	Neuron 1 Value
1, -100, 0, 0	1, 0, 0, 0
1	0
18	6
20	0
1	0
Normal Negative Saturation	Normal Negative Saturation

(b) Tonic Bursting Neuron Parameters

Table 4: Neuron Validation Parameters.



(a) Izhikevich phasic spiking run.

(b) Izhikevich tonic bursting run.

Figure 17: Two Izhikevich Validation Run TNLIF Neuron Parameters and Results.

To validate *NeMo*, we implemented two of the Izhikevich models as done in [40] using the TNLIF model. Our goal was to match the behavior of these models, showing that *NeMo* correctly simulates the TNLIF model. To do this, we used the same parameters for each neuron as were used to generate the original results. A Phasic spiking neuron was configured this way, with a single axon input set to send spikes out every 200 ticks. The results of this run are shown in Figure 17a.

We then implemented a tonic bursting neuron, again following the parameters used to recreate this behavior using the TNLIF model in COMPASS. In this configuration, we used two neurons and three axons. One axon was configured to send input spikes every 300 ticks. The neuron parameters used for this run are shown in Table 4b, and the membrane potential results are shown in Figure 17b.

The information shown in Figures 17b and 17a visually presents neuron behavior that is nearly identical to the behavior observed using COMPASS. Slight differences in the values are a result of neurons updating state only when events warrant. We also do not record the membrane potential of the input axons. Despite this, we do see qualitatively similar neuron behaviors. Thus, the *NeMo* simulation model is able to recreate the simulation results produced by the IBM simulation tool, COMPASS.

4.1.2 *NeMo-ES* Performance Results

Experimental Setup: For each of the following experiments, we simulate TrueNorth-like neurosynaptic cores using the ROSS framework. Each neurosynaptic core connects 256 axon LPs, 65,536 synapse LPs, and 256 neuron LPs for a total of 66,048 LPs per core. We perform experiments with up to 32,768 neurosynaptic cores, giving a maximum number of 2,164,260,864 LPs in our largest simulation.

To fully test the performance of our model, we used a neurosynaptic core model which generates over 1,500 events per neurosynaptic core per tick. For this benchmark, each core consists of an "identity-matrix" of neurons. In this model, axon i will trigger synapse i, i, which triggers the neuron at i. The output destination of each neuron is set randomly with an 80% chance that it will output to a different neurosynaptic core. To start, each axon in the simulation fires. Overall, this creates an immense number of events, a larger workload than would be expected in a real-world application.

All simulations were performed on an IBM Blue Gene/Q machine. Each node of the Blue Gene/Q features eighteen 1.6 GHz processor cores, 16 of which are devoted to application use [79]. For the two remaining cores, one conducts operating system functionality while the other serves as a spare. All nodes are connected by an effective, high-speed communication network [43].

The 16 GB of DDR3 memory on each Blue Gene/Q node can be a limiting factor in memory intensive simulations. To allow for maximum utilization, each node is highly configurable in terms of parallelism. Each of the 16 processors can run up to 4 hardware threads (for a total of 64 MPI ranks per node) or the processor cores can be under-subscribed (with a minimum of 1 MPI rank per node). Our experiments test several parallel configurations.

All experiments were performed using the time-warp based optimistic synchronization algorithm in ROSS.

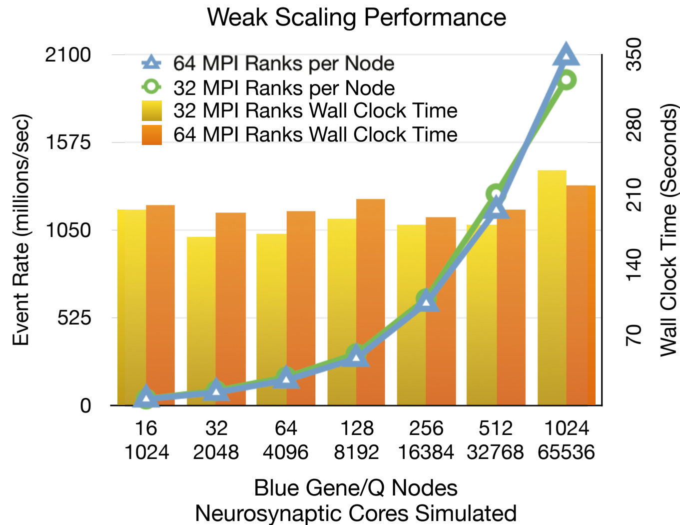


Figure 18: Weak scaling performance experiments.

Weak Scaling Experiment: Our first set of experiments tested two configurations: one and two neurosynaptic cores per MPI rank. These configurations ran on either 64 or 32 MPI ranks per Blue Gene/Q node, scaling from 16 to 1024 Blue Gene/Q nodes (see Figure 18). We achieved a

peak performance of over 2 billion events per second when simulating 65,536 neurosynaptic cores on 1024 Blue Gene/nodes with 64 MPI ranks per node. These experiments simulated a total of 1,000 neurosynaptic core ticks.

Table 5: Breakdown of time spent during the simulation of 65,536 neurosynaptic cores on 1024 Blue Gene/Q nodes each with 64 MPI ranks.

Clock Cycle Category	Time Taken (seconds)	Percentage
Priority Queue (enq/deq)	1.8825	0.86%
AVL Tree (insert/delete)	0.0192	0.01%
Event Processing	38.2921	17.52%
Event Cancel	0.7486	0.34%
GVT	154.6155	70.75%
Fossil Collect	12.7742	5.85%
Primary Rollbacks	5.1278	2.35%
Network Read	5.0715	2.32%

Table 5 presents a breakdown of time spent during our peak performance simulation. These statistics are representative of all of our weak scaling experiments. The most noteworthy statistic is the time that the ROSS simulator spent performing GVT calculations. With less than 20% of simulation time being spent performing local event processing, we observe over 70% of the simulation time is spent performing GVT calculations. Since the GVT calculation is based around an `MPI_all_reduce` calculation, this indicates that there is a load imbalance within the simulation. That is, not all MPI ranks are reaching the blocking MPI reduction operation at the same time.

The slight load imbalance is to be expected. Every time a neuron fires, it has an 80% chance to send a signal to a neuron within a different synaptic core. Since the location of the receiver neuron is also chosen randomly, there is in an unpredictable, yet expected load imbalance across the simulation.

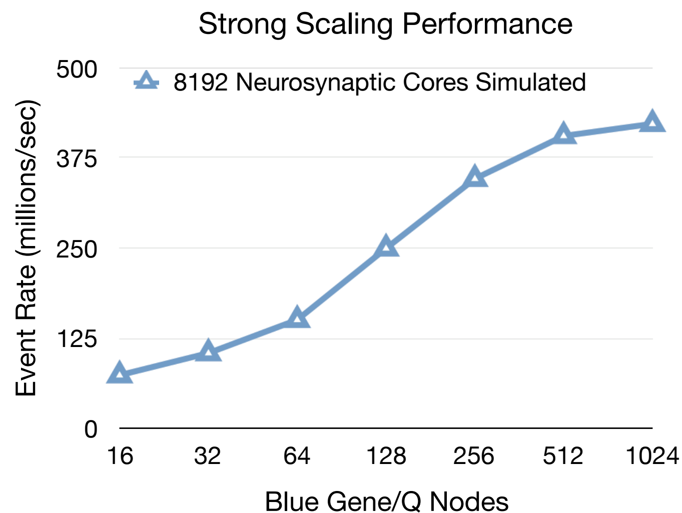


Figure 19: Strong scaling performance experiments.

Approved for Public Release; Distribution Unlimited.

Strong Scaling Experiment: To understand the ways in which the *NeMo* model scales as parallelism increases, we ran a series of strong scaling experiments. Figure 19 shows performance results for a simulation of 8,192 neurosynaptic cores using 16 to 1,024 Blue Gene/Q nodes. These experiments were run for 1,000 ticks resulting in more than 13 billion net events. We achieved peak performance when we used 1,024 Blue Gene/Q nodes, where we observed over 421 million events per second. This benchmark was run with the same randomly generated neuron model as the weak scaling experiments, with an 80% chance of neurons communicating to remote cores. One interesting thing to note is that *NeMo* does not place a neurosynaptic core across multiple MPI ranks. This is a limiting factor in the strong scaling results, as simulating 8,192 neurosynaptic cores gives a maximum of 8,192 MPI ranks. When running on 512 Blue Gene/Q nodes, there are 32,768 possible MPI ranks, and on 1,024 nodes there are 1,048,576 ranks available. We ran at these scales with 8,192 ranks, and the lack of increase in performance is attributable to this limitation in the *NeMo* system.

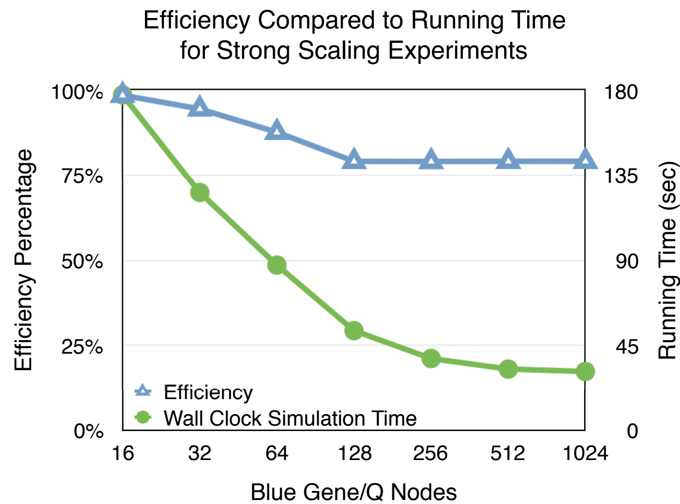


Figure 20: Comparison of the efficiency and the running time of the strong scaling experiments.

One interesting phenomena is observed when analyzing the running time and efficiency of the strong scaling experiments, Figure 20. Here, we see that an unexpected correlation between overall simulation efficiency and the running time of the simulation: a *decrease* in efficiency corresponds to faster running times. This indicates a very low cost for performing an event rollback. Overall, the optimistic simulation is able to find more parallelism (and thus more speedup) despite incurring an increased number of rollback events.

Figure 21 shows a breakdown of the rollbacks observed during the strong scaling experiments. At first the number of rollbacks does increase as the parallelism increases however all experiments on 128 nodes or more each incur approximately 7.7 million rollbacks. This indicates that our simulations have a maximum amount of parallelism where increases in hardware do not correlate to increases in performance.

Both Figures 20 and 21 show that increasing the number of parallel nodes is only effective up to a point. At more than 128 Blue Gene/Q nodes, we see diminishing returns in performance scaling. In the 128 node experiment, each node simulates 64 neurosynaptic cores. We see that the overall workload is balanced (i.e., there is no decrease in performance which would indicate an over decomposition of the system). For the 128, 256, 512, and 1,024 node experiments, the communication

Approved for Public Release; Distribution Unlimited.

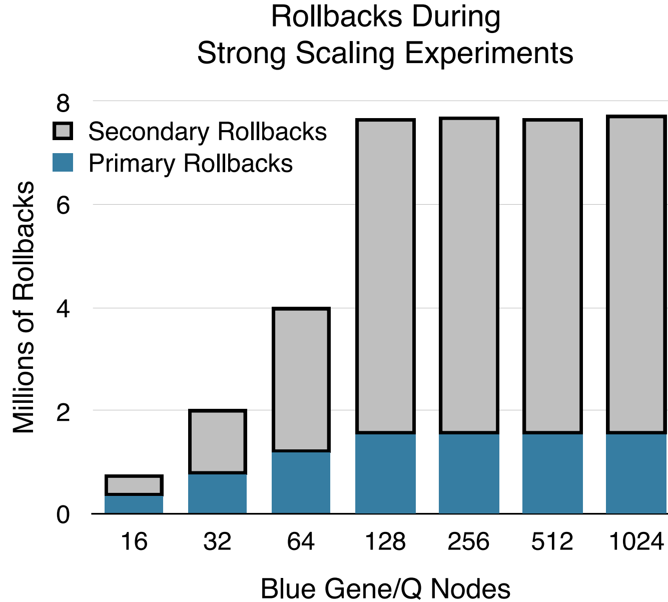


Figure 21: Breakdown of the primary and secondary rollbacks for the strong scaling simulation. Note that the net event population for these experiments is 13 billion events, but the maximum number of rollbacks observed is only 7.6 million.

overheads surpass the time spent doing local event processing. What is most intriguing about these experiments is that the number of rollbacks remains constant, despite an increase in parallelism.

Comparison with Compass: Comparing *NeMo* with IBM's own early parallel neuromorphic simulation software, *Compass*, poses some challenges. The intention of *NeMo* is to provide an open-source way to simulate various neuromorphic hardware designs. The *Compass* simulator is tuned for a similar purpose, but is tied into the TrueNorth architecture. Furthermore, *Compass* is proprietary software that we are unable to use on our benchmarking hardware. IBM Blue Gene/Q support for *Compass* was also eliminated in favor of focusing support on x86 architectures. These factors make a meaningful direct comparison impossible. However, in [140], benchmark runs of *Compass* are done using several models. Using these existing benchmarks, we can create a rough comparison between *NeMo* and *Compass* [1].

While both *NeMo* and *Compass* can exploit massively parallel supercomputer systems like the Blue Gene/Q, there are a number of key differences between the two. First, *Compass* employs a time-stepped algorithm which iterates over the set of neurons assigned to a particular thread (or MPI rank) and then iterates over each synapse event that targets that neuron from the enclosing iteration loop. *NeMo* is implemented using a pure event-driven approach with all neuron, axon and synapse events being enqueued into a single priority queue. This approach avoids the cost of iterating over neurons which do not have any posted synapse events.

In [140], *Compass* weak-scaling benchmarks are presented using the *CoCoMac* neuron model. We do not have access to the specific implementation details of the neuron model used in the paper, so we can not explicitly re-create the author's benchmarks in *NeMo*. CoCoMac is a message-sparse model, generating on average 1.3 spikes per simulated neurosynaptic core [140].

[1] Today the IBM simulator does not execute in parallel on a Blue Gene supercomputer and has been renamed *NSCS* which was used in other parts of our research.

$$e_{\text{total}} = s_{\text{reported}} \times 2 \times f \tag{17}$$
$$e_{\text{second}} = s_{\text{total}}/t \tag{18}$$
$$e_{\text{second/rack}} = e_{\text{second}}/16 \tag{19}$$

Figure 22: Calculating *Compass*'s events per second per Blue Gene/Q Rack

Given that we are not able to run the same model and we are not able to run *Compass* on our benchmark hardware, we have decided to compare the events per second produced by *Compass* with *NeMo*. To find this value for *Compass*, we took the number of spike events per simulated tick reported in [140] and made some reasonable assumptions about the underlying model. We took the values from the largest run of *Compass* that ran on 16 racks of IBM Blue Gene/Q that generated 22 million spikes per simulated tick. The paper does not specify if that value is for all spikes generated simulating the model, or if it was for remote-core spikes (spikes originating on a different node than the destination). To compare *NeMo*'s performance with *Compass*, we will assume the best-case scenario for *Compass*: that the 22 million spikes generated are only the remote spikes. The paper further specifies that the CoCoMac model simulated has some cores that generate a ratio of 80 remote spikes to 20 local spikes, and other cores that generate a 60/40 ratio. For the purposes of our comparison, we assume a best-case scenario for *Compass* and assume a 50/50 ratio of remote spikes versus local spikes. This assumption would mean that the 22 million spikes per simulated tick would be 50% of the total spikes generated by the simulation. For comparison, our benchmark model generated an average of 80% remote spikes per core.

The set of equations that approximate the performance of the *Compass* simulation are shown in Equations 17-19. To find the number of events per second, we assume that 1 axon and 256 synapse fan-out events, f are scheduled for each spike s event over the paper's reported 500 ticks t_{sim}, shown in Equation 17. We also multiply the total spikes reported, e_{total} by 2, per our assumption that 50% of the spikes are not remote. We then divide the calculated total number of events by the wall clock time taken by *Compass*, t_{wall} to get the spikes per second, e_{second}, shown in Equation 18. We then divide by the number of Blue Gene/Q racks used in the simulation r, to get *Compass*'s events per second per rack, shown in Equation 19.

These values were chosen to help represent the number of events *NeMo* produces. For every neuron spike event in *NeMo*, there are 256 neuron events generated with one axon event. Our largest current *NeMo* simulation ran on one rack of Blue Gene/Q, thus comparing *Compass*'s event rate per rack p er second will provide a roughly accurate gauge of performance.

This gives $((22\text{M} \times 2 \times 257) \times 500\,\text{ticks})/\,194\,\text{secs}/16\,\text{racks} = 1,821\text{M}\,\text{events}/\,\text{second}/\,\text{rack}$. Our benchmark runs of *NeMo* showed an event rate of $2,082\text{M}\,\text{events}/\,\text{rack}$. The weak scaling experiments run on *NeMo* show 261M events per second more than *Compass*. While a direct comparison between *NeMo* and *Compass* is currently impossible, this result shows that *NeMo* is on par with the performance of *Compass*, and a viable option for simulation of neuromorphic hardware.

4.1.3 *NeMo-SS* Performance Results

Understanding the performance of *NeMo-SS* within a massively parallel environment is important. The purpose of *NeMo-SS* is to allow for simulations of novel neuromorphic hardware and designs, for

Approved for Public Release; Distribution Unlimited.

Table 6: Experimental Run Configurations.

Parameter					
Neurons Per Core	256	256	256	512	512
Synchronization Mode	GVT	Real-Time	GVT	Real-Time	GVT
NeMo Version	1	2	2	2	2

both exploration of novel neuromorphic hardware as well as simulation of neuromorphic hardware within a simulated heterogeneous HPC system. Providing the ability to simulate extremely large neuromorphic hardware networks will provide insights into massively connected hardware networks. To show that *NeMo-SS* will be able to simulate large structures of neuromorphic hardware, we ran *NeMo-SS* with extremely large networks in a massively parallel environment.

To accomplish this, we first examine a weak scaling experiment done on an IBM Blue Gene/Q, where we simulate up to 8,388,608 neurosynaptic cores with a total of 4,269,367,296 neurons. We then examine the strong scaling performance of a 65,536 neurosynaptic core simulation. We also compare these results with the first version of *NeMo*. A smaller run on an Intel based cluster is also examined, showing performance results that compare with the Blue Gene/Q architecture.

Experimental Setup: For each of the following experiments, we simulate TrueNorth-like neurosynaptic cores using the ROSS framework. In the weak scaling experiments, we compare a simulation containing neurosynaptic cores with 256 axon LPs, 1 synapse LP, and 256 neuron LPs, and a simulation containing neurosynaptic cores with 512 axon LPs, 1 synapse LP, and 512 neuron LPs. The largest *NeMo-SS* simulation contains a total of 8,598,323,200 LPs.

To test the performance of our model, we used a neurosynaptic core design that generates over 1,500 events per neurosynaptic core per tick. The neurons are configured such that they will fire a spike if they receive an input spike from an axon with the same ID. Each neurosynaptic core in this benchmark has weights such that an input from axon i will trigger neuron i to send one spike.

We have also implemented a "neuron connection pool" benchmark. In this network, every neuron is connected to 20 randomly selected axons (a pool of connections). All connected axons have a weight value of 1, and neurons have a threshold value of 5. As in the other benchmark model, neurons are connected to a pseudo-random output, with a probability of .9 of a connection to a different core.

The output destination of each neuron is set randomly with a 90% chance that it will output to a different neurosynaptic core. When the benchmark starts, each axon fires once. This benchmark setup generates an extremely large number of events, resulting in a larger workload than would be expected in a real-world application.

These simulations were performed on both an IBM Blue Gene/Q machine, and an Intel based cluster. Each node of the Blue Gene/Q features eighteen 1.6 GHz processor cores, 16 of which are dedicated to application use [79]. For the two remaining cores, one conducts operating system functions while the other serves as spare. All nodes are connected by an effective, high-speed communication network [43].

The 16 GB of DDR3 memory on each Blue Gene/Q node can be a limiting factor in memory intensive simulations. To allow for maximum utilization, each node is highly configurable in terms of parallelism. Each of the 16 processors can run up to 4 hardware threads (for a total of 64 MPI ranks per node) or the processor cores can be under-subscribed (with a minimum of 1 MPI rank

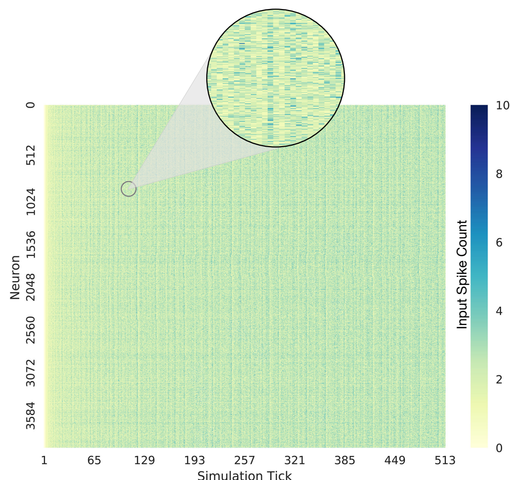


Figure 23: Neuron activity in the identity matrix benchmark simulation. This chart represents the number of spikes sent by each neruon in a 4096 neuron simulation across 512 ticks, demonstrating the workload generated by the benchmark model run. We observed an even distribution of neuron activity across the simulation running this benchmark.

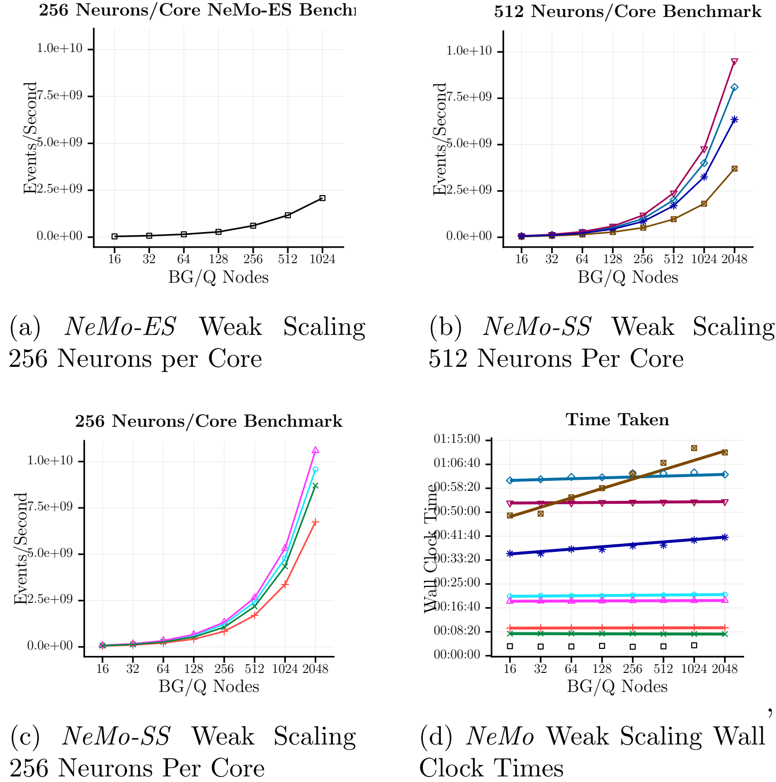
per node). Our experiments test several parallel configurations.

Blue Gene/Q Weak Scaling Experiments: Our first set of experiments tested several configurations. For each *NeMo-SS* experiment, we configured the simulation with 16 neurosynaptic cores per MPI rank, for a total of 64 neurosynaptic cores per Blue Gene/Q node. In this weak scaling model, we ran *NeMo-SS* with 64 MPI ranks per Blue Gene/Q node. We ran two different neurosynaptic core configurations with *NeMo-SS*: One with 256 neurons per core, and the other with 512 neurons per core. The TrueNorth architecture has 256 neurons per neurosynaptic core. By adding a configuration with 512 neurons per core, we were able to simulate a theoretical hardware configuration. We ran the simulation for a total of 1,000 neurosynaptic core ticks. This is equivalent to running the TrueNorth hardware for 1 second, as the hardware runs at 1,000 Hz. The results of these runs are shown in Figure 24.

In Figure 24a, we also show the results of a weak scaling experiment running *NeMo-ES*. In these runs, *NeMo-ES* was configured to simulate 256 neurons per neurosynaptic core, running the same benchmark configuration as *NeMo-SS*. Due to memory limitations, the *NeMo-ES* simulation was run with 16 neurosynaptic cores per MPI rank. This run was limited to 1024 Blue Gene/Q nodes.

The *NeMo-ES* experiment achieved a peak performance of over 2 billion events per second when simulating 65,536 neurosynaptic cores on 1024 Blue Gene/Q nodes with 64 MPI ranks per node. In Figure 24d, *NeMo-ES* shows a near linear performance across the weak scaling simulation. Towards the upper end of the simulation, the wall-clock time increased. This is due to a slight load imbalance, caused by excessive GVT calculations. Every time a neuron fires, it has a 90% chance to send a signal to a neuron within a different neurosynaptic core. As the simulation encounters more network latency, the effects of rollbacks become more apparent in the simulation time. This long event chain (seen in Figure 4), coupled with the large number of random remote messages, results in an expected load imbalance. Generally, *NeMo-ES* shows near linear performance in weak scaling experiments.

We ran *NeMo-SS* using two synchronization modes: GVT calculated based on the number of events processed and real-time GVT. In Figures 24b and 24c, we show the performance of *NeMo-SS*



(a) *NeMo-ES* Weak Scaling 256 Neurons per Core

(b) *NeMo-SS* Weak Scaling 512 Neurons Per Core

(c) *NeMo-SS* Weak Scaling 256 Neurons Per Core

(d) *NeMo* Weak Scaling Wall Clock Times

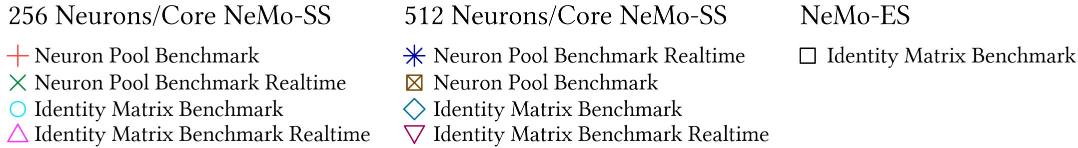


Figure 24: Blue Gene/Q weak scaling performance experiments. Figure (a) shows the results of running *NeMo-ES* with 256 neurons per core with 16 cores per rank. Figure (b) shows the results of running *NeMo-SS* with 512 neurons per core with 64 cores per rank. Figure (c) shows the results of *NeMo-SS* with 256 neurons per core with 64 cores per rank. Figure (d) shows the wall clock time taken for each run.

using these two different synchronization modes. Interestingly, real-time GVT provides significantly better performance, especially when simulating 512 neurons per neurosynaptic core.

In Figure 24d, the wall-clock time for these runs is shown. We found that both implementations of *NeMo* showed near linear wall-clock time across all identity matrix benchmark runs. *NeMo-SS*, running with 512 neurons per core, showed a slight increase in wall-clock time as the simulation size increased. This is likely due to the increased amount of time spent in GVT, as it can be seen in Figure 25.

The results of the neuron pool benchmark show the impact on *NeMo* of the significant increase in neuron activity within this benchmark network. The wall-clock time of the 512 neuron per core pool benchmark illustrates the potential limits of optimistic simulation in this scope. Switching from the standard GVT synchronization technique to the real-time technique improved scaling results as

well.

Figure 25 shows the wall clock time spent in each phase of simulation during these runs. We observed that real-time GVT significantly reduced the amount of time spent processing GVT. *NeMo-SS* benefits greatly from the real-time GVT, showing reduced time spent waiting for synchronization.

Optimistic synchronization in ROSS creates MPI barriers at each MPI rank after a specific number of events have been processed. Real-time synchronization creates these barriers after a set period of time. Real-time synchronization can provide performance increases over standard optimistic methods if many MPI ranks are waiting for a GVT to occur, while few MPI ranks are still processing events. This algorithm is based on the real time algorithm proposed by [67], but adapted from a shared memory system to a fully distributed system.

In *NeMo*, we observed a significant burst effect of messages. Some MPI ranks will generate large numbers of messages quickly, while others are waiting to process new events. Given the burst-like nature of the simulation, the real-time synchronization protocol will generally provide better performance in this case. This performance increase is specific to the neurosynaptic model, as there are many bursts of high message activity followed by slower periods of message activity.

Figure 24d also shows the wall clock time taken for the *NeMo-SS* weak scaling simulation runs. For most of the *NeMo-SS* runs, a near linear wall-clock time is observed. The exception occurs when running the simulation with 512 neurons and standard GVT synchronization. This increase in execution time is most likely due to increased network communication overhead, along with the significantly higher time spent on event processing, as seen in Figure 7.

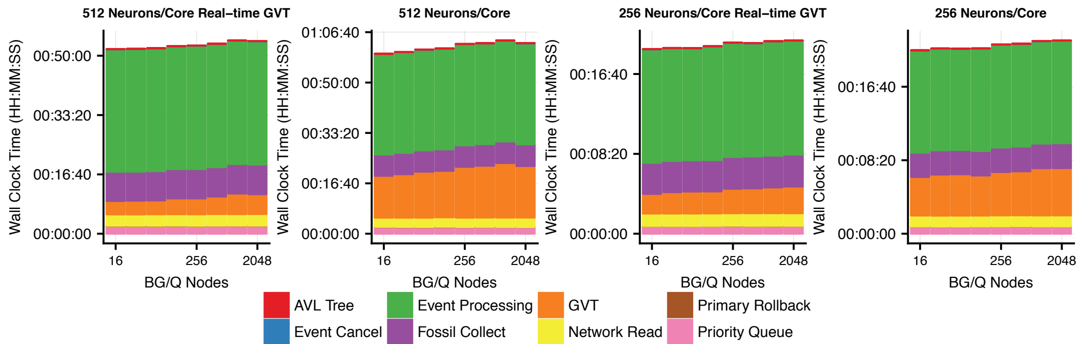


Figure 25: *NeMo-SS* BG/Q Weak Scaling Time Detail.

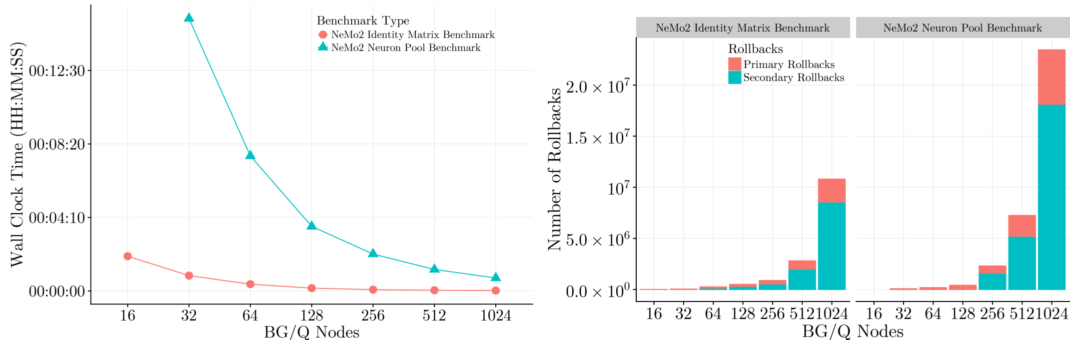
Blue Gene/Q Strong Scaling Experiments: To understand the ways in which the *NeMo-SS* model scales as parallelism increases, we ran a series of strong scaling experiments. Figure 26 shows performance results for a simulation of 65,536 neurosynaptic cores using 16 to 1024 Blue Gene/Q nodes and the number of rollbacks observed. These experiments were run for 1,000 ticks resulting in over 9 billion net events. We achieved a peak performance of 5,834,092,242 events per second when we used 1,024 Blue Gene/Q nodes. This benchmark was run with the same randomly generated neuron model as the weak scaling experiments.

One thing to note is that *NeMo-ES* and *NeMo-SS* do not place a neurosynaptic core across multiple MPI ranks. This limits the maximum parallelization possible during strong scaling experiments. In this experiment, running 65,536 neurosynaptic cores gives a maximum of 65,534 MPI ranks. Figure 26a shows the simulation performance increase at a near linear rate until the number of neurosynaptic cores starts to equal the number of available MPI ranks. Eventually, the Blue

Table 7: Breakdown of time spent during the simulations on 2048 Blue Gene/Q nodes each with 64 MPI ranks. The columns labeled with "Real-Time" text indicate the NeMo-SS runs that use the Real-Time GVT synchronization protocol, in contrast to the event count based GVT method.

	512 Cores	512 Cores Real-Time GVT	256 Cores	256 Cores Real-Time GVT
Priority Queue (enq/deq)	130615 ms	134795 ms	49065 ms	49778 ms
AVL Tree (insert/delete)	1316 ms	1239 ms	764 ms	666 ms
Event Processing	2005299 ms	2079298 ms	697964 ms	713461 ms
Event Cancel	4577 ms	2808 ms	1027 ms	1002 ms
GVT	1023165 ms	333997 ms	321908 ms	167047 ms
Fossil Collect	428499 ms	497998 ms	169134 ms	200164 ms
Primary Rollback	2379 ms	2308 ms	881 ms	45 ms
Network Read	187937 ms	195584 ms	74874 ms	78527 ms

Gene/Q's individual node compute power eclipses the available network bandwidth, slowing strong scaling performance gains. Figure 26b shows the increase in primary and secondary rollbacks as the communication overheads surpass the time spent during local event processing.



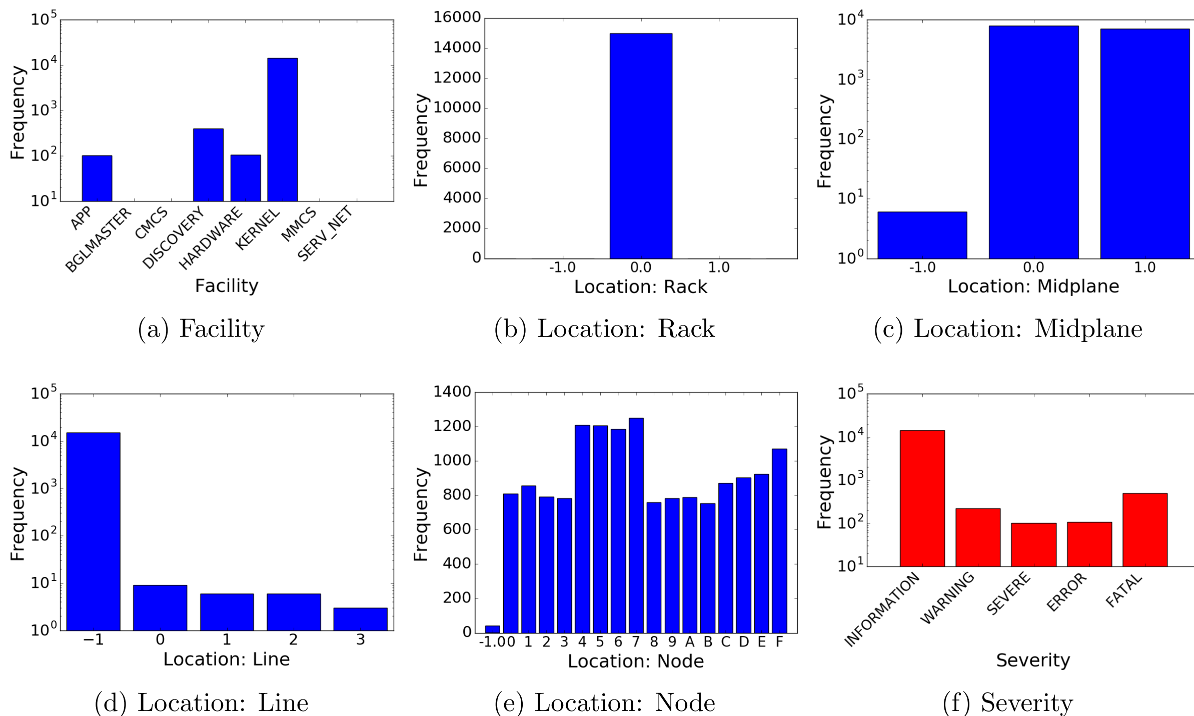
(a) Blue Gene/Q Strong Scaling Performance. (b) Blue Gene/Q Strong Scaling Rollbacks.

Figure 26: *NeMo-SS* Blue Gene/Q Strong Scaling Experiment Results.

A major factor in the increase in overhead as the simulation reaches maximum parallelism is the time spent rolling back events. In Figure 26b, the number of rollbacks dramatically increases as the number of nodes increases. When run at 1024 Blue Gene/Q nodes, the number of rollbacks approaches the number of events computed.

The largest simulation done on the Blue Gene/Q used two racks, 2048 nodes, to simulate 4,294,967,296 neurons in 8,388,608 neuromorphic cores. This simulation achieved a maximum event rate of 9,524,353,605 events per second when run with real-time synchronization. When the number of neurons per core was reduced to 256, the event rate increased to 10,589,662,119 events per second across 2,147,483,649 neurons.

Approved for Public Release; Distribution Unlimited.



(a) Facility (b) Location: Rack (c) Location: Midplane

(d) Location: Line (e) Location: Node (f) Severity

Figure 27: Histograms of Categorical Variables.

4.2 Classification of Supercomputer Failures Using TrueNorth

4.2.1 Exploratory Data Analysis (EDA)

The first step after cleaning the data was to perform Exploratory Data Analysis (EDA). EDA summarizes the main characteristics of the dataset so that we can form a solid understanding of the dataset and decide on further data analysis. Because we are using machine learning and deep learning in our analysis, we cannot perform exploratory data analysis on the whole dataset. This is in line with the machine learning principle that once any kind of inference is drawn from any part of the dataset, we should not use it to train the learning models. We ran a Singular Value Decomposition (SVD) on the dataset to check if dimensionality reduction was possible. This revealed that all the chosen features were significant and dimensionality reduction was not necessary.

The next step in EDA was to plot histograms of the categorical variables and the output variable, i.e. `SEVERITY` to get an idea of their distributions (see Figure 27). We used a value of -1.0 for all those data entries which did not have a value for the variable. The variable `FACILITY` is bimodally distributed with majority of log files initiated from `DISCOVERY` and `KERNEL`. The histogram for the Rack location is unimodal, meaning that in the 10% of the data that forms the EDA set, all the data points happened to have a Rack value of zero. It is, however, very unlikely that this would be true for the training and test datasets as well. Histogram for Midplane is uniformly distributed with some noise present in the data. The midplane can have two values, and in the EDA set, they are more or less equally distributed. When it comes to the `LINE` location, the EDA dataset was extremely noisy, as is evident from a large portion of -1.0 values. The histogram for the `NODE` location is more or less uniformly distributed around the frequency value of 1000 and contains very

Table 8: Performance Comparison of All Techniques.

ML/DL Technique	Training Accuracy (%)	Validation Accuracy (%)	Testing Accuracy (%)
Logistic Regression	93.59	98.63	94.52
K-NN (K=3)	97.84	98.67	96.75
K-NN (K=$\sqrt{83,903}$ = 289)	94.14	98.48	94.97
SVM	92.85	97.65	93.75
DNN	96.57	98.54	96.23
RNN	96.27	98.36	95.98
SNN (TrueNorth)	**99.41**	**98.12**	**99.80**

little noise.

4.2.2 Classification Results and Comparative Analysis

Our Spiking Neural Network (SNN) model consists of an input layer followed by a transduction layer, which is followed by three convolutional layers (see Fig. 6 and Fig. 7). The output layer is a softmax layer. This spiking convolutional neural network uses a total of 45 TrueNorth cores. Given that each TrueNorth core has 256 neurons, this gives us an upper bound of 11,520 TrueNorth neurons. We use the phrase 'upper bound' because it could be the case that not all 256 neurons in each of the 45 cores are being used. Nevertheless, we count it as part of the total system cost because most likely, these unused neurons cannot be used for other applications running on the TrueNorth chip. We used IBM's EEDN software framework, which is built in conjunction with MatConvNet (MATLAB) for SNN.

Our TensorFlow DNN model consists of six fully connected layers containing 105 neurons and 1750 synapses. The layer-wise neuron configuration is as follows: 30-25-20-15-10-5. We use hyperbolic tangent (tanh) and rectified linear unit (relu) as our activation functions and we alternate them for each layer. The output layer is a softmax layer. Our TensorFlow RNN model consists of a Long Short Term Memory (LSTM) layer followed by a softmax layer. The maximum sequence length of this RNN was 100, meaning we would monitor information from the previous 100 log entries to make a prediction for the current log entry. To store relevant information from older as well as current sequence of log entries, we chose the RNN cell size as 50. We also observed that the DNN and RNN models with number of neurons in ballpark of the SNN overfitted the training data, and were thus discarded. The DNN and RNN configurations mentioned above gave the best performance metrics amongst all the configurations that we tested.

