Pacific Northwest
NATIONAL LABORATORY

PNNL-34841

Proxy Applications for Converged Workloads

DMC LDRD Initiative

September 2023

Sayan Ghosh
Milan Jain
Hyungro Lee
Kenneth Roche

U.S. DEPARTMENT OF
ENERGY

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

PNNL-34841

Proxy Applications for Converged Workloads

DMC LDRD Initiative

September 2023

Sayan Ghosh
Milan Jain
Hyungro Lee
Kenneth Roche

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99354

Abstract

Modern scientific applications are complicated and require coordination of several components. Proxy application driven software-hardware co-design plays a vital role in driving innovation among the developments of applications, software infrastructure and hardware architecture. Proxy applications are self-contained and simplified codes that are intended to model the performance-critical computations within applications.

Applications executing on modern High Performance Computing (HPC) systems are susceptible to network congestion, insufficient memory bandwidth within and across compute nodes, and inadvertent loss of performance due to bugs and unoptimized programming models. Modern numerical simulations and machine learning models play a critical role in studying physical phenomenon under myriad uncertainties. Such applications often exhibit irregular computation and memory accesses at specific regions of the application code, which can contribute to various performance bottlenecks at scale. To mitigate such issues and prepare the next generation hardware for a variety of computation and data movement contingencies, a well-known practice is to consider "proxy" applications as representative motifs for various classes of scientific applications. While there is disagreement in the HPC community on the mechanisms of construction of the proxy applications, there is a strong consensus on their positive impact in co-design.

Proxy Applications for Converged Workloads (PACER) is about facilitating software-hardware co-design through proxy applications with the goal of improving the performance of converged science workflows on heterogeneous systems.

Summary

We discuss several proxy applications from multiple computational domains, and apart from the variety of the application patterns, the need to expose myriad customizable trade-offs impacting the resource usage within the proxies is a crucial aspect, often requested by hardware designers. This deviates slightly from the traditional concept of proxies, which mandates a reasonable degree of equivalence with the parent or original application. In our work, we have been cognizant of the importance of similarity between parent and a proxy application, however, we have put more weightage on customizability and parameterizability to be able to test a spectrum of application scenarios. This is particularly important for irregular cases like graph analytics, where it is challenging to capture the relative structuredness of multifarious input graphs. Due to this choice, our artifacts can be made to behave traditionally, following a specific (larger) application scenario, or in potentially new ways to trade-off solution quality with performance (or vice versa). Another crucial choice we made early in the project lifetime was the usage of community-driven programming models for constructing the proxy applications, which can work on laptops to supercomputers and can be interoperable. As a result, proxies discussed in this report can work with a range of computational resources, on a variety of input data sizes and parameters.

Acknowledgments

This research was supported by the **Data Model Convergence Initiative**, under the Laboratory Directed Research and Development (LDRD) Program at Pacific Northwest National Laboratory (PNNL). PNNL is a multi-program national laboratory operated for the U.S. Department of Energy (DOE) by Battelle Memorial Institute under Contract No. DE-AC05-76RL01830.

Contents

Figures

1.0 Introduction

Classic HPC applications have been dominated by Floating Point Operations (FLOPS) intensive linear algebra operations. On the other hand, modern science workflows are complicated tasks that coordinate data collection (e.g., from a beamline) and poised to perform simulations on multiple computational sites. Current heterogeneity in the hardware (GPUs, CPUs, and AI Accelerators [AIA]) and software/programming models' maturity has further set the stage for converged applications --- potentially combining HPC programming models, ML frameworks, combinatorics algorithms, scientific toolkits, etc. This increased convergence requires coordination between disparate programming systems, dealing with different data feeds and speeds, co-location of irregular and data intensive computation scenarios with regular arithmetic intensive cases, etc.