Table 8 shows the results of our ML/DL classifiers. Although we present the results for just one hyperparameter configuration of each of the ML/DL techniques described above, we fine tuned the hyperparameters for these techniques over multiple iterations to obtain the best possible configuration. So, for example, the DNN model presented in Table 8 was seen to be the best performing DNN model. We used three performance metrics – Training Accuracy, Validation Accuracy and

Approved for Public Release; Distribution Unlimited.

Table 9: Design Index Computation.

ML/DL Technique	Training Error (%)	Validation Error (%)	Accuracy Index (I_a)	Overfitting Index (I_o)
Logistic Regression	6.41	1.37	1.1931	-0.6701
K-NN (K=3)	2.16	1.33	1.6655	-0.2106
K-NN (K=289)	5.86	1.52	1.2321	-0.5861
SVM	7.15	2.35	1.1457	-0.4832
DNN	3.43	1.46	1.4647	-0.3709
RNN	3.73	4.02	1.4283	-0.3569
SNN (TrueNorth)	**0.59**	**1.88**	**2.2291**	**0.5033**

Test Accuracy. All our ML/DL models perform better than 92% classification accuracy. The best model as per all the performance metrics was SNN with a test accuracy of 99.80%. KNN ($K = 3$) seems to be the second best model with test accuracy of 96.75%. The two deep learning models (DNN and RNN) show comparable results with respect to each other with test accuracy of 96.23% and 95.98% respectively. These are followed by KNN ($K = 289$), logistic regression and SVM with test accuracy of 94.97%, 94.52% and 93.75% respectively.

We also computed the design index [52] for all seven ML/DL and neuromorphic models used in this paper. The design index is a tool which functions as an aid during the designing phase of DNN. Although it was defined for DNNs, it can easily be extended to other ML/DL and neuromorphic computing techniques. It consists of an index tuple containing Accuracy Index (I_a) and Overfitting Index (I_o), which gives a quantitative estimate of how accurate and how overfitted a trained DNN model is. Moreover, it is easy to represent visually. In this work, the threshold error was chosen as 1.0, the accuracy threshold was chosen as 1.0 and the overfitting threshold was chosen as 2.0. The threshold error is the order of magnitude of the smallest enumerated label. In our case, since the enumerated labels go from 1 to 5, the order of magnitude of smallest enumerated label is 0 and thus, the threshold error error was $10^0 = 1$. We chose an accuracy threshold of 1.0 because for this application, we would be happy to have a training error which is an order of magnitude less than the smallest enumerated label. We chose an overfitting threshold of 2.0 because for our application, we would render a trained model as overfitted if the training error is two orders of magnitude less than the validation error.

Table 9 and Figure 28 show the computation and scatter plot of of design indices. A peculiar feature in this figure is that the overfitting indices of all ML/DL techniques except SNN are negative, meaning that their validation errors were less than their training errors. We think this happened because in spite of picking the log entries identically and independently for training and validation datasets, their distributions were slightly different. However, as denoted by the green star in Figure 28, the SNN was able to correctly model the error data, showing the highest accuracy index (2.2291) and an acceptable overfitting index (0.5033). The blue region in the figure shows the acceptable region – we will accept a particular model if it lies in this region. All of our ML/DL and neuromorphic models lie in the acceptable range – i.e. not only do they demonstrate a certain acceptable level of accuracy, but also show that they do not overfit or underfit the training data. This confirms that ML/DL and neuromorphic techniques can be used to model and classify node

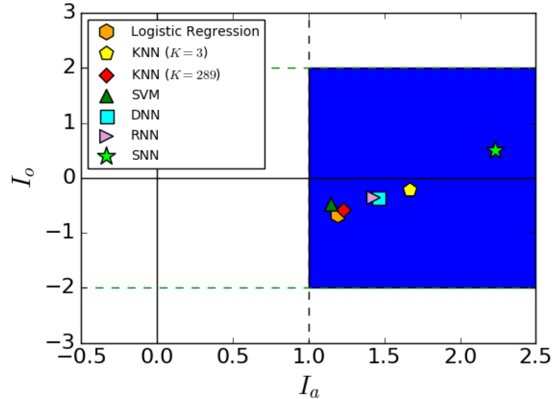


Figure 28: Plot of Design Index for All Techniques. Accuracy Index is computed as $\log_{10}\left(\frac{e_p}{e_t}\right)$, where e_p is the threshold error and e_t is the training error [52]. Overfitting Index is computed as $\log_{10}\left(\frac{e_v}{e_t}\right)$, where e_v is the validation error and e_t is the training error [52].

failures on supercomputers as they display a respectable level of accuracy for this application.

4.2.3 Speed and Power Consumption

We now shed some light on the running times and power estimates of our models. During the test phase, all the techniques used in this study were able to classify 29,965 data points in the test dataset fairly quickly. The fastest was DNN (9.6 milliseconds) and the slowest was RNN (2.2 seconds). The TrueNorth SNN took 23.5 milliseconds to classify the test dataset. When it comes to power consumption, DNN, RNN and Logistic Regression models were running on GPUs and consumed around 250 Watts of power. The models running on CPUs (i.e. K-NN and SVM) consumed around 60 Watts of power. The SNN ran on the TrueNorth chip and was estimated to consume less than 2 milliwatts of power – five orders of magnitude less than GPU and four orders of magnitude less than CPU. To the best of the authors' knowledge, a more detailed breakdown of power consumption is not possible with the TrueNorth development kit at our disposal.

4.2.4 Discussion

From the analysis so far, we understand that while all the ML/DL techniques fall within the acceptable range, the convolutional SNN seems to be performing the best for our dataset as per the performance metrics used in this work. This result is unexpected because CNNs are inherently not designed for sequential data such as the error data used in this paper. The model that is known to perform well on sequential datasets is the RNN owing to its inherent dynamical nature. Consequently, we expected RNN to outperform all other models, but that was not the case. Although our error log dataset is a sequential dataset, it also is a spatio-temporal dataset, because it contains information about the physical location and timestamp of the log entries. Spiking Neural Networks (SNN) have been shown to perform well on spatio-temporal datasets [94, 58]. While the exact reason for good performance of SNN on spatio-temporal datasets is unknown, a brief intuition is as follows. SNNs were modeled after the human brain and resemble it very closely. Since human

brain is excellent at perceiving, interpreting and analyzing spatio-temporal data, one might expect SNNs to behave similarly – and they indeed perform well on spatio-temporal data, as demonstrated previously in the literature and by our results.

It must be pointed out that although SNN used 45 TrueNorth cores, which correspond to a maximum of 11,520 neurons, they cannot be compared directly to hundreds of neurons used in our DNN and RNN models. First of all, 11,520 is the number of *hardware* neurons used and is different from the *software* neurons used by DNN and RNN models. A single software neuron usually maps to multiple hardware neurons – the exact mapping depends on the encoding algorithm. Furthermore, we had to add two layers of padding to our dataset, which required 25 times more neurons to represent a single data point – this directly adds to the neuron cost as previously indicated. Secondly, the 11,520 neurons is just an upper bound – it does not necessarily mean that our SNN used all 11,520 neurons. With the TrueNorth development kit that we had, it was not possible to know the exact number of neurons used.

4.3 Durango – A Hybrid System Performance Modeling Framework

Here, we present two series of experiments. The first uses the LULESH miniapp and compares it with Durango's generated facsimile of its communication pattern. For these experiments, a 64-way multicore system is used with 64 GB of RAM and 2.2 TB of disk space. Here, the real LULESH DUMPI trace for a 4x4x4 grid (64 MPI ranks) configuration is compared with that of the Durango generator, which also runs and creates a DUMPI trace. For the comparison, both DUMPI traces are run through the CODES torus network simulator, which is configured with a torus network topology in a 4x4x4 configuration.

In the second series of experiments, we demonstrate the efficacy of our direct integration approach. Performance results are shown for Durango when a computational kernel implemented in Aspen is linked into a executable network model. For this series of experiments, both a torus and a dragonfly network are used. All simulations are run in parallel on the "AMOS" IBM Blue Gene/Q supercomputer located at the Center for Computational Innovations at Rensselaer.

4.3.1 Durango Generated vs. Real LULESH Results

4.3.1.1 LULESH Proxy Application

LULESH [4, 93] is a scientific computing application that performs explicit shock hydrodynamics calculations on an unstructured grid. It has been ported to several programming models. To explore modeling of the communication behavior, we studied the parallel version of LULESH implemented by using MPI for interprocess communication.

LULESH has a processor decomposition that is regular. While the mesh elements are defined with explicit connectivity allowing for unstructured elements other than hexahedra, the implementation of communication between the problem domains has a logical structure that allows nearest neighbor communication along the three-dimensional mesh $i/j/k$ directions.

LULESH has three styles of communication:

- **(A)** bidirectional nearest-neighbor communication across only faces;
- **(B)** bidirectional nearest-neighbor communication across faces, edges, and corners and

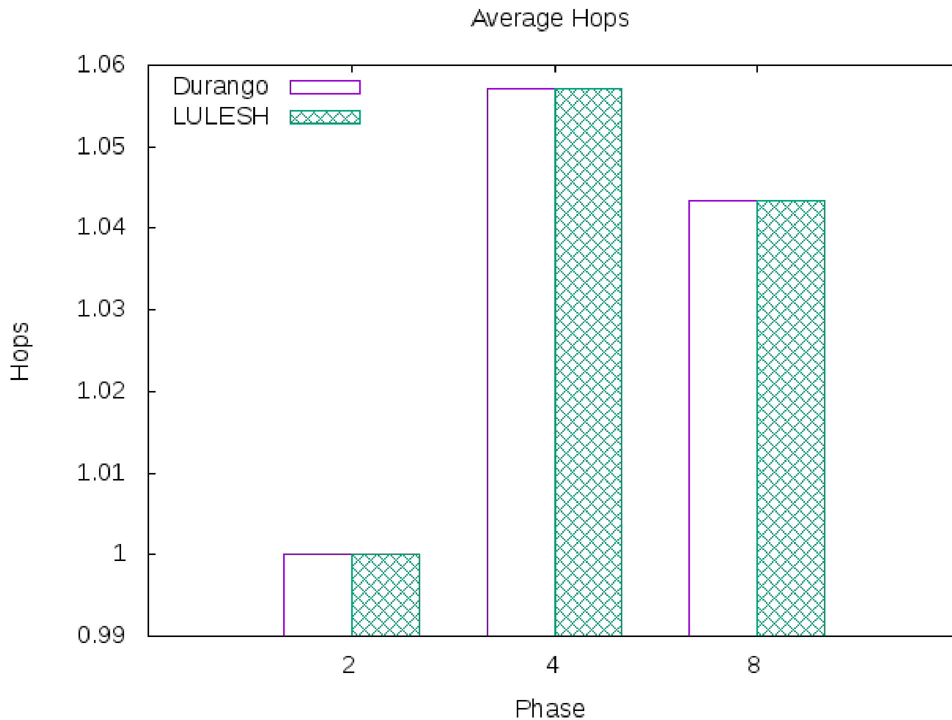


Figure 29: LULESH vs. Durango: Average torus network packet hop count as a function of the different LULESH phases.

(C) unidirectional communication across faces, edges, and corners, from a lower-rank task to a higher-rank task.

LULESH has four phases of communication, each conforming to one of these three styles. We label experiments using a bit pattern representing which phase is active: "1" for the first phase (style B), "2" for the second phase (style A), "4" for the third phase (style B), and "8" for the fourth phase (style C). This allows us to describe the behavior of the full application with multiple phases active by summing the bit patterns of every active phase. Because the first communication phase in LULESH is used only for problem initialization, we focused on the analysis of the other three phases, which occur every time step. The analysis results are labeled "2," "4" and "8" for their respective LULESH phases.

4.3.1.2 Experimental Results

In this first series of experiments, we compare the CODES network output statistics for the Durango generated and the real LULESH miniapp MPI communication traces. The key network statistics are as follows:

- **Average packet hop count:** the total number of hops traversals divided by the total number of packets sent into the network
- **Finished packets:** the total number of packets that reached their final destination

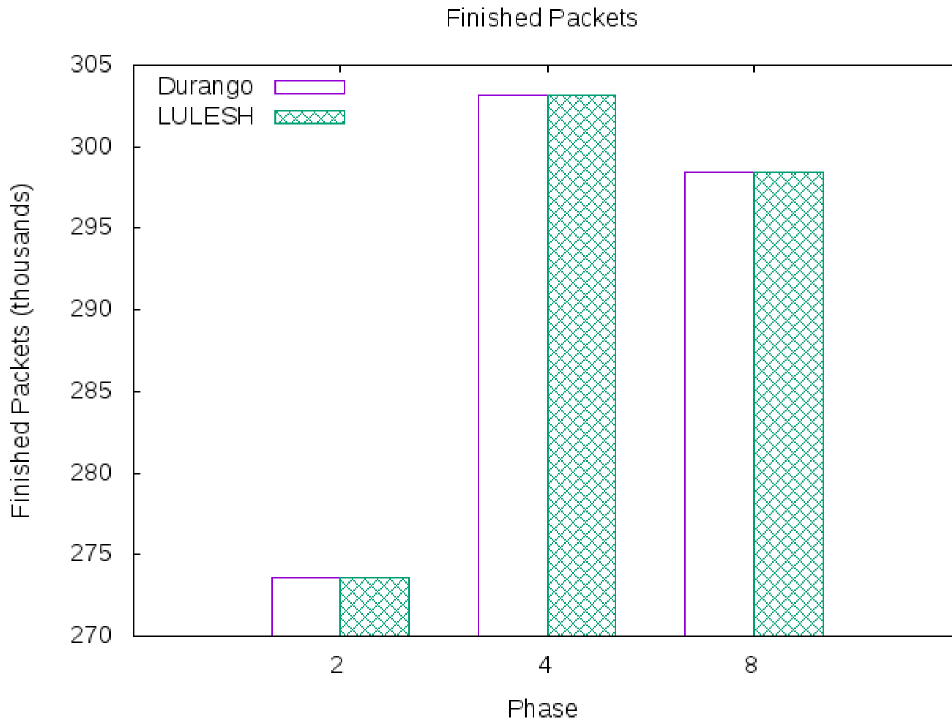


Figure 30: LULESH vs. Durango: Finished torus network packet count as a function of the different LULESH phases.

- **Finished packet hop count:** the total number of hop traversals divided by the total number of packets that reached their final destination
- **Total bytes sent:** total amount of data sent into the network

These statistics avoid any measure of time either absolute or relative such as an interarrival time. While such information is important, it largely depends on the amount of compute time between MPI messages and/or architectural features of the underlying machine model. The four statistics we use capture the communication behaviors that are independent from computation.

In validating Durango's generated LULESH communication patterns relative to the original LULESH miniapp code, we discovered an incorrect use of the `MPI_Waitall` operation by LULESH, whereas the Aspen-generated code was correct. Specifically, the number of wait requests sent is hard coded to 26 when only a small fraction (e.g., about 8) of the 26 available `MPI_Isend` and `MPI_Irecv` requests were used in this specific scenario. This error resulted in the CODES simulation prematurely terminating because of unmatched requests within the `MPI_Waitall`, which is a correct behavior for the simulator since it denotes a "bad" trace. Once provided the right number of active requests, the `MPI_Waitall` trace events were correct, and the CODES simulator completed without error. This finding underscores the potential need for a tool such as Durango beyond its benefits for flexible workload generation and modeling.

With regards to the network results, Figure 29 shows the average packet hop count as a function of different LULESH phases. The average ranges from 1.0 in phase 2 to nearly 1.06 in phase 4.

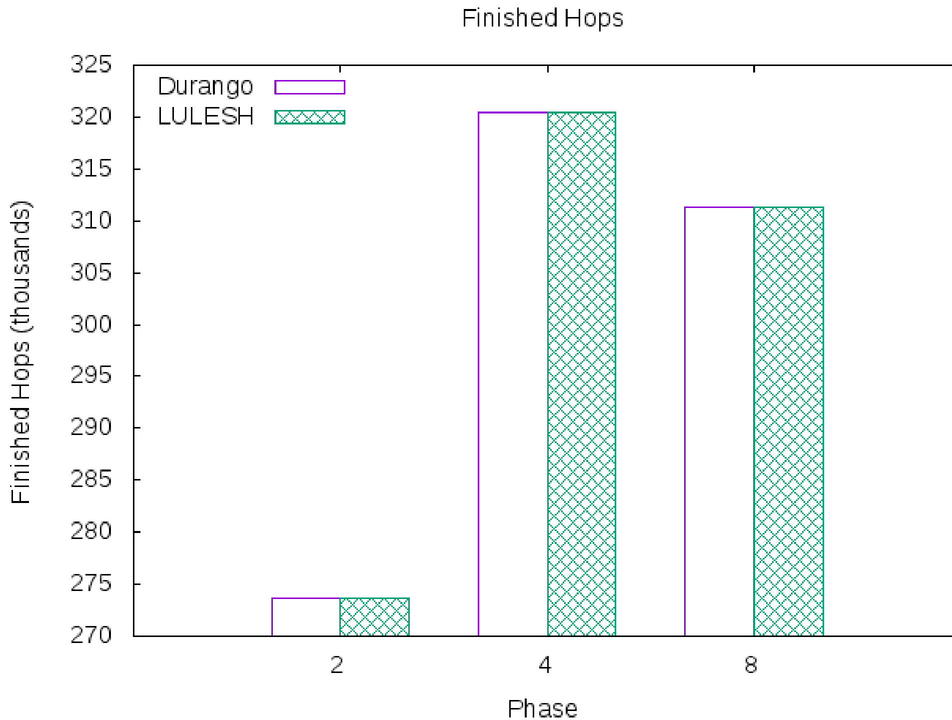


Figure 31: LULESH vs. Durango: Finished torus network packet hop count as a function of the different LULESH phases.

Across all the phases, we observe identical results between the Durango generated and the real LULESH communication patterns.

Figures 30 and 31 show the number of finished packets and hops as a function of LULESH phases. In Figure 30 the number of finished packets ranges between 272K and 320K packets; similarly, the finished hop counts are in the same range because the hop count for the majority of LULESH packets is a single hop away in the 3D torus network. Across all the phases, we observe identical results the Durango finished packets/hop count and the real LULESH finished packets/hop count.

Figure 32 reports the total number of packets sent as a function of LULESH phases. As before, we observe no difference between the Durango generator and the real LULESH miniapp. The range in packets is 140MB for phase 2 up to 151MB for phases 4 and 8.

4.3.2 Evaluation of Durango Direct Integration

The Durango direct integration approach was executed on *AMOS*, an IBM Blue Gene/Q supercomputer located at the Rensselaer Polytechnic Institute Center for Computational Innovation. *AMOS* has 5 racks, each with 1,024 nodes. Each node contains 16 GB of DDR3 RAM and one IBM A2 processor, clocked at 1.6 GHz with 16 compute cores and 64 hardware threads.

Across all tests, Durango was run using BG/Q node counts ranging from 4 to 4096 nodes and 32 to 32,768 MPI ranks. For the network simulation component, torus and dragonfly network models were configured with a nearest-neighbor traffic pattern, and a custom Aspen machine model based on the AMD processors was written for calculating runtimes with a matrix multiplication kernel

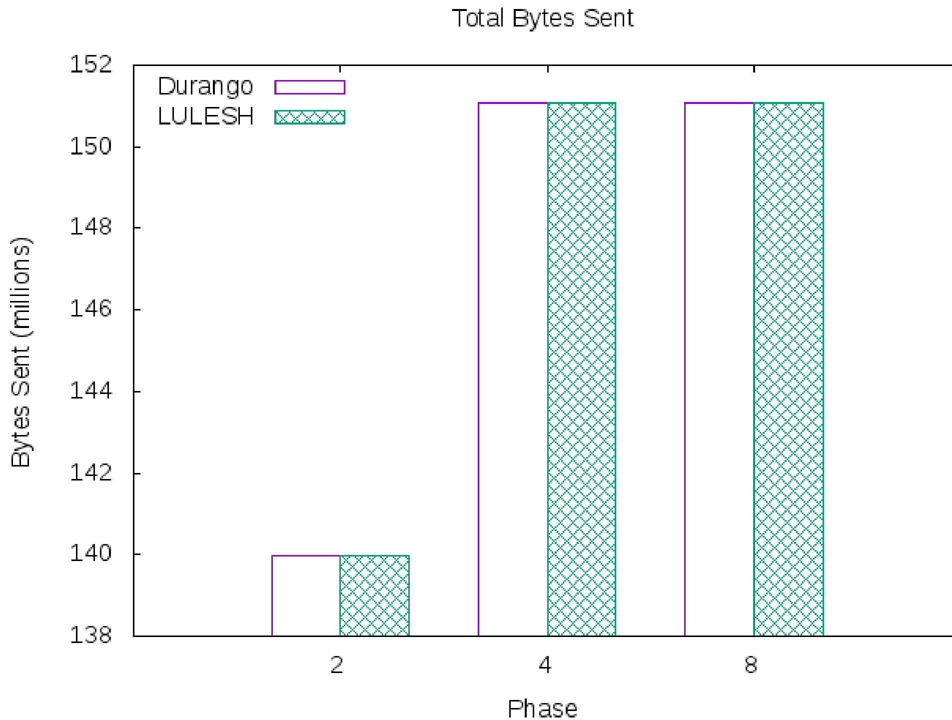


Figure 32: LULESH vs. Durango: Total torus network bytes sent as a function of the different LULESH phases.

model. The torus network is a 5D (8^5) topology yielding 32K nodes with each link having 2 GB/sec bandwidth. The dragonfly network's configuration is taken from [126] and has 1.3M terminal nodes.

Unlike the previous test, Aspen is driving the compute time between phases of nearest-neighbor communications. The purpose of this performance study is to demonstrate that the Aspen compute model does impede the parallel network simulation. Future work on Durango will enable the Aspen system model to drive both compute node timing activity and network patterns at the same time.

4.3.2.1 Runtime Configuration

Configuration of Aspen when executed in this direct integration mode with the CODES parallel simulation framework is accomplished through a primary and secondary configuration file. The primary configuration file is used to specify the Aspen compute kernel and machine models that will be utilized during each network-computation round. This configuration file also allows per-round selection of the socket on each node of the Aspen machine to be used to "run" the compute kernel, as well as the number of rounds to be simulated. The number and size of the "ping" requests sent between Aspen Server logical processes are configured as well. A sample primary configuration file is shown in part in Listing 6, where Durango has been configured to simulate two rounds of matrix multiplication on an AMD-based cluster using the CPU in the first round and the GPU in the second round over the CODES-provided "SimpleNet" testing topology. Note that in the primary configuration file, the level of debug information can be adjusted as follows:

- Level 0: supresses all debug output

- Level 1: allows configuration details to be printed
- Level 2: allows runtime progress messages to be printed

Listing 6: Aspen parameter configuration excerpt.

```
ASPEN_PARAMS
{
    debug_output="1";
    network_conf_file="simplenet.conf";
    network_traffic_pattern="random";
    num_rounds="2";
    aspen_mach_path="./models/
                    machine/BigTestRig.aspen";
    socket_choice000="amd_830";
    socket_choice001="amd_HD5770";
    aspen_app_path000="./models/matmul/
                    matmul.aspen";
    aspen_app_path001="./models/matmul/
                    matmul.aspen";
}

server_pings
{
     num_reqs="128";
     payload_sz="1024";
}
```

The second configuration file used by Durango is included in the simulator runtime by a parameter in the primary configuration file, called `network_conf_file`. The parameters in the network configuration file are directly passed on to the underlying CODES-Net framework, and allow for network topology and size settings to be adjusted. The network configuration file also controls how logical processes are organized in the simulator. Durango contributes the Aspen Server LP. Required by the CODES framework are the network-level "server" LPs and the chosen network topology's routing LPs. Listing 7 shows the basic network configuration file for the SimpleNet network topology.

LPGROUPS contains the ***ASPEN_SERVERS*** subcategory, and lists the classes and organization of the LPs in the simulation. For the SimpleNet topology the only two LP types needed are the CODES-level "modelnet_simplenet" LPs and the "server" LPs, which drive the overall Aspen-CODES simulation layer. The other two supported topologies, torus and dragonfly, also require two CODES-level LPs to function.

Listing 7: Aspen SimpleNet network configuration.

```
LPGROUPS
{
   # simplenet has a set of servers, each with
   # point-to-point access to each other
```

```
    ASPEN_SERVERS
    {
       # required: number of times to repeat
       the following key-value pairs
       repetitions="4096";
       # LP types
       server="1";
       modelnet_simplenet="1";
    }
}
# Network Params:
PARAMS
{
   # ROSS-specific parmeters:
   # - message_size:
   message_size="340";
   pe_mem_factor="512";
   # model-net-specific parameters:
   # - individual packet sizes for network
   #      operations where each "packet" is
   #      represented by an event
   # - independent of underlying network being
   #     used
   packet_size="512";
   # - order that network types will be presented
   #      to the user in
   #    model_net_set_params. In this example,
   #      we're only using a single
   #      network topology
   modelnet_order=( "simplenet" );
   # - packet scheduling algorithm
   modelnet_scheduler="fcfs";
   # - simplenet-specific parameters
   net_startup_ns="1.5";
   net_bw_mbps="20000";
}
```

`PARAMS` includes topology-specific configuration details; for the excerpt shown here include the packet size, network bandwidth, latency, and packet scheduling algorithm. The dragonfly network topology adds to this list the number of routers in each subgroup, the global and local channel bandwidths, and several other topology-specific parameters. The torus topology also adds several unique parameters to this section, including the torus dimensionality (and corresponding dimension sizes).

4.3.2.2 Performance Results

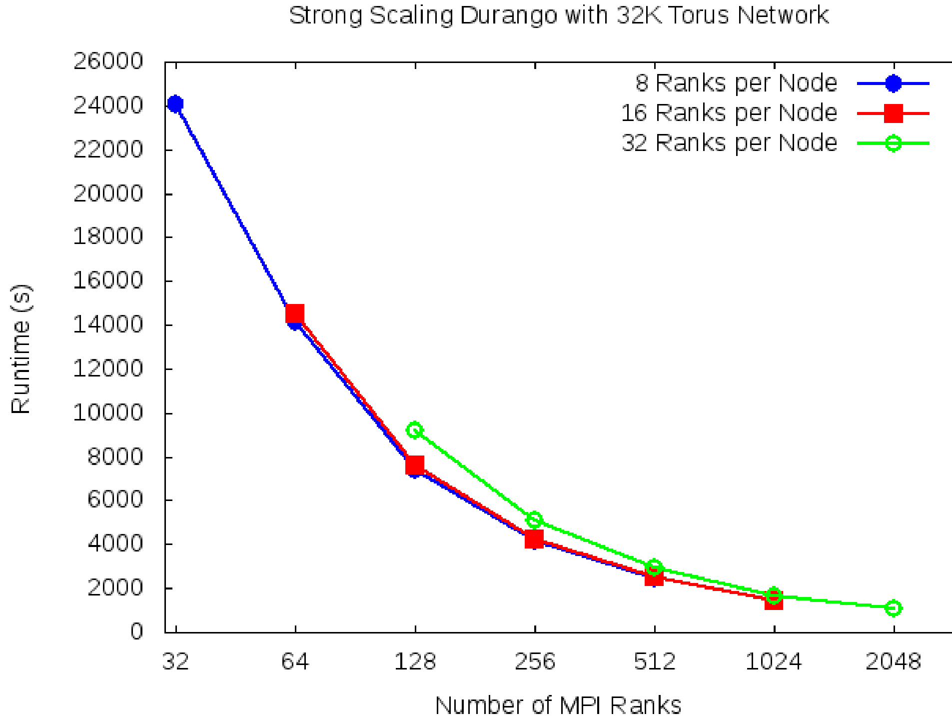


Figure 33: Durango in direct integration mode with 32K node torus network and Aspen compute node generator for 32 to 2,048 MPI ranks.

Figure 33 shows strong-scaling results for Durango when configured with a 32K node torus network and executed using 32 to 2048 MPI ranks across a varing number of ranks per compute node on the Blue Gene/Q supercomputer. Here, the execution time ranges from 24,000 seconds down to just over 1,000 seconds. This implies an overall worst-case to best case speedup of 24x using just 16x the hardware (e.g., 4 nodes to 64 nodes). This superlinear performance is attributed to performance gains because of a smaller memory "working set" which yields higher cache hit rates as the core counts increase. Similar performance gains were reported by Barnes et al. [24]. Parallel simulation efficiency ranges from a peak of 92% on 4 nodes with 32 ranks to a low of 61% on 64 nodes using 2048 ranks. This is to be expected because of the likelihood of out-of-order event computations grows as the number of nodes increase.

The out-of-order event computations become more problematic at larger MPI rank counts, as shown Figure 34. Here, the same 32K node torus network configured with an Aspen LP for determining the compute phase timing for the matrix-multiplication kernel is being scaled from 1K to 16K MPI ranks and 128 to 512 Blue Gene/Q compute nodes. The worst-case efficiency of -314% is report when using 512 nodes and 16K MPI ranks. This implies that nearly three events are being rolled back for each forward event per the efficiency definition used in [24]. Clearly, the simulation has become overly speculative. The fastest execution case is with 256 nodes and 4,096 MPI ranks and completes in 726 seconds.

If we increase the network simulation workload by using a much larger network, we observe

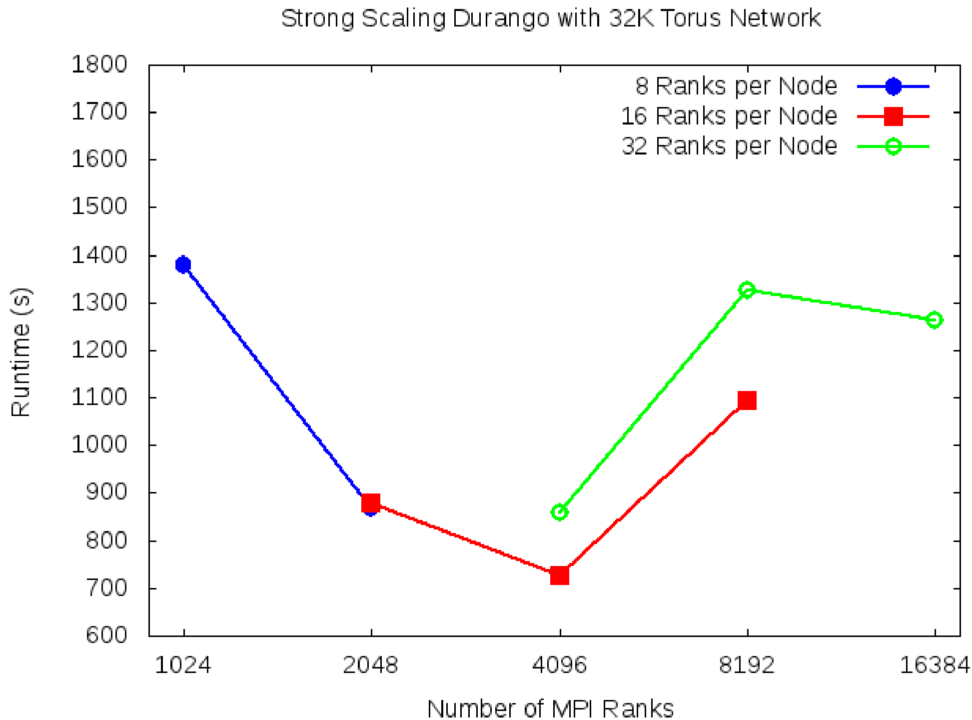


Figure 34: Durango in direct integration mode with 32K node torus network and Aspen compute node generator for 1K to 16K MPI ranks.

much better Durango performance, as shown in Figure 35. Here, a 1.3M node dragonfly network is configured with the Aspen computation timing event generation. The parallel simulation run with 128 nodes and 1K MPI ranks completes in 24,000 seconds while the run with 4,096 node and 32K MPI ranks completes in just 2200 seconds. The observed performance is nearly a 11x speedup for 32x the hardware, which is in line with results reported in previous dragonfly network simulation studies [127].

Overall, the integration of Aspen's code timing event generation does not impact the overall Durango performance. Also, we avoid the performance penalty of reading in large, memory-intensive traces datasets.

4.4 HPC Network Models

4.4.1 Model Validation

4.4.1.1 Slim Fly

We begin with a comparison with published Slim Fly network results by Kathareios et al. [95] to validate the implementation of our Slim Fly model. Validation with a real testbed Slim fly system is optimal. Unfortunately, we do not currently have access to such a system so we resort to published simulation results from the creators of the Slim Fly topology. The specifics of the IBM-ETH-SF simulator that we compare with are not provided, but the authors do mention that it is based on the OMNeT++ simulator, which also employs parallel discrete-event simulation [159]. This

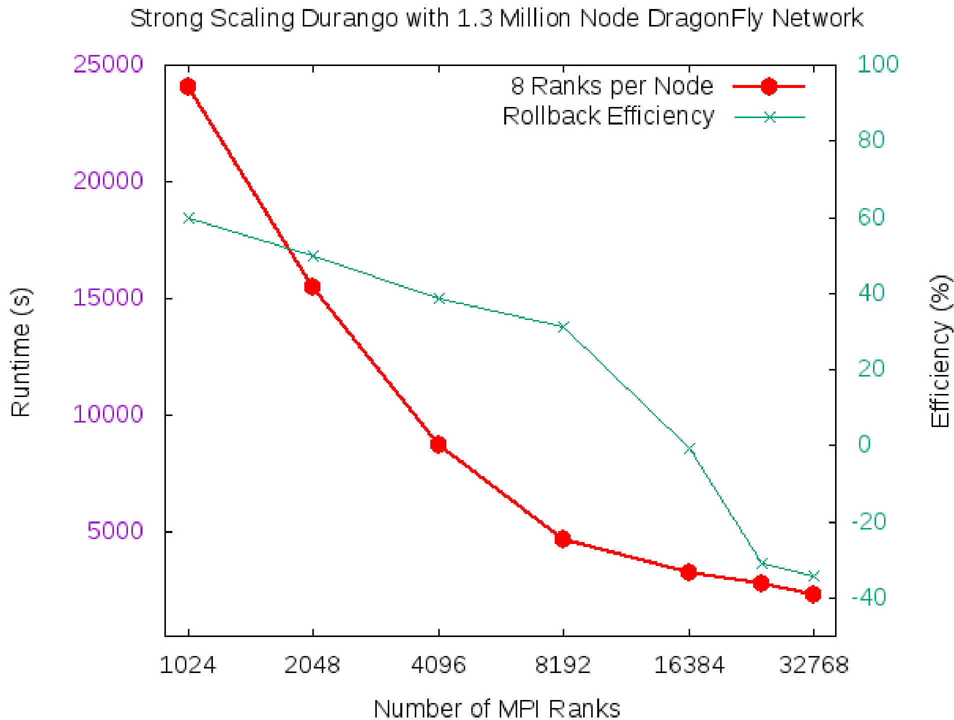


Figure 35: Durango in direct integration mode with 1.3M node dragonfly network and Aspen compute node generator for 1K to 32K MPI ranks.

IBM-ETH collaborative work presents throughput results for a slim fly network with the 3K-node configuration.

Slim Fly Synthetic Traffic Workloads: To accurately simulate and analyze the network communication using a slim fly interconnection topology, we leverage two traffic workloads. The first workload is the uniform random (UR) traffic pattern which sends packets to randomly selected compute nodes anywhere in the network. The second workload is a worst-case (WC) traffic pattern designed specifically as the worst-case adversarial workload for Slim Fly minimal routing algorithm. The WC workload leverages the minimal path information to simulate an application that pairs compute nodes for communication such that their minimal paths all take the maximum two hops and all flow through a select few links in the network and thus creates a bottleneck for minimal routing. In this workload, each compute node in a router, R1, will communicate to a node within a paired router that is the maximum two hops away. Another pair of routers that share the same middle link with the previous pair of routers will be established to fully saturate that center link. As shown in the example in Figure 36, all compute nodes connected to R1 communicate with nodes connected to R3 along the blue path. Also, the reverse communication is true, because all nodes connected to R3 communicate with nodes connected to R1 along the red path. The router pair R2 and R4 are set up in the same manner communicating along the gray and green paths, respectively. This setup of network communication puts a worst-case burden on the link between routers R2 and R3 as $4p$ nodes are creating $2p$ data flows (where p is the number of compute nodes in the system). With all nodes paired in this configuration, congestion quickly builds up for all nodes in the system and limits maximum throughput to $1/2p$.

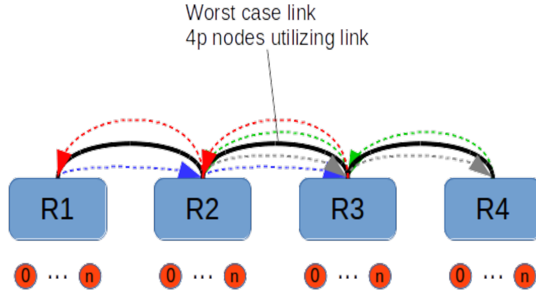


Figure 36: Worst-case traffic layout for the slim fly topology.

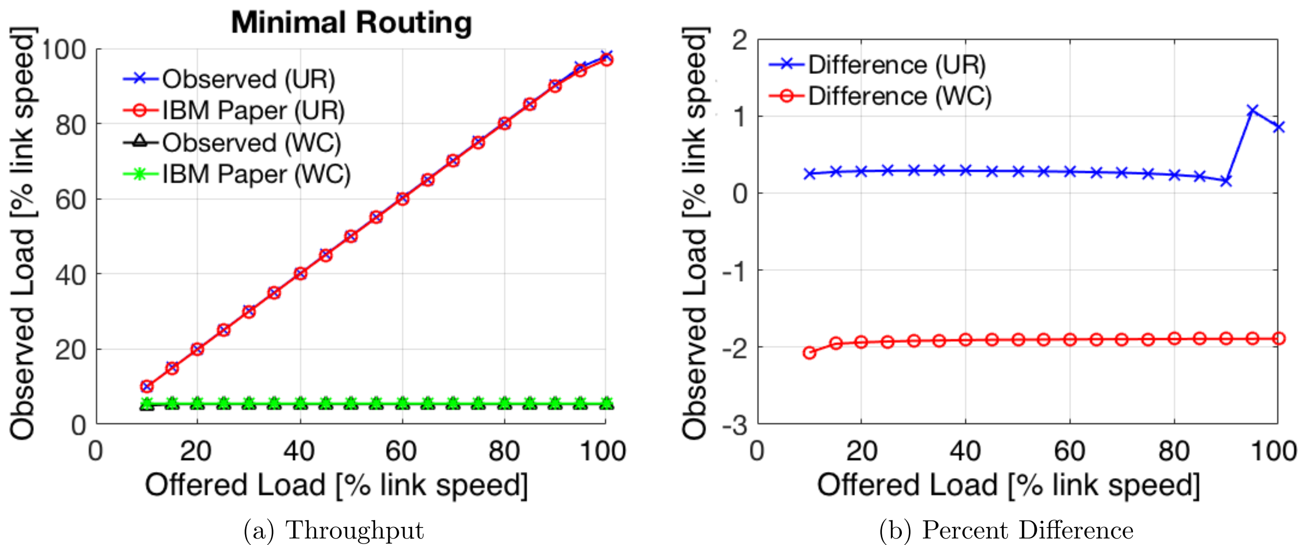
Synthetic workloads generate and inject messages into the network following an input *load* variable. The *load* governs the percent of linkspeed to inject messages of a given size into the network. By providing the *load* and message size, the simulation can calculate the ns time delay between message injections such that the injection rate is *load*% of linkspeed.