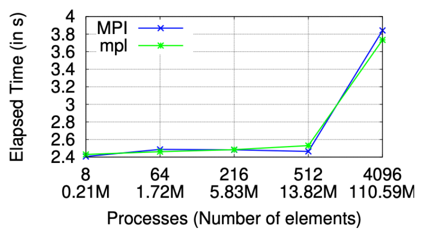
As the hardware landscape becomes more heterogeneous and unconventional, it is important for applications to be adaptive and customizable. Current trends in advanced chip packaging demonstrate reduced memory per processing element (PE), requiring distributed-memory considerations. As a result, strong scalability is becoming a hard requirement for efficiency. Unfortunately, community-driven distributed-memory models still suffer from limited composability to express a variety of application patterns. Additionally, some data-intensive applications such as graph analytics inherently exhibit patterns that are difficult to optimize at scale due to repetitive uncoalesced memory accesses. Since graphs are rapidly increasing in size, graph-based analytics such as similarity detection of proteins are becoming exorbitant to run on traditional GPU-based systems (thousands of core-hours), necessitating newer algorithms and methodologies to limit intermediate data. Data manipulation and movement operations are endemic in modern machine learning workflows, which requires careful exploitation of the reduced precision units in modern GPUs and AI Accelerators (AIA), in addition to optimal scheduling of myriad computational tasks. Machine Learning methods can complement simulation-driven analytics, such as molecular property prediction in biophysical systems and cascading failure detection in power grids.

As a result, it is important to invest in new algorithms, heuristics, and trade-offs to leverage the massive parallelism available in modern extreme-scale systems. We need a multi-pronged strategy to 1) develop/utilize high-level distributed-memory programming abstractions for rapid prototyping of application scenarios, 2) devise efficient heuristics and methods to reduce data movement for (irregular) graph analytics and combinatorial methods, and 3) quantitative evaluation of machine learning methodologies to predict domain properties efficiently. Forthcoming sections discuss these three application thrusts in details. These workloads can be constructed following the best practices of proxy application development [1], capturing the compute/throughput patterns in existing or prospective applications. The goals of our proxy application-based development strategy are as follows:

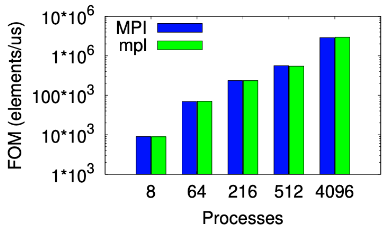
- Quantify bottlenecks in hardware and software – influence future systems.
- Expose fundamental trade-offs (e.g., memory and computation).
- Sandbox to design and expand applications on future systems.
- Enable systematic profiling and analysis.
- Caters to a wide variety – researchers, vendors, tool/runtime/compiler developers, etc.

2.0 Distributed-Memory Prototyping and Analysis

To improve the composability of existing distributed-memory models such as Message Passing Interface (MPI), we investigated the application of modern C++ programming paradigm. The C++ programming language has made significant strides in improving performance and productivity across a broad spectrum of applications and hardware. The C++ language bindings to MPI had been deleted since MPI 3.0 (circa 2009) because it reportedly added only minimal functionality over the existing C bindings relative to modern C++ practice while incurring significant amount of maintenance to the MPI standard specification. Two years after the MPI C++ interface was eliminated, the ISO C++ 11standard was published, which paved the way for modern C++ through numerous improvements to the core language. Since then, there has been continuous enthusiasm among application developers and the MPI Forum for modern C++ bindings to MPI. In this paper, we discuss ongoing efforts of the recently formed MPI working group on language bindings in the context of providing modern C++ (C++11 and beyond) support to MPI. Because of the lack of standardized bindings, C++-based MPI applications will often layer their own custom subsets of C++ MPI functionality on top of lower-level C; application- and/or domain-specific abstractions are subsequently layered on this custom subset. From such efforts, it is apparently a challenge to devise a compact set of C++ bindings over MPI with the “right” level of abstractions to support a variety of application uses cases under the expected performance/memory constraints. This work is used to identify and eventually standardize a normative set of C++ bindings to MPI that can provide the basic functionality required by distributed-memory applications. To engage with the broader MPI and C++ communities, we discuss a prototypical interface derived from mpl, an open-source C++17 message passing library based on MPI (referred to as mpl-subset) [2]. We port LULESH mini-application using mpl-subset and demonstrate identical performance to baseline MPI version, enhancing the productivity and composability without affecting performance.



(a) Elapsed time (in secs.)