Slim Fly Routing Validation: The necessary simulation parameters are disclosed by the IBM-ETH simulator and replicated in our Slim Fly simulation for an accurate comparison of both UR and WC workloads using minimal, non-minimal and adaptive routing algorithms. The simulations are set up to use a 100 Gbps link bandwidth for all links with a latency of 50 ns. The routers utilize virtual channels, a buffer space of 100 KB per port (equally divided among the VCs), and a 100 ns traversal delay. Flow control is done with the use of credits, and messages are 256 byte packets. Simulation time for the IBM-ETH-SF was 200 μs with a 20 μs warmup. In our simulation, we include the warmup time in the total execution and therefore run the simulation for 220 μs. The results include minimal, non-minimal, and adaptive routing for uniform random and worst-case traffic workloads. Our simulation results in comparison with the IBM-ETH-SF results are presented in Figures 37, 38, and 39. The metric used for comparison is throughput percentage and is a measure of observed system throughput as a percentage of the aggregate network injection load. The $observed_throughput$ is obtained from our Slim Fly model by performing a sum reduction to get the total number of packets transferred by all compute nodes, multiplying by the 256 byte packet size and dividing by the total number of compute nodes.

$$throughput_percent = \frac{observed_throughput_Gbps}{0.71 * 100Gbps} * 100 \quad (20)$$

Slim Fly Minimal Routing Comparison: Figure 37 presents the throughput analysis for the minimal routing algorithm under input loads varying from 10% to 100% link bandwidth. Focusing on the uniform random workload case, our Slim Fly model closely matches that of the IBM-ETH-SF. As expected, the minimal routing algorithm excels under uniform random workloads. Both simulations show the Slim Fly network throughput matching the injection load from 10% load to about 95% load, at which point the throughput trails off to roughly 98% throughput at 100% load. The difference between the two simulators is consistently under 1%. In the worst-case workload results, the two results are again a close match within 2% difference of one another. Both show roughly 5.5% throughput utilization from 10% to 100% load.

Slim Fly Non-minimal Routing Comparison: The results comparing throughput for non-minimal routing are shown in Figure 38. In this case, the results also compare favorably as both



(a) Throughput

(b) Percent Difference

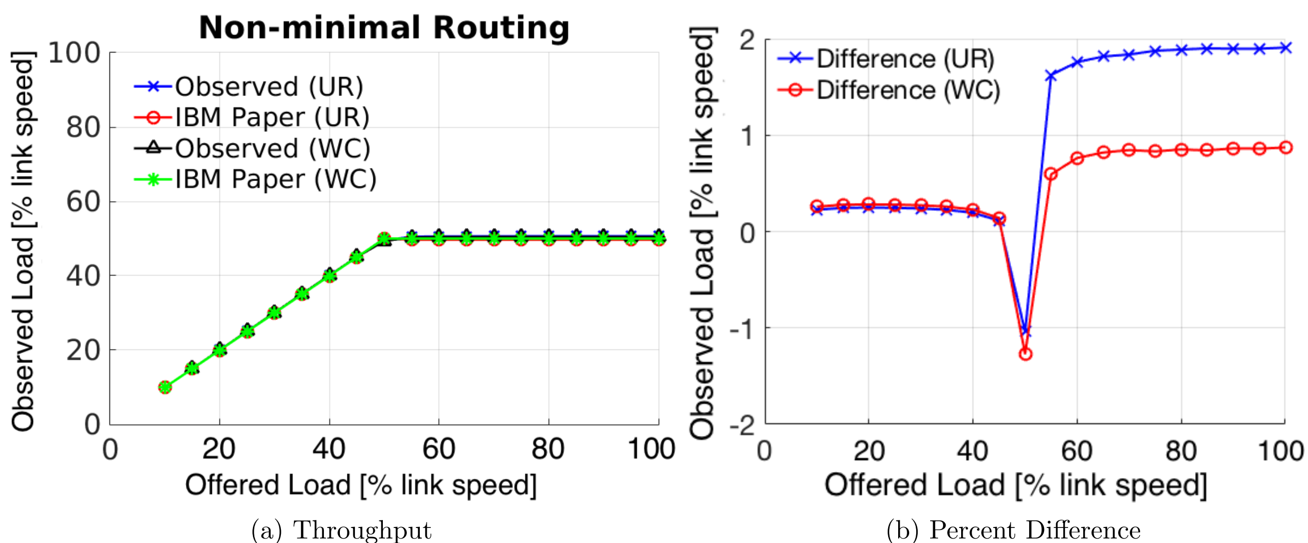
Figure 37: Throughput comparison of minimal routing for uniform random (UR) and worst-case (WC) traffic workloads. Figures are best viewed in color.

simulators observe less than 2% difference. Under both uniform random and worst-case traffic routing, the Slim Fly network achieves a throughput equal to the injection load until 50% load is reached. At this point, the network throughput reaches a bottleneck and maintains just under half-link bandwidth up to 100% injection load. Non-minimal routing under-performs compared with minimal routing for uniform random traffic because the maximum path length of all routes is twice as long at four hops compared with two hops in minimal routing. Therefore, non-minimal routing reaches congestion in UR traffic at roughly 50% load, roughly half the load of minimal routing. However, non-minimal routing outperforms minimal routing in worst-case traffic because of its ability to perform a uniform load balancing of traffic as it selects a random intermediate router along its path.

Slim Fly Adaptive Routing Comparison: The throughput comparison results for the adaptive routing algorithm are shown in Figure 39. In all cases, we set the number of indirect routes, $n_i = 3$, and balancing constant, $c_{SF} = 1$. Once again, the observed results for our Slim Fly model agree with those of the IBM-ETH-SF simulator with no more than 2% difference. In both uniform random and worst-case traffic workloads, the network throughput matches the injection load until 55% load, at which point the worst-case traffic results reach congestion and are limited at 58%. The uniform random traffic results continue with optimal throughput and reach nearly full system throughput at 100% load. Adaptive routing outperforms both minimal and non-minimal routing for worst-case traffic because of its ability to dynamically select between the minimal and non-minimal routes.

Slim Fly Validation Visualization: Continuing the analysis, we show visual representations of router occupancy and message sends and receives for both router LPs and compute node LPs during the validation simulations. These visualizations provide visual confirmation of correct packet routing functionality by explaining the low level network statistics that contribute to the Slim Fly's observed load thresholds under the synthetic traffic.

The router occupancy metric collects the number of packets sitting in queue waiting for space to



(a) Throughput

(b) Percent Difference

Figure 38: Throughput comparison of non-minimal routing for uniform random (UR) and worst-case (WC) traffic workloads. Figures are best viewed in color.

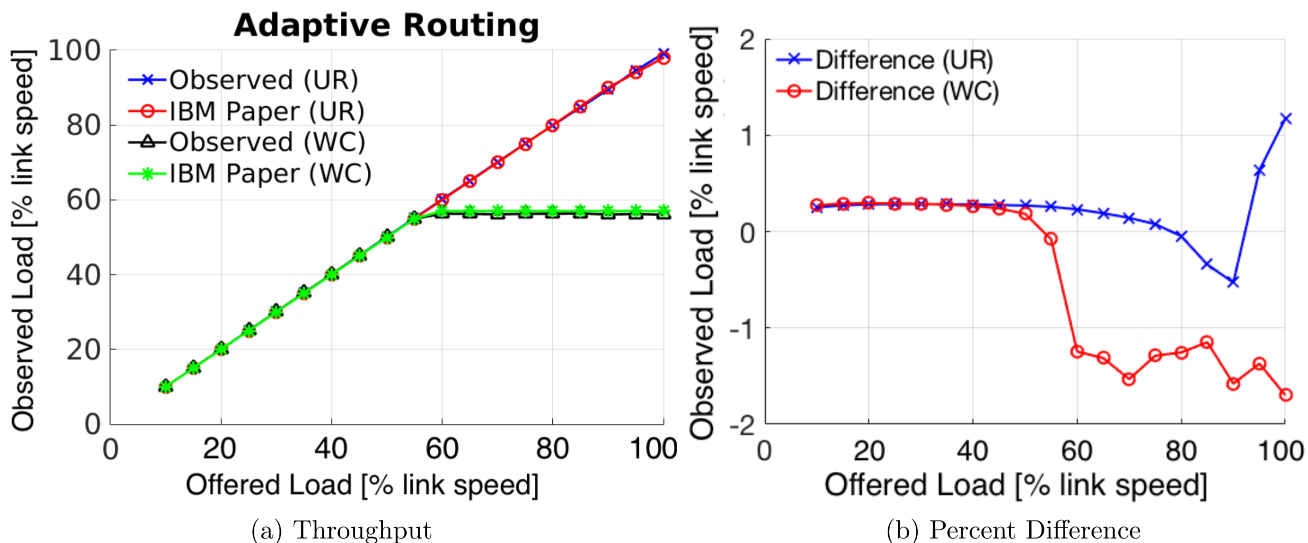
open up on the necessary router output port. Since we use VCs for deadlock avoidance, the Slim Fly router occupancy metric can be further divided into virtual channels with 2 VCs per port per router used in the case of minimal routing and 4 VCs per port per router used in non-minimal routing.

Figure 40 presents a number of graphs visualizing the occupancy of all virtual channels for all ports on all routers in the simulation. All four Slim Fly test cases are from simulation data displayed in Figure 37, in which we run the 3K-node Slim Fly model using minimal routing for uniform random traffic. As shown in Table 11, the 3K-node Slim Fly consists of 338 routers with a network radix of 19 and 2 VCs per port. The result is a total of 6,422 ports, each with 2 virtual channels. Figures 40a–40d display the occupancy of VC0 with increasing load from 50% to 100%, and Figures 40e–40h display the same for VC1.

The 3K-node Slim Fly model experiences little congestion for this traffic pattern until about 90% injection load, where VC0 sees a uniform distribution of roughly 20% congestion in the network. At 100% injection load, the network begins to reach the buffer space limit as packets enter the network at an increased rate, further explaining why we see a slight dip in throughput for minimal routing under uniform random traffic in Figure 37. The VC0 buffer fills up first, because it is always used for the first hop in a packet's path. VC0 is always the gate.

In addition to buffer occupancy, the number of message packets sent and received by all routers and compute nodes is visualized over the simulation time. The results are collected during the same simulation in Figures 40d and 40h and are displayed in Figure 41. The first noticeable feature is the large spike in the beginning of the compute node sends (Figure 41a). Occurring at the beginning of the simulation, this spike is a result of the initial packet burst into the system, which is followed by a balancing out as the network reaches a steady state. The same phenomenon is reflected as a slow start in router sends and receives plots in Figures 41c and 41d. These figures all resemble the uniform random traffic workload being simulated, except for the initial start-up phase.

The added capability of visual analysis of these important network metrics at this fidelity not only



(a) Throughput

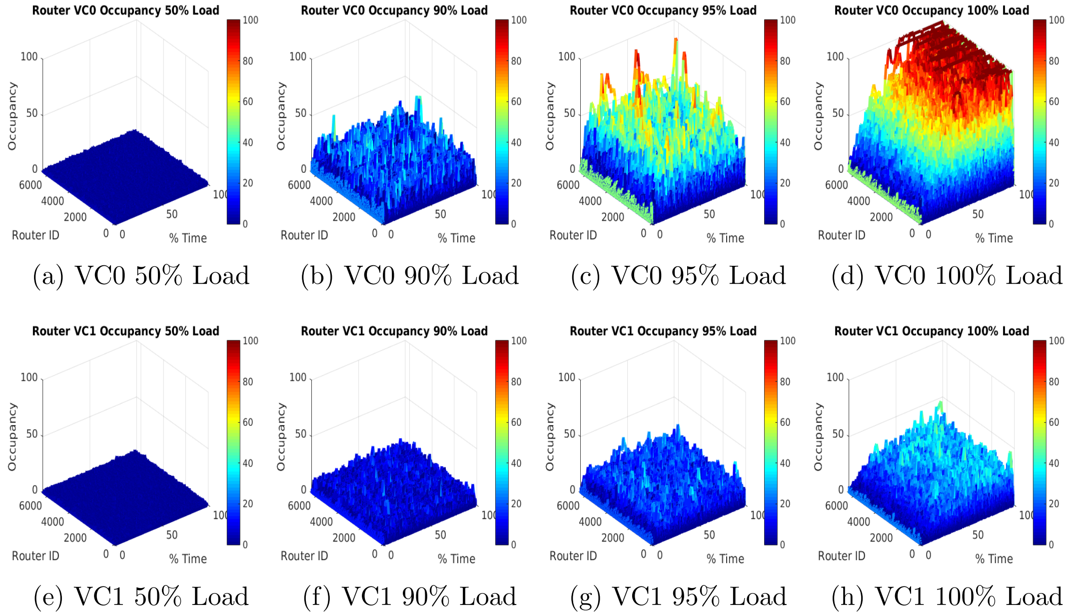
(b) Percent Difference

Figure 39: Throughput comparison of adaptive routing for Uniform Random (UR) and worst-case (WC) traffic workloads. Figures are best viewed in color.

helps detect network congestion but also helps identify the time, location, and effect of congestion on the entire network simulation.

Extreme-Scale Slim Fly: Here, we validate large-scale Slim Fly networks with 74 K and 1 M compute nodes to confirm correct model performance at extreme-scale. We again confirm both large-scale configurations achieve expected performance using all three routing algorithms—minimal, non-minimal, and adaptive—under both uniform random and worst-case traffic workloads. Both throughput utilization for each system and average packet latency are shown in Figure 42. The results follow the same general trend as was observed for the 3K-node Slim Fly verification. Minimal routing performs at nearly full bandwidth under uniform random traffic. Simulating worst-case traffic, minimal routing maintains roughly half the throughput achieved with 10% load of uniform random traffic. non-minimal routing hits congestion at 50% load under both uniform random and worst-case traffic. Again, this is the result of the non-minimal routing algorithm forcing path lengths to be twice as long as minimal routing. Adaptive routing achieves better throughput over minimal and non-minimal routing for worst-case traffic because it has the added benefit of selecting between the minimal or non-minimal route.

Unlike the 74K-node Slim Fly performance, which trails off to a maximum throughput of 8.2 GBps, the million-node model achieves a maximum of 8.7 GBps. As shown in the Slim Fly network configuration parameters of Table 11, the million-node configuration has only 19 compute nodes per router, well below the suggested provisioning of $p = \lfloor \frac{k'}{2} \rfloor$ nodes per router. Therefore, the million-node Slim Fly model does not experience any congestion under uniform random traffic with minimal routing because there is ample network bandwidth to satisfy the much smaller injection load bandwidth.



(a) VC0 50% Load (b) VC0 90% Load (c) VC0 95% Load (d) VC0 100% Load

(e) VC1 50% Load (f) VC1 90% Load (g) VC1 95% Load (h) VC1 100% Load

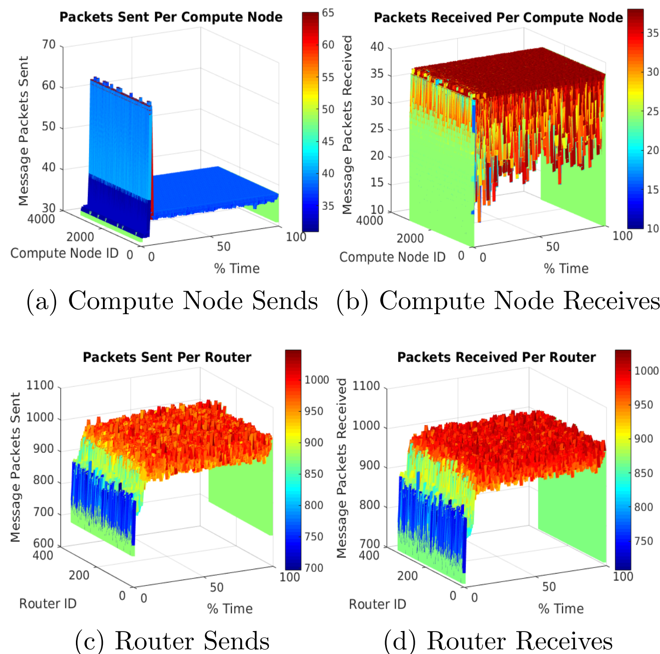
Figure 40: Router occupancy comparison for simulations as a function of the Router ID and percentage of simulation time across different offered loads for virtual channels (VCs) 0 and 1 using UR traffic and minimal routing with increasing injection load. Figures are best viewed in color.

4.4.1.2 Fat-Tree Validation:

The fat-tree network simulation model used in this paper is an extension of previously validated work [90]. Previously, the Fat-Tree network was capable of fully provisioned single-rail networks. In this work, the CODES Fat-Tree model has been extended to include the simulation of pruned Fat-Tree configurations with multiple rails. In this section we validate the extensions by showing the new configurations still maintain full bisection bandwidth under a bisection workload. Additionally, we further validate the CODES Fat-Tree model by comparing results with a physical Fat-Tree system in a controlled MPI packet latency test.

Fat-Tree Routing Validation: In this validation study we ensure that a model configured with a pruned Fat-Tree and multiple rails still observes expected full-bisection bandwidth. For this test we use the single-rail 3,240 node pruned Fat-Tree previous described as the base configuration and then scale the number of rails to generate models with two, four, and eight rails. The tested synthetic workload is based on bisection pairing, which establishes unique compute node pairs in the network to transfer messages. Using static routing in combination with the bisection-pairing results in each pair of compute nodes having their own distinct path in the fat-tree network to transfer messages without any contention.

All nodes should have an observed throughput performance equal to the injection load up until 100% link bandwidth, at which point all links of a single-rail network will be fully saturated. When injecting beyond 100%, each additional rail in the network should provide a corresponding increase in observed bandwidth above the 100% injection load. These expectations are validated in Figure 43, which shows the observed network throughput matching the injection load. This confirms the expected performance for multi-rail networks under ideal workload conditions.



(a) Compute Node Sends (b) Compute Node Receives

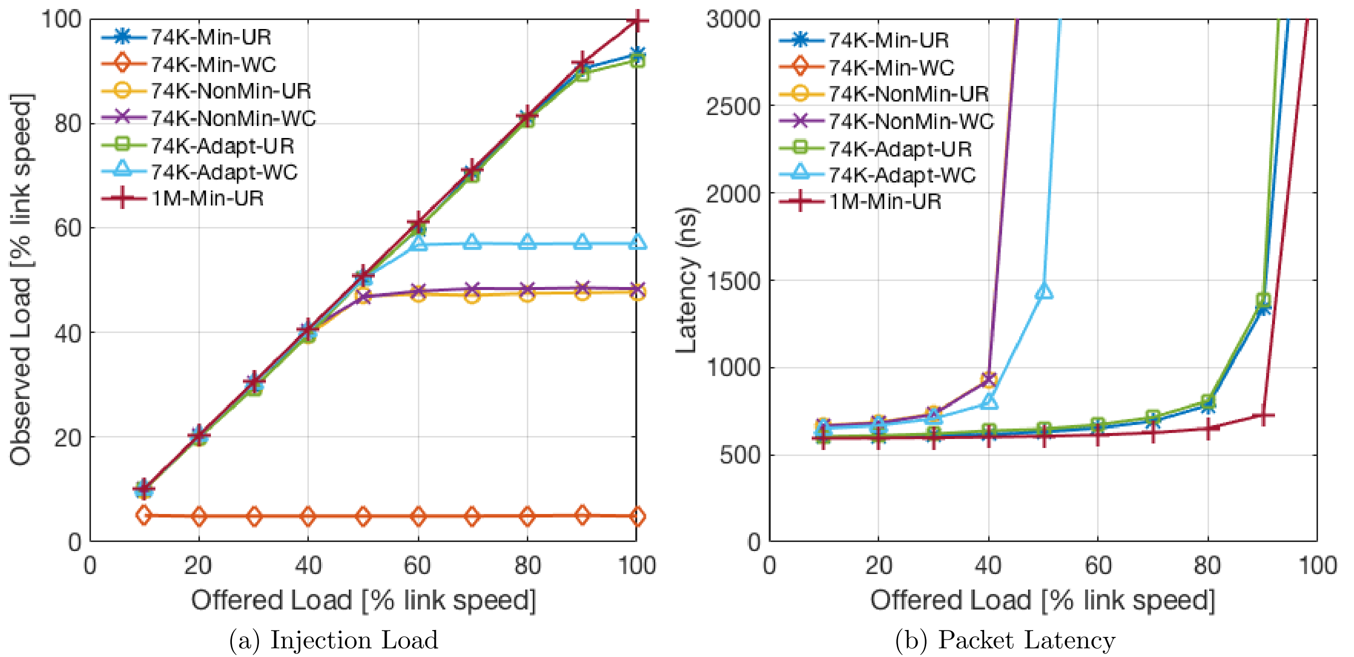
(c) Router Sends (d) Router Receives

Figure 41: Messages sent and received over time for the simulation across different router IDs using UR traffic and minimal routing using 100% load. Figures 41a and 41b show the number of sends and receives sampled over the simulation run time for all the compute nodes. Figures 41c and 41d show the same for all routers in the simulation.

Fat-Tree Network Hardware Validation: Previously, work has been performed to validate the Fat-Tree simulation model with hardware results using the Trace-R trace replay utility [13]. The experiments were conducted to see how well the simulator predicts run time performance of synthetic workloads with increasing message sizes and mini application benchmarks [91]. The simulator results were compared with a 2:1 tapered two level Fat-Tree. The summarized results showed the Fat-Tree model consistently predicting synthetic ping-pong traffic within 10% error of hardware with no more than 30% error.

In this test we use the DRP cluster at the Rensselaer Polytechnic Institute Center for Computational Innovations to validate the accuracy of the CODES Fat-Tree model using the DUMPI trace replay utility [144]. The DRP is a 64 node Intel based system interconnected via a non-tapered 2-level Fat-Tree network with 56Gbps FDR Infiniband. Testing is performed using the ping-pong benchmark provided by the MPPTest program. MPPTest is a program consisting of of multiple MPI message passing routines specifically for accurately measuring MPI performance at a high resolution [74]. The ping-pong workload (labeled as “roundtrip” in MPPTest [7]) is a traffic pattern that first sends a message of selected size from the source MPI process to a selected destination process. Upon receiving the message, the destination MPI process then sends an acknowledgement message of the same size back to the source MPI process. The time reported is the end-to-end latency from the time the first message is generated, to the time the acknowledgment message is received.

For realistic simulation results, constant latency and overhead delays for the software layer and hardware devices are extracted from the DRP system for accurate representation in the CODES Fat-



(a) Injection Load (b) Packet Latency

Figure 42: Distribution of simulation time for the 74K-node Slim Fly model with minimal routing, UR traffic, and 10% load.

Tree simulation model. Testing with increasing message sizes showed a sharp change in performance going from 4KB to 8KB message sizes indicating a change in MPI transfer protocol from eager to rendezvous, so we set the eager threshold in the simulator to 8KB. Sending a zero byte message between the same source and destination MPI process showed a combined MPI overhead and NIC delay of $1.25us$. In the simulator we use an MPI overhead value of $1us$ and NIC delay of $250ns$. Router traversal delay is set to $90ns$ and was extracted from sending a zero byte sized message between two MPI processes connected to the same switch and subtracting the end time from the MPI self message found earlier. Finally, the overhead associated with copying messages using the MPI eager protocol is set to $0.65ns$ after observing the increase in end time associated with sending increasing messages from 4B to 8KB.

The network performance of the DRP system is first measured using the MPPTest ping-pong benchmark workload. The end-to-end latencies are collected for increasing message sizes doubling from 4B to 32KB sent between two MPI processes. Each process is mapped to a compute node in the system located the maximum distance apart separated by three intermediate switches. The ping-pong pattern is repeated and averaged over 2,000 message iterations to alleviate the effect of performance anomalies. At the same time the ping-pong workload is run through the DRP system, the DUMPI tracing library is used to record and save the actual ping pong benchmark traffic in a dumpi trace file. With the necessary latencies and MPI parameters extracted and set in the simulator, each of the collected workload traces are replayed in CODES through a similarly configured Fat-Tree model to compare end-to-end latency. In order to remove the effects of communication interference from other active jobs running on the system, test runs of the Ping Pong benchmark were made during a maintenance day after given solitary access to the system.

Figure 44 presents the average message end-to-end latency for both the DRP system and the

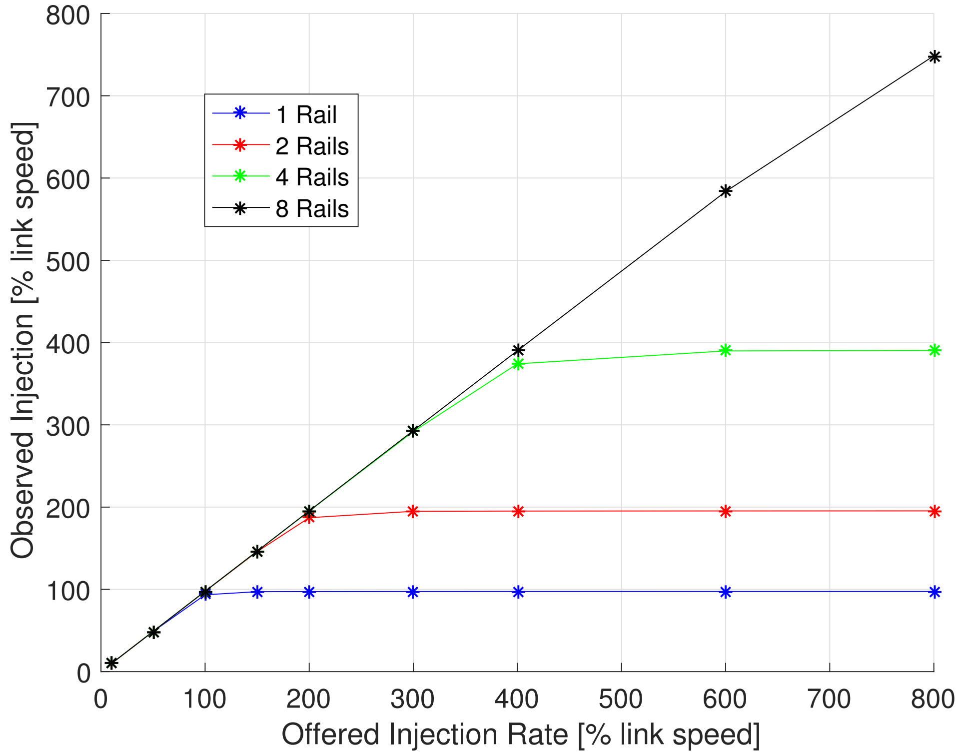
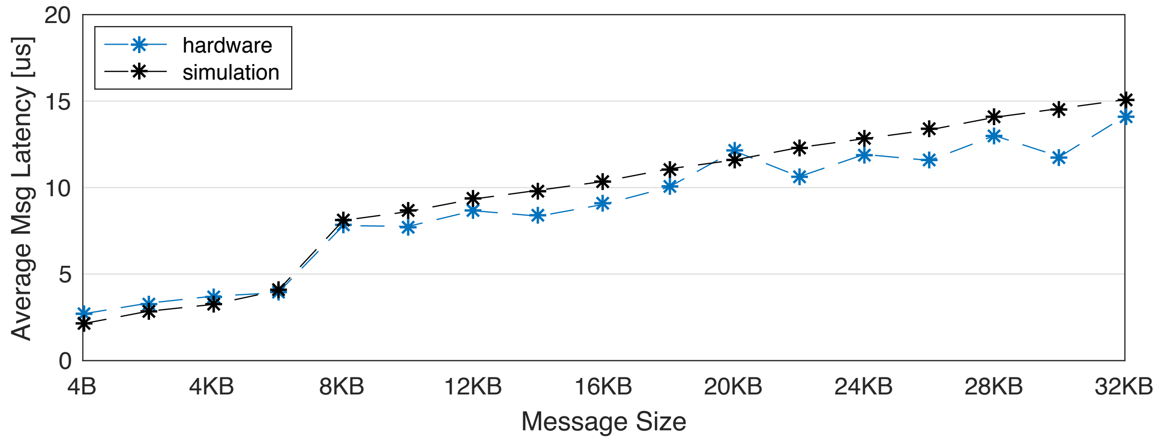


Figure 43: Verification experiments showing observed network performance while increasing the number of rails under a synthetic bisection-pairing workload. Both axes show the relative injection in percent of the link speed (12.5 GB/s).

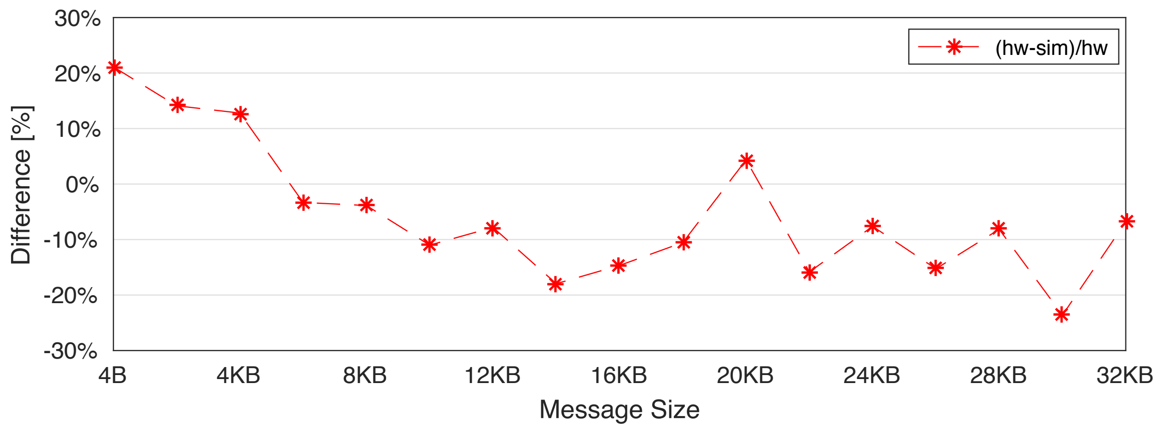
CODES Fat-Tree simulation. Overall, the simulation results match the trend of the DRP hardware system. From the 4B message size to 6KB we see a constant increase in latency resulting from the eager protocol copy overhead. At 8KB the protocol switches and we again see a constant increase in latency resulting from the increased transmission delays for the larger message sizes. While the simulation results show very consistant increases, the hardware results show variance resulting in up to 20% difference with simulation predictions. The differences are believed to be the result of operating system jitter as the current configuration of the linux operating system for the DRP nodes does not allow pinning operating system task threads to one specific core. Instead, threads are able to migrate across cores and can interfere with performance measurements especially at the larger message sizes where the results show some of the most variance.

4.4.2 Experimental Setup

All networks used in this thesis are configured with the parameters shown in Tables 10 and 11. Table 11 lists and compares the network configurations of selected Dragonfly-1D, Dragonfly-2D, single-rail Fat-Tree and dual-rail Fat-Tree topologies. Table 10 lists the Slim Fly configurations. The 74K and 1M node Slim Fly configurations are used for large-scale analysis and validation of the Slim Fly model. Each of the remaining network configurations are leveraged in prior the network performance comparison evaluations and are therefore generated to provide a fair comparsion prior-



(a) Round Trip Time

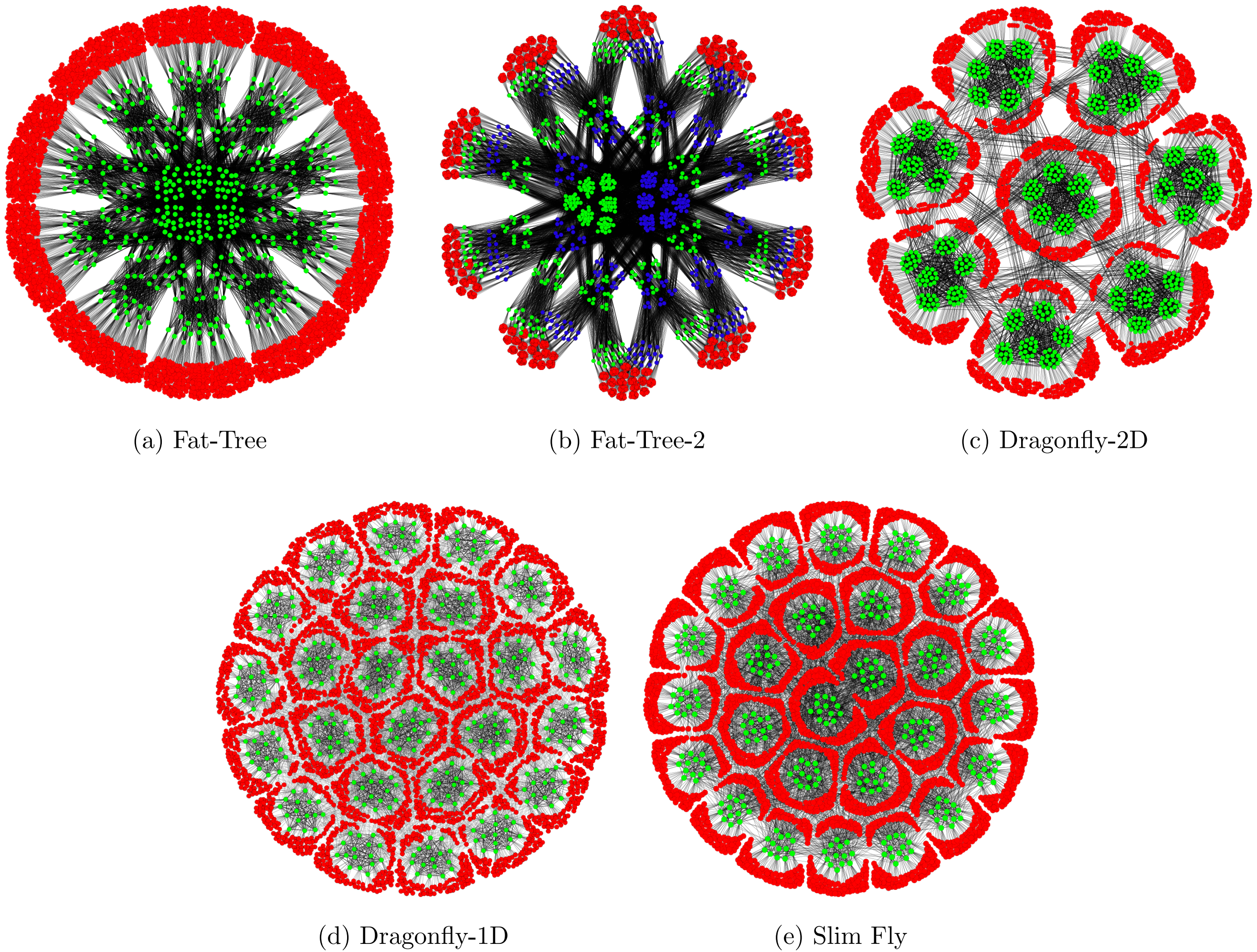


(b) Percent Difference

Figure 44: Validation results comparing network performance between the physical DRP system and a similarly configured simulation system using the CODES Fat-Tree model. Both simulation and hardware perform the ping-pong benchmark pattern between two MPI processes on separate compute nodes. Subfigure 44a shows the packet end-to-end latencies and Subfigure 44b presents the error between hardware and simulation.

itizing similar compute node counts of roughly 3,000 nodes. Other parameters such as router radix are unrestrained to avoid placing design constraints which limit the specific benefits of the different topologies. Figure 45 provides a visual representation of each 3K node system configuration showing the overall composition and connectivity of each network topology.

All parameters independent of the network topology such as link speed, buffer space, and router latency are the same across all six of the 3K node systems to maintain a fair comparison. Also, buffer spaces are allocated per-vc and per-direction. The system diameter indicates the maximum number of links between any two compute nodes in the configured system. Max path length indicates the maximum number of link traversals between any two compute nodes using the selected routing protocol. For adaptive routing, the max path length is given from the worst-case non-minimal path. Table entries for local, global and compute node connections are per router. Router Radix Total describes the commodity type switch used for each configuration and Router Radix Used indicates how many ports are utilized. Injection bandwidth ($total_terminal_to_router_links * link_speed$) and



(a) Fat-Tree (b) Fat-Tree-2 (c) Dragonfly-2D

(d) Dragonfly-1D (e) Slim Fly

Figure 45: Visualization of the networks utilized in this paper. Green and blue nodes are routers (green being routers in the first plane/rail and blue routers being in the second plane/rail, if one exists). Red nodes are compute nodes.

network bandwidth ($total_router_to_router_links * link_speed$) calculations are provided for each configuration in Tables 10 and 11 showing a simple high level comparison of bandwidth available for compute nodes to inject traffic into the network and bandwidth available for transferring the data throughout the network.

4.4.2.1 Slim Fly Model Configurations

It's important for a network simulator to provide the capability to model a wide range of network sizes in order to keep up with ever-changing node counts for HPC systems. We test this capability in the CODES Slim Fly model by constructing and simulating networks from 3K to 1M compute nodes.

3K-Node Slim Fly Model: This configuration is of particular interest because it yields a total number of compute nodes that is similar to the number of nodes in the Summit supercomputer housed at Oak Ridge National Laboratory [133]. Using 338 routers, each with a radix of $k = 28$ and 9 compute node connections, results in a network size of 3,042 nodes.

Table 10: Network configurations (Fat-Tree & Dragonfly).

Metric	***Fat-Tree*** **Single**	***Fat-Tree*** **Dual**	***Dragonfly*** **1D**	***Dragonfly*** **2D**
Approximated System	Summit	Summit	n/a	Theta
Network Diameter	6	6	5	7
Max Path Length	6	6	8	12
Avg Path Length	5.79	5.79	4.80	6.51
Levels	3	3	n/a	n/a
Pods/Groups	10	20	25	8
Routers per Pod/Group	36	36	16	96
Planes/Rails	1/1	2/2	1/1	1/1
Compute Nodes	3,240	3,240	3,200	3,072
Routers	468	936	400	768
Total Links	9,720	19,440	8,600	11,424
Router Radix Total	36	36	36	48
Router Radix Used	36	36	35	42
Router Latency	90ns	90ns	90ns	90ns
Routing Protocol	static	static	adaptive	adaptive
Virtual Channels	1	1	6	8
Buffer Space per VC	64 KB	64 KB	64KB	64 KB
Local Connections	18	18	15	30
Global Connections	n/a	n/a	12	8
Compute Node Connections	18	18	8	4
Link Speed	100 Gbps	100 Gbps	100 Gbps	100 Gbps
Injection Bandwidth	316 Tbps	632 Tbps	313 Tbps	300 Tbps
Network Bandwidth	648 Tbps	1.55 Pbps	540 Tbps	835 Tbps

Approved for Public Release; Distribution Unlimited.

Table 11: Network configurations (Slim Fly).

Metric	***Slim Fly* 3K**	***Slim Fly* 74K**	***Slim Fly* 1M**
Network Diameter	4	4	4
Max Path Length	6	6	6
Avg Path Length	3.93		
Groups	26	74	326
Routers per Group	13	37	163
Planes/Rails	1/1	1/1	1/1
Compute Nodes	3,042	73,926	1,009,622
Routers	338	2,738	53,138
Total Links	6,253	149,221	4,394,123
Router Radix Total	36	108	324
Router Radix Used	28	82	255
Router Latency	90ns	90ns	90ns
Routing Protocol	adaptive	adaptive	adaptive
Virtual Channels	4	4	4
Buffer Space per VC	64 KB	64KB	64 KB
Local Connections	6	18	73
Global Connections	13	37	163
Compute Node Connections	9	27	19
Link Speed	100 Gbps	100 Gbps	100 Gbps
Injection Bandwidth	297 Tbps	7.0 Pbps	96 Pbps
Network Bandwidth	321 Tbps	7.2 Pbps	323 Pbps

Approved for Public Release; Distribution Unlimited.

74K-Node Slim Fly Model: Here, we describe a Slim Fly model at the expected scale of Aurora, the future supercomputer to be deployed at Argonne National Laboratory. Aurora is expected to have over 50,000 nodes, which is significantly larger than Summit [1]. This 73,926-node Slim Fly model is the smallest configuration that can obtain at least 50,000 nodes without exceeding the $p = \lfloor \frac{k'}{2} \rfloor$ restriction for obtaining optimal system throughput. It requires 2,738 routers, each with a radix of $k = 82$.

Million-Node Slim Fly Model: Further showcasing the scalability of the Slim Fly model, we scale the topology over 1 million nodes. To the best of our knowledge, this is the largest simulated Slim Fly network model. The feasibility of such a large Slim Fly topology must take into account the requirement of a router/switch containing at least 264 ports. Also, utilizing only 19 node connections per router leaves a significant amount of bandwidth on the network side of the router and provides the ability to scale the system up to 6.4 million nodes with up to p=122 nodes per router. This number of nodes reaches the desired $p = \lfloor \frac{k'}{2} \rfloor$ where we still achieve full link bandwidth throughout the system [31]. Unfortunately, this also raises the necessary router radix k' to 367.

4.4.2.2 Fat-Tree Model Configuration

Two dual-rail pruned Fat-Tree configurations are generated with similar node counts for testing. We generated a 3,564 compute node system to approximate the system size of the Summit supercomputer and used for multi-rail performance studies. We also generated a slightly smaller 3,240 compute node system for a closer comparison with the other network topologies. The two configurations are similar except one additional pod of switches and nodes added to generate the 3,564 node system. A third single-rail version of the 3,240 Fat-Tree configuration is generated for comparison as well. Each Fat-Tree setup uses 36-port switches with a 90 ns switch traversal delay and one virtual channel.