(b) Figure of Merit (Elements/us)

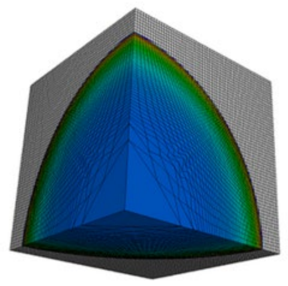
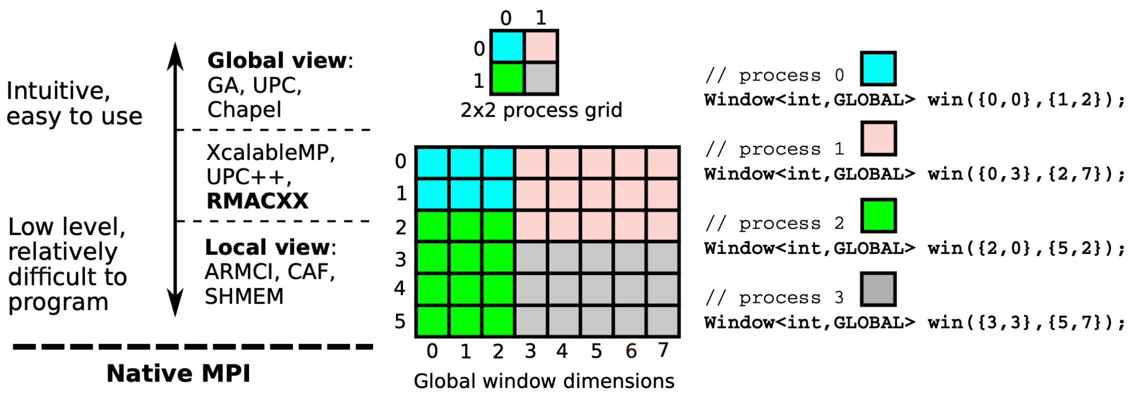


Figure 1. MPL(-subset) relative to MPI performances are identical (left) and an unstructured hex mesh used in LULESH (right).

Parallel scientific applications can benefit from decoupling communication and synchronization. One-sided programming abstractions, which separate communication from synchronization, have in fact served as a motivation for Partitioned Global Address Space (PGAS) models. However, the use of PGAS models in application codes in a manner that fully exploits the benefit of these programming models requires significant development effort. Meanwhile, a vast majority of scientific codes already use the Message Passing Interface (MPI) and need convenient features to support application-specific one-sided communication scenarios. MPI Remote Memory Access (RMA)can be employed for this purpose. MPI is a low-level API, however, and developing applications with MPI RMA requires programmers to be well versed in its nuances. We developed RMACXX [3], a compact set of C++ bindings to MPI-3 RMA, to ease the use of MPI RMA. Unlike other PGAS models, which may have interoperability issues with

MPI, RMACXX is written on top of MPI and uses the same runtime as MPI. The basic functionality of RMACXX adds only a relatively small number of extra instructions (about 20) to the critical communication path. Moreover, RMACXX provides an intuitive API for building a wide variety of scientific applications while enjoying performance matching handwritten MPI-3 RMA codes.


Intuitive, easy to use
Low level, relatively difficult to program
Global view: GA, UPC, Chapel
XcalableMP, UPC++, RMACXX
Local view: ARMCI, CAF, SHMEM
Native MPI
0 1
0
1
2x2 process grid
0 1 2 3 4 5
0 1 2 3 4 5 6 7
Global window dimensions

```
// process 0
Window<int,GLOBAL> win({0,0},{1,2});

// process 1
Window<int,GLOBAL> win({0,3},{2,7});

// process 2
Window<int,GLOBAL> win({2,0},{5,2});

// process 3
Window<int,GLOBAL> win({3,3},{5,7});
```

Figure 2. RMACXX relative to MPI and other models (left) and demonstrating distributed global array creation using RMACXX -- 6x8 array distributed over a logical 2x2 grid (right).

We further developed a state-of-the-art application proxy for electronic structure calculations that forms the basis of several accurate time-dependent studies of unitary fermi gas, cold atom systems, and nuclear phenomenon on distributed-memory systems derived from [4]. The proxy performs a superfluid Time-Dependent Density Functional Theory (TDDFT) on a lattice is dominated by the evaluation of first and second order derivatives on the lattice, function evaluations for the time-stepping scheme, and regular density computations. The current version uses MPI and OpenMP code to time evolve a fictitious 4-component superfluid system based only on densities required for unitarity from dimensional analysis easily extensible to real microscopic systems. Subsets of the solutions are evolved independently in different PEs for scaling and efficient evaluations, and in each time step, three forward and inverse discrete 3D complex to complex Fourier transforms are performed on each solution to obtain highly accurate numerical derivatives and Laplacian terms from the theory.