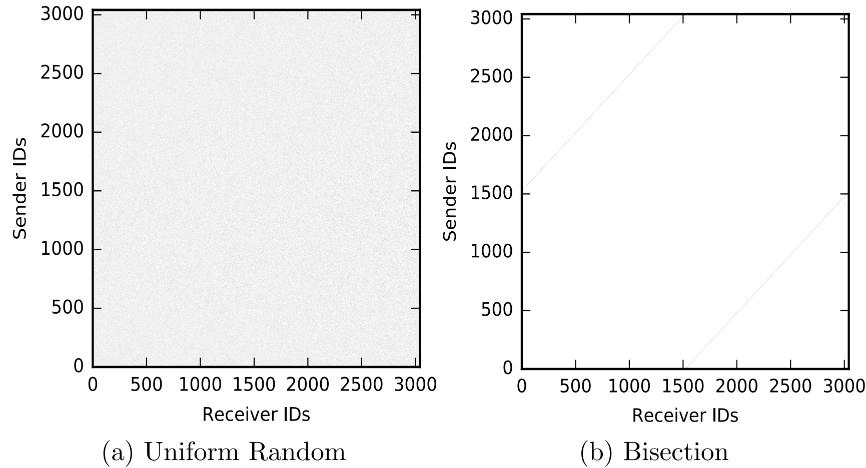
4.4.2.3 Dragonfly Model Configuration

We generate one network configuration for both the 1D and 2D Dragonfly models. Again, for fair comparison with Slim Fly and Fat-Tree, the priority is focused on generating systems with just over 3,000 nodes. Without placing any other restraints on each Dragonfly model, we end up with a Dragonfly-1D configuration with 3,200 compute nodes and a Dragonfly-2D configuration with 3,072 compute nodes. The Dragonfly-2D configuration has a smaller number of groups than Dragonfly-1D and Slim Fly but many more routers per group resulting in roughly double the number of routers than Dragonfly-1D and Slim Fly.

4.4.2.4 Synthetic Workloads

The analysis of network performance benefits greatly from the existence of workloads that represent real system use cases. These representative workloads can be broken down into two categories, namely synthetic workloads and trace workloads. Synthetic workloads are manually constructed by the user to replicate specific activity observed from applications running on the system. In this section we describe the communication patterns of a select number of synthetic workloads used to quantify performance of the network topologies.

The rate at which each synthetic workload process generates and injects messages into the network is governed by an input *load* variable. The *load* variable is in the form of a percentage ranging



(a) Uniform Random (b) Bisection

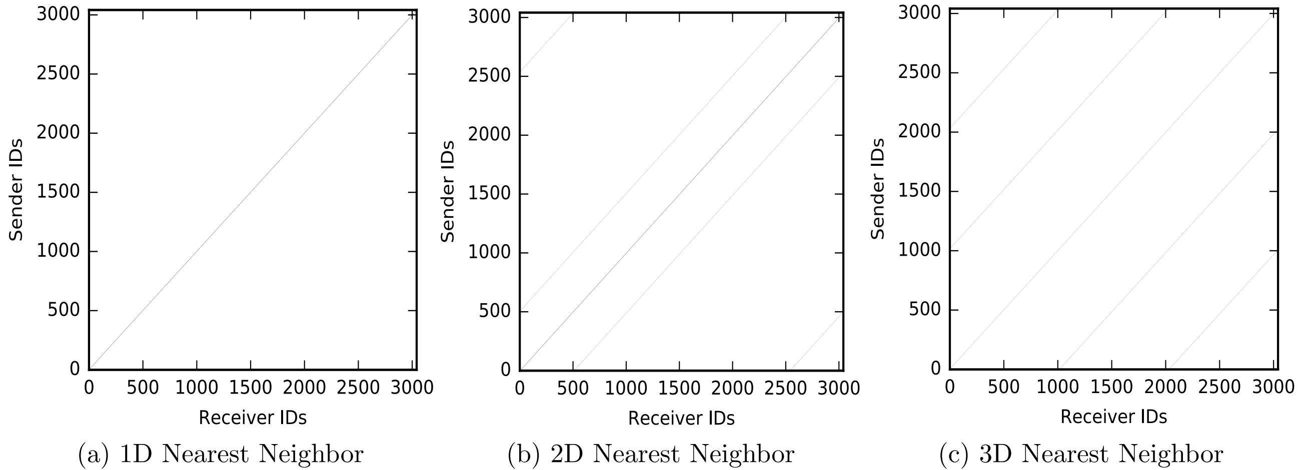
Figure 46: Communication pairing diagrams showing the level of communication between all MPI processes in uniform random, and bisection workloads. Sending process IDs are on the left axis and receiving process IDs are on the bottom axis. *Note, the bisection heapmap lines are faint due to resolution limitations of the image coupled with the very high number of MPI ranks used in this graphic.*

from $0 - 100$, and dictates the desired percentage of link speed to inject the packets into the network. Knowing *load*, each simulated synthetic workload process repeatedly generates synthetic messages of size *msg_size* to be injected into the network at time *send_time*. The *send_time* is calculated following equation 21. The result of the quotient is the exact time in nano seconds that a message of size *msg_size* needs to be sent to achieve a constant link utilization of $load * link_bandwidth$. The $Rand_Exp(x)$ function provides a slight deviation from the exact timing by returning an exponentially distributed random number with mean x.

$$send_time = last_send_time + Rand_Exp(\frac{msg_size}{load * link_bandwidth}) \tag{21}$$

Uniform Random: Uniform random is a workload with communication load that is well balanced across the entire network typically resulting in high bandwidth utilization. Each destination is generated from a uniform distribution resulting in a random destination MPI process ID between 0 and $P - 1$ where P is the total number of MPI processes in the simulation. The generated random destination must be different from the source, otherwise a new destination is sampled until satisfied. A communication pairing heatmap for an example uniform random simulation on Slim Fly with each process sending a total of 100 messages is presented in Figure 46a. The figure shows what looks like white noise as each sending MPI rank sends 100 messages to random destinations. Each MPI process injects one message at a time into the network with a delay that follows the user provided injection *load* for the user selected message size.

Bisection: In a bisection workload, all compute nodes are divided into two equal groups and paired to communicate with one and only one compute node in the opposite group. In the common minimal bisection, the bisecting of compute nodes into two equal partitions is done such that the number of links between each partition is the minimum [70]. It's arguable as to how representative a true minimum bisection workload for the Slim Fly and Dragonfly networks would be of a real



(a) 1D Nearest Neighbor (b) 2D Nearest Neighbor (c) 3D Nearest Neighbor

Figure 47: Communication pairing diagrams showing the level of communication between all MPI processes in one, two, and three dimensional nearest neighbor workloads. Sending process IDs are on the left axis and receiving process IDs are on the bottom axis. *Note, the heatmap lines are faint due to resolution limitations of the image coupled with the very high number of MPI ranks used in this graphic.*

application workload. To maintain the same logical communication mapping across each topology we adopt a simple "middle cut" bisection which generates two equal groups of compute nodes by splitting compute nodes in half using their node IDs. Compute nodes are then paired to communicate using an offset of $N/2$ following Equation 22 where N is the total number of compute nodes in the network and % is the modulo operator. An example communication heatmap for the 3K node Slim Fly topology is shown in Figure 46b confirming the $N/2$ mapping offset. MPI processes are mapped one-to-one to compute nodes and transfer one message at a time according to the input *load* following the bisection node pairing.

$$destination_id = (source_id + \frac{N}{2})\%N \tag{22}$$

Nearest Neighbor: Nearest neighbor traffic replicates the communication patterns typically observed in grid/mesh based applications. The approach for solving equations on a grid in parallel across many processes usually follows a domain decomposition where each MPI process gets a piece of the overall domain and is required to transfer the boundaries of that simulated domain with nearest neighbors. In this work we consider the case where the simulated grid has one, two, and three dimensions that need to be exchanged between processes with respective grid sizes shown in Equations 23, 24, and 25. Example communication heatmaps for 1D, 2D, and 3D workloads are shown for the 3K Slim Fly in Figure 47 showing the mapping offset of the dimensions in the simulated domain to the compute node IDs. At each sending iteration, the workloads send two messages per dimension for the neighbors in the positive and negative direction of each dimension. Again, the input *load* value controls the time between each dimension exchange in order to maintain the correct % utilization of injection bandwidth.

$$\text{1D Grid: } N \times 1 \tag{23}$$

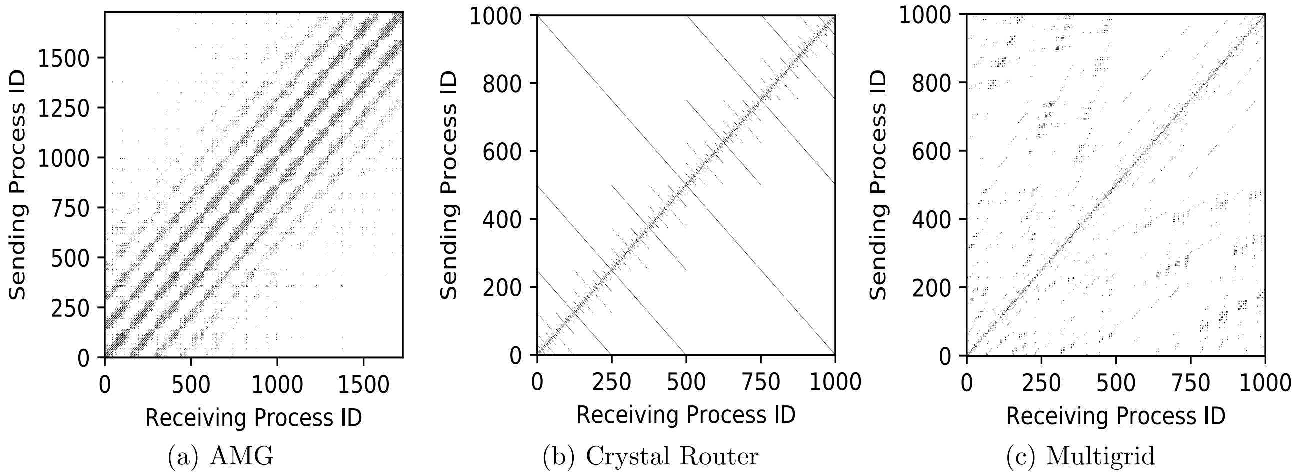


Figure 48: Communication pairing diagrams showing the level of communication between all CPU MPI processes in each CPU application. Sending process IDs are on the left axis and receiving process IDs are on the bottom axis.

Table 12: CPU workload comparison.

Workload	***Processes***	***End Time***	***Msgs***	***Msg Size***	***Waits***
AMG	1,728	0.50ms	2.2M	0.79KB	101.1K
CR	1,000	258.48ms	39.9M	7.95KB	39.9M
MG	1,000	5.51ms	0.5M	9.30KB	0

End time is the virtual time to replay the workload through the Fat-Tree configuration. Msg size is the average size of all messages transfered across all processes.

$$\text{2D Grid: } (N/6) \times 6 \tag{24}$$

$$\text{3D Grid: } (N/9) \times 3 \times 3 \tag{25}$$

4.4.2.5 Application Trace Workloads

Trace workloads, on the other hand, are generated from an actual execution of an application on a real hardware system and are therefore able to replicate the activity of that application in a simulated environment. In this section, we describe select trace application workloads extracted from both neuromorphic and CPU architectures used in the next chapter to benchmark the performance of each network topology.

CPU Three trace workloads are used from common HPC scale CPU applications. All three applications are part of the DOE Design Forward initiative [55]. Conforming to the DUMPI trace format [142], each workload is read in from file by the CODES framework and replayed over the configured HPC network system allowing network performance analysis studies in response to realistic traffic. The three CPU workloads considered are Algebraic Multigrid, Crystal Router, and Multigrid each of which emphasize a different load on the HPC network.

DUMPI: DUMPI traces provide detailed information on the type of MPI point-to-point and collective operations executed by the application. CODES' MPI simulation layer ingests these

operations and replays them on the network models. This layer acts as a bridge between the network workload and interconnect model, and is responsible for maintaining the correct causality order between messages/events of the trace [127]. The DUMPI traces in this paper are run with compute times disabled. In this case, the collected compute times within each trace are ignored and messages are sent without observing any corresponding compute delay.

The communication traces for this work are selected from the Design Forward program because they represent a variety of communication patterns and intensities at varying application scales [55]. These application traces are important tools for next generation interconnection network development because they provide examples of the network workload requirements for the Department of Energy. In the following, we provide details of these applications and communication traces, which provide the workload for the evaluations of the various HPC network configurations.

Design Forward Trace – Algebraic Multigrid: *AMG application trace:* AMG is a parallel Algebraic Multigrid solver mini-application derived from the BoomerAMG solver [84]. The AMG solver approach is used for linear systems on unstructured grids and was developed at LLNL [47]. AMG decomposes the data grid into 3D chunks, resulting in a 3D nearest neighbor communication pattern among processes as shown in Figure 48a. Our application trace of AMG, collected for 1,728 MPI processes, represents the communication in a single V-cycle of the multigrid sequence. MPI operations account for 52.9% of its run time.

Design Forward Trace – Crystal Router: *Crystal Router application trace:* Crystal Router represents the many-to-many communication pattern of the highly scalable Nek5000 spectral element code developed at ANL [150]. The MPI ranks in Crystal Router perform large data transfers in the form of a n-dimensional hypercube. Crystal Router is a very structured and synchronization heavy workload. In this work we use the trace for 1 000 MPI ranks which shows an overall communicating time of 68.5% of the application's runtime. MPI processes each communicate with 10 other processes, sending and receiving a total of 4000 messages at roughly 5 KB each with synchronization after each message transfer.

Design Forward Trace – Multigrid: *Multigrid application trace:* The Geometric Multigrid production application implements a single production cycle of the elliptic solver used in BoxLib [54], an adaptive mesh refinement code typically for structured grids. Multigrid is similar to the AMG mini-app but each use a different solver algorithm. In this case, processes communicate along the diagonals, which results in many-to-many communication. The two Multigrid communication traces used in the paper have been executed on 10,648 and 110,592 MPI ranks with roughly 5% and 4% of runtime spent in MPI communication respectively.

For Multigrid, extrapolation of statistics from application analysis available online for the 1K MPI rank case [54] shows each MPI process in Multigrid communicates with up to 2.2% of the total MPI processes in the simulation. Each one of these communication pairings sends and receives a total of 104 messages varying in size from 8 B to 9.4 KB. One major difference between Multigrid and Crystal router is that Multigrid has much less synchronization sending large amounts of data in many bursty phases.

In summary, Table 12 shows Crystal Router is a highly synchronized application that consistently injects data over a longer period of time. The one-to-one ratio of msg transfers to waits makes Crystal Router a latency sensitive workload. Multigrid, in comparison, can be classified as a bandwidth sensitive workload because it has no synchronization between MPI processes and sends a smaller number of larger messages. Finally, AMG is the most network resource heavy application injecting relatively small messages in a short amount of time with a decent amount of synchronization.

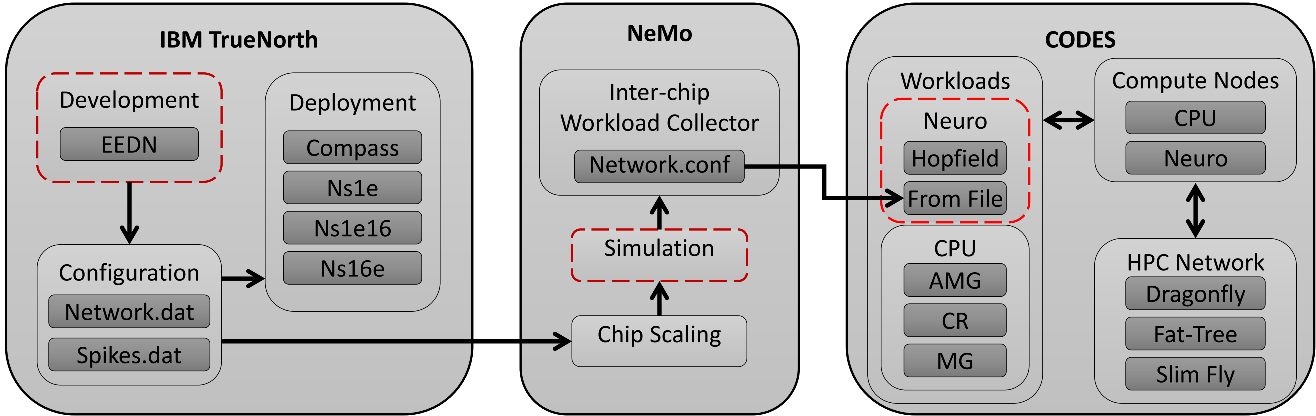


Figure 49: Integration of IBM TrueNorth, NeMo, and CODES ecosystems to form the large-scale neuromorphic systems simulation workflow. The red dashed boxes indicate the various entry points for users to setup a large-scale neuromorphic simulation.

Neuromorphic Workflow and Trace Generation: In this section, we describe the simulation framework being used for exploring the design space of HPC-neuromorphic systems. We present a complete workflow capable of taking predefined spiking neural networks, scaling them up beyond a single chip and simulating them on a large-scale neuromorphic system to study application and system performance. A high level overview of the workflow is shown in Figure 49. The workflow is composed of three loosely integrated components including the IBM TrueNorth, NeMo and CODES frameworks. The exploration process requires an ensemble of simulation runs to analyze the permutations of different parameters where both HPC and neuromorphic architectures are modeled at a detailed fidelity.

TrueNorth: The first component in the workflow is the IBM TrueNorth Neurosynaptic System [145]. TrueNorth is a currently available neuromorphic processor capable of deploying neural network applications with up to 1M neurons. Using the TrueNorth ecosystem we can generate realistic neuromorphic application workload traces.

NeMo: The second component to the workflow is NeMo, a general purpose neuromorphic processor simulator. NeMo [139] is a simulation model leveraging the high-performance and massively parallel Rensselaer Optimistic Simulation System (ROSS) discrete-event simulation framework [36] and validated against the IBM TrueNorth Simulator. In addition to simulating the TrueNorth style neuromorphic processor, NeMo provides the ability to read in and simulate applications developed within the IBM TrueNorth ecosystem. From there, both custom NeMo developed neuromorphic applications as well as TrueNorth applications can be scaled to run on thousands of chips utilizing NeMo's chip-scaling capability.

NeMo is able to generate partially synthetic multi-chip workloads for the purposes of benchmarking novel processor designs. These multi-chip benchmarks are created through interpolation of existing models. NeMo extracts the communication between cores to generate an approximation of a multi-chip neural network. If the number of desired virtual chips is greater than or equal to the number of physical neurosynaptic cores, NeMo simply splits the core-to-core communication across that many chips. For smaller numbers of simulated chips, NeMo will generate chip-to-chip communication based on the underlying core communications. For benchmarking network traffic, this is a reasonable approach, as the traffic of trained classification spiking neural networks tends to

be extremely structured, as observed in other distributed spiking neural network research [38], [39]. This is in contrast to what is observed when simulating biological simulations of spiking networks, where traffic is generally homogenous and complex [157]. If the underlying model is constructed in NeMo, rather than imported from a trained TrueNorth network, NeMo's multi-chip simulation will produce complete chip-to-chip traffic using the underlying synthetic neuromorphic model.

A limitation of *NeMo* is that it lacks the ability to simulate inter-chip communication through an external network such is the case in large-scale HPC systems. When inter-chip spikes are detected, NeMo routes them instantaneously to their destination. We address this limitation by using the HPC network models provided by CODES, as explained in the next section. To allow for external processing of inter-chip communication, NeMo collects the chip-to-chip connections as well as the time dependent spiking information, computes the average number of spike transfers per connection per tick, and saves it all to file. The result is a neuromorphic many-chip communication workload representing the average number of spikes transferred between all chip connections during one tick of execution. This serves as an input to the CODES network simulation framework, which is described next.

CODES: The final component in the workflow is the CODES framework for HPC network simulations. Two new options for running neuromorphic workloads have been added to the CODES HPC network simulation toolkit. The first feature allows CODES to read in inter-chip communication data files to accurately replay real neuromorphic workloads in an HPC environment. In this approach, CODES leverages a NeMo generated input file to establish the network of neuromorphic chips, their connections, and their number of spikes transferred to each connected chip over the length of the simulation. When inter-chip spikes are observed, they are sent in an `MPI_Isend` and routed through the network to their destination chip. Alternatively, a second feature allows the modeler to run a fully synthetic neuromorphic workload based on a hopfield style neural network [85]. Following this type of neural network, in which all neurons are fully connected, results in all-to-all communication between chips. During each 1ms tick, each chip sends a user-configurable number of spikes to all chip destinations using the `MPI_Isend` operation.

Application Development: Currently, modelers have three different points of entry into the neuromorphic HPC workflow, each providing a different level of detail/accuracy. First, neural network applications can be generated using the IBM TrueNorth development platform. This level provides an easy adoption by TrueNorth developers. Entering at this level benefits from the data being generated from a real application source. However, this approach requires knowledge of and access to the TrueNorth specific hardware and software systems.

Modelers without any knowledge of TrueNorth can enter the workflow by generating a neural network workload within NeMo. Developing an application within NeMo allows for custom neuromorphic hardware constraints but requires learning the NeMo development API. Finally, a neuromorphic workload can be constructed at a much higher level of abstraction using the CODES framework. CODES specifically deals with only the inter-chip communication. Therefore, if the modeler understands their neural network application in terms of number of connections per chip, and number of spikes per tick transferred over those connections, they can implement a strong application workload approximation at scale directly. Additionally, if the connections and spiking data is available, modelers can convert the data to the CODES neuromorphic input file format to be read and replayed over the network.

When mapping the virtual neuromorphic cores to simulated chips we use a simple contiguous placement approach. Sawada et. al. [145] describe the IBM heuristic placement algorithm which

Approved for Public Release; Distribution Unlimited.

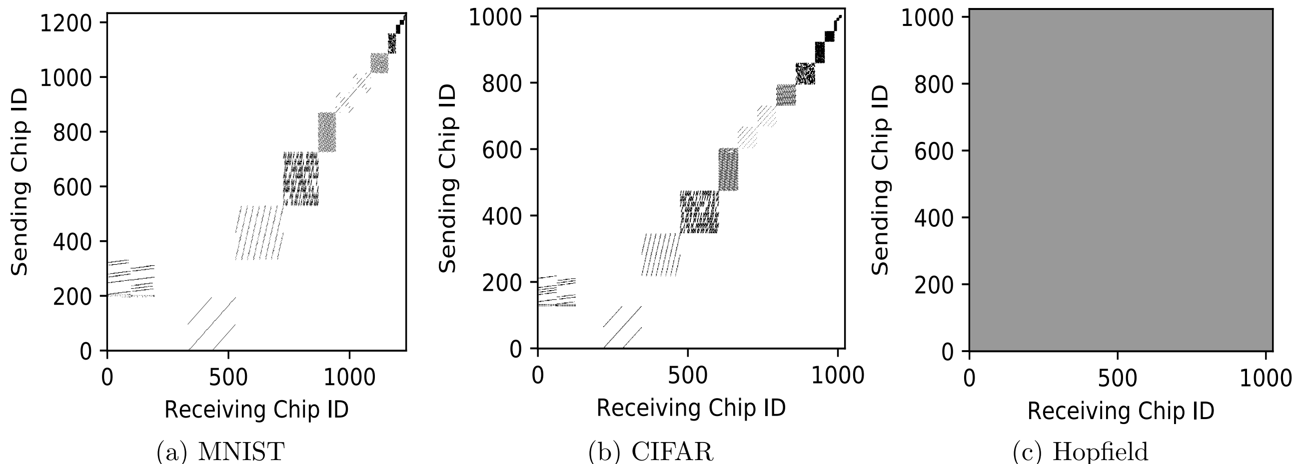


Figure 50: Communication pairing diagrams showing the level of communication between all neuromorphic chips in each neural network application. Sending chip IDs are on the left axis and receiving chip IDs are on the bottom axis.

Table 13: Neuromorphic workload comparison.

Workload	***Chips***	***Connectivity***	***Spikes/Tick***	***MB/Tick***	***Waits***
MNIST	1234	15 layers	577K	4.4MB	0
CIFAR	1024	15 layers	647K	4.9MB	0
Hopfield	1024	all-to-all	10.2M	80MB	0

A tick is 1ms of simulated time.

optimizes the placement of virtual neurons to the TrueNorth hardware in order to minimize message travel distance. The algorithm is optimized for placing virtual cores on hardware that uses a 2D connectivity of chips. However, we aren't using a 2D connectivity of chips. We are using, dragonfly, Fat-Tree and Slim Fly connectivity of chips and opt for a simple linear scaling and placement approach which is non-optimized but provides a consistent placement for many chips connected via Dragonfly, Fat-Tree, and Slim Fly topologies. Therefore, our results can be seen as a non-optimized or possibly even worst-case outcome. We created two trained neural network trace workloads and one synthetic workload to test the simulated distributed neuromorphic hardware systems. Using the IBM TrueNorth EEDN framework [61], we trained two convolutional neural networks based on common neural network datasets, MNIST and CIFAR. These applications were then imported into NeMo, scaled to thousands of chips, and simulated to obtain theoretical large-scale communication workloads. Starting from real TrueNorth neural network applications, we are able to generate approximations of traffic produced by potential large distributed neural networks. The following neuromorphic workloads are studied in this work.

Neuromorphic Workload – MNIST: MNIST is a dataset containing 28x28 pixel images of handwritten digits zero through nine [108]. The MNIST workload used in this study is a convolutional network trained on the MNIST dataset to classify handwritten digits. This network workload is of interest because it leverages the common convolutional neural network structure. MNIST can also be considered a gold standard benchmark for neural networks as a number of different implementations and results have been very well documented. For the implementation, we are using a prebuilt MNIST configuration provided in the set of IBM examples. The network consists of 15

layers of neurons using 2,468 neurosynaptic cores which is roughly 60% of one TrueNorth chip. For the large-scale neuromorphic studies, we have scaled the workload from 1 chip to 1234 chips. Figure 50a presents a heatmap showing the layered connections between chips in the network.

Neuromorphic Workload – CIFAR: CIFAR is a much larger and complex image dataset containing 32x32 pixel images of objects that belong to one of 100 different classes [103]. The classes include various animals, foods, and inanimate objects. The workload used in this study is again, a convolutional neural network implemented in the IBM TrueNorth ecosystem and provided as an IBM generated example. The network consists of 15 layers of neurons using 4,044 neurosynaptic cores, totaling roughly 99% of one TrueNorth chip. The CIFAR workload again, represents the very popular convolutional network approach, but the increased size and complexity of the input data results in a more dense connectivity between layers as compared to MNIST. For the large-scale neuromorphic studies, we have scaled the workload from 1 chip to 1024 chips. Figure 50b shows a heatmap indicating the different layers and connections between chips in the network.

Neuromorphic Workload – HF: The third network workload is a synthetic implementation of a Hopfield network. We have chosen Hopfield because it represents another commonly used neural network approach posing a different communication workload from the TrueNorth implementations of MNIST and CIFAR. Unlike the layered connectivity of the CIFAR and MNIST convolutional networks, the Hopfield network consists of one layer of fully connected neurons [85]. The high degree of connectivity poses it's own concerns for large-scale HPC networking and therefore an interesting case to study. The Hopfield network is also a recurrent network, and unlike the convolutional networks of CIFAR and MNIST, TrueNorth does not currently support recurrent networks. Therefore, we implemented a synthetic representation of the Hopfield network workload at the HPC network level within the CODES framework. We configure the Hopfield workload to utilize all neuromorphic chips in each simulated HPC system configuration. For comparison with the MNIST and CIFAR workloads, a chip connectivity heatmap for the Hopfield workload running on 1024 chips is provided in Figure 50c along with workload details in Table 13. While Hopfield has more data transfered per tick than CIFAR and MNIST workloads, its communication spans the entire network following all-to-all connections instead of tightly grouped layered connections as seen in the convolutional workloads. Note, the data values presented in Table 13 are aggregate totals for the entire workload and not per MPI rank.

4.4.3 HPC Network Model Evaluations

We first start off with a study analyzing the scaling and run time performance of the discrete-event simulation framework used to perform the network evaluations. Achieving high performance from the underlying simulation framework is key to enable the large ensembles of runs necessary for an exhaustive set of results. From there, we focus on the individual performance of the Slim Fly and Fat-Tree topologies. For Slim Fly, we look at the effect of routing algorithms on CPU application traces on 3K node and 74K node systems. For Fat-Tree, we focus on answering questions regarding the benefits of a multi-rail configuration by performing experiments on rail scaling, job placement, and scaling up compute power per node.

Finally, we get into studies focused on comparing the performance of similarly configured Dragonfly, Fat-Tree and Slim Fly networks. First, we perform a load analysis to see how well the networks are able to respond to increasing message injection rates. Then, we focus on workload specific performance as we study run times for synthetic, CPU and neuromorphic application workloads,

as well as interference of CPU and neuromorphic workloads running in parallel. Finally, a cost analysis is computed to provide a picture of performance per dollar for each network topology.

4.4.3.1 Discrete-Event Simulation Analysis

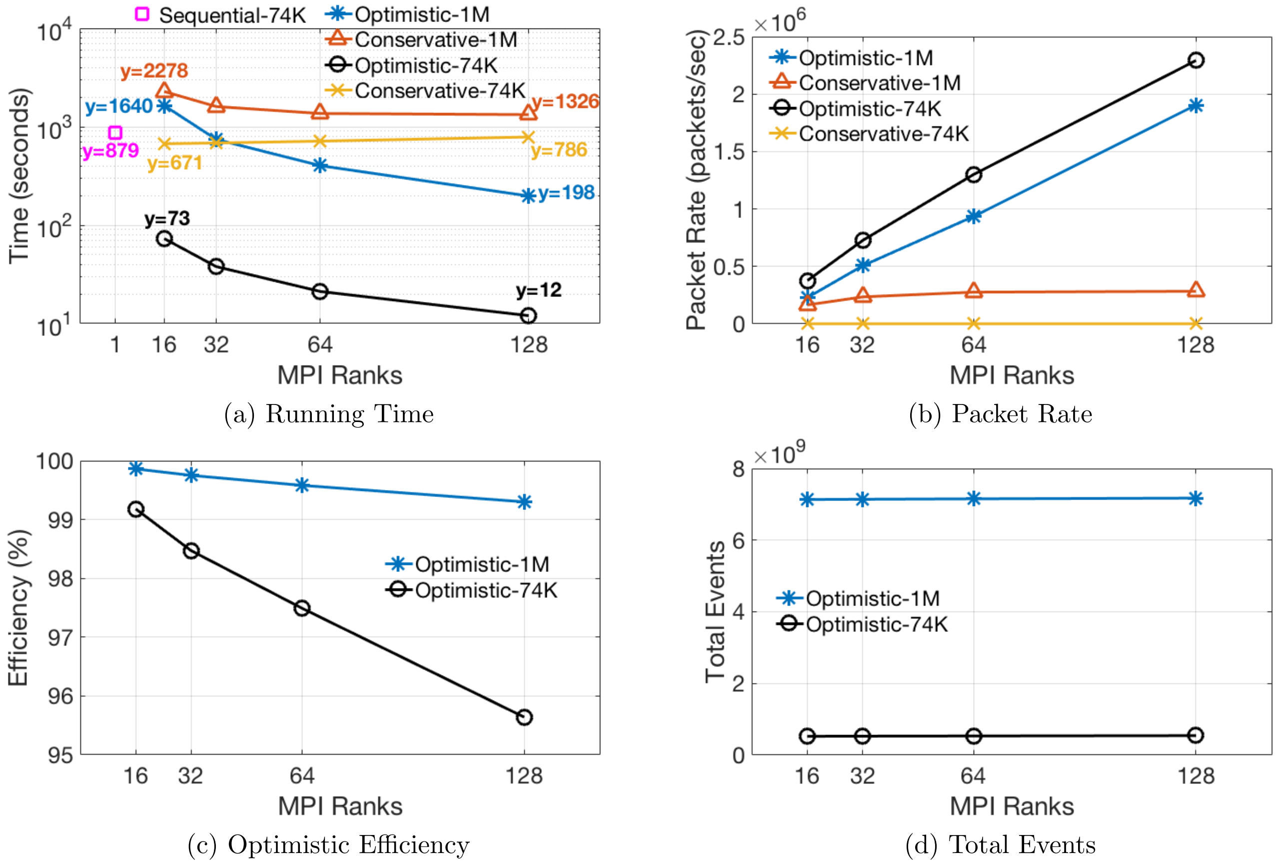
Performing HPC network analysis can quickly result in hundreds or even thousands of executions as we try to investigate many independent variables such as routing algorithms, workloads, job mappings, etc. The ability to execute a fixed problem on increasing numbers of resources provides the opportunity to accelerate discovery. In this section, we study the simulator performance of the Slim Fly network model, in terms of node scaling and clock cycle analysis, to show the efficiency of the model in using available compute resources and speeding up the discovery process.

Scaling Analysis: In this section, we present the strong-scaling performance analysis of the extreme-scale 74K and 1M-node configurations of the Slim Fly network model using an Intel Xeon cluster at Rensselaer Polytechnic Institute (RPI). The system has 34 nodes, each node consisting of two 4-core Intel Xeon E5-2643 3.3 GHz processors and 256 GB of RAM. We scale the simulations from 16 processes to 128. Each execution of the 74K-node Slim Fly model is allocated with 8 MPI ranks per node, while the million-node model is allocated with 4 because it's larger memory footprint requires at least 4 nodes. Following the same simulation parameters as in previously described, we use 100 Gb/s link bandwidth with a latency of 50 ns. Routers utilize virtual channels, a buffer space of 100 KB per port, and a 100ns traversal delay. Messages are broken down into 256 B packets and simulations are executed for 220 μs of virtual time. Finally, all simulations are executed using minimal routing, uniform random traffic, and an injection load of 10%.

Additionally, ROSS uses simulation specific parameters that can be used to tune the simulation performance by controlling the frequency of global virtual time (GVT) calculation [35]. These parameters are the "batch" and "gvt-interval." The batch size is the number of events the ROSS event scheduler will process before checking for the arrival of remote events (events issued from other MPI ranks) and anti-messages (messages indicating an event was issued out of time stamp order and needs to be rolled back in optimistic execution). The GVT interval is the number of times through the main scheduler loop before a GVT computation will be started. The default values "batch=16" and "gvt-interval=16" are used in the optimistic event scheduling simulations and the default lookahead value of 1 is used in the conservative executions.

The scaling performance is evaluated using several measurements that provide insight into how well the Slim Fly model performs as a large-scale parallel discrete-event simulation. The following measurements are collected:

- *Simulation Run Time:* The real-time taken to complete the simulation on the cluster.
- *Packet Rate* The total number of simulated network packets successfully transfered divided by the runtime of the simulation.
- *Event Efficiency:* Measured as a percentage, the efficiency provides insight into how much work is being performed in processing the forward events. Instead of using traditional state saving techniques, ROSS uses reverse event handlers that undo events executed out of order. This technique saves memory by not having to save the state but requires extra compute to unroll the events processed out of order [25]. The efficiency is computed using Equation 26.



(a) Running Time

(b) Packet Rate

(c) Optimistic Efficiency

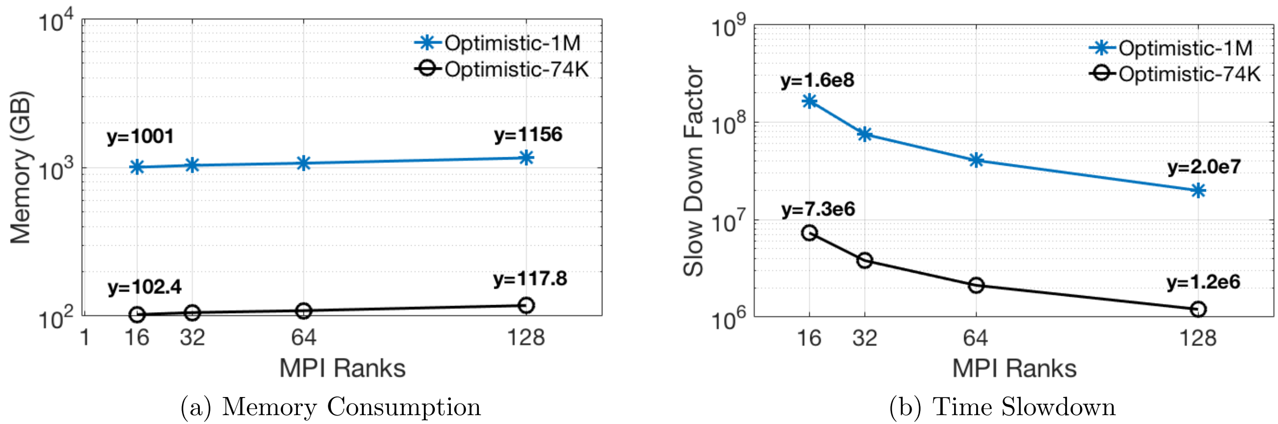
(d) Total Events

Figure 51: Million-node compute scaling analysis simulating 100 μs using minimal Routing, UR traffic, and 10% network load. Figures are best viewed in color. Added result for sequential execution.

- *Total Events:* Total events collects the total number of completed events in both the forward and reverse directions.
- *Memory Consumption:* Memory consumption measures the physical amount of system memory required to initialize the model and run the simulation.
- *Slowdown:* Slowdown is a measurement indicating how much slower the simulation is compared to the real-world experiment being modeled. For example, a simulation taking 1000 seconds to simulate 10 seconds of network traffic has a slowdown of 100.

$$efficiency = 1 - \frac{rolled_back_events}{total_events} \tag{26}$$

As shown in Figure 51, utilizing optimistic event scheduling results in solid compute performance speedups for both the Slim Fly million-node model and the 74K-node model. The largest packet rate is achieved running the 74K-node Slim Fly model on 128 MPI ranks, executing 2.3 million network packets per second and processing 543 million discrete events. Not far behind, the million-node model achieves a rate of 1.9 million simulated packets per second processing 7 billion events.



(a) Memory Consumption

(b) Time Slowdown

Figure 52: Million-node memory and time scaling analysis simulating 100 μs using minimal Routing, UR traffic, and 10% network load. The time scaling shows the slowdown factor of simulation compared to real hardware as the number of MPI ranks increases. Figures are best viewed in color.

Additionally, with an 8x increase in compute performance, the 74K-node and 1M-node models achieve 6.1x and 8.3x improvements respectively in run time. Comparing with sequential execution on only one node, optimistic execution achieves a 12x improvement with 16 nodes. Conservative, on the other hand, achieves only a 1.7x speedup for the 1M-node model and actually gets worse for the 74K-node model at .85x speedup.

The super linear speedup of the optimistic 1M-node model is a result of memory consumption during the 16 rank execution. As can be seen in Figure 52a, at 16 ranks the 1M-node model consumes 1001GB of memory. Taking into account memory used by the operating system and other resident background tasks, the 1024GB of total available memory provided from the four nodes is closely maximized. Increasing the number of nodes in the simulation provides relief from the memory pressure. Specifically, going from 16 ranks on four nodes to 32 ranks on eight nodes, we see a super linear 2.2x improvement in runtime with only a 3% increase in memory footprint for the 1M-node model. Ignoring the speedup due to the memory footprint, we can calculate the speedup using 128 nodes from the 32 rank case to be 3.75x.

A direct cause for the strong compute scaling results for optimistic event scheduling is the uniform random workload. The workload optimally distributes the simulated network packets across the network, resulting in a balanced workload across all MPI ranks. This balanced workload helps reduce the chances of one or more of the MPI ranks getting ahead of others in virtual time, executing events out of order, and requiring costly rollbacks. Furthermore, the mapping of 7 billion events to 128 MPI processes translates to each process staying saturated with events and leads to an event efficiency above 99%. Running the smaller 74K-node Slim Fly configuration on as many processes leads to a drop in event efficiency because of less available work. At 128 MPI processes, the smaller 74K-node Slim Fly model has a 3% lower event efficiency than does the million-node model but manages to execute packets at a 20% faster packet rate. The smaller number of packets per PE in the 74K-node model translates to less overhead reordering discrete events to maintain timestamp order but also translates to less work mapped per PE, and as a result, only achieves a 6.8x increase in performance at 128 MPI processes with an 8x increase in compute.

The last measurement analyzed is the slowdown of the simulation model, shown in Figure 52b.

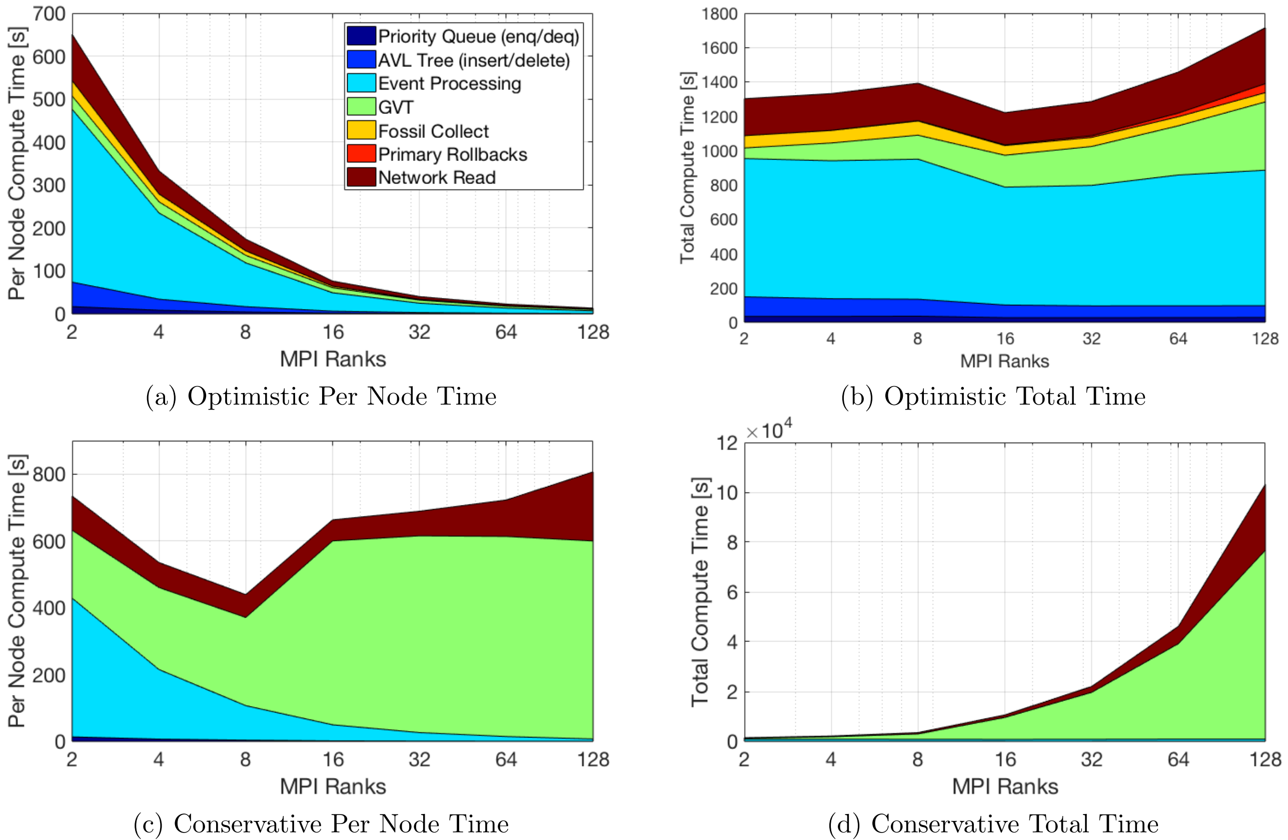
Slowdown indicates how far away the simulation runtime is from executing workloads in real-time. In the context of our scaling tests, real-time is the $100\mu s$ length of the uniform random workload. Starting with 16 process, simulation of the $100\mu s$ workload takes a total of 1,640s for the 1M-node model and 73s for the 74K-node model. Therefore at 16 processes the models are 160 million and 7.3 million times slower than real-time execution. Following the curves in Figure 52b, the best performance for both models is achieved at 128 nodes where the simulations are 20 million and 1.2 million times slower than real-time execution. In order to improve analysis and development of future network systems, it's vital to improve the execution time of these models to closer approach real-time execution and allow for simulation of much longer workloads in a reasonable amount of time.

4.4.3.2 Clock Cycle Analysis

In order to understand the performance of the Slim Fly model within the context of the underlying discrete-event simulation engine, this section sheds light on the tasks during which the model spends the majority of its clock cycles. Figure 53 presents four area plots showing the distribution of time the 74K-node Slim Fly model simulation spends in each phase of the ROSS discrete-event computation when using optimistic or conservative event scheduling protocols. This study utilizes a 220 μs virtual time simulation using minimal routing under uniform random traffic with an increasing number of MPI ranks (PEs). Figures 53a and 53b present data for the model under optimistic event scheduling while Figures 53c and 53d present the conservative event scheduling execution results. Within the figures, the timings for each compute task are stacked on top of one another so the vertical length of each colored section indicates the time spent performing the associated task for the execution with the corresponding x-axis number of ranks. The max height of all stacked color sections represents the sum of all compute tasks for the execution with the given number of ranks. Finally, the optimistic simulations use a batch size of 16 and GVT interval of 16, and the conservative executions use a lookahead of 1.

Focusing first on the time spent per node in the optimistic execution, we see in Figure 53a that the Slim Fly model scales well, roughly cutting its total time spent in the compute tasks in half for each doubling of MPI ranks allocated. Moving on to Figure 53b we see that the total time to completion stays fairly constant with a slow upwards trend in response to increasing MPI rank counts. The distribution of computation by task starts off at 2 ranks spending the majority of the time making forward progress processing events. This trend is consistent regardless of the number of MPI ranks utilized. In addition, the distribution of time spent in each aspect of the simulation stays constant for optimistic scheduling as the number of MPI processes increases. This denotes an ideal distribution of events to LPs, and even further, an ideal distribution of LPs to PEs. This characteristic allows the simulation to scale strongly as there is an equal amount of work for each processor, preventing the case where some processors have less work. Less work causes the PE to advance its local time further than the global virtual time. The result is a much higher chance of processing an event out of order and forcing a large number of primary and secondary rollbacks.

Note, at 16 ranks, there is a unique drop in total compute time. Using 16 ranks appears to be the sweet spot balancing the tradeoffs of the default parameters of GVT-interval=16 and batch=16, which allows each processing element to freely execute events far enough in the future without getting too far ahead of others and computing events out of time-stamp order. Moving above 16 ranks, time spent computing GVT increases as it takes more time to synchronize global time across



(a) Optimistic Per Node Time

(b) Optimistic Total Time

(c) Conservative Per Node Time

(d) Conservative Total Time

Figure 53: Distribution of simulation time for the 74K-node Slim Fly model with minimal routing, UR traffic, and 10% load. Per node time is the average time each node spent in each task while the total time is the average multiplied by the number of MPI ranks in the simulation. Figures are best viewed in color.

the growing number of process. Also, starting at 32 ranks, the time processing rollbacks increases which indicates our load distribution per rank is varying, due to less total work available per node. All in all, the Slim Fly model excels under optimistic event scheduling.

In conservative event scheduling using a lookahead value of 1 and starting with 2 MPI ranks, we see a large portion of the Slim Fly model compute cycles spent in GVT. Unlike optimistic scheduling where each PE can maintain its own local time and process events accordingly, conservative execution forces all PEs to maintain the same virtual time, essentially executing in a semi-lockstep manner. This event scheduling guarantees that no messages are processed out of order, but it requires more interaction from GVT, and as shown in Figure 53c results in processors being blocked from productive event processing. Moving from 2 to 8 MPI ranks, the execution time decreases because there is enough work per processor to keep busy between each execution of GVT but at the same time performing GVT computation slightly rises. At 16 MPI ranks, the amount of work available for the number of processes decreases to the point that GVT must intervene more often to keep the processes in order, so we experience a large increase in GVT cycles. As the number of MPI processes increases, so does the number of number of PEs the event scheduler must keep locked at the same virtual time. This situation inevitably leads to PEs sitting idle waiting for GVT

to advance the time window.

4.4.3.3 Slim Fly CPU Trace Evaluation

In this study, we investigate the performance of the Slim Fly network in response to real application workloads and the effects of routing algorithms. Both Crystal Router and Multigrid application workloads are executed on the 3 K and 74 K node Slim Fly systems while performance metrics are collected to compare the performance of minimal, non-minimal and adaptive routing algorithms. All network parameters such as link speed and router buffer sizes are the same as in previous experiments except now a packet size of 4 KB is used. The specific application workloads used in this study are a 1K MPI rank Crystal Router trace as well as 10K and 110K rank Multigrid traces as previously described.

The following metrics are studied in this section:

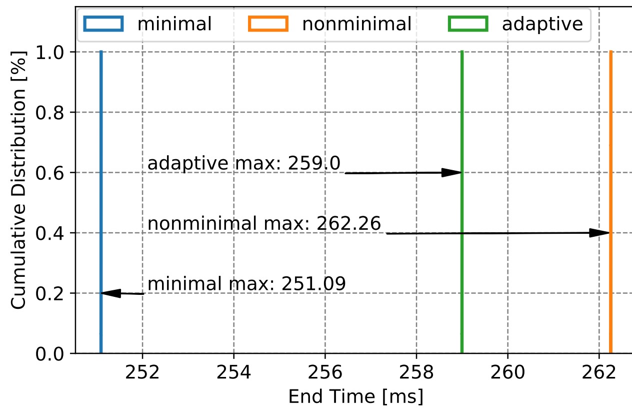
- *Observed Bandwidth:* Observed bandwidth measures the rate at which each node is able to inject data into the network. Each compute node calculates the observed bandwidth by dividing the total amount of bytes transferred by the total time spent transferring the data.
- *Average Congestion:* The average terminal congestion delay quantifies the amount of time (in microseconds) a compute node spends with its NIC's buffers completely filled. Lower is better, indicating less injection congestion.
- *Average Packet Latency:* Average latency describes the average source to destination delay of all packets injected per compute node.
- *Average Hops:* Average hops collects the average number of links traversed within the network by each packet per compute node. It does not include the compute node injection link.

CDF plots are chosen to visualize the results data because they provide robust analysis for large data sets that have a high variance in data values. Within the CDF figures, each plotted line corresponds to the values for all compute nodes in that given simulation. Each (x; y) coordinate along a plotted line indicates the percentage y of compute nodes that took on a value equal to or less than x. Portions of a line that have a very small slope (horizontal line) indicate few compute nodes, if any, took on values within the given range of x values. On the other hand, a very high slope (vertical line) indicates many compute nodes took on similar x values.

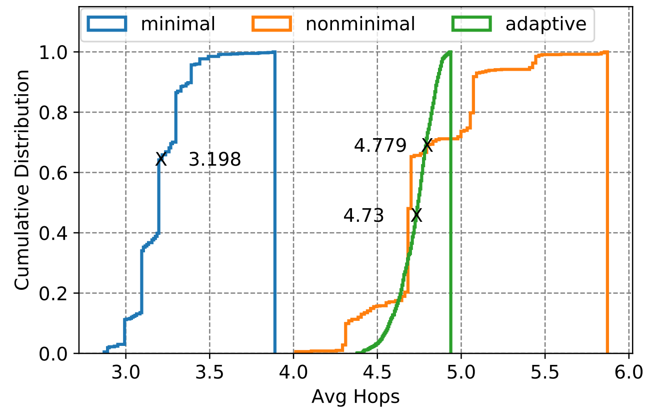
3K-Node Slim Fly: Figure 54 presents the results for the 1 K MPI rank Crystal Router application running on the 3 K Slim Fly network under minimal, non-minimal and adaptive routing algorithms. The simulated 1 K MPI ranks are contiguously mapped to compute nodes at a ratio of 1:1, leaving 2,042 nodes in the system unallocated and sitting idle. Focusing first on Figure 54a, we can see that the 1K Crystal Router trace performs best under minimal routing where it finishes 7.96 ms faster than adaptive for a 3.1% improvement, and 11.2 ms faster than non-minimal routing for a 4.3% improvement. Moving on to average number of hops per packet in Figure 54b, we see minimal routing has a mean average hop count improvement of roughly 47% over non-minimal and adaptive routing protocols.

Due to the large amount of synchronization and the few number of messages transfered between each synchronization barrier, the amount of congestion in the network for the Crystal Router application is low regardless of routing algorithm. As shown in Figure 54d, minimal routing, which

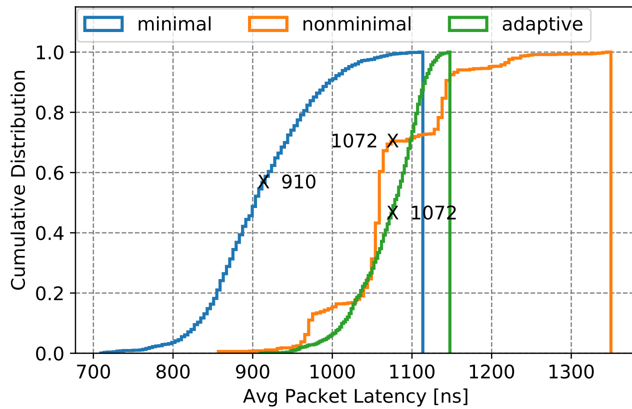
Approved for Public Release; Distribution Unlimited.



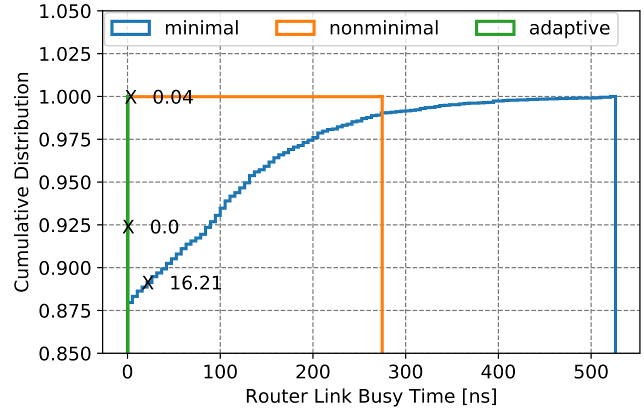
(a) Simulation End Time



(b) Packet Hops



(c) Packet Latency



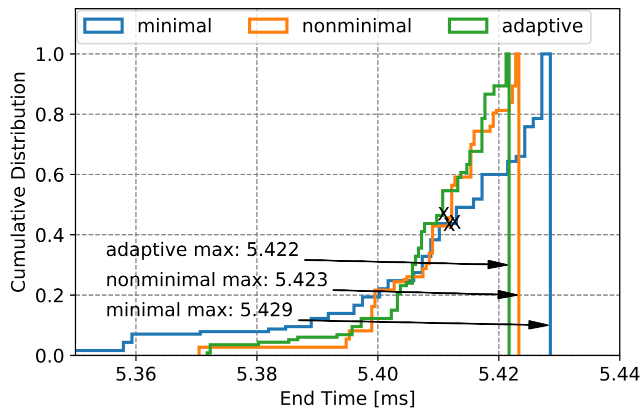
(d) Network Congestion

Figure 54: Performance results for the Crystal Router application running with 1K MPI ranks on the 3,042-node Slim Fly network. Black 'X's indicate the mean value across all compute nodes. Figures are best viewed in color.

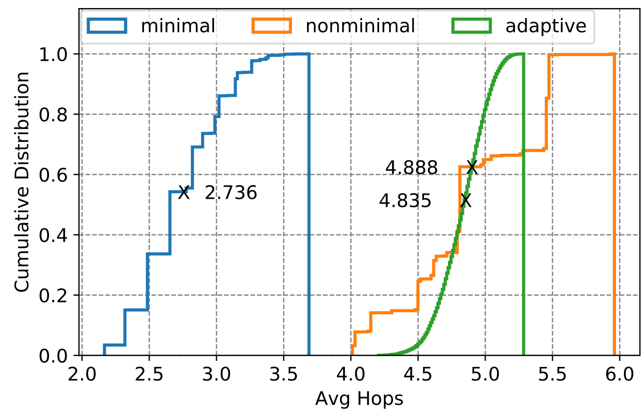
has the highest potential for network congestion, observes an average of only 16.21ns of busy time per link over the length of the simulation and at most 500ns for a few links. The packet latency correlates more closely to the number of hops a packet takes en route to it's destination. In the case of adaptive routing, it continually sees traffic on the minimal path option (as the Crystal Router workload only communicates with 10 other process), and as a result frequently selects one of the four random non-minimal paths. For Crystal Router, taking the shorter number of hops is worth the minimal extra delay due to network congestion.

In summary, Slim Fly's routing algorithm can be set to minimal as the more complex routing algorithms non-minimal and adaptive do not appear to offer significant performance benefits for the highly synchronized Crystal Router application workload. Adaptive routing is able to mitigate network congestion observed by minimal and nonminimal routing but at the cost of an increased number of average hops per packet. In the case of the Crystal Router application, the path length is most critical to network performance and therefore minimal routing performs best with a 3.1% improvement in simulation end time with adaptive routing being second best.

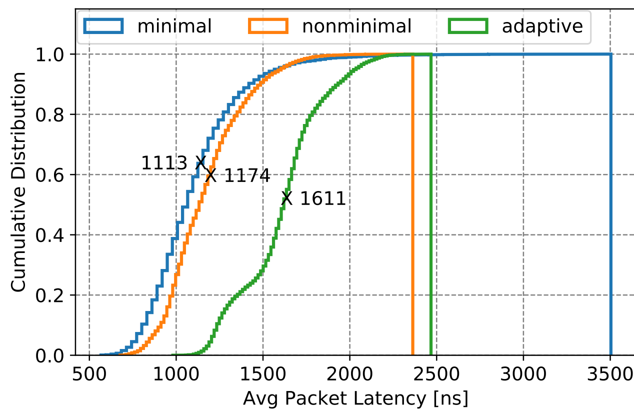
Approved for Public Release; Distribution Unlimited.



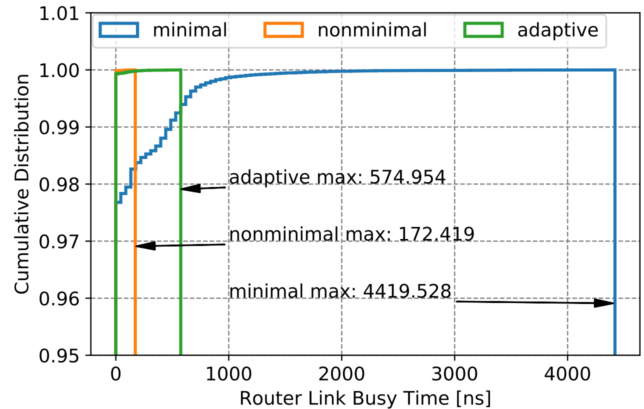
(a) Simulation End Time



(b) Packet Hops



(c) Packet Latency



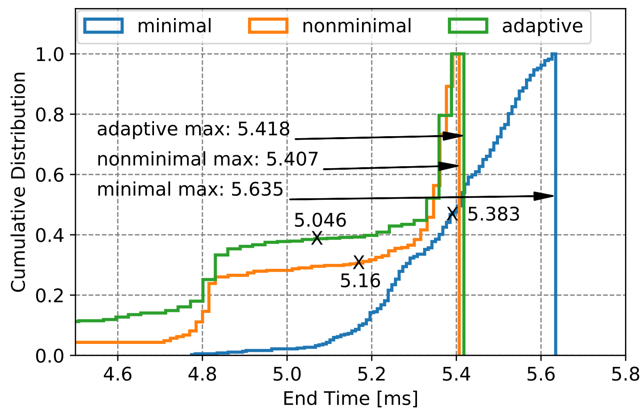
(d) Network Congestion

Figure 55: Performance results for the Multigrid application running with 10K MPI ranks on the 74K-node Slim Fly network. Black 'X's indicate the mean value across all compute nodes. Figures are best viewed in color.

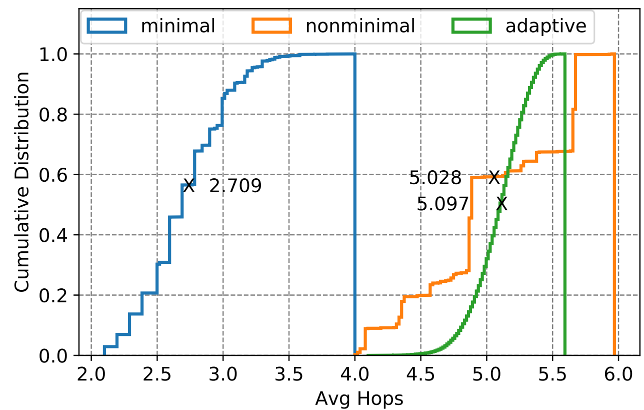
74K-Node Slim Fly: We now present network performance under real application workloads for the 74 K node Slim Fly model. In this study we use the 10 K and 110 K MPI rank Multigrid applications as they each contain enough MPI processes to utilize a significant fraction of the 74 K node network. Again, we test network performance using minimal, non-minimal and adaptive routing approaches. All of the 10 K MPI ranks are contiguously mapped to compute nodes at a ratio of 1:1, while the 110 K MPI ranks are mapped at 2:1 because the number of ranks exceeds the number of compute nodes in the system. These configurations respectively leave 86% and 26% of the nodes in the system idle.

Starting with the smaller 10 K MPI rank Multigrid application workload, we observe similar overall performance from all three routing algorithms. As shown in Figure 55a, the median, mean, and max end times for all compute nodes are very close between minimal, non-minimal, and adaptive routing. In fact, the most variance is in the max end time where adaptive routing provides only 0.1% and 0.02% speedups over minimal and non-minimal respectively.

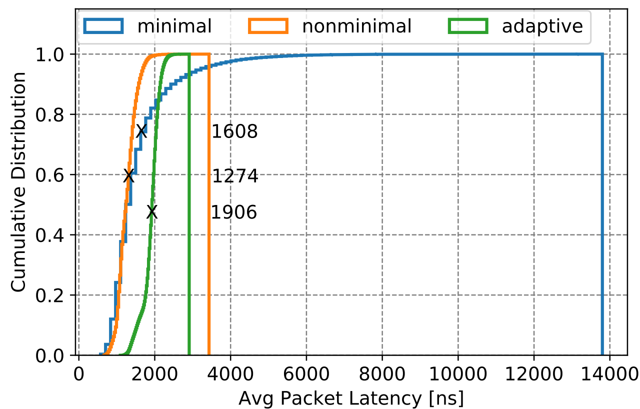
Diving deeper into minimal routing, the remaining subfigures in Figure 55 show that minimal



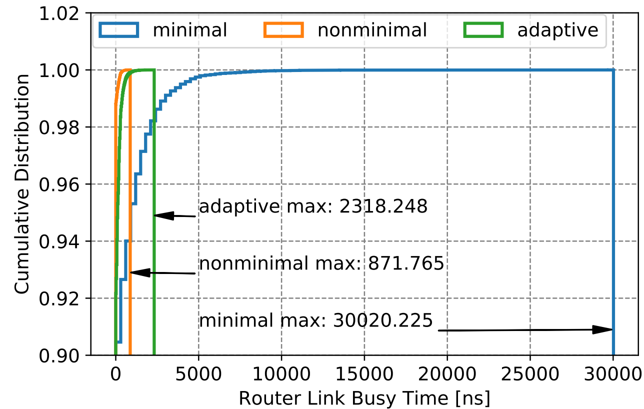
(a) Simulation End Time



(b) Packet Hops



(c) Packet Latency



(d) Network Congestion

Figure 56: Performance results for the Multigrid application running with 110K MPI ranks on the 74K-node Slim Fly network. Figures are best viewed in color.

routing is able to significantly lower the average packet hop count per compute node, and achieve a lower average packet latency than non-minimal and adaptive routing. However minimal routing suffers from small communication hotspots in the network, which prevents some compute nodes from finishing their message transfers in a timely manner. As seen in Figure 55d, roughly 98% of the links in the network have zero congestion while roughly 0.01% of links observe close to 4000ns of busy time. Compute nodes using these congested links observe large packet latencies that increase the simulation end time.

In contrast, non-minimal and adaptive routings mitigate network congestion with their ability to distribute packets throughout the network via random intermediate routers. Of course, the improved path diversity comes at the cost of extra hops, resulting in additional transmission and router traversal delays. The key result which gives non-minimal and adaptive routing the benefit over minimal is the much smaller variance in average packet latencies. Network hotspots associated with minimal routing significantly reduce performance for a few compute nodes while increased hop counts for non-minimal and adaptive provide a smaller consistent delay to all compute nodes.

Finally, we have the performance results for the largest application workload, 110 K MPI rank

Approved for Public Release; Distribution Unlimited.

Multigrid, shown in Figure 56. In this case, the performance trends are similar to the 10 K rank Multigrid application with a few differences relating to the increased number of compute nodes utilized in the system by 110 K Multigrid. Overall adaptive and non-minimal routings performs better than minimal routing. Non-minimal has a 0.2% increase over adaptive with respect to the max compute node end time, while adaptive has a 2.3% increase over non-minimal with respect to the mean compute node end time. Minimal performs 4.2% and 6.7% worse than the non-minimal routing algorithm in terms of max and average end time respectively.

The main deployment difference between the 10 K and the 110 K MPI rank Multigrid applications is the number of compute nodes utilized in the system. Mapping to more compute nodes increases the number of links injecting packets into the network. Starting with the number of hops in Figure 56b, little change is observed between the 10 K and the 110 K applications with respect to minimal and non-minimal routing. However, the curve for adaptive routing shifts to the right, indicating adaptive is selecting more non-minimal paths for the 110 K trace as the increase in compute node utilization translates to increased network congestion. In response to the increased congestion, adaptive routing issues more non-minimal routes to help distribute the network load and minimize congestion. The same trend is observed in Figures 55c and 55d as the maximum observed values for packet latency and network congestion grow substantially for the 110 K workload under minimal routing as compared to non-minimal and adaptive. The max value for avg packet latency grows by a factor of 4x for minimal while non-minimal and adaptive increase by only 1.4x and 1.2x, showing the latter's capability to redistribute traffic in response to increased levels of network congestion.

In summary, all three routing algorithms make different trade-offs in hop count and path diversity that result in similar network performance for the Multigrid workload. Minimal routing maintains a low hop count per packet but creates hotspots in the network delaying communication for compute nodes. Non-minimal routing eliminates network hotspots at the cost of increased number of packet hops. Finally, adaptive routing balances between the two to arrive at a marginally optimal middle ground.

The results from both Crystal Router and Multigrid application studies indicate routing protocols do not significantly influence network performance for the Slim Fly topology in the case of practical workloads. In these communication workload examples, the Slim Fly routing algorithm can be set to the simple minimal routing to observe similar performance without added complexity. Clearly, synthetic workloads that are created to exploit weaknesses in the topology and routing algorithms, such as the worst-case workload, will perform poorly in those scenarios but it's unclear if such communication workloads exist in practice.

4.4.3.4 Multi-Rail Fat-Tree Evaluation

Using multiple separate studies, we focus on characterizing general trends as well as quantifying network performance for a Summit supercomputer approximate dual-rail Fat-Tree system under three application workloads. A multi-job test configuration consisting of the AMG, Crystal Router, and Multigrid workloads is used throughout our studies to explain expected performance of the system in a multi-job HPC environment. We present and discuss the results from four performance studies including single-rail vs. dual-rail comparison, rail scaling, job placement and network response to increasing compute performance. All executions simulate a multi-job run on the 3,564-node system using concurrently executing AMG, Crystal Router, and Multigrid application traces. The trace

workloads have 13,824, 1,000, and 10,648 MPI processes, respectively, and each compute node hosts eight MPI processes. Hence, our simulated workload utilizes 3,184 compute nodes, while the rest are kept idle.

To quantify network performance, each node within the 3,564-node simulated system collects and aggregates statistics for all MPI processes mapped to it. In multi-rail cases, the metrics are further aggregated over all NICs within a node. The following metrics are collected:

- *Bandwidth:* Observed bandwidth measures the rate at which each node is able to inject data into the network. Each compute node calculates the observed bandwidth by dividing the total amount of bytes transferred by the total time spent transferring the data.

- *Injected Traffic:* Injected traffic describes the total amount of data transferred throughout the network by each compute node over the length of the simulation.

- *Total Congestion Delay:* The total congestion delay quantifies the amount of time (in bytes) a node spends with one or more of its corresponding NIC's buffers completely filled. Lower is better, indicating less injection congestion.

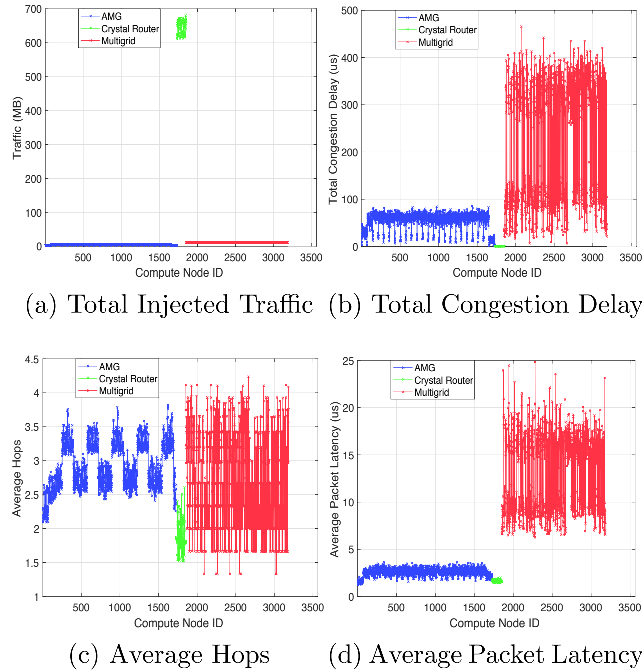
- *Average Latency:* Average latency describes the average source to destination delay of all packets injected per node.

- *Average Hops:* Average hops collects the average number of links traversed within the network by each packet. It does not include the compute node injection link.

We use a quad-plot layout with clustered error bars throughout this section to present bandwidth performance results for the different job placement and routing schemes. The sub-figure labeled "System Aggregate" represents the data aggregated over all active compute nodes in the 3 564-node system, i.e., idle nodes are excluded from the data aggregation. The remaining trace specific sub-figures present data from the same parallel multi-job run extracted with respect to the corresponding application trace workload. The y-axis in each plot shows observed bandwidth in GB/s. The height of each bar represents the average bandwidth among all compute nodes, while the upper and lower limits on each bar indicate the maximum and minimum compute node bandwidths, respectively. Furthermore, each cluster of bars has a corresponding hyphenated x-axis label indicating intra-rail routing and inter-rail injection policies. For example, a "Static-Adapt" label indicates all runs in that cluster used static intra-rail routing with adaptive inter-rail injection.

Multi-Job Trace Workload: We first perform analysis of the multi-job workload to characterize each application's base load on the Fat-Tree network. For the test we use the 3,564-node Summit approximation system previously described. For this study, all 3 application traces are executed in parallel. The simulation uses adaptive intra-rail routing, adaptive rail injection, and contiguously maps the processes of each job to compute nodes in the dual-rail system (eight processes per node), as shown in Figure 57a.

Among the three application traces, AMG and Multigrid have similar network characteristics opposite that of Crystal Router. Both, AMG and Multigrid, transfer much smaller amounts of data than Crystal Router, i.e., 246× less for AMG and 34× less for Multigrid, as shown in Figure 57a. The higher node counts of AMG and Multigrid traces also result in traffic that traverses longer distances in the Fat-Tree and with higher variance in average hop count than Crystal Router, see



(a) Total Injected Traffic (b) Total Congestion Delay

(c) Average Hops (d) Average Packet Latency

Figure 57: Network statistics per compute node comparing AMG, Crystal Router, and Multigrid application traces running in parallel on the 3,564-node dual-rail Fat-Tree system.

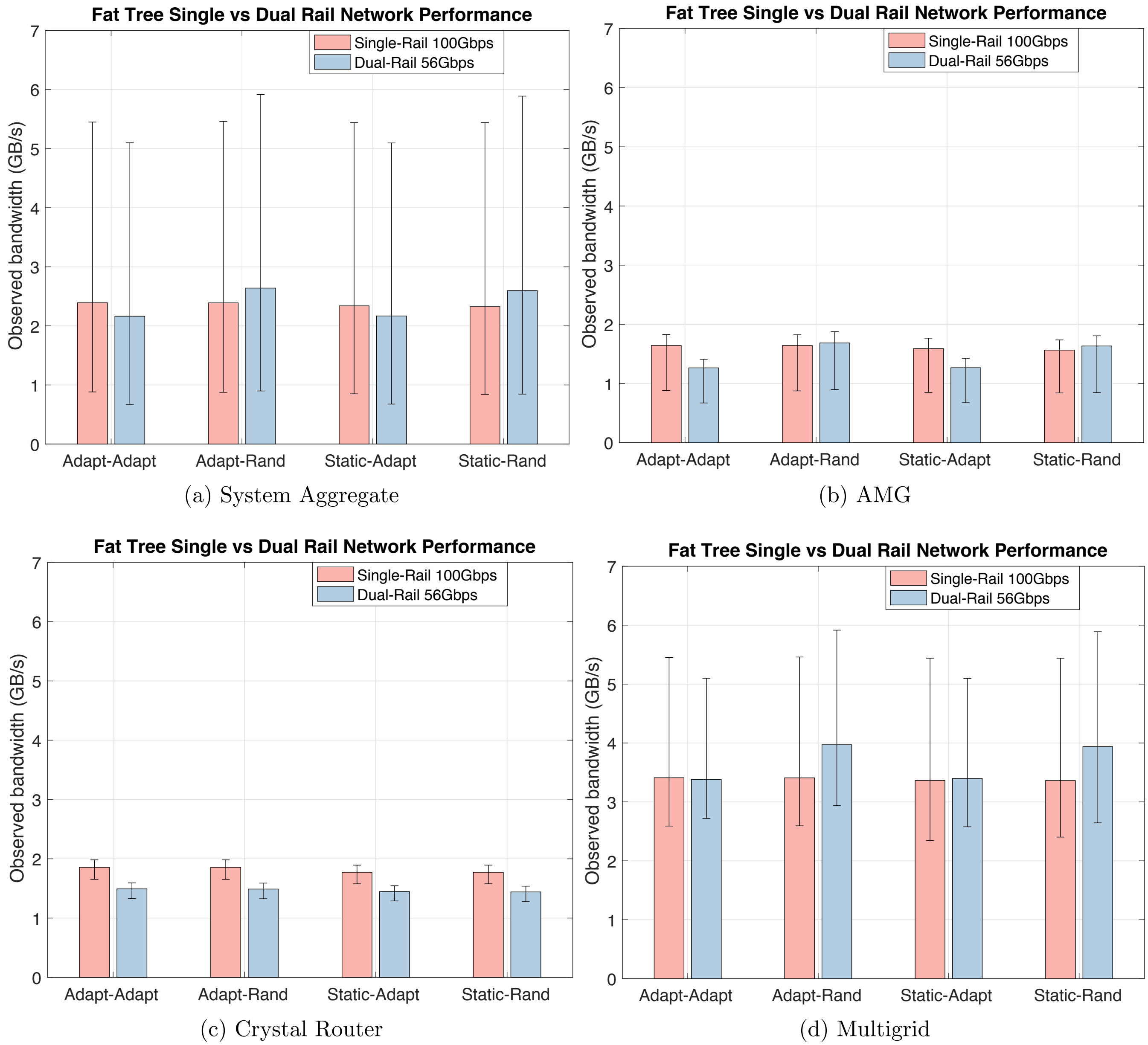
Figure 57c. Using only 124 nodes, the entire Crystal Router job fits within one pod of the Fat-Tree network lowering the maximum number of hops between any two MPI processes.

The total congestion delay, presented in Figure 57b, shows AMG and Multigrid experience more congestion at the node level than Crystal Router. The bursty communication pattern of AMG and Multigrid workloads follow periods of high injection rates resulting in long buffer saturation times coupled with lower data transfer totals. In comparison, the highly synchronized communication pattern of Crystal Router uses many blocking wait-all operations which provide a more balanced injection bandwidth, resulting in less congestion at each node's NIC. Finally, Crystal Router experiences less average packet delay. The lower hop count coupled with lower congestion delay of Crystal Router translates to a lower average packet latency compared to AMG and Multigrid, as shown in Figure 57d.

Overall, AMG and Multigrid follow similar non-synchronized communication patterns transferring small messages in high activity periods which results in higher NIC congestion than Crystal Router in the dual-rail Fat-Tree network. Crystal Router, with its smaller node count and synchronized communication pattern, transfers larger amounts of data resulting in longer average packet latencies.

Fair Comparison of Single-Rail vs. Dual-Rail: Here, we conduct a performance comparison of single-rail and dual-rail networks under comparable conditions. Theoretically adding an additional rail to a network increases the bisection bandwidth by two. We seek to determine the ability of the dual-rail network to match or even exceed network performance of a similar single-rail configuration using currently available InfiniBand link speeds under AMG, Crystal Router and Multigrid workloads.

Both test cases use the same 3,564-node system described previously, running the three contigu-



(a) System Aggregate

(b) AMG

(c) Crystal Router

(d) Multigrid

Figure 58: Network performance comparison of a single rail 100 Gb/s network and a dual rail 56 Gb/s network.

ously allocated trace jobs. The only differences between the two configurations are the number of rails and the link speeds. The dual-rail configuration has twice the number of switches and links using the slower FDR link speed of 56 Gbps. The single-rail configuration has half as many switches and links using the faster EDR links speed of 100 Gbps, i.e., nearly double the speed of FDR used for the dual-rail case. Figure 58 presents the observed bandwidth performance of the single-rail and dual-rail network configurations under all combinations of intra-rail routing and inter-rail injection policies. Rail injection has obviously no influence for the single-rail but is included in Figure 58 for easier comparison with dual-rail.

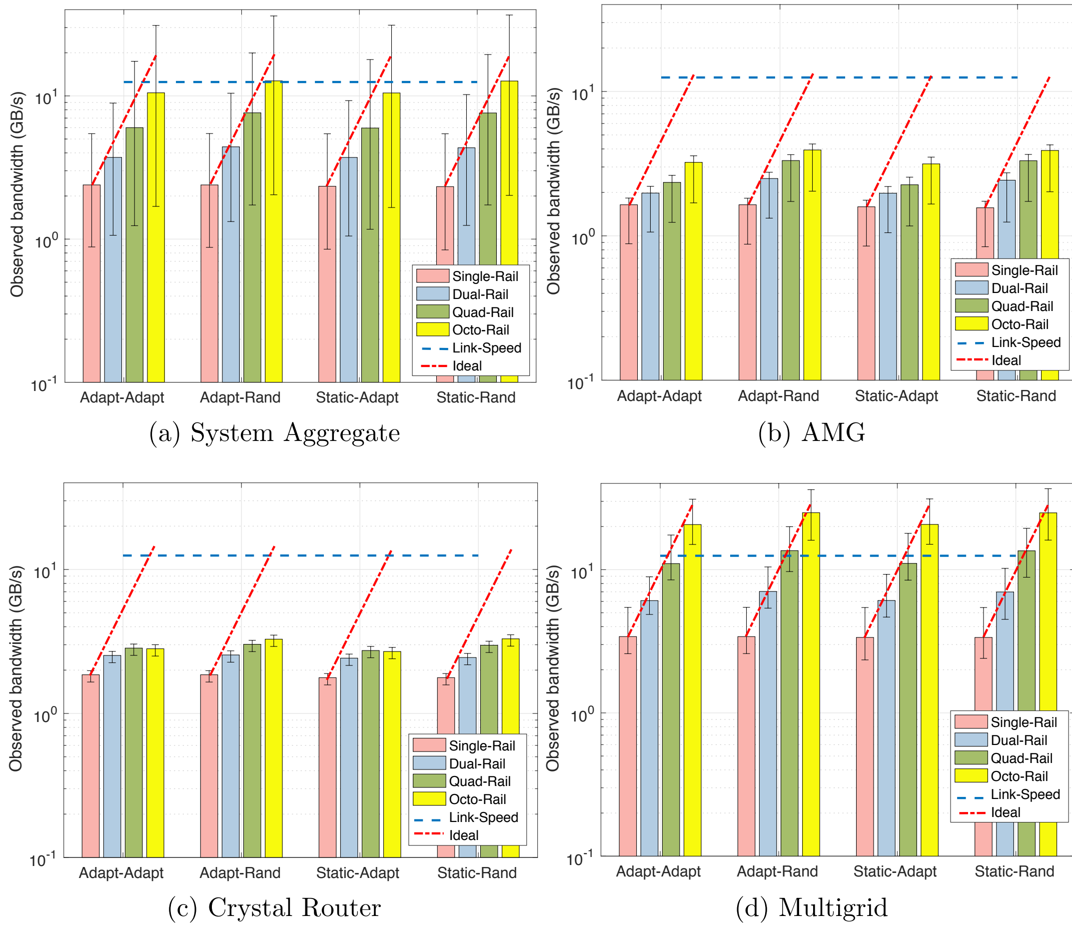
Overall, the observed bandwidth performance is almost identical for both rail configurations. A slight variance in performance between the two networks can be seen depending on the application trace and dual-rail injection policy. For example, Crystal Router achieves better single-rail performance for all rail injection and routing algorithms with up to 25% improvement over dual-rail. Multigrid observes equal or better performance on the dual-rail configuration with up to 17% improvement compared to single-rail. Finally, AMG sees split preference to single-rail and dual-rail depending on the dual-rail injection policy. Adaptive rail injection suffers up to 26% lower bandwidth on the dual-rail configuration while random injection allows for up to 4% improved bandwidth on the dual-rail network. When injection bandwidth for an application trace falls below link bandwidth, which is the observed outcome in the bursty AMG and Multigrid workloads, then output buffers on the compute nodes are emptied by the time packets are generated. In this case, packets are repeatedly issued on the first rail and can lead to a slight network load imbalance and congestion within the network which compounds over time. In contrast, random rail injection is impervious to injection loads and balances communication among available rails at all times, resulting in less congestion and slightly higher bandwidth performance under bursty communication patterns.