The proxy demonstrates strong scalability with larger number of PEs. Following table lists the results obtained from NERSC Cori Haswell supercomputer comparing 4/8/16-OpenMP threads per process (tppn) varying #processes/node (ppn), keeping tppn*ppn quantity the same. The scalability is better with larger #threads (52x between 1 & 64 nodes for 16-tppn, vs. 45x for 8-tppn and 36x for 4-tppn).

NERSC Cori Haswell	(20 20 20 10 10)	NERSC Cori Haswell	(20 20 20 10 10)	NERSC Cori Haswell	(20 20 20 10 10)
n-ppn(tppn)	Avg Time	n-ppn(tppn)	Avg Time	n-ppn(tppn)	Avg Time
1-4(16)	146.90	1-8(8)	74.60	1-16(4)	42.16
2-4(16)	73.03	2-8(8)	37.26	2-16(4)	21.13
4-4(16)	36.79	4-8(8)	18.93	4-16(4)	10.93
8-4(16)	18.72	8-8(8)	9.74	8-16(4)	5.75
16-4(16)	9.62	16-8(8)	5.24	16-16(4)	3.12
32-4(16)	5.04	32-8(8)	2.84	32-16(4)	1.81
64-4(16)	2.78	64-8(8)	1.67	64-16(4)	1.16

Table 1. Performance on distributed memory (NERSC Cori supercomputer) with varying number of threads/processes exhibit strong scalability.

3.0 Graph Analytics Co-Design

Distributed-memory graph algorithms are fundamental enablers in scientific computing and analytics workflows. Most graph algorithms rely on the graph neighborhood communication pattern, i.e., repeated asynchronous communication between a vertex and its neighbors in the graph. The pattern is adversarial for communication software and hardware due to high message injection rates and input-dependent, many-to-one traffic with variable destinations and volumes. This pattern also gives rise to a high amount of irregularity and poses a challenge for GPGPU porting of graph workloads. We have performed elementary analysis of the impact of graph structure on communication performance and observe multiple orders of degree slowdown in terms of average and 99th percentile latencies when the communication pattern mimics a graph. We develop a proxy application that provides several options in tweaking the graph structure, to observe the impact of communication and memory-access overheads on heterogeneous systems [5].