In summary, the dual-rail network matches and even exceeds bandwidth performance of a similar single-rail network with roughly twice the link speed. The network performance is application and rail injection policy dependent with Crystal Router outperforming on the single-rail and Multigrid outperforming on the dual-rail configuration with random rail injection.

Rail Scaling: Next, we seek to analyze the improvement in network performance as the number of rails increases. Ideally, scaling the number of rails in the network should proportionally increase the observed bandwidth. However, taking into account that multiple jobs are running on the HPC system with each job having its own communication pattern, such linear increases in observed performance is doubtful. To quantify bandwidth performance under realistic HPC center conditions, we configured our simulated Fat-Tree networks for single, dual, quad, and octo-rail configurations all with link speeds of 100 Gbps. Again, using the same 3,564-node system as previously described, each rail configuration is subjected to a multi-job execution of the three contiguously allocated jobs. Each rail configuration is tested with varying intra-rail routing and inter-rail injection combinations. The resulting observed bandwidths are shown in Figure 59.

At the aggregated system level, see Figure 59a, observed network bandwidth starts off increasing from one to two rails across all rail injection and routing policy executions. Eventually, the increases in observed bandwidth start to trail off at four and eight rails. Similar to previous experiments, the performance depends more on the rail injection policy than the intra-rail routing algorithm. The best performance at the system level is achieved by random rail injection allowing up to 5.3× speedup in observed bandwidth at eight rails over one rail. As discussed in the previous experiments, adaptive rail injection resorts to an over-utilization of the first rail and under-utilization of the remaining rails in multi-rail configurations under the bursty communication patterns such as Multigrid. The net effect is less performance compared to random rail injection which balances the workload regardless of injection load utilizing both rails equally.

Rail scaling performance of the individual traces shows mixed results. Crystal Router observes poor scaling across the board achieving at most 1.86× increase in observed bandwidth under static intra-rail routing with random rail injection while using eight rails compared to one rail. Increasing the number of rails provides extra links at the same link speed. However, Crystal Router synchronizes after every few messages making it more sensitive to end-to-end packet latency than injection bandwidth and therefore limiting the effectiveness of additional rails.



(a) System Aggregate

(b) AMG

(c) Crystal Router

(d) Multigrid

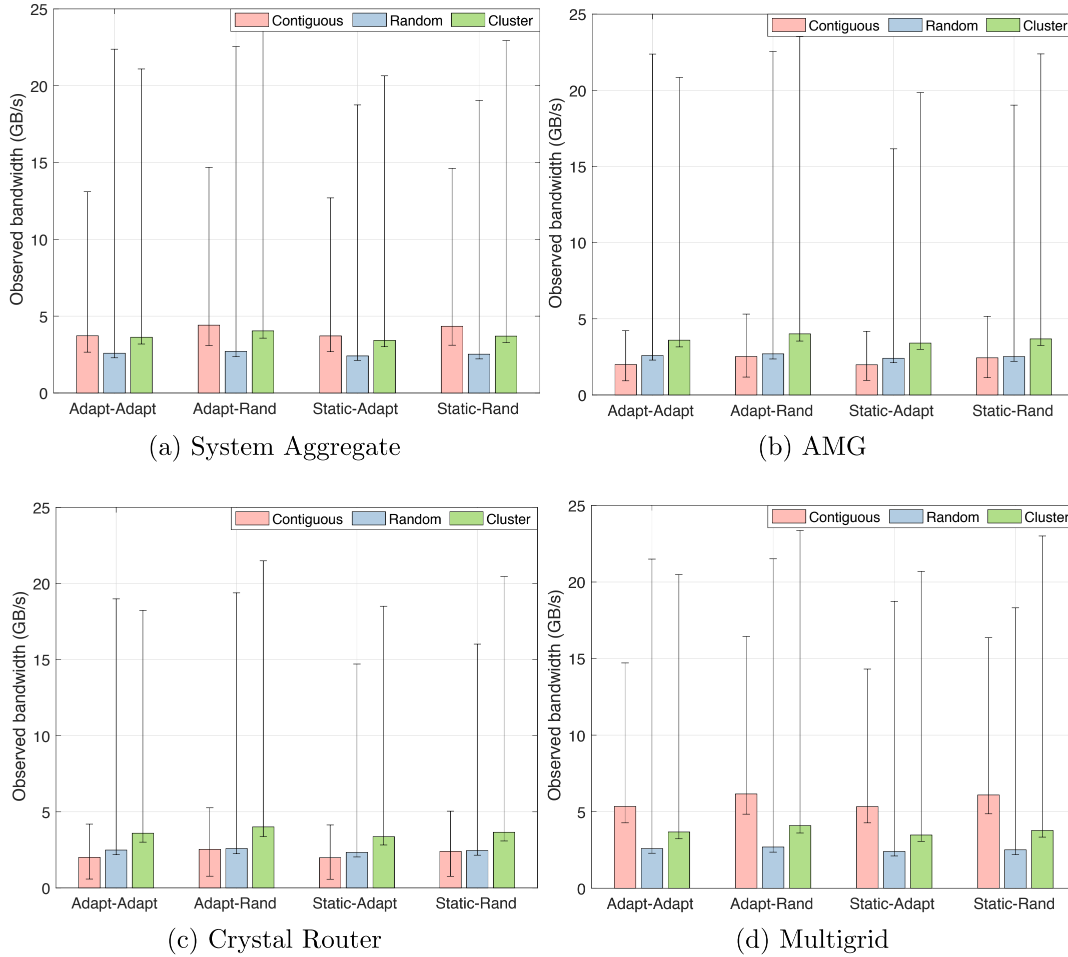
Figure 59: Observed network bandwidth for one, two, four, and eight rails.

In comparison, Multigrid strongly benefits from additional rails, see Figure 59d. Under random inter-rail injection, Multigrid obtains a 7.3× increase in observed bandwidth when going from one to eight rails. The reason is Multigrid's communication pattern which follows bursty periods of high injection rates sending large quantities of small messages with fewer synchronization points. As shown in Figure 57, the Multigrid application trace observes significantly more total congestion delay than Crystal Router. Hence, each additional rail increases the available bandwidth and path diversity between source and destination nodes.

AMG's performance, shown in Figure 59b, falls between Multigrid and Crystal Router in response to an increasing rail count. Performance increases decently with 1.5× speedup for two rails but quickly tapers off with a maximum speedup of 2.4× when using eight rails. Less network congestion, see comparison to Multigrid in Figure 57b, indicates fewer queued packets available to take advantage of additional rails.

In summary, performance improvements resulting from additional rails in a Fat-Tree network depend on the application's communication pattern. Applications similar to Crystal Router, transferring small numbers of larger packets and doing frequent synchronization, see diminished improvement with additional rails. However, applications sending large numbers of small packets with fewer synchronization points, such as AMG and Multigrid, gain major improvements in observed

bandwidth.



(a) System Aggregate

(b) AMG

(c) Crystal Router

(d) Multigrid

Figure 60: Multi-job allocation study on the dual-rail Fat-Tree network using contiguous, random, and clustered allocation policies.

Multi-Job Placement: Job placement describes the selection and mapping of MPI processes in one or more applications to available compute nodes in an HPC system. Mapping processes can have a large effect on the performance of the workload depending on the specific traffic pattern. Many times, in an HPC environment, mapping selection is highly limited due to the fragmented availability of nodes as applications are constantly allocated and completed at various times and requiring varying quantities of nodes. In this section we investigate Fat-Tree network response to the following three placement policies that are either in practice at HPC centers or have been shown in previous works to provide performance increases [168].

- *Contiguous:* Each job is assigned a consecutive set of available nodes. This method helps to maintain spatial locality of nodes, reducing the number of hops, but also increasing the chances of congestion along the localized links.
- *Random:* All jobs receive a random selection of available nodes. This approach can help to distribute the communication load throughout the network, but may increase the number of

hops on Fat-Tree networks as packets are increasingly forced to traverse spine switches to reach their destination.

- *Clustered:* The clustered placement randomly assigns each job a central node. Subsequent node selections are then clustered around the center for each job, where the distance to the center follows a geometric distribution. Hence, most allocated nodes maintain a close spatial locality, while few nodes are distributed across the entire system. This allocation scheme tries to mimic the natural fragmentation found in batch systems of production HPC systems.

For this set of experiments, the variable parameters are job allocation, rail injection and intra-rail routing policies. In each test case, the link speed remains constant at 100 Gbps. The results are visualized in Figure 60.

Starting with the aggregate system level in Figure 60a, both contiguous and clustered allocation policies outperform random for all intra-rail routing and rail injection combinations. Contiguous allocation also shows a significantly smaller variance in maximum observed bandwidth among the compute nodes. At the application level, the contiguous allocation policy results in the worst performance for both AMG and Crystal Router applications. Both applications perform better under the clustered approach of structurally spreading out communication to small groups of compute nodes instead of one contiguous block or a completely random scattering. Nodes within the structured small groups have a high chance of residing in the same pod and therefore reduce the number of hops between one another. It also provides a better opportunity to distribute different applications and their different network loads throughout the system to avoid self starvation of network resources in workloads such as Crystal Router which transfer large quantities of data.

Contiguous allocation for Multigrid offers best overall performance, where it sees up to 1.6× improvement in bandwidth over the second best option of clustered allocation. Compared to clustered and random approaches which can intertwine MPI processes of different applications, contiguous allocation helps to minimize interference by providing potentially disjoint groupings of network resources in the Fat-Tree topology. This helps applications such as Multigrid with sporadic bursts of communication from competing with large data transfer applications like Crystal Router.

Similar to the other studies, the injection policy provides a stronger influence on network performance than the intra-rail routing policy. Choosing random injection over adaptive provides slight improvements up to 15% for contiguous and cluster allocations. Random allocation policy uniformly distributes the workload throughout the system and sees virtually no benefit to the rail injection and intra-rail routing policies which also serve to further balance network load.

In summary, network performance as it relates to job placement again largely depends on the parallel job applications and their workloads. The cluster approach provides a fair distribution of Fat-Tree network resources to all jobs while contiguous mapping can deliver the strongest observed bandwidth for applications similar to Multigrid. Random rail injection can further contribute to improve network performance across the system for contiguous and cluster allocation policies.

Increased Computational Power per Node: In this final multi-rail study, we present results quantifying the ability of the dual-rail Fat-Tree network to match increasing compute node performance. We capture the response of the network to the increasing compute load by increasing the number of application trace processes mapped to each node in the system. Compute nodes of the Summit HPC system will have multiple IBM Power9 CPUs [133]. With 24-cores per POWER9 CPU [117], two CPUs per node will result in up to 48 MPI processes mapped onto a single node. Therefore, we simulate configurations of 8, 16, and 48 application trace processes per node. Since

Approved for Public Release; Distribution Unlimited.

the application size remains constant as the number of processes per node increases, the number of compute nodes utilized by each application job decreases. Finally, we assume a contiguous job placement policy for this study. However, we are varying the intra-rail routing and inter-rail injection combinations.

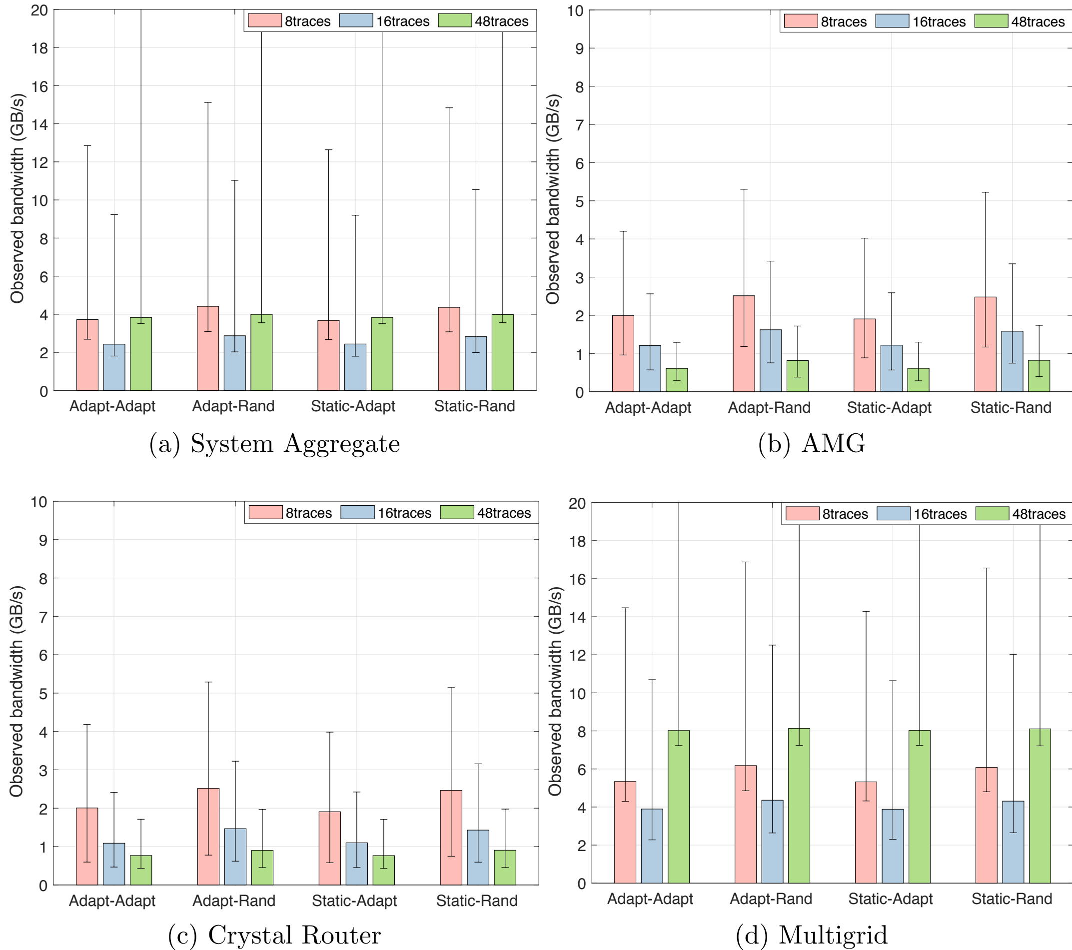


Figure 61: Dual-rail network performance when increasing the number of trace processes mapped to a node from 8 to 48.

Figure 61 shows our simulation results for increasing the number of trace processes per node on the 3,564-node dual-rail system. Once again, the rail injection policy has a larger effect on network performance than intra-rail routing, with random injection consistently outperforming adaptive. Both, AMG and Crystal Router, experience a constant decrease of observed bandwidth in response to increasing trace processes per node, see Figure 61b and 61c. The compressed mapping condenses network traffic to a smaller set of available links resulting in increased packet latency and increased congestion. In Figure 61d, Multigrid follows a unique trend, i.e., first the bandwidth decreases similarly when increasing from 8 to 16 processes per node, but then improves by up to 130% when scaling further to 48 processes per node. The explanation for the initial drop in performance arises from the growth in network traffic over a smaller set of links. However, the performance increases again due to a significant drop in average packet hop count. At 8 processes per node, Multigrid

requires 1,331 nodes or just over four pods. Mapping 48 processes per node further decreases the footprint to 222 nodes fitting all Multigrid processes within one pod, completely removing the need for any packets to traverse through the spine of the Fat-Tree to reach the destination.

Regardless of application workload, the general trend shows that network performance decreases in response to more dense MPI process per node mappings. Nonetheless, with the Multigrid example we see that positive performance can be achieved through the ability to isolate applications in distinct pods within the network lowering link traversal counts and network interference.

4.4.3.5 Synthetic Workload Performance

We evaluate performance of each network topology under synthetic workloads. In the first part, the evaluations look at the ability of each network to handle increasing levels of injection bandwidth and determine the saturation point at which the given topology is no longer able to match increasing rates of messages entering the network. The second part considers only the end time performance of each traffic workload as we increase the message injection rates.

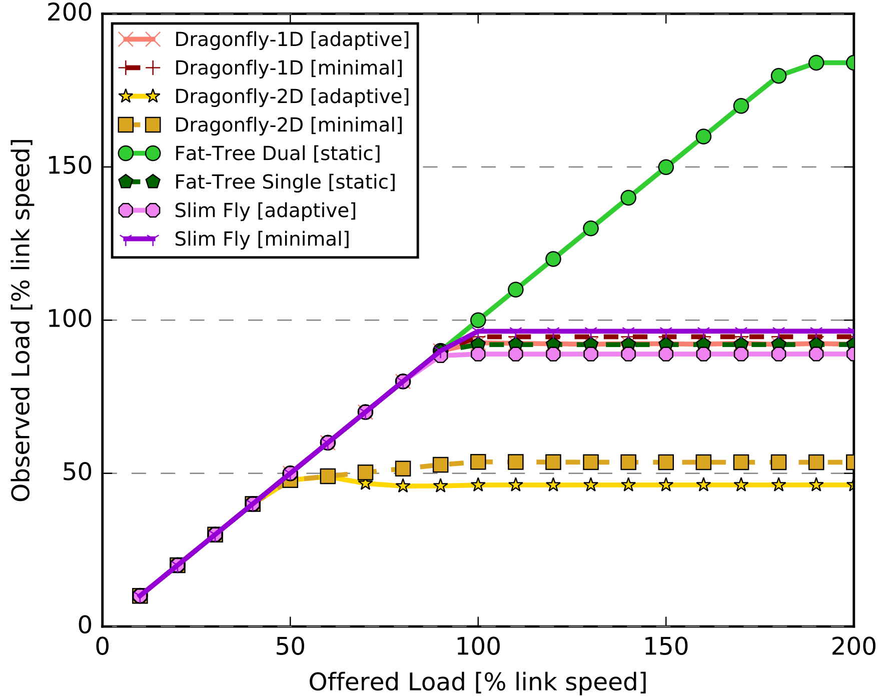
Offered Load Analysis: This section evaluates the performance of each network topology as we increase the traffic injection rate to test how much effective bandwidth each network configuration is able to utilize before reaching network saturation. We test all 3K node network configurations including Dragonfly 1D and 2D, single and dual-rail Fat-Tree and Slim Fly. Additionally, Dragonfly and Slim Fly models are tested using both adaptive and minimal routing. Each of the configured systems is tested using both uniform random and bisection synthetic traffic workloads with increasing message injection rates going from 10% link speed up to 200% link speed.

Uniform Random Traffic: Starting with uniform random traffic, Figure 62 presents the results for all network configurations. The overall trends show the networks matching injection bandwidth up until the network is fully saturated with traffic and has reached its peak effective bandwidth for the uniform traffic workload. Slim Fly performs as expected achieving just short of 100% and 200% observed load.

Dragonfly 1D and the single-rail Fat-Tree achieve very similar performance for uniform random traffic as Slim Fly. Fat-Tree leverages full bisection bandwidth to provide congestion free transfers up to roughly 98% offered load. Dragonfly 1D matches the performance benefiting from low hop counts due to the all-to-all connectivity within groups and strong ratio of 15 to 12 for global to local links per router. The dual-rail Fat-Tree performs as expected, achieving roughly double the performance of the single-rail version matching offered load up until about 195%.

Dragonfly 2D reaches network saturation at roughly 50% offered load which is roughly half the performance of the other single-rail configurations. Dragonfly 2D has a higher ratio of local links to global links (30:8) for each router compared to Dragonfly 1D (15:12) and Slim Fly (13:9). This results in a larger hop count when sending messages between nodes in different groups as is the case with uniform random traffic. The end result is a larger network utilization per message and reaching network congestion at only 50% offered load.

Routing performance shows minimal routing consistently outperforming adaptive by a few percent. Uniform random traffic is naturally distributed across the network and does not focus traffic along any specific points of congestion. Therefore, selecting a non-minimal path when doing adaptive routing results in increased path length with no improvement in congestion. The adaptive routing algorithms are set to perform minimally when no packets exist on the minimal path port. However if there exists a non-minimal path port without a packet in it's occupancy then that non-



(a) Uniform Random Traffic

Figure 62: Observed load of the networks in response to increasing offered load (measure as a percentage of link speed) for the uniform random synthetic traffic workload.

minimal path will be selected. The selection of non-minimal paths rarely occurs at the lower values of offered load but when offered loads reach the point at with it pushes each network to its point of saturation, the increased congestion forces the occasional selection of non-minimal paths and results in slightly lower observed load for adaptive over minimal routing.

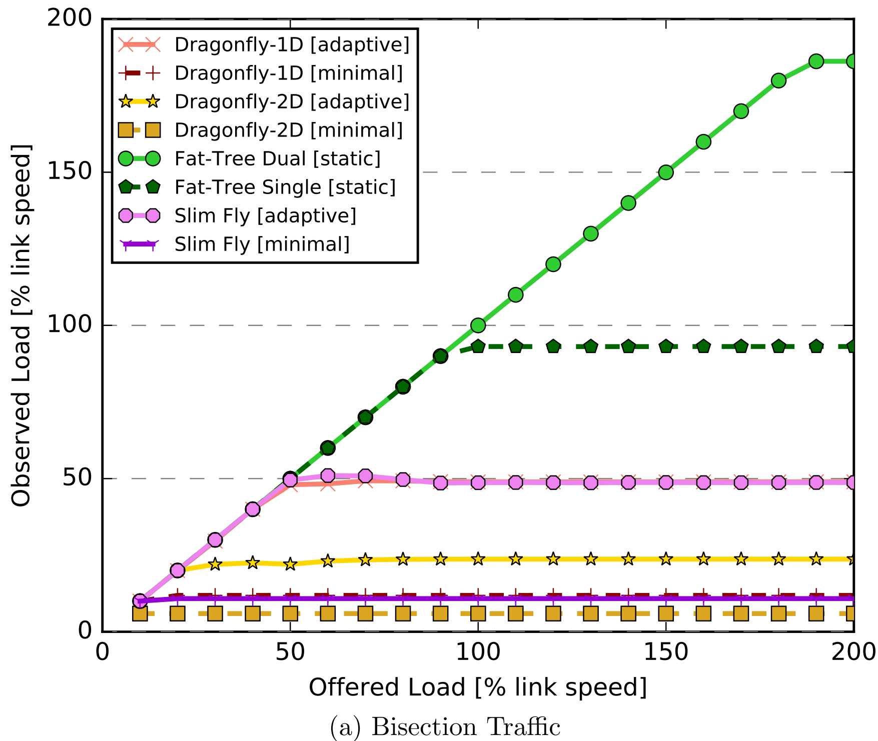
Bisection Traffic: Offered load results for the network configurations are shown in Figure 63. Again the figure highlights the point at which each network becomes fully saturated with traffic and no longer able to transfer data at the rate in which the data is injected. In this case, the results show lower overall performance than that of uniform random as the bisection workload presents a more adversarial load for Dragonfly and Slim Fly configurations than for Fat-Tree.

The best performing configuration is the Fat-Tree. By design, the Fat-Tree has full bisection bandwidth so using static routing results in each compute node pairing communicating using unique path's in the network. The single-rail and dual-rail Fat-Tree networks observe close to ideal performance of 100% and 200% link speed.

The routing comparison shows adaptive largely outperforming minimal routing. Unlike UR traffic, in which all compute nodes randomly select a new destination before sending a new message, the bisection traffic pattern sends all messages to the same destination node. For minimal routing, which selects the same static shortest path for every message, this results in a constant load on one specific path through the network. The links along the shortest path quickly saturate and minimal routing suffers as a result. Adaptive routing balances the benefits of shortest path provided by minimal routing with the benefits of distributing the workload across the network provided by the non-minimal paths.

Slim Fly and Dragonfly 1D show similar results for both adaptive and minimal routing. Adaptive reaches a max observed load of 50% while minimal routing achieves a max observed load of 10%. As mentioned, minimal routing suffers from the hotspots following the static communication pattern of the bisection workload. Adaptive routing is able to increase performance by using non-minimal paths to distribute the workload across the network but the trade off is additional hops and using additional network bandwidth.

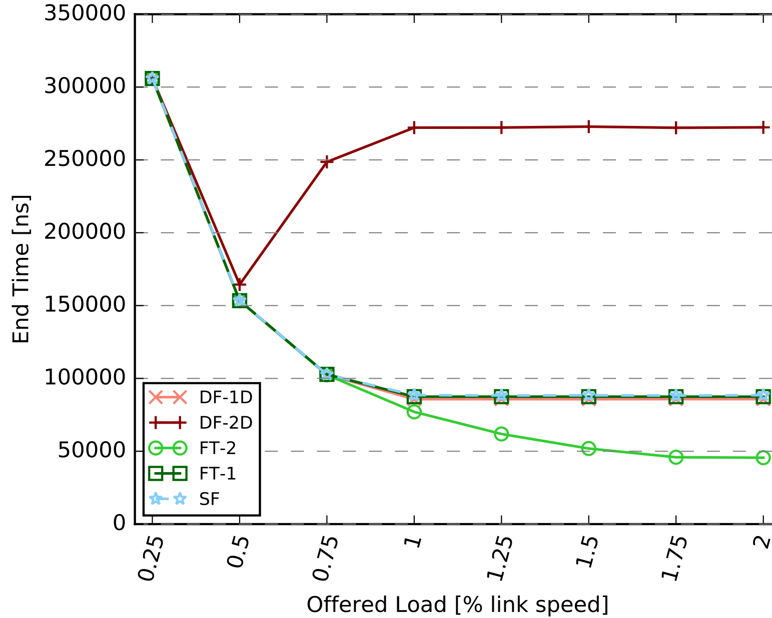
Finally, Dragonfly 2D yields the lowest observed load performance as it reaches saturation at 22% and 5% respectively for adaptive and minimal routing. Similar to the uniform random traffic pattern, the bisection workload results in a large amount of traffic traversing between groups. The longer hop counts means the messages are spending more time in the network, starving additional messages from utilizing available resources.



(a) Bisection Traffic

Figure 63: Observed load of the networks in response to increasing offered load (measure as a percentage of link speed) for the bisection synthetic traffic workload.

End Time Performance: In this section we study the end time performance of each topology under synthetic workloads. At the end of the day, the most important result from an application developer's perspective is the time it takes to complete a given workload. The less time that is spent performing the execution, the more time that can be put towards additional executions and/or analysis. Again, testing is done on the Dragonfly 1D, Dragonfly 2D, single and dual-rail Fat-Tree and Slim Fly 3K node network configurations. The synthetic workloads chosen for testing include MPI collectives, nearest neighbor traffic, uniform random and bisection traffic. Each workload is executed with one MPI process per compute node and the data sizes, and patterns are consistent between the different network systems to maintain a fair comparison. For each workload, a total of 4,000 messages are sent by each compute node and a message size of 256 bytes is used. The



(a) Uniform Random

Figure 64: End time performance for all network configurations running uniform random synthetic workload with increasing offered load.

offered load is increased from 25% link speed to 200% link speed to evaluate network performance in response to increasing traffic.

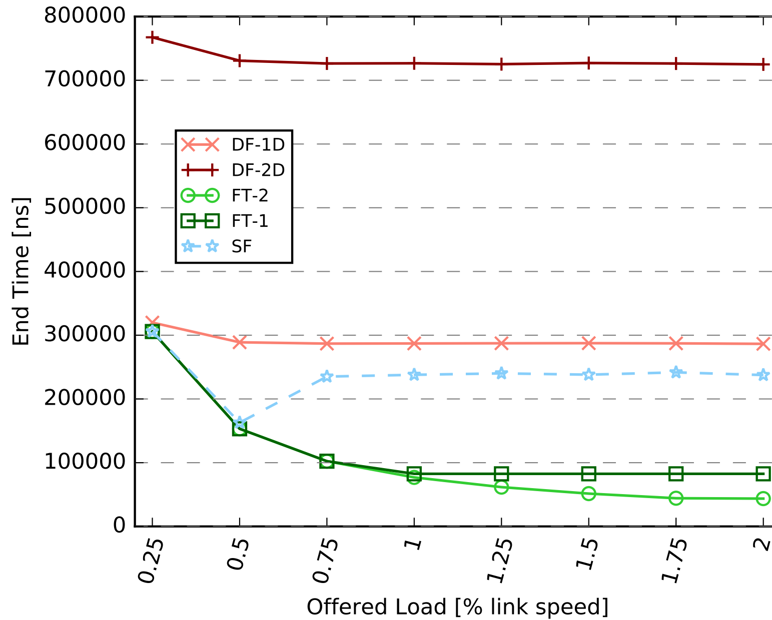
Starting with the straightforward uniform random workload, we can see in Figure 64 that the general response from the network configurations is a reduction in end time as the offered load is increased. Dragonfly-1D, Slim Fly, and the single-rail Fat-Tree configurations see very similar end time performance improving up until messages are injected into the network at 100% link speed and the networks are fully saturated.

The dual-rail Fat-Tree sees continued improvement in end time up until messages are injected at 200% link speed. In the end, the additional bandwidth provided by the extra rail provides the dual-rail Fat-Tree networks with a 1.7x speedup in end time over the single-rail networks for uniform random traffic offered at 200% link speed.

The one exception to the general trend is the Dragonfly-2D which first matches an improved end time performance going from 25% to 50% offered load but then the trend shifts and end time worsens at 75% and 100%. Interestingly, at 50% offered load, Dragonfly-2D reaches the saturation point on the offered load curve seen in Figure 62 of the previous results section. At this point, Dragonfly-2D is able to benefit from non-minimal paths without the extra hops stealing resources from other messages. Going above that point, the faster injection rate of 75% link speed means more messages in the network at any one point in time, creating more congestion, forcing more non-minimal paths, and translating to even further congestion and packet latencies. The end time results settle at 100% where Dragonfly-2D is roughly 3x slower than the other topologies.

For the bisection workload running with increasing offered load, Figure 65 shows a more varied end time performance than was observed for uniform random. As was observed in the offered-load analysis of Figure 63, the Fat-Tree configurations perform very well with the bisection workload

resulting in consistent improvements in end time up to 100% offered load for the single-rail Fat-Tree and up to 200% for the dual-rail Fat-Tree.



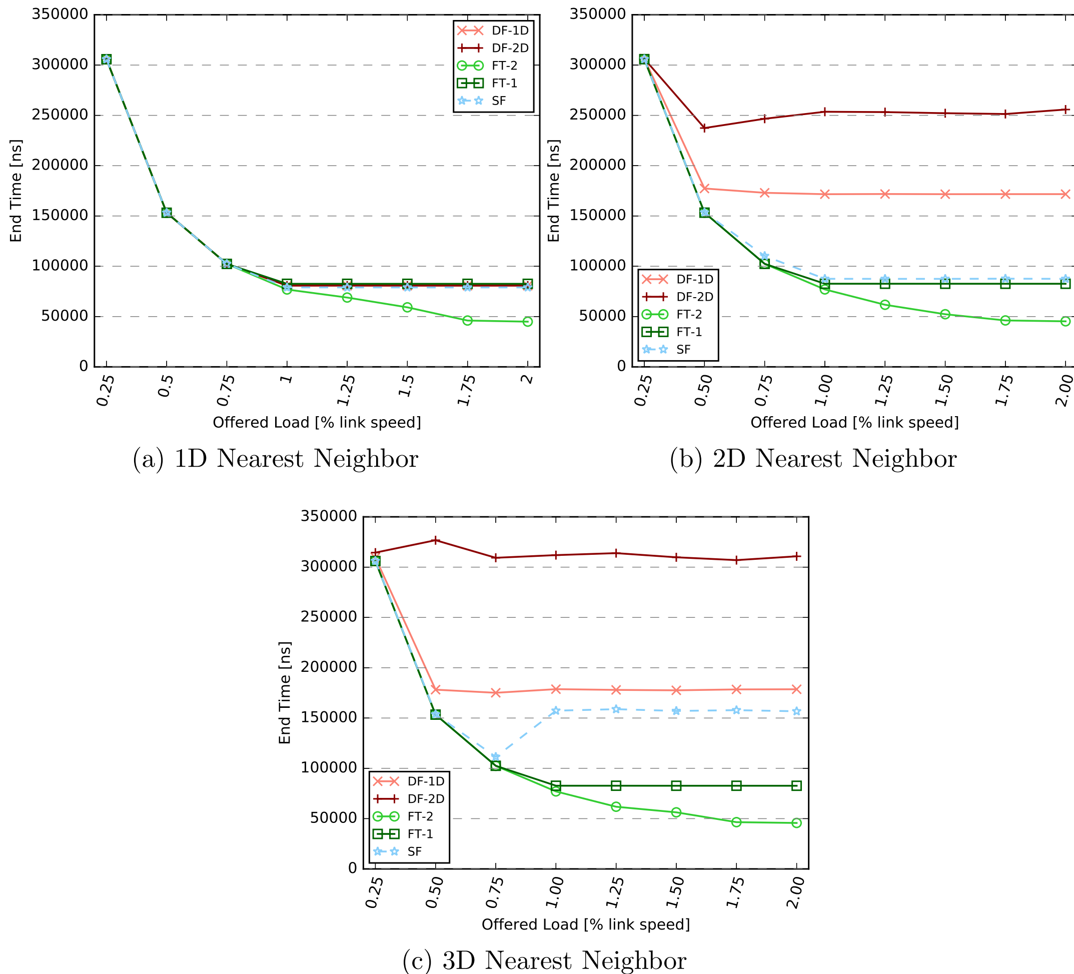
(a) Bisection

Figure 65: End time performance for all network configurations running bisection synthetic workload with increasing offered load.

Performance for non Fat-Tree networks is not as ideal. The bisection workload is most adversarial for the Dragonfly-2D network as the traffic pattern results in a large amount of inter-group traffic and therefore larger hop counts and early saturation of the network. Dragonfly-2D ends up 9x slower at 100% offered load than the best case end times of Fat-Tree. The 1D Dragonfly network follows the same trend of the 2D Dragonfly with little improvement in end time performance in response to increasing offered load. However, Dragonfly-1D does observe much better overall end time reaching a speedup of 2.5x over Dragonfly-2D at 100% offered load.

Slim Fly sees a slight overall improvement in performance over the 1D Dragonfly. Interestingly, the Slim Fly observes the same dip in performance phenomena observed by Dragonfly-2D for uniform random traffic. At 50% load Slim Fly reaches the network saturation point where the congestion avoiding benefits of non-minimal paths outweigh the benefits of the additional hops. Unfortunately, with increasing offered load, the additional messages in the system create even more congestion which non-minimal path selection exacerbates by taking longer paths through the network.

Figure 66 presents the results for end time performance of the one, two, and three dimensional nearest neighbor traffic patterns on the six network configurations. Overall, the workloads follow expected results with the networks achieving the best end time for the 1D exchange and the worst performance for the 3D exchange. The 1D nearest neighbor pattern translates to each compute node in the system communicating with only it's nearest connected compute nodes. All networks efficiently handle the simple communication pattern and observe similar increases up to 4x end time performance at 100% offered load for single-rail configurations and up to 6x increase at 200% offered load for dual-rail networks.



(a) 1D Nearest Neighbor (b) 2D Nearest Neighbor

(c) 3D Nearest Neighbor

Figure 66: End time performance for all network configurations running 1D, 2D, and 3D nearest neighbor synthetic workloads with increasing offered load.

Moving to the two dimensional nearest neighbor workload, we continue to see the improved end time performance from Slim Fly and the Fat-Tree configuration but the Dragonfly networks observe no improvement going past 50% offered load. It's important to note that while the second dimension in the 2DNN workload is a nearest neighbor in the simulated domain, that does not translate to nearest neighbors in the physical HPC system domain. What we end up with is compute nodes having two communication pairings with it's nearest neighbor compute nodes as well as two new communication pairings with compute nodes $+offset$ and $-offset$. For the Dragonfly networks, the second dimension communication pairing results in frequent inter-group communication. Furthermore, the communication consistently follows along the same inter-group links causing congestion and pushing the adaptive routing to select more non-minimal paths and resulting in only a 1.2x improvement in end time performance at 100% offered load. Dragonfly-1D has a better inter-group connectivity per each router and achieves a best case end time that is roughly 1.5x times faster than the best case Dragonfly-2D performance.

Slim Fly also observes inter-group communication resulting from the second dimension of communication but minimal routing is effective for this specific communication pattern. Specifically,

even if Slim Fly is communicating with nearest neighbor compute nodes, the traffic is by design routed in a way such that the minimal path distributes the traffic across the whole network. In this way, the network achieves a strong load balance and maintains minimal hop counts.

Finally, with six total messages exchanged each iteration, the 3D nearest neighbor workload further decreases end time performance for all networks except the Fat-Tree and Dragonfly-1D.

Dragonfly-2D is forced to use already limited inter-group connections to handle the third dimension of traffic resulting in end-times that do not improve with increasing offered-load and instead reaches saturation with an offered load of only 25% link speed. For Dragonfly-1D, the addition of traffic along the third dimension is similar to that of the second dimension. It increases the inter-group traffic with a new communication pairing pattern following a new offset. The all-to-all connectivity within a group and large number of inter-group links per router allows the Dragonfly-1D network to absorb the third dimension traffic using available bandwidth and paths without increasing congestion along existing communication paths of the first and second dimensions of traffic. The outcome is an end-time performance curve similar to the 2D nearest neighbor result.

Slim Fly sees mixed results for 3D nearest neighbor. At 75% Slim Fly reaches a point where it effectively balances the workload across the network alleviating hot spots along the 3D NN paths by using non-minimal paths and resulting in no added end-time over the 2D nearest neighbor workload. Beyond 75% Slim Fly becomes over-saturated and the non-minimal paths starve additional messages from using network resources. The net result is an increase in end-time performance at 100% offered load that is 1.8x slower than 2D NN performance and only 1.2x faster than Dragonfly-1D performance for 3D NN.

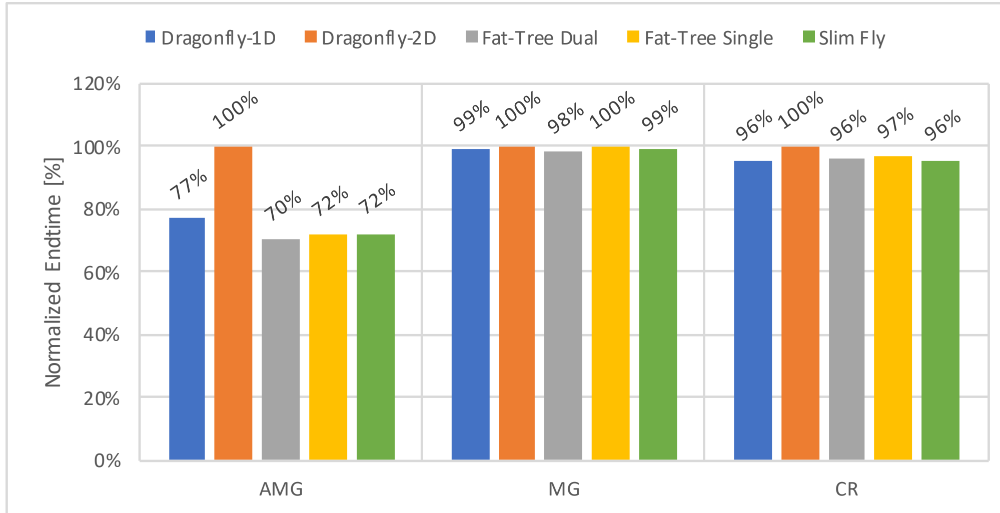
Both single-rail and dual-rail Fat-Tree continue the same end time performance scaling with 3D NN as they did with 2D and 1D. Fat-Tree does well with paired communication as each compute node pairing gets a unique path through the network. A bisection pairing achieves a completely unique path for each pairing but even with a structured pairing like 3D nearest neighbor, results in path selections that minimize path sharing and therefore result in the strong performance.