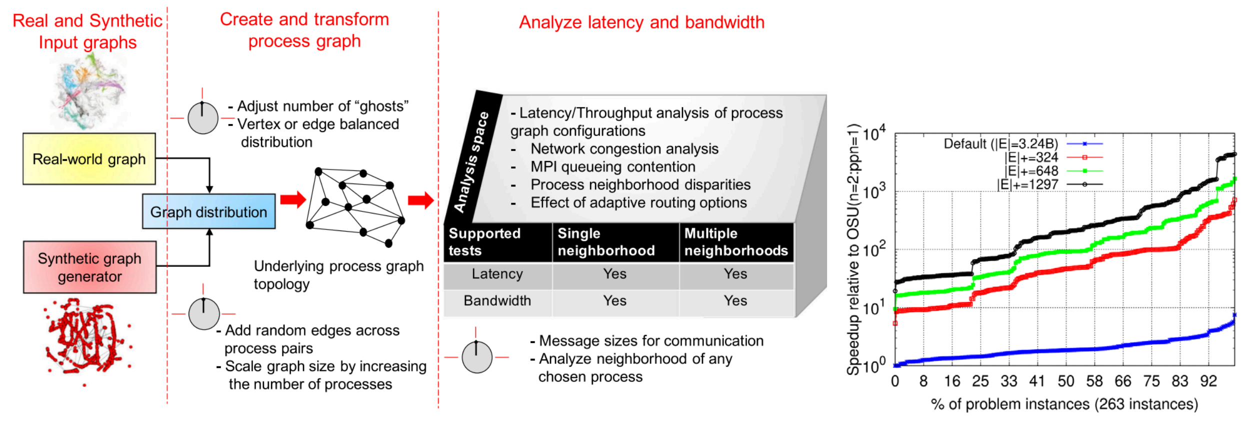

Real and Synthetic Input graphs
Create and transform process graph
Analyze latency and bandwidth
Real-world graph
Graph distribution
Synthetic graph generator
- Adjust number of "ghosts"
- Vertex or edge balanced distribution
Underlying process graph topology
- Add random edges across process pairs
- Scale graph size by increasing the number of processes
Analysis space
- Latency/Throughput analysis of process graph configurations
- Network congestion analysis
- MPI queueing contention
- Process neighborhood disparities
- Effect of adaptive routing options
Supported tests
Single neighborhood
Multiple neighborhoods
Latency
Yes
Yes
Bandwidth
Yes
Yes
- Message sizes for communication
- Analyze neighborhood of any chosen process
Speedup relative to OSU(n=2;ppn=1)
Default (|E|=3.24B)
|E|+=324
|E|+=648
|E|+=1297
% of problem instances (263 instances)

Figure 3. Graph proxy application generator for analysis (left) and average latencies of data exchanges for minor variations of a billion-edge graph relative to best latency (right).

Apart from distributed-memory evaluation, we use the same proxy application for single-node partitioned-memory analysis, exploiting the same graph neighborhood access pattern for multiple PEs within a single node. We measure the overhead of loading large graphs (requiring terabytes of memory) by leveraging persistent memory systems such as Intel Optane [6] and observe same access patterns but different allocation locations can cause major performance divergences. These issues only creep up for irregular graph workloads and remain hidden for regular memory-access benchmarks that measure best-case memory bandwidth.

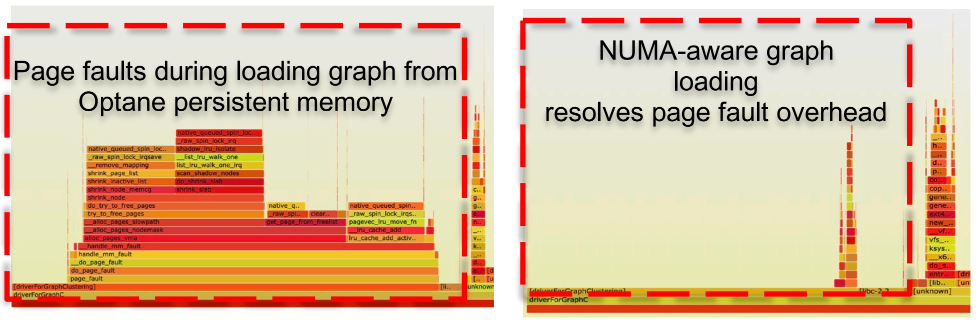
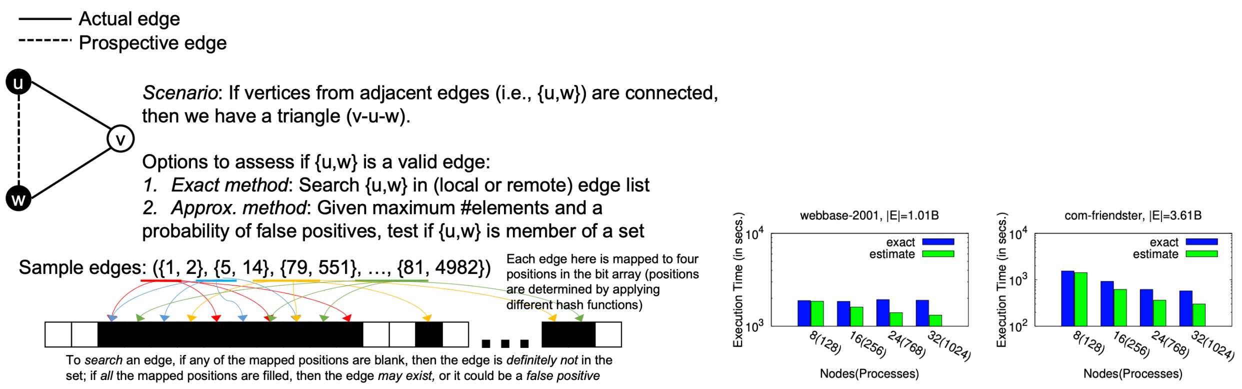