4.4.3.6 CPU Trace Performance

In this study, we run each CPU workload individually to benchmark performance on each of the four HPC network configurations. The CPU results are collected over the entire workload execution. Figure 67 shows the results for each network clustered by the application labeled on the x-axis. The end times for each network within a clustered group are normalized with respect to the worst-case end time within each clustered group of results.

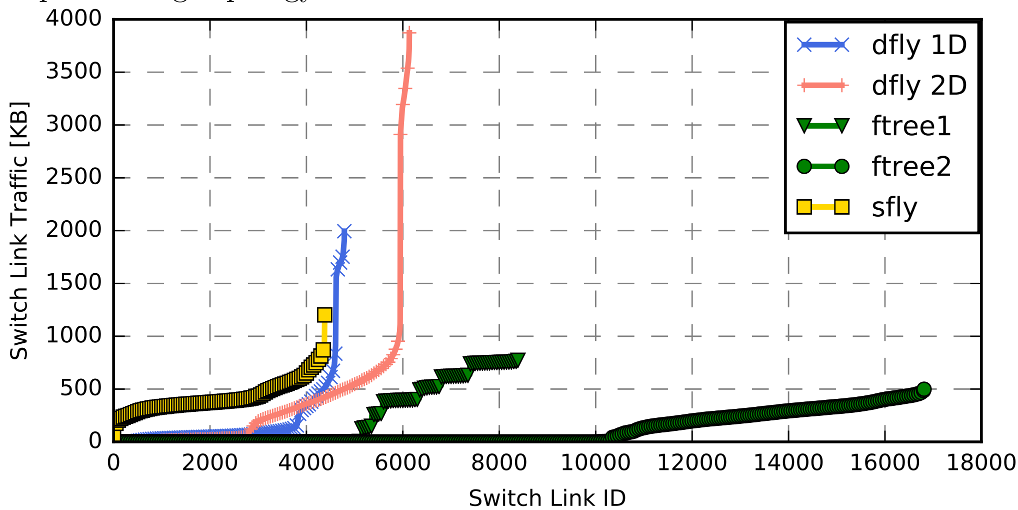
Here, we compare performance between topologies using application virtual end time and we use observed link traffic to explain the differences. The applications studied have widely varying end times, so we normalize each end time to clearly show how much faster each topology is in comparison to the worst-case performer. The bars within each bar plot are grouped into clusters as indicated by the x-axis workload labels. Link traffic data is presented in line plots sorted by increasing value. Each point on a switch traffic plot represents the aggregate traffic transmitted across a global link over the length of the simulation. In the case of Fat-Tree, global links are those connecting the core and aggregate level switches. For clarity, markers on switch traffic plots are shown every 50 points.

AMG shows the largest variance in end time between the network topologies. In this case, Dragonfly-1D is 23% faster than Dragonfly-2D, and best performance is observed on Slim Fly and Fat-Tree which are 28% faster than Dragonfly-2D. The traffic data presented in Figure 68,



(a) CPU Performance

Figure 67: Simulated end time results for CPU applications running alone on the Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly HPC systems. End times are normalized within a workload to the slowest performing topology result. Lower is better.



(a) Switch Traffic

Figure 68: Aggregate global link traffic for Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly running the single-job AMG execution.

shows Dragonfly-2D has a small number of global links that are heavily saturated compared to the remaining links. High saturation of global links for Dragonfly indicates packets are traversing deeper into the network, taking additional hops, and increasing chances of congestion. The high utilization hot spots along the global link effect many compute nodes and result in the slowdown in run time performance.

Moving on to the remaining CPU workloads, we see MG and CR observe similar performance across all topologies. Note, MG and CR aren't as communication intensive as AMG for a couple reasons. There are fewer processes, data is injected at a lower rate (quantity of data over time), and the communication patterns of MG and CR send more data directly along the diagonal. The

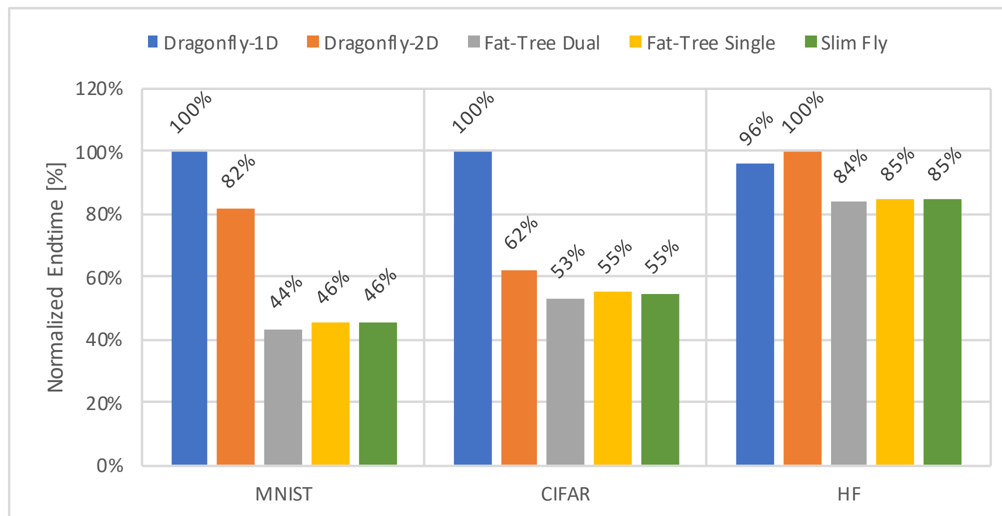
Approved for Public Release; Distribution Unlimited.

combination of these workload characteristics results in less demand on the network than AMG and leads to all networks being able to finish the MG and CR workloads in roughly the same time when run individually.

The dual-rail Fat-Tree configuration overall sees little improvement over the single-rail Fat-Tree. The improvement is roughly 2% on average over the single-rail version. The additional rail provides little improvement in end time performance because the workloads are mostly latency dependent in which case additional paths providing additional bandwidth does not improve end-to-end latency when little congestion is observed.

4.4.3.7 Neuromorphic Trace Performance

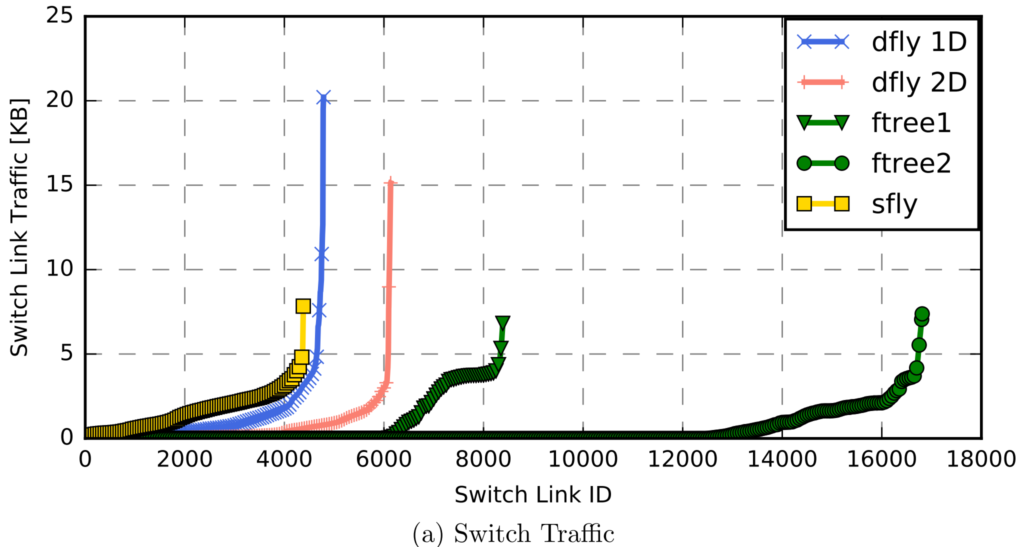
We want to say something which motivates the need to study the single job neuromorphic trace performance and then describe the setup. The neuromorphic results are collected for only one $1ms$ tick of execution. Performance results are presented in normalized end time bar plots and sorted line plots as discussed in the previous experiments.



(a) Neuro Performance

Figure 69: Simulated end time results for Neuromorphic applications running alone on the Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly HPC systems. End times are normalized within a workload to the slowest performing topology result. Lower is better.

Overall, the Hopfield workload achieves closer end times across all three HPC network topologies than the convolutional workloads of CIFAR and MNIST. In this case, Dragonfly-2D execution is 4% slower than Dragonfly-1D and both Fat-Tree and Slim Fly are 15% faster than the worst case Dragonfly-2D. While the Hopfield workload has more connections and sends more messages than MNIST and CIFAR, the all-to-all communication pattern helps to evenly distribute the messages across the entire HPC network and avoid hotspots. In this case, hop counts play a large role in end time as each extra hop incurs additional router traversal delays resulting in increased end-to-end message latency for the small 8 byte spike messages. Fat-Tree using static routing and Slim Fly using adaptive routing both have a max hop count of 6. Dragonfly-1D and Dragonfly-2D have larger max hop counts of 8 and 12 and end up 11% and 15% slower respectively than Fat-Tree and Slim Fly.



(a) Switch Traffic

Figure 70: Aggregate global link traffic for Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly running the single-job MNIST execution.

For MNIST and CIFAR, there is a significant difference in simulation end time between the HPC topologies as Fat-Tree and Slim Fly outperform worst-case performer Dragonfly-1D by 54% and Dragonfly-2D outperforms Dragonfly-1D by 18%. The layered connectivity and reduction in layer size of the convolutional workload focuses communication through a small subset of network links that can result in adversarial communication for the Dragonfly networks. Figure 70 shows Dragonfly-1D and 2D have global links that observe up to 2.7x and 2x more traffic than the worst case global links for Fat-Tree and Slim Fly. In this case, the specific convolutional network of MNIST translates to an adversarial communication mapping for the Dragonfly network resulting in adaptive routing moving traffic to global links to distribute the load.

CIFAR results are similar to MNIST with Fat-Tree and Slim Fly finishing in roughly half the time of Dragonfly-1D but in this case Dragonfly-2D also finishes much quicker at 38% less run time than Dragonfly-1D. CIFAR has the same number of layers as MNIST but the layers implement different numbers of connections and placements of connections again resulting in hotspots for Dragonfly-1D. Interestingly, it does not translate to as severe of hotspots for Dragonfly-2D. Running the Dragonfly topologies with random allocation of chips to nodes results in performance similar to Fat-Tree and Slim Fly and confirms the long end time is a result of an adversarial mapping of the convolutional network to the Dragonfly topologies. In summary, the performance for convolutional neural network applications can be highly dependent on the specific connections of each application.

4.4.3.8 Neuromorphic-CPU Interference

Neuromorphic processing provides a means of increasing computational power while maintaining a small power footprint, making it an option for integration in next generation supercomputers alongside a traditional computing architecture like the CPU. To test the impact of large-scale hybrid neuromorphic computing, we conducted a series of studies to quantify performance and interference of such a proposed system.

In this section, we construct theoretical hybrid neuromorphic/CPU HPC systems with a total of 3,000 nodes and using Dragonfly, Fat-Tree, and Slim Fly interconnect topologies. Each simulated

compute node consists of one neuromorphic processor representing the equivalent of one TrueNorth processor as well as one CPU processor. Both processors on a node share one network interface card.

To study the interference of neuromorphic and traditional CPU workloads running in a hybrid compute system, we execute the neuromorphic workloads in parallel with the three CPU application workloads on the four HPC network topologies. Each CPU and neuromorphic workload is mapped linearly to compute nodes with one CPU and one Neuromorphic MPI process per compute node. We use the simulation end time to measure the high-level interference.

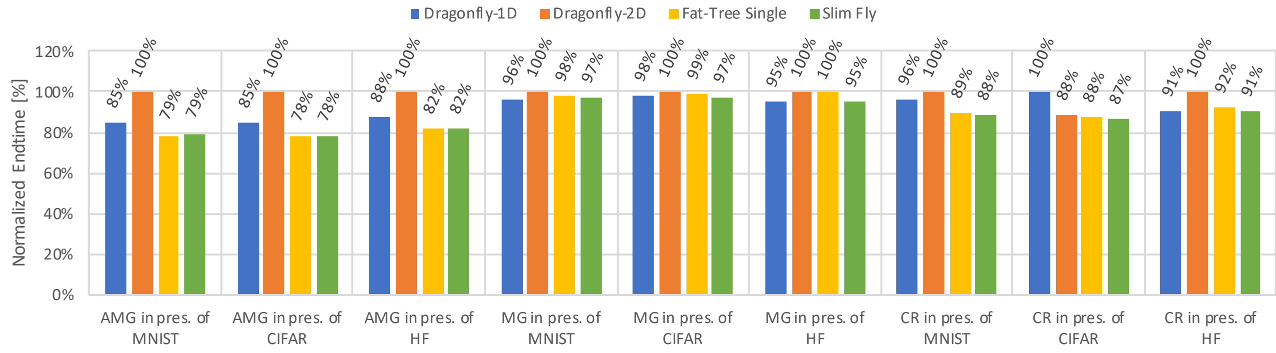
Measuring interference between the tick synchronized neuromorphic applications and unrestrained CPU architecture workloads poses a unique challenge. To study the effect of CPU applications on neuromorphic workloads, the neuromorphic workload is executed in parallel with the CPU application for the entire length of the CPU application. The longer the CPU workload end time, the more ticks of the neuromorphic workload are simulated. Since the progression of neuromorphic workloads is governed by the 1ms tick delay, and not simply by the availability of work, we compute the end time for the neuromorphic workloads, during multi-job executions, to be the average time to complete the work within each 1ms tick. This allows us to extract the true effect of the CPU workloads on the neuromorphic spike generation and sending process within each tick.

Lastly, to compare interference of the multi-job execution results between topologies, we compute and analyze the slowdown. This is computed as the difference in end time of the application in the multi-job and single-job executions divided by the single-job end time. We leverage the single-job CPU and neuromorphic application performance from prior CPU workload results and performance and interference results are presented in the form of bar clustered bar plots similar to the previous CPU and neuromorphic workload results.

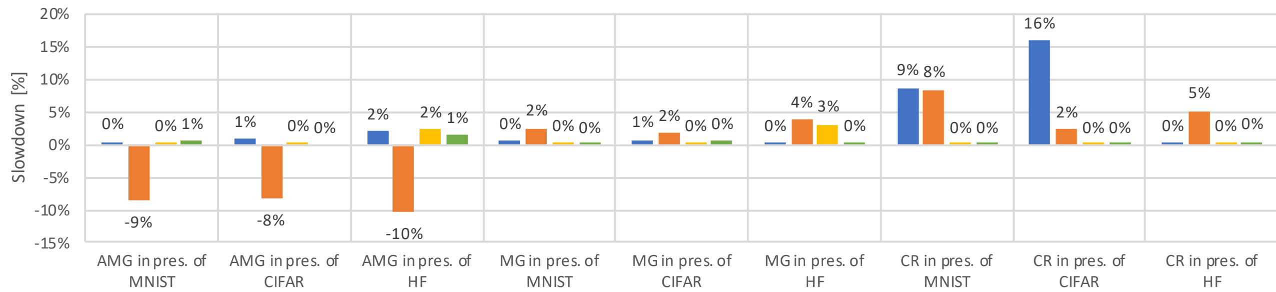
CPU in Presence of Neuro: Overall, end time performance and interferance results in Figure 71 show AMG, Multigrid and Crystal Router CPU workloads with little effect from running in the presence of neuromorphic workloads. Multigrid end time performance closely matches the trends observed for single-job execution with all four network topolgies finishing within a max of 5% of one another. Of the four neuromorphic workloads, Hopfield creates the most interference for Multigrid resulting in only 4% and 3% slowdowns for Multigrid.

The CPU workload Crystal Router, on the other hand, shows the largest negative effect observing up to 16% slower end time running in parallel with CIFAR. Dragonfly-1D sees more slowdown than others to make it the worst-case performer when running in the presence of MNIST and CIFAR. The convolutional neural networks present adversarial traffic creating hotspots along critical paths for Crystal Router. Crystal Router is a highly synchronized workload performing a wait operation after each send, making it a latency sensitive application. The global link hot spots seen from the MNIST and CIFAR convolutional NN type applications in Figure 69, force some traffic in CR to take longer paths through the network to avoid congestion resulting in delays for both Dragonfly configurations.

Finally, the AMG CPU workload shows some interesting results with Dragonfly-2D improving performance in the presence of all three neuromorphic workloads while the other network topologies see up to 2% slowdown. Dragonfly-2D sees improved performance benefiting from additional traffic on the network to increase network utilization and decrease congestion on global links. Switch traffic for the AMG workload running in the presence of MNIST is presented in Figure 72 and shows a small decrease in traffic for all Dragonfly-2D global links. Most notably, the max link traffic decreased by roughly 30% compared to the max link traffic observed in the single-job AMG

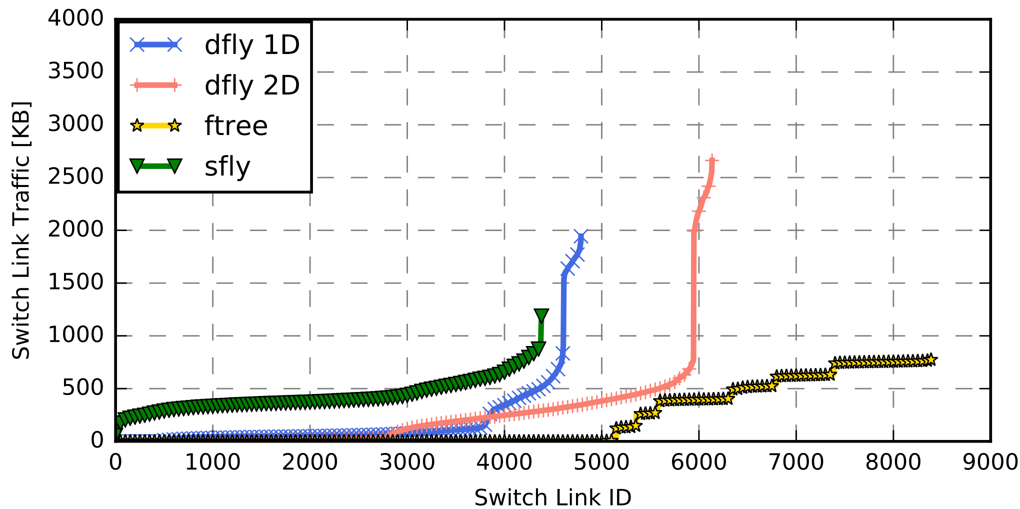


(a) CPU Job Performance (in presence of Neuro)



(b) CPU Job Interference (in presence of Neuro)

Figure 71: The top subfigure presents simulated end time results for the CPU applications when running in the presence of neuromorphic applications on the Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly HPC systems. End times are normalized within each workload pairing to the slowest performing topology result. The bottom subfigure presents the net slowdown in end time performance for CPU workloads when running in the presence of neuromorphic workloads. In both subfigures, lower is better.



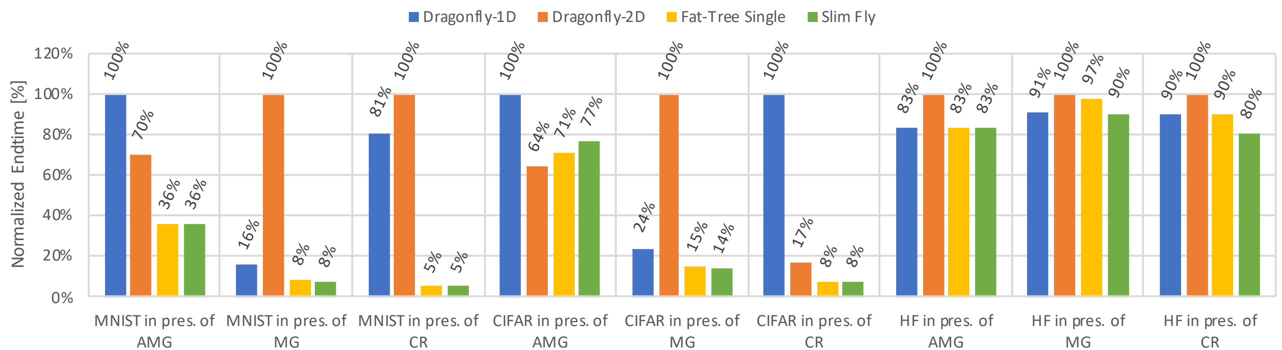
(a) Switch Traffic

Figure 72: Aggregate global link traffic for Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly running the hybrid AMG-MNIST execution.

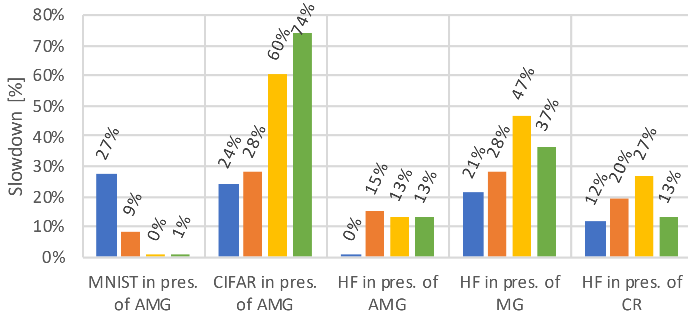
Approved for Public Release; Distribution Unlimited.

execution.

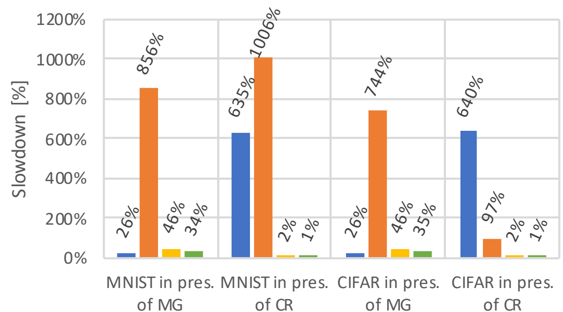
Neuromorphic in Presence of CPU: The results for neuromorphic application end time performance and interference in the presence of CPU workloads is presented in Figure 73. Overall, Neuromorphic workloads are highly susceptible to network interference from CPU workloads. Both Dragonfly-1D and Dragonfly-2D observe severe slowdowns for the convolutional NN workloads of MNIST and CIFAR when running in the presence of Multigrid and Crystal Router. Both Multigrid and Crystal Router have a large portion of their communication along the diagonal and that consistent nearest neighbor pattern can saturate local links. Adaptive routing, in response, pushes the traffic further into the network to global links to minimize congestion and resulting in overlap with the global link hotspots for the convolutional workloads. Fat-Tree, using static routing always take the same shortest path regardless of congestion. Adaptive routing for Slim Fly performs better largely because of the smaller hop counts for minimal and nonminimal paths as well as the ability of minimal routing to naturally distribute the workload throughout the network to utilize all available resources.



(a) Neuro Job Performance (in presence of CPU)



(b) Neuro Job Interference (in pres. of CPU) [Mild Cases]



(c) Interference (Neuro in pres. of CPU) [Severe Cases]

Figure 73: The top subfigure presents simulated end time results for neuromorphic applications when running in the presence of CPU applications on the Dragonfly-1D, Dragonfly-1D, Fat-Tree, and Slim Fly HPC systems. End times are normalized within each workload pairing to the slowest performing topology result. The bottom subfigures present the net slowdown in end time performance for neuromorphic workloads when running in the presence of CPU workloads. In all subfigures, lower is better.

The Hopfield neuromorphic workload observes the least amount of slowdown running in the presence of CPU workloads. The average slowdown for Hopfield across all topologies and CPU

workloads is 21% compared to 148% and 220% for CIFAR and MNIST. Additionally, end times are consistently close between the topologies for Hopfield further indicating the all-to-all communication pattern is equally adversarial for all topologies and with distributed communication that isn't readily susceptible to traditional CPU workloads.

Neuromorphic workload performance also varies widely between the tested network topologies. On average, Slim Fly, Fat-Tree, Dragonfly-1D, and Dragonfly-2D are 45%, 46%, 76%, and 83% respectively faster than the worst case performing topology in each hybrid workload configuration. While the Dragonfly configurations many times have the worst-case end times, we do believe the causes (namely link traffic hot spots) can be significantly reduced and possibly even alleviated with approaches to better balance the communication workloads such as minimal path bias, an adaptive threshold, or randomly mapping processes to compute nodes.

4.4.3.9 Topology Performance Summary

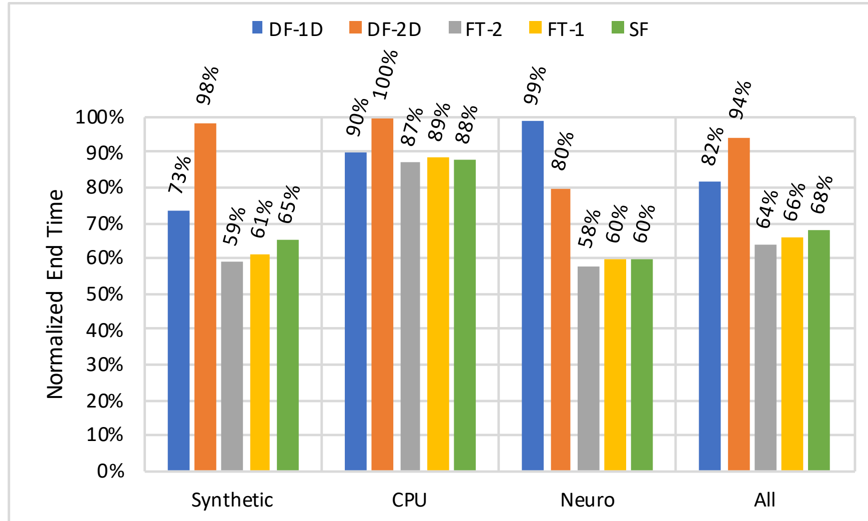
Up to this point, multiple studies have been conducted individually to test and understand run time performance of the Dragonfly, Fat-Tree and Slim Fly topologies under different traffic workloads and run conditions. This section summarizes those results into a set of workload performance scores to provide an overall performance comparison for the different HPC topologies. The workloads taken into consideration are single-job executions of synthetic traffic, CPU trace, and neuromorphic trace workloads. The results are those generated from the 3K node system configurations of Dragonfly-1D, Dragonfly-2D, single-rail and dual-rail Fat-Tree and Slim Fly.

The simulation end time is collected from each execution and normalized with respect to the worst-case performer within each workload. The result is a score for each topology that is comparable within the given workload and on the same scale as the other workloads. The score ranges from 1 to 100 indicating how each topology performed as a percentage of the worst-case topology. Clearly, smaller values are better (indicating the workload finished in only X% of the time required for the worst-case result) with 100 indicating that result is the worst-case for the given workload. For example, if the score is a 78% for the bisection workload, then that topology finished the workload 22% faster than the slowest topology end-time. Each network topology score within a workload is directly comparable.

To generate average scores across multiple workloads, and even workload types, we use the geometric mean to find a single "figure of merit." A geometric mean uses a product of values (instead of the sum of values used in the traditional arithmetic mean) to effectively remove the bias of workloads that have more traffic and take longer to complete on average than other workloads which would place an unfair influence on the workloads with higher end times.

For a fair comparison between the single-rail and dual-rail configurations, synthetic workload results are only considered for offered loads equal to or less than 100% link speed. In that case the synthetic score for each topology is the average normalized end time averaged over the 25%, 50%, 75%, and 100% injection load cases. Again, when we take the "average," we use the geometric mean to remove end time bias.

The summarized performance for each topology is computed individually for synthetic, CPU, and neuromorphic workload categories as well as across all workloads. The results are labeled accordingly as "Synthetic", "CPU", "Neuro" and "All" and are presented in Figure 74. Starting with the summarized results for the synthetic workloads, it's clear Dragonfly-2D is consistently the worst-case performing topology with an average normalized end time close to 100. Of the remaining



(a)

Figure 74: Normalized end time results for each topology averaged over each workload category including synthetic, CPU and neuromorphic workloads. Additionally a fourth summary is provided averaging normalized end time across "all" workloads.

single-rail networks, the Fat-Tree achieves the best performance for synthetic workloads roughly 39% faster than the worst-case topology while Slim Fly and Dragonfly-1D are 35% and 27% faster respectively than worst-case. While the dual-rail configured Fat-Tree network provides twice the network bandwidth and additional path diversity, the added benefits do not translate to substantial performance improvement for the synthetic workloads. On average, the dual-rail Fat-Tree achieves normalized end times are 41% and 42% faster respectively than worst-case. However, it's important to note that, for a fair comparison, the synthetic results only consider message injection rates up to 100% link speed which limits the benefit of doubled bandwidth of the dual-rail networks. The improvements of dual-rail Fat-Tree over the single-rail networks observed here are strictly a result of the increased path diversity. Considering message injection rates above 100% link speed will only add more separation between the dual-rail and single-rail network performance.

The summarized CPU workload results show much less variance between the different network configurations. Dragonfly-2D again sees the lowest normalized end time on average but it's not as far behind the other networks as was the case for the synthetic workloads. In fact, the next closest in end time is the Dragonfly-1D at 10% faster, and the fastest end time is observed by the dual-rail Fat-Tree topology at 13% faster than the worst-case topology. The relatively close set of end time performance results shows the tested CPU workloads, when run as single job executions, do not necessarily contain traffic patterns or other characteristics that are especially adversarial for any one network over the others.

Overall, the neuromorphic workloads presented a traffic pattern that was highly latency sensitive. The densely connected neuron layers communicating via very small 8B messages translated to a workload that was very susceptible to longer path lengths. Both Dragonfly networks attempted to redistribute the dense communication across the network resulting in excessive end-to-end latencies for the small messages and low end time performance. Overall, the end time performance summarized over all neuromorphic workloads show Dragonfly-1D with the lowest performance achieving

Approved for Public Release; Distribution Unlimited.

around 98% of worst-case and Dragonfly-2D at 80% of worst-case end time. Slim Fly and single-rail Fat-Tree provided lower path length routing resulting in improved performance taking only 60% of worst-case end time. The dual-rail Fat-Tree network provides marginal improvement over the single-rail Fat-Tree because the neuromorphic workloads do not have enough substantial traffic to exploit the additional bandwidth and extra path diversity.

Finally, the average performance across all synthetic, CPU, and neuromorphic workloads, reiterates similar topology performance trends observed within each workload category. The two Dragonfly configurations observe end times on average that are at least 20% slower than Fat-Tree and Slim Fly. The Fat-Tree network achieves the best average end time performance among all single-rail configurations, only a few percent faster than the Slim Fly. As noted earlier, the dual-rail Fat-Tree network provides little improvement in average end time, failing to justify the added increase in bandwidth and path diversity provided by the additional rail.

4.4.3.10 Topology Cost Analysis

Performance is typically one part of a larger equation when determining best fit interconnects for an HPC system. Another important part from a deployment perspective is cost. Money spent on interconnecting nodes, translates to less money spent on larger quantities of compute nodes or more computationally powerful compute nodes. In an ideal world, an HPC interconnection network would consist of one layer of all-to-all connections between compute nodes but besides being unfeasible to realize with today's hardware such a system would be cost prohibitive. An attractive HPC interconnect isn't one that simply allocates more hardware resources. It's the HPC interconnect that can achieve the best performance (above some desired lower bound) for a given dollar amount. In this section a cost comparison is provided to show how optimal each configuration is in terms of performance per dollar.

In this study, cost is computed taking into account the total number of links and network switches. Rack hardware, filesystems, and other HPC components are not included in the cost calculation. For simplicity, we assume all links to be copper cables while in practice, vendors typically deploy a mixture of copper and optical cables depending on the given topology and physical length of the cables between nodes in racks. The cost per link is based on the Mellanox MCP1600 copper cable supporting up to Infiniband EDR 100Gb/s link speeds which costs $134.00 [5]. The cost for switches is based off of the Mellanox MSB7800 Infiniband EDR 36 port switch which costs $25,633 [6]. Dragonfly-2D is constructed using 48 port switches but unfortunately, Mellanox does not provide pricing for such a switch with 48 ports and rated for EDR link speeds. Therefore, the cost of a 48 port switch is extrapolated from the cost per port of the Mellanox 36 port switch. The final cost of the 48 port switch comes to $34,177.33.

Table 14 presents the associated costs for the Dragonfly-1D, Dragonfly-2D, single-rail Fat-Tree, dual-rail Fat-Tree and Slim Fly 3K node systems. Within the rows, the lowest value is colored green and the largest value is colored red to highlight the two extremes for the given metric. Overall, the Slim Fly is the network configuration with the lowest total cost resulting from it's low number of routers and links. In comparison, Slim Fly is 17%, 29%, 64% and 66% cheaper to construct than Dragonfly-1D, single-rail Fat-Tree, dual-rail Fat-Tree, and Dragonfly-2D respectively.

Taking it a step further, Table 15 presents a combined performance per dollar metric for each of the six tested networks. The table values describe how much speedup (measured in percent) the given topology achieves over the worst-case performing topology per every $1M spent. The score is

Table 14: Network cost comparison.

Metric	***DF-1D***	***DF-2D***	***FT-1***	***FT-2***	***SF***
Links	8,600	11,424	9,720	19,440	6,253
Routers	400	768	468	936	338
Router Radix	36	48	36	36	36
$ Per Link	$134	$134	$134	$134	$134
$ Per Router	$25,633	$34,177	$25,633	$25,633	$25,633
Link Total $	$1,152K	$1,531K	$1,302K	$2,605K	$838K
Router Total $	$10.3M	$26.2M	$12.0M	$24.0M	$8.7M
Total $	$11.4M	$27.8M	$13.3M	$26.6M	$9.5M

Table 15: Speedup per $1M.

Network	***Synthetic***	***CPU***	***Neuro***	***All***
DF-1D	2.3	0.9	0.1	1.6
DF-2D	0.1	0.0	0.7	0.2
FT-1	2.9	0.9	3.0	2.6
FT-2	1.5	0.5	1.5	1.3
SF	3.7	1.3	4.2	3.3

computed by first taking the performance scores provided in the performance summary Figure 74 and subtracting them from 100 to get the speedup achieved by each topology over the worst-case performing topology. Then the calculated speedup value is divided by the total cost of the network to arrive at a speedup per $1M value. The "Synthetic", "CPU", "Neuro", and "All" columns indicate the performance per dollar calculation is computed with respect to tested synthetic, CPU, neuro, or all workload results. The Slim Fly, for example, achieves the highest value of 3.3 for the category of "All" workloads indicating that the Slim Fly is 3.3% faster (on average across the tested synthetic, CPU, and neuro workloads) than the worst case topology for every $1M spent. Dragonfly-2D, with the lowest value of 0.2 indicates it was many times the worst-case performing topology across all the workloads and only achieves 0.2% speedup over the worst-case performing topology per every $1M spent.

Overall, The Slim Fly topology proves to have the best performance per dollar across all workloads. The topology with the closest performance per dollar result is the single-rail Fat-Tree which many times achieves faster end times than Slim Fly but the performance difference is not enough to offset the 30% extra cost over the Slim Fly configuration.

This performance per dollar calculation provides a good single value measure important for deciding between available network configurations. It's important to note this is by no means an end-all value. Other network qualities not accounted for but also important to consider for next generation systems are the ability to scale to larger node counts and the resiliency to hardware failures. Each of these network characteristics can effect the preferred topology selection.

4.5 Classification of AFRL Data Using IBM TrueNorth & *NeMo*

The first test involved the accuracy of the *NeMo* importing system. In this test, we took the EEDN trained network and imported it into *NeMo*. For each neuron in the simulation, we saved it's

Table 16: MNIST Data Set: *NeMo* vs. NSCS Spike Counts .

	NeMo Spikes	NSCS Spikes	Spikes Not in *NeMo*	Spikes Not in NSCS
MNIST	564,532	490,790	0	73,742
MNIST - Filtered	490,790	490,790	0	0

connected components to a file. We then ran the NSCS simulation, and output all neuron activity. From that file, we took all active connections between neurons, and produced an edge connection table.

Comparing these two files shows that the *NeMo* importer was able to extract the model information from NSCS. Reviewing these model files revealed that there were some inconsistencies in *NeMo* compared to the NSCS spike record file. We found that there were cores included in the configuration file that had connections to many cores in the model which *NeMo* read in, that the NSCS spike records did not represent. For example, *NeMo* found a neurosynaptic core with ID -1, that was at the center of 512 different cores. In NSCS, this core was not active. Further examination revealed that these cores were defined in the EEDN model file, but were not used by the NSCS simulation. These cores are used by the TrueNorth system as a monitoring layer, providing links for debugging and visualization when running the model. Given that the cores do not connect to other cores (they are strictly sinks in the network), they do not affect the output of the simulation in either *NeMo* or NSCS, save for counting the number of spikes produced by the simulation. Further work is warranted, however, as we should completely rule out any effects these cores may have on the underlying simulation.

Next, we looked at the full spike data from each of these runs. For these runs, we generated a list of spikes that took place during a run through the MNIST and AFRL networks. We then produced a table that shows the total number of missing spikes and extra spikes produced by *NeMo*. Here, we noticed a few interesting things. First, in the MNIST run, *NeMo* reproduced all NSCS reported spikes. However, *NeMo* also produced extra spike communications. Based on the examination of *NeMo*'s parsing of the EEDN data, we realized that *NeMo* was generating spikes for the previously mentioned monitoring cores. To resolve this, we filtered spikes with a destination core matching those of the monitoring cores. Once done, *NeMo*'s spike output matched the NSCS simulation's spike data. Further investigation is still warranted however, as these spikes could represent some inconsistency in the *NeMo* simulation. The aggregate results of these spikes are shown in table 16.

We then took a look at a TrueNorth model trained using AFRL supplied MSTAR data. To produce these comparison results, we looked at the output layers of the network, and counted spikes going to each output over the simulation time. This allowed us to compare classification accuracy between the *NeMo* simulator and the NSCS simulator. The results of this comparison are shown in table 17. Comparing the output spikes with the NeMo output showed a perfect match. Further work is needed, as a detailed comparison with the TrueNorth hardware would provide greater details on *NeMo*'s performance as a neurosynaptic hardware simulation model.

We then took a look at a TrueNorth model trained using AFRL supplied data. To produce these comparison results, we looked at the output layers of the network, and counted spikes going to each output over the simulation time. This allowed us to compare classification accuracy between the *NeMo* simulator and the NSCS simulator. The results of this comparison are shown in table 17. Comparing the output spikes with the NeMo output showed a perfect match. Further work is needed, as a detailed comparison with the TrueNorth hardware would provide greater details on

Approved for Public Release; Distribution Unlimited.

Table 17: AFRL MSTAR Data Set: *NeMo* vs. NSCS neuromorphic Output Core Spike Counts.

Output Core	NSCS Spikes	NeMo Spikes
-63	3812	3812
-62	4604	4604
-61	4608	4608
-60	4604	4604
-59	4176	4176
-58	4608	4608
-57	4608	4608
-56	4608	4608
-55	4320	4320
-54	4608	4608
-53	4608	4608
-52	4608	4608
-51	4320	4320
-50	4608	4608
-49	4608	4608
-48	4604	4604
-47	4176	4176
-46	4608	4608
-45	4608	4608
-44	4608	4608
-43	4316	4316
-42	4604	4604
-41	4608	4608
-40	4608	4608
-39	4320	4320
-38	4608	4608
-37	4608	4608
-36	4608	4608
-35	4464	4464
-34	4608	4608
-33	4608	4608
-32	4608	4608

Approved for Public Release; Distribution Unlimited.

NeMo's performance as a neurosynaptic hardware simulation model.

5 CONCLUSIONS

5.1 NeMo: A Massively Parallel Neuromorphic Simulator

The main contributions of this core research thrust (CRT) are:

1. ***NeMo*: A PDES neuromorphic hardware simulation model:** Development of a massively parallel discrete event simulation based neuromorphic hardware modeling tool. This simulator allows for massive scale simulations of both existing and novel neuromorphic hardware. Shown to have neuron model level accuracy, the simulator can be used to explore the hardware design space of neuromorphic processor designs. The model is shown to be extremely powerful; scaling up to billions of neurons while simulating even larger numbers of spike events.

2. ***NeMo-SS*: An improved neuromorphic hardware simulation model:** Further work was done in an attempt to optimize the original *NeMo* neuromorphic simulation model. While the first design showed promising results, the underlying model was memory bound. In this section, work done to optimize the simulation model is discussed, along with new performance benchmarks and a comparison of results with the original *NeMo* design.

3. **Neuromorphic hardware simulation integration with CODES** In order to explore the effects of neuromorphic hardware on next-generation HPC systems, we explored ways to simulate a hybrid HPC system. This hypothetical system would contain neuromorphic hardware working as an accelerator processor alongside traditional CPUs within each node of the system. Here, we present work that allows *NeMo* to expand a neuromorphic simulation from a single processor to multiple "virtual" processors. We added further features NeMo that allows for the capture of neuromorphic network activity. Using these new features, we developed a toolchain that allows NeMo to generate neuromorphic hardware network traffic traces that can integrate into the CODES HPC network simulation tool. We then discuss the techniques used to create these virtual network traces as well as some avenues for future work and validation.

5.2 Classification of Supercomputer Failures Using TrueNorth

In this CRT, We set out to explore the possibility of using a neuromorphic computing approach to classify node failures on supercomputers. Furthermore, we compared this approach to five other machine learning and deep learning approaches. We demonstrated that the neuromorphic computing approach outperforms all the other machine learning and deep learning approaches for our application. Moreover, we also showed that all the techniques used in this work do a good job of classifying node failures. We used the IBM Blue Gene/L (BG/L) dataset, which consisted of RAS logs spanning over a year of the machine's operational lifetime. We used Python (TensorFlow and Scikit-Learn) and MATLAB (IBM's EEDN framework) to run our ML and DL techniques.

The bigger picture that we are trying to paint is to design the next generation of supercomputers that will have a Neuromorphic Processing Unit (NPU) on every node of the supercomputer. In this

setting, our paper sheds light on classifying node-level failures on such supercomputers. As part of our future work, we would like to put the neuromorphic computing approach in a live setting where failure data is read as a live stream of data. Another aspect that we would like to incorporate is to use more complex SNN architectures running on TrueNorth to see if they deliver better results.

Also, in this paper, we have established that node failures can be classified using ML/DL techniques. We would also like to get an estimate of how far into the future can we *predict* these failures. Furthermore, KNN ($K = 3$) seems to be the best machine learning technique for the task solely from an accuracy stand point but it suffers from some inherent drawbacks. It would be interesting to see how some of the variants of KNN that tackle its inherent drawbacks compare to the other techniques used in the paper. Moreover, in line with designing next generation neuromorphic supercomputers, we would like to explore other applications that could potentially be running on the NPUs.

Through the results in this core research thrust, we have successfully demonstrated that node failures on supercomputers can be modeled and classified using machine learning and deep learning techniques, some of which can also be deployed on neuromorphic hardware.

5.3 *Durango* – A Hybrid System Performance Modeling Framework

In this core research thrust (CRT), we introduce a new performance analysis tool called *Durango* that integrates the analytical performance modeling capabilities of the Aspen domain specific language with the efficient, massively parallel network simulation capabilities of CODES. Aspen has been extended to enable communication pattern specification. The efficacy of Durango is demonstrated as a new approach to the performance modeling of extreme-scale systems in two ways:

- Comparing the Aspen generated communication patterns with real application network communications via traces that are run through the CODES packet-level network simulation framework. Durango shows strong agreement with the real application trace data for key network performance statistics.

- Performing a scaling study of Durango's direct integration approach that links Aspen with CODES as part of the running network simulation model. Here, Aspen generates the application-level computation timing events, which in turn drive the start of a network communication phase. Results show that Durango's performance scales well when executing both torus and dragonfly network models on up to 4K Blue Gene/Q nodes using 32K MPI ranks.

We plan to extend Durango's capabilities by enabling Aspen to drive both the compute kernel timing and the network communication patterns for key supercomputing applications. This extension will enable end-to-end performance prediction capabilities for current and future extreme-scale systems.

5.4 HPC Network Models

Understanding methods to improve network performance for existing and new network topologies is important to the effectiveness of future HPC systems. Focusing on understanding and quantifying HPC network performance, in this thesis, we extended the CODES system simulation framework to support Slim Fly and multi-rail pruned fat-tree networks as well as replay novel neuromorphic

computing application workloads. The Slim Fly and Fat-Tree additions have been validated with published results and the Fat-Tree model has been additionally validated with a real hardware system. We presented simulation results: (1) benchmarking the discrete-event simulation performance at scale (2) comparing the performance benefits of additional rails in the Fat-Tree network, (3) studying the effect of routing on CPU application end times in large-scale Slim Fly Networks, and (4) testing equally provisioned Dragonfly, Fat-Tree and Slim Fly networks under synthetically generated workloads as well as real CPU and novel neuromorphic computing application trace workloads to provide a fair comparison of expected performance.

In the Fat-Tree multi-rail investigation, we found observed network bandwidth performance across all studies shows high dependency on the communication pattern. We have found that applications, such as Multigrid with communication patterns that inject large quantities of small messages into the network at high rates, yield an increase of up to 7.3× in observed bandwidth when going from one to eight rails. In a fair comparison study, we showed that a dual-rail network matches and even exceeds observed bandwidth by up to 17% over a similar single-rail network with twice the link speed. In the job allocation study, we demonstrated strong application independent performance from the cluster policy and best overall speedup of 1.6× for Multigrid using the contiguous policy. Finally, we observed a general decreasing trend in application network performance on the dual-rail fat-tree system in response to increasing numbers of trace processes mapped to nodes. However, our results also suggest that positive performance is achievable when increasing the number of processes per node results in applications that can be isolated in the system, resulting in lower link traversal counts and lower network interference.

The CODES-ROSS discrete-event simulation framework was tested and found to provide improvement in run times under balanced workloads. Using the Slim Fly model, we have shown linear scaling in execution up to 128 MPI ranks on the CCI RSA Intel cluster, achieving a peak event rate of 43 million events per second with 543 million total events processed for the 74K-node slim fly model. The million-node model achieves 36 million events per second processing 7 billion events.

Evaluation studies were presented to compare slim fly network response to real communication workloads from Crystal Router and Multigrid applications using minimal, non-minimal, and adaptive routing algorithms. Overall, non-minimal achieves the best observed bandwidth for both applications. While minimal routing significantly lowers the average hop count when compared to minimal and even adaptive, it results in a consolidation of network traffic through specific paths in the slim fly network resulting in higher packet latencies. In terms of application end times, minimal routing provides results only marginally slower than non-minimal and adaptive routing making the argument that minimal routing is sufficient and the excess system utilization of non-minimal routing and complex implementation of adaptive routing can be avoided.

Finally, new simulation workflow has been presented allowing exploration of real, extreme-scale neuromorphic workloads and their interference with traditional CPU workloads when executed on a hybrid HPC system. We studied single-job and multi-job executions analyzing performance of systems configured with Dragonfly, Fat-Tree and Slim Fly network topologies.

Neuromorphic workloads representing convolutional neural network and Hopfield network applications pose little effect on traditional CPU applications when running in parallel in a multi-job hybrid HPC environment. In the worst-case, the Crystal Router CPU workload observed 16% slower end time performance. Furthermore, performance for the Dragonfly-1D, Dragonfly-2D, Fat-Tree, and Slim Fly topologies matched the trends of the single-job CPU end times with Fat-Tree and Slim Fly finishing faster than the Dragonfly configurations with some instances where Dragonfly-1D

matched Fat-Tree and Slim Fly's performance.

The neuromorphic workloads, on the other hand, are largely susceptible to the network workloads generated by the traditional CPU applications. Convolutional NN workloads MNIST and CIFAR have been shown to observe up to 10x slowdown in end time running in parallel with CPU workloads. Slim Fly and Fat-Tree are roughly 40% faster on average than Dragonfly-1D and Dragonfly-2D for neuromorphic workloads in the presence of CPU workloads. However the Dragonfly configurations may still achieve the performance of Fat-Tree and Slim Fly with approaches such as minimal path bias, adaptive thresholds, or randomly mapping processes to compute nodes.

Future directions for this work include investigating performance of larger network configurations of around 60,000 nodes, closer to the expected node counts of anticipated Argonne National Laboratory Aurora exascale machine. Analysis of the exascale machine could also be performed using a more diverse set of applications of varying heterogeneity such as combined CPU-GPU workloads as well as multi-job workloads consisting of tens of applications running in parallel to study overall system congestion and utilization under traditional HPC run time and scheduling environments. From the neuromorphic perspective, applications workloads need to be expanded to cover a more realistic large-scale neuromorphic application. The current approach scales a real application with a small number of inputs and outputs. A realistic large-scale neuromorphic application will conceivably have a much larger input size taking in one large input as apposed to many replicated small inputs.

5.5 Classification of AFRL Data Using TrueNorth & *NeMo*

For both the MNIST benchmark and MSTAR datasets, *NeMo* demonstrated a very high degree of per spike accuracy when compared with the IBM NSCS simulator. In the case of MSTAR, we observed no differences inference accuracy or in total spike counts from the output layer neuromorphic cores.

References

[1] Alcf aurora 2021 early science program: Data and learning call for proposals. (Accessed on: Nov. 3, 2018).

[2] Cori computational system. (Accessed on: Nov. 3, 2018).

[3] Edison computational system. (Accessed on: Nov. 3, 2018).

[4] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[5] Mellanox mcp1600-e003 passive copper cable. (Accessed on: Nov. 3, 2018).

[6] Mellanox msb7800-es2f switch-ib 2. (Accessed on: Nov. 3, 2018).

[7] Mpptest - measuring mpi performance. (Accessed on: Nov. 3, 2018).

[8] Theta computational system. (Accessed on: Nov. 3, 2018).

Approved for Public Release; Distribution Unlimited.

[9] Mellanox OFED for Linux User Manual Rev. 2.0-3.0.0, August 2013. (Accessed on: Nov. 3, 2018).

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

[11] Jemal H Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 238. IEEE, 2004.

[12] Yaser S Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from data*, volume 4. AMLBook New York, NY, USA:, 2012.

[13] Bilge Acun et al. Preliminary evaluation of a parallel trace replay tool for hpc network simulations. In Sascha Hunold, Alexandru Costan, Domingo Gimnez, Alexandru Iosup, Laura Ricci, Mara Engracia Gmez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander, editors, *Euro-Par 2015: Parallel Processing Workshops*, volume 9523 of *Lecture Notes in Computer Science*, pages 417–429. Springer Int. Publishing, 2015.

[14] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. Hyperx: Topology, routing, and packaging of efficient large-scale networks. In *Proc. of the Conf. on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 41:1–41:11, New York, NY, USA, 2009. ACM.

[15] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, Oct 2015.

[16] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, 2015.

[17] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proc. of the ACM SIGCOMM 2008 Conf. on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

Approved for Public Release; Distribution Unlimited.

[18] Mahmoud Al-Nsour and Hoda S Abdel-Aty-Zohdy. Implementation of programmable digital sigmoid function circuit for neuro-computing. In *mwscas*, page 571. IEEE, 1998.

[19] S. R. Alam et al. Cray xt4: an early evaluation for petascale scientific simulation. In *SC '07: Proc. of the 2007 ACM/IEEE Conf. on Supercomputing*, pages 1–12, Nov 2007.

[20] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In *Proc. of the Seventh Annu. ACM Symp. on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[21] A. Amir, P. Datta, W. P. Risk, A. S. Cassidy, J. A. Kusnitz, S. K. Esser, A. Andreopoulos, T. M. Wong, M. Flickner, R. Alvarez-Icaza, E. McQuinn, B. Shaw, N. Pass, and D. S. Modha. Cognitive computing programming paradigm: A corelet language for composing networks of neurosynaptic cores. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, Aug 2013.

[22] S. Azizian, K. Fathi, B. Mashoufi, and F. Derogarian. Implementation of a programmable neuron in 0.35m cmos process for multi-layer ann applications. In *2011 IEEE EUROCON - International Conference on Computer as a Tool*, pages 1–4, April 2011.

[23] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative Study of Caffe, Neon, Theano, and Torch for Deep Learning. *CoRR abs/1511.06435*, 2015.

[24] P. D. Barnes, C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: executing time warp on 1,966,080 cores. In *Proc. of the 2013 ACM SIGSIM Conf. on Principles of Advanced Discrete Simulation (PADS)*, pages 327–336, May 2013.

[25] Peter D. Barnes, Jr., Christopher D. Carothers, David R. Jefferson, and Justin M. LaPre. Warp speed: Executing time warp on 1,966,080 cores. In *Proc. of the 1st ACM SIGSIM Conf. on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, pages 327–336, New York, NY, USA, 2013. ACM.

[26] K Basterretxea, JM Tarela, and I Del Campo. Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons. *IEE Proceedings-Circuits, Devices and Systems*, 151(1):18–24, 2004.

[27] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 32. ACM, 2011.

[28] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[29] B. V. Benjamin et al. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proc. of the IEEE*, 102(5):699–716, May 2014.

Approved for Public Release; Distribution Unlimited.

[30] M. Besta and T. Hoefler. Slim fly: A cost effective low-diameter network topology. In *Proc. of the Int. Conf. for High Performance Comput., Networking, Storage and Anal. (SC)*, pages 348–359, 2014.

[31] M. Besta and T. Hoefler. Slim Fly: A Cost Effective Low-Diameter Network Topology. Nov. 2014. Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC14).

[32] Maciej Besta and Torsten Hoefler. Slim fly: A cost effective low-diameter network topology. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 348–359, Piscataway, NJ, USA, 2014. IEEE Press.

[33] Julian A Bragg, Edgar A Brown, Paul Hasler, and Stephen P DeWeerth. A silicon model of an adapting motoneuron. In *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, volume 4, pages IV–IV. IEEE, 2002.

[34] Aaron Brown and David A Patterson. Embracing failure: A case for recovery-oriented computing (roc). In *High Performance Transaction Processing Symposium*, volume 10, pages 3–8, 2001.

[35] Christopher D. Carothers, David Bauer, and Shawn Pearce. Ross: A high-performance, low memory, modular time warp system. In *Proc. of the Fourteenth Workshop on Parallel and Distributed Simulation*, PADS '00, pages 53–60, Washington, DC, USA, 2000. IEEE Computer Society.

[36] Christopher D Carothers, David Bauer, and Shawn Pearce. Ross: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[37] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, July 1999.

[38] S Carrillo et al. Hierarchical Network-on-Chip and Traffic Compression for Spiking Neural Network Implementations. In *2012 Sixth IEEE/ACM Int. Symp. on Networks-on-Chip (NoCS)*, pages 83–90. IEEE, 2012.

[39] Snaider Carrillo et al. Scalable Hierarchical Network-on-Chip Architecture for Spiking Neural Network Hardware Implementations. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2451–2461, 2013.

[40] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, B. Jackson, R. Alvarez-Icaza, P. Datta, J. Sawada, T. M. Wong, V. Feldman, A. Amir, D. B. D. Rubin, F. Akopyan, E. McQuinn, W. P. Risk, and D. S. Modha. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, Aug 2013.

Approved for Public Release; Distribution Unlimited.

[41] Andrew S Cassidy, Rodrigo Alvarez-Icaza, and Others. Real-time Scalable Cortical Computing at 46 Giga-synaptic OPS/Watt with 100x Speedup in Time-to-solution and 100,000x Reduction in Energy-to-solution. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 27–38, Piscataway, NJ, USA, 2014. IEEE Press.

[42] Andrew S Cassidy, Paul Merolla, John V Arthur, Steve K Esser, Bryan Jackson, Rodrigo Alvarez-Icaza, Pallab Datta, Jun Sawada, Theodore M Wong, Vitaly Feldman, et al. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–10. IEEE, 2013.

[43] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The ibm blue gene/q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM.

[44] Dong Chen, Philip Heidelberger, Craig Stunkel, and Yutaka Sugawara. An Evaluation of Network Architectures for Next Generation Supercomputers. In *Proc. of the 7th Int. Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '16, pages 2:11–2:21, New York, NY, USA, 2016. ACM.

[45] Zizhong Chen, Graham E Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 213–223. ACM, 2005.

[46] George Chrysos. Intel® Xeon Phi™ coprocessor-the architecture. *Intel Whitepaper*, 2014.

[47] *Co-design at Lawrence Livermore National Laboratory. Algebraic Multigrid Solver (AMG).* (Accessed on: Nov. 3, 2018).

[48] Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, and Leonid Gurvits. Using multirail networks in high-performance clusters. *Concurrency and Computation: Practice and Experience*, 15(7-8):625–651, 2003.

[49] W. J. Dally and B. P. Towles. *Principles and Practices of Interconnection Networks.* Burlington, MA, USA: Morgan Kaufmann, January 2004.

[50] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[51] W.J. Dally. Virtual-channel flow control. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):194–205, Mar 1992.

[52] Prasanna Date, James A Hendler, and Christopher D Carothers. Design index for deep neural networks. *Procedia Computer Science*, 88:131–138, 2016.

Approved for Public Release; Distribution Unlimited.

[53] M. Davies et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, Jan. 2018.

[54] Department of Energy. *AMR Box Lib*. (Accessed on: Nov. 3, 2018).

[55] Department of Energy. *Design Forward - Exascale Initiative*. (Accessed on: Nov. 3, 2018).

[56] Jens Domke, Torsten Hoefler, and Satoshi Matsuoka. Fail-in-place Network Design: Interaction Between Topology, Routing Algorithm and Failures. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 597–608, Piscataway, NJ, USA, 2014. IEEE Press.

[57] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[58] Bard Ermentrout. Neural networks as spatio-temporal pattern-forming systems. *Reports on progress in physics*, 61(4):353, 1998.

[59] S K Esser, A Andreopoulos, and Others. Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores. In *The 2013 International Joint Conference on Neural Networks*, pages 1–10. IEEE, August 2013.

[60] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S Modha. Backpropagation for energy-efficient neuromorphic computing. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1117–1125. Curran Associates, Inc., 2015.

[61] Steven K Esser, Paul A Merolla, John V Arthur, Andrew S Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J Berg, Jeffrey L McKinstry, Timothy Melano, Davis R Barch, et al. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences*, page 201604850, 2016.

[62] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. Cray cascade: A scalable hpc system based on a dragonfly network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 103:1–103:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[63] Graham E Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J Dongarra. Process fault tolerance: Semantics, design and applications for high performance computing. *The International Journal of High Performance Computing Applications*, 19(4):465–477, 2005.

[64] F. Ferrari. System-on-a-chip verification methodology and techniques. *IEEE Circuits and Devices Magazine*, 18(6):39–39, Nov 2002.

Approved for Public Release; Distribution Unlimited.

[65] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 78. IEEE Computer Society Press, 2012.

[66] Jose Flich et al. A Survey and Evaluation of Topology-Agnostic Deterministic Routing Algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 23(3):405–425, March 2012.

[67] Richard M Fujimoto and Maria Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 7(4):425–446, October 1997.

[68] I. Fujiwara, M. Koibuchi, H. Matsutani, and H. Casanova. Skywalk: A topology for hpc networks with low-delay switches. In *2014 IEEE 28th Int. Parallel and Distributed Processing Symp.*, pages 263–272, May 2014.

[69] Ana Gainaru and Franck Cappello. Errors and faults. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 89–144. Springer, 2015.

[70] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[71] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity.* Cambridge university press, 2002.

[72] SAMANWOY GHOSH-DASTIDAR and HOJJAT ADELI. SPIKING NEURAL NETWORKS. *International Journal of Neural Systems*, 19(04):295–308, August 2009.

[73] M Grattarola, M Bove, S Martinoia, and G Massobrio. Silicon neuron simulation with spice: tool for neurobiology and neural networks. *Medical and Biological Engineering and Computing*, 33(4):533–536, 1995.

[74] William Gropp and Ewing Lusk. Reproducible measurements of mpi performance characteristics. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[75] J. P. Grossman, B. Towles, J. A. Bank, and D. E. Shaw. The role of cascade, a cycle-based simulation infrastructure, in designing the anton special-purpose supercomputers. In *2013 50th ACM/EDAC/IEEE Design Automation Conf. (DAC)*, pages 1–9, May 2013.

[76] T. Groves et al. (sai) stalled, active and idle: Characterizing power and performance of large-scale dragonfly networks. In *2016 IEEE Int. Conf. on Cluster Computing (CLUSTER)*, pages 50–59, Sept 2016.

[77] Thomas J Hacker, Fabian Romero, and Christopher D Carothers. An analysis of clustered failures on large supercomputing systems. *Journal of Parallel and Distributed Computing*, 69(7):652–665, 2009.

[78] Paul R. Hafner. Geometric realisation of the graphs of mckay-miller-siran. *Journal of Combinatorial Theory, Series B*, 90(2):223–232, 2004.

[79] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, March 2012.

[80] Jennifer Hasler and Bo Marr. Finding a roadmap to achieve large neuromorphic hardware systems. *Frontiers in neuroscience*, 7, 2013.

[81] Jennifer Hasler and Harry Bo Marr. Finding a roadmap to achieve large neuromorphic hardware systems. *Frontiers in Neuroscience*, 7:118, 2013.

[82] T. Hatazaki. Tsubame-2 - a 2.4 PFLOPS peak performance system. In *2011 Optical Fiber Communication Conf. and Exhibition/National Fiber Optic Engineers Conf. (OFC/NFOEC)*, pages 1–3, Mar. 2011.

[83] Almoatazbellah M Hegab, Noha M Salem, Ahmed G Radwan, and Leon Chua. Neuron model with simplified memristive ionic channels. *International Journal of Bifurcation and Chaos*, 25(06):1530017, 2015.

[84] Van Emden Henson and Ulrike Meier Yang. Boomeramg: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155 – 177, 2002. Developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss.

[85] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc. of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[86] Eric Hunsberger and Chris Eliasmith. Spiking deep networks with lif neurons, 2015.

[87] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, Sept 2004.

[88] E M Izhikevich. Hybrid spiking models. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 368(1930):5061–5070, October 2010.

[89] Eugene M. Izhikevich. Resonate-and-fire neurons. *Neural Networks*, 14(6–7):883 – 894, 2001.

[90] Nikhil Jain, Abhinav Bhatele, Sam White, Todd Gamblin, and Laxmikant V. Kale. Evaluating HPC Networks via Simulation of Parallel Workloads. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '16 (to appear), 2016.

[91] Nikhil Jain et al. Predicting the performance impact of different fat-tree configurations. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 50:1–50:13, New York, NY, USA, 2017. ACM.

[92] S Jeyanthi and M Subadra. Implementation of single neuron using various activation functions with fpga. In *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on*, pages 1126–1131. IEEE, 2014.

Approved for Public Release; Distribution Unlimited.

[93] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[94] Nikola K Kasabov. Neucube: A spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data. *Neural Networks*, 52:62–76, 2014.

[95] G. Kathareios, C. Minkenberg, B. Prisacari, G. Rodriguez, and T. Hoefler. Cost-effective diameter-two topologies: Analysis and evaluation. Nov. 2015. Accepted at IEEE/ACM Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC15).

[96] Rafid Ahmed Khalil et al. Fpga implementation of artificial neurons: Comparison study. In *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pages 1–6. IEEE, 2008.

[97] M. M. Khan et al. Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *2008 IEEE Int. Joint Conf. on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 2849–2856, June 2008.

[98] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 77–88. IEEE Computer Society, 2008.

[99] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *SIGARCH Comput. Archit. News*, 36(3):77–88, June 2008.

[100] M. Koibuchi, H. Matsutani, H. Amano, D. F. Hsu, and H. Casanova. A case for random shortcut topologies for hpc interconnects. In *2012 39th Annu. Int. Symp. on Computer Architecture (ISCA)*, pages 177–188, June 2012.

[101] Stefanos Kollias and Andreas Stafylopatis. In Ioannis Pitas, editor, *Parallel Algorithms*, chapter Parallel Implementations of the Backpropagation Learning Algorithm Based on Network Topology, pages 233–258. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[102] Sotiris B Kotsiantis, Ioannis D Zaharakis, and Panayiotis E Pintelas. Machine learning: a review of classification and combining techniques. *Artificial Intelligence Review*, 26(3):159–190, 2006.

[103] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[104] N. T. Kung and R. Morris. Credit-based flow control for atm networks. *IEEE Network*, 9(2):40–48, Mar. 1995.

[105] Brett Lantz. *Machine learning with R.* Packt Publishing Ltd, 2013.

[106] Justin M LaPre, Christopher D Carothers, Kenneth D Renard, and Dale R Shires. Ultra large-scale wireless network models using massively parallel discrete-event simulation. *Transactions of The Society for Modeling and Simulation Int.*, October 2012.

[107] Lawrence Livermore National Laboratory. *Sierra Advanced Technology System*. `http://computation.llnl.gov/computers/sierra-advanced-technology-system`. (Accessed on: Nov. 3, 2018).

[108] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[109] Seyong Lee, Jeremy S. Meredith, and Jeffrey S. Vetter. COMPASS: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 405–414, Newport Beach, California, USA, 2015. ACM.

[110] Seyong Lee and Jeffrey S Vetter. OpenARC: extensible openACC compiler framework for directive-based accelerator programming study. In *Proceedings of the First Workshop on Accelerator Programming using Directives (with SC14)*, pages 1–11, New Orleans, 2014. IEEE Press.

[111] Charles E. Leiserson et al. The network architecture of the Connection Machine CM-5. In *SPAA '92: Proc. of the 4th Annu. ACM Symp. on Parallel Algorithms and Architectures*, pages 272–285, New York, NY, USA, 1992. ACM.

[112] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. Failure prediction in ibm bluegene/l event logs. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 583–588. IEEE, 2007.

[113] Ning Liu, Adnan Haider, Xian-He Sun, and Dong Jin. Fattreesim: Modeling large-scale fat-tree networks for hpc systems and data centers using parallel and discrete event simulation. In *Proc. of the 3rd ACM SIGSIM Conf. on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, pages 199–210, New York, NY, USA, 2015. ACM.

[114] C. J. Lobb, Z. Chao, R. M. Fujimoto, and S. M. Potter. Parallel event-driven neural network simulations using the hodgkin-huxley neuron model. In *Workshop on Principles of Advanced and Distributed Simulation*, PADS 2005, pages 16–25, June 2005.

[115] Qingyun Ma, Mohammad Rafiqul Haider, Vinaya Lal Shrestha, and Yehia Massoud. Bursting hodgkin–huxley model-based ultra-low-power neuromimetic silicon neuron. *Analog Integrated Circuits and Signal Processing*, 73(1):329–337, 2012.

[116] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[117] Brad McCredie. Openpower and the roadmap ahead, 2016. (Accessed on: Nov. 3, 2018).

[118] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[119] Brendan D McKay, Mirka Miller, and Jozef Siran. A note on large graphs of diameter two and given maximum degree. *Journal of Combinatorial Theory, Series B*, 74(1):110 – 118, 1998.

Approved for Public Release; Distribution Unlimited.

[120] K. Meier. A mixed-signal universal neuromorphic computing system. In *2015 IEEE Int. Electron Devices Meeting (IEDM)*, pages 4.6.1–4.6.4, Dec 2015.

[121] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *IEEE Conference on Custom Integrated Circuits*, CICC '11, pages 1–4, Sept 2011.

[122] Paul A. Merolla et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[123] Miller, Mirka, Siran, and Jozef. Moore graphs and beyond: a survey of the degree/diameter problem. *The Electronic Journal of Combinatorics [electronic only]*, DS14:61 p., electronic only–61 p., electronic only, 2005.

[124] Cyriel Minkenberg and Germán Rodriguez. Trace-driven co-simulation of high-performance computing systems using omnet++. In *Proc. of the 2Nd Int. Conf. on Simulation Tools and Techniques*, Simutools '09, pages 65:1–65:8, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[125] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. In *High Performance Comput., Networking, Storage and Anal. (SCC) SC Companion*, pages 366–376, 2012.

[126] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. A case study in using massively parallel simulation for extreme-scale torus network codesign. In *Proc. of the 2nd ACM SIGSIM/-PADS Conf. on Principles of Advanced Discrete Simulation*, pages 27–38, 2014.

[127] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. Enabling parallel simulation of large-scale hpc network systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):87–100, Jan 2017.

[128] Misbah Mubarak, Christopher D. Carothers, Robert Ross, and Philip Carns. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. In *Proc. of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 366–376, Washington, DC, USA, 2012. IEEE Computer Society.

[129] Misbah Mubarak, Christopher D. Carothers, Robert B. Ross, and Philip Carns. A case study in using massively parallel simulation for extreme-scale torus network codesign. In *Proc. of the 2Nd ACM SIGSIM Conf. on Principles of Advanced Discrete Simulation*, SIGSIM PADS '14, pages 27–38, New York, NY, USA, 2014. ACM.

[130] Misbah Mubarak, Christopher D. Carothers, Robert B. Ross, and Philip Carns. Enabling Parallel Simulation of Large-Scale HPC Network Systems. In *IEEE Transactions on Parallel and Distributed Systems*. IEEE, 2016.

[131] Misbah Mubarak et al. Quantifying i/o and communication traffic interference on dragonfly networks equipped with burst buffers. In *Cluster Computing (CLUSTER), 2017 IEEE Int. Conf. on*, pages 204–215. IEEE, 2017.

[132] David M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM*, 40(2):304–333, April 1993.

[133] Oak Ridge National Laboratory. *Summit, Oak Ridge's next High Performance Supercomputer*. `https://www.olcf.ornl.gov/summit/`. (Accessed on: Nov. 3, 2018).

[134] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber. Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, 48(8):1943–1953, Aug 2013.

[135] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[136] F. Petrini and M. Vanneschi. k-ary n-trees: high performance networks for massively parallel architectures. In *Proc. of the 11th Int. Parallel Processing Symp.*, pages 87–93, April 1997.

[137] Fabrizio Petrini and Marco Vanneschi. k-ary n-trees: High performance networks for massively parallel architectures. In *Parallel Processing Symp., 1997. Proc.., 11th Int.*, pages 87–93. IEEE, 1997.

[138] Mark Plagge, Christopher D. Carothers, and Elsa Gonsiorowski. Nemo: A massively parallel discrete-event simulation model for neuromorphic architectures. In *Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*, SIGSIM-PADS '16, pages 233–244, New York, NY, USA, 2016. ACM.

[139] Mark Plagge, Christopher D. Carothers, and Elsa Gonsiorowski. Nemo: A massively parallel discrete-event simulation model for neuromorphic architectures. In *Proc. of the 2016 Annu. ACM Conf. on SIGSIM Principles of Advanced Discrete Simulation*, SIGSIM-PADS '16, pages 233–244, New York, NY, USA, 2016. ACM.

[140] Robert Preissl, Theodore M. Wong, Pallab Datta, Myron Flickner, Raghavendra Singh, Steven K. Esser, William P. Risk, Horst D. Simon, and Dharmendra S. Modha. Compass: A scalable simulator for an architecture for cognitive computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 54:1–54:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[141] Kenneth L Rice, Mohammad A Bhuiyan, Tarek M Taha, Christopher N Vutsinas, and Melissa C Smith. Fpga implementation of izhikevich spiking neural networks for character recognition. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 451–456. IEEE, 2009.

[142] Arun Rodrigues et al. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, Mar. 2011.

[143] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

Approved for Public Release; Distribution Unlimited.

[144] Sandia National Labs. *SST DUMPI trace library*. (Accessed on: Nov. 3, 2018).

[145] Jun Sawada, Filipp Akopyan, Andrew S Cassidy, Brian Taba, Michael V Debole, Pallab Datta, Rodrigo Alvarez-Icaza, Arnon Amir, John V Arthur, Alexander Andreopoulos, et al. Truenorth ecosystem for brain-inspired computing: scalable systems, software, and applications. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 130–141. IEEE, 2016.

[146] Jürgen Schmidhuber. Deep Learning in Neural Networks: An Overview. *Neural Networks*, 61:85–117, March 2014.

[147] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.

[148] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[149] Catherine D Schuman, Thomas E Potok, Robert M Patton, J Douglas Birdwell, Mark E Dean, Garrett S Rose, and James S Plank. A Survey of Neuromorphic Computing and Neural Networks in Hardware. *arXiv.org*, May 2017.

[150] Jaewook Shin et al. Speeding up nek5000 with autotuning and specialization. In *Proc. of the 24th ACM Int. Conf. for Supercomputing*, pages 253–262. ACM, 2010.

[151] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, and D. Mavriplis. Cfd vision 2030 study: A path to revolutionary computational aerosciences. Technical Report NASA/CR-2014-21878, NASA, March 2014.

[152] Shane Snyder et al. A case for epidemic fault detection and group membership in hpc storage systems. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science*, pages 237–248. Springer Int. Publishing, 2015.

[153] Shane Snyder et al. Techniques for Modeling Large-scale HPC I/O Workloads. In *Proc. of the 6th Int. Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, pages 5:1–5:11, New York, NY, USA, 2015. ACM.

[154] K. L. Spafford and J. S. Vetter. Aspen: A domain specific language for performance modeling. In *SC12: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Salt Lake City, 2012.

[155] RB Stein, Ao S French, and AV Holden. The frequency response, coherence, and information capacity of two neuronal models. *Biophysical journal*, 12(3):295–322, 1972.

[156] Doron Tal and Eric L. Schwartz. Computing with the leaky integrate-and-fire neuron: Logarithmic computation and multiplication. *Neural Computation*, 9(2):305–318, 1997.

Approved for Public Release; Distribution Unlimited.

[157] G. Urgese, F. Barchi, E. Macii, and A. Acquaviva. Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms. *IEEE Transactions on Emerging Topics in Computing*, PP(99):1–1, 2017.

[158] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, 1982.

[159] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proc. of the 1st Int. Conf. on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[160] A. Vedaldi and K. Lenc. Matconvnet – convolutional neural networks for matlab. In *Proceeding of the ACM Int. Conf. on Multimedia*, 2015.

[161] Jeffrey S Vetter and Jeremy S Meredith. Synthetic program analysis with aspen. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*, pages 1–6. University of Edinburgh, 2015.

[162] Sying-Jyan Wang. Load-balancing in multistage interconnection networks under multiple-pass routing. *Journal of Parallel and Distributed Computing*, 36(2):189 – 194, 1996.

[163] Ke Wen et al. Flexfly: Enabling a reconfigurable dragonfly through silicon photonics. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 15:1–15:12, Piscataway, NJ, USA, 2016. IEEE Press.

[164] S Williams, A Waterman, and D Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[165] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[166] N. Wolfe et al. Preliminary performance analysis of multi-rail fat-tree networks. In *2017 17th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGRID)*, pages 258–261, May 2017.

[167] Noah Wolfe, Misbah Mubarak, Christopher D. Carothers, Robert B. Ross, and Philip H. Carns. Modeling large-scale slim fly networks using parallel discrete-event simulation. *ACM Trans. Model. Comput. Simul.*, 28(4):29:1–29:25, August 2018.

[168] Xu Yang, John Jenkins, Misbah Mubarak, Robert Ross, and Zhiling Lan. Watch out for the bully! job interference study on dragonfly networks. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[169] Li Yu, Ziming Zheng, Zhiling Lan, and Susan Coghlan. Practical online failure prediction for blue gene/p: Period-based vs event-driven. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 259–264. IEEE, 2011.

Approved for Public Release; Distribution Unlimited.

[170] Eitan Zahavi, Gregory Johnson, Darren J. Kerbyson, and Michael Lang. Optimized InfiniBand fat-tree routing for shift all-to-all communication patterns. *Concurr. Comput. : Pract. Exper.*, 22(2):217–231, February 2010.

[171] Mohammed J. Zaki, Christopher D. Carothers, and Boleslaw K. Szymanski. Vogue: A variable order hidden markov model with duration based on frequent sequence mining. *ACM Trans. Knowl. Discov. Data*, 4(1):5:1–5:31, January 2010.

[172] Ziming Zheng and Zhiling Lan. Reliability-aware scalability models for high performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–9. IEEE, 2009.

[173] Ziming Zheng, Li Yu, Wei Tang, Zhiling Lan, Rinku Gupta, Narayan Desai, Susan Coghlan, and Daniel Buettner. Co-analysis of ras log and job log on blue gene/p. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 840–851. IEEE, 2011.

Approved for Public Release; Distribution Unlimited.

A Publications

The following is the list of publications published as part of this research project.

1. P. Date, C. D. Carothers, J. Hendler and M. Magdon-Ismail, "Efficient Classification of Supercomputer Failures using Neuromorphic Computing", To appear in *Proceedings of the 2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, November 18–21, Bengaluru, India, 2018.

2. N Wolfe, M. Plagge, M. Mubarak, **C D. Carothers**, R. B. Ross. "Evaluating the Impact of Spiking Neural Network Traffic on Extreme-Scale Hybrid Systems", In *Proceedings of the 9th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS 2018) as part of Supercomputing (SC'18).* Dallas, Texas, November 2018.

3. N. Wolfe, M. Mubarak, C. D. Carothers, R. B Ross, P. Carns, "Modeling Large-Scale Slim Fly Networks Using Parallel Discrete-Event Simulation", In *ACM Transactions on Modeling and Computing Simulation*, part of Special Issue on *Best of 2016 ACM-SIGSIM-PADS Conference*, Volume 28 Issue 4, page 29:1–29:25, October 2018.

4. M. Plagge, C. D. Carothers, E. Gonsiorowski and N. McGlohon, "NeMo: A Massively Parallel Discrete-Event Simulation Model for Neuromorphic Architectures", In *ACM Transactions on Modeling and Computing Simulation*, part of Special Issue on *Best of 2016 ACM-SIGSIM-PADS Conference*, pages 30:1–30:25, October 2018.

5. **C. D. Carothers**, J. S. Meredith, M. P. Blanco, J. Vetter, M. Mubarak, J. LaPre and S. Moore, "Durango: Scalable Synthetic Workload Generation for Extreme-Scale Application Performance Modeling and Simulation", In *Proceedings of the 2017 ACM SIGSIM-PADS Conference*, pages 97–108, Singapore, May 24–26, 2017.

6. N. Wolfe, M. Mubarak, N. Jain, J. Domke, A. Bhatele, **C. D. Carothers** and R. B. Ross, "Methods for Effective Utilization of Multi-Rail Fat-Tree Interconnects", In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).* Madrid, Spain, May 14-17, 2017.

7. P. Date, J. A. Hendler and **C. D. Carothers**, "Design Index for Deep Neural Networks", In *Proceedings of the 2016 Annual International Conference on Biologically Inspired Cognitive Architectures (BICA 2016)*, New York, New York, July 16-19, 2016. `http://www.sciencedirect.com/science/article/pii/S1877050916316726.`

8. M. Plagge, **C. D. Carothers** and E. Gonsiorowski, "NeMo: A Massively Parallel Discrete-Event Simulation Model for Neuromorphic Architectures", In *Proceedings of the 2016 ACM SIGSIM-PADS Conference*, May 15-18, 2016, Banif, Alberta Canada.

9. N. Wolfe, **C. D. Carothers**, M. Mubarak, R. B. Ross and P. Carns, "Modeling a Million-Node Slim Fly Network using Parallel Discrete Event Simulation", In *Proceedings of the 2016 ACM SIGSIM-PADS Conference*, May 15-18, 2016, Banif, Alberta Canada.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

The following is the list of symbols, abbrevations and acronyms used in this report.

- **AVL Tree:** A self-balancing search tree used by ROSS.
- **AFRL:** Air Force Research Laboratory.
- **Aspen:** analytic performance modeling language developed at ORNL
- **Compass**: IBM's original massively parallel simulator for the TrueNorth processor.
- **CODES:** Co-design of extreme-scale systems simulation system.
- **DOE:** Department of Energy.
- **Durango:** a hybrid system performance modeling framework built using *Aspen* and *CODES*.
- **fps:** Frames-per-second.
- **GVT:** Global Virtual Time - which is a time computed and used by ROSS to determine which events can be safely garbage collected as part of optimitic event processing.
- **HPC:** High-Performance Computing.
- **LIF:** Leak Integrate and Fire neuron model.
- **LP:** Logical Process - a base simulation modeling entity used by ROSS.
- **MNIST:** A hand writting image classification benchmark developed at NIST.
- **MPI:** Message-Passing Interface. A standard used in the development of massively parallel software systems.
- **MSTAR:** Moving and Stationary Target Acquisition and Recognition (MSTAR) data is a SAR image data-set of military hardware.
- **mW:** milli-watts - a measure of power comsumption.
- ***NeMo***: A massively parallel neuromorphic processor simulator design using ROSS.
- **NIST:** National Institute of Standards and Technology.
- **NSCS:** 2nd generation simulator for IBM TrueNorth processor.
- **LLNL:** Lawrence Livermore National Laboratory.
- **LULESH:** DOE mini-application used in system performance modeling and bechmarking.
- **ORNL:** Oak Ridge National Laboratory.
- **PDES:** Parallel discrete-event simulation.

- **ROSS:** Rensselaer Optimistic Simulation System.
- **SAR:** synthetic apeture radar.
- **TNLIF:** LIF neuron model implemented on TrueNorth.
- **TrueNorth:** neuromorphic processor designed and built by IBM.

Approved for Public Release; Distribution Unlimited.