Page faults during loading graph from Optane persistent memory
NUMA-aware graph loading resolves page fault overhead

Figure 2 Loading large graphs can expose issues in Linux memory management system when persistent memory is involved

Certain classes of graph workloads generate arbitrary intermediate data, which loosely resembles the structure of the graph. A canonical example is patter enumeration, which involves enumeration or counting specific patterns of a subgraph found in a larger graph. A more specific example is that of graph triangle counting, which entails counting the number of “triangles” in a graph (a triangle is referred to as a 3-vertex subgraph where every vertex is connected to the remaining vertices, forming a triangle). Since the underlying scheme require explicit assessment of the edges, the volume of intermediate data and communication depends on the structure/connectivity of a graph. Since the number of edges in a graph can be significantly larger than the number of vertices, and graphs can exhibit diverse node-degree distributions, exact counting methodologies are challenging to sustainably scale on distributed-memory systems. One of the first practical extensions to our distributed-memory approach was allowing users to specify a buffer size for the communication to happen over increments. The buffered version ensures that the code will run to completion on a set of nodes large enough to approximately hold the entire graph (regardless of the volume of the intermediates when edge enumeration begins for triangle counting). We also identified a few simple heuristics (constant time overhead) to reduce the number of remote edges for enumeration, just by checking whether a remote node ID falls within the edge range held by a specific process. We also realized that aside from communication, searching for an edge (in the Compressed Sparse Row representation of a graph) was another bottleneck affecting the performance. We tried using different C++ associative data structures to store a partial list of edges, leading to a further 20–30% improvement in the overall execution times. Encouraged by these advances, we explored other means of overhauling the edge membership query performance. To that end, we observed remarkable performance enhancements (up to 7x) by employing a probabilistic data structure such as Bloom Filter [7].



Scenario: If vertices from adjacent edges (i.e., {u,w}) are connected, then we have a triangle (v-u-w).

Options to assess if {u,w} is a valid edge:
1. *Exact method*: Search {u,w} in (local or remote) edge list
2. *Approx. method*: Given maximum #elements and a probability of false positives, test if {u,w} is member of a set

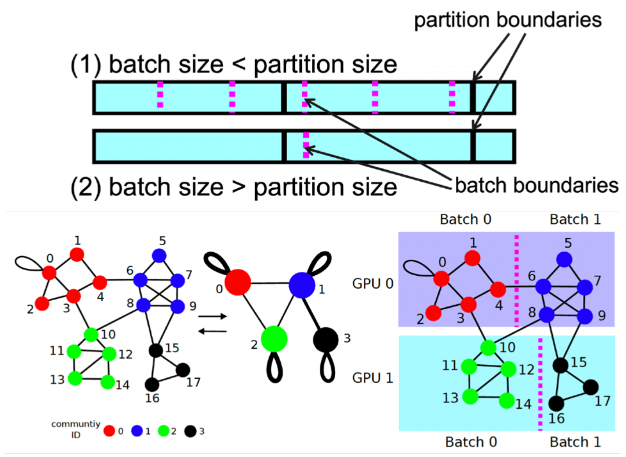
Sample edges: ({1, 2}, {5, 14}, {79, 551}, …, {81, 4982})

Each edge here is mapped to four positions in the bit array (positions are determined by applying different hash functions)

To *search* an edge, if any of the mapped positions are blank, then the edge is *definitely not* in the set; if *all* the mapped positions are filled, then the edge *may exist*, or it could be a *false positive*

Figure 3 Edge query scenario in computing triangle counts, depicting exact and estimated methods (left) and performance of exact and estimated for large graphs (right).

Our last discussion on graph analytics concerns GPU implementation and trade-offs for a complex graph algorithm – namely graph clustering. Graph clustering or community detection is a novel method to identify closely related structures within a graph. Applications of graph clustering can be found in cybersecurity, computational biology, proteomics, power grids, et al. Graph algorithms/heuristics are usually part of existing data analytics and scientific computing pipelines (such as NVIDIA RAPIDS, Tridata, Apache Spark, etc.) and have the potential to be embedded into a larger "converged" application context. As such, developing high performance implementations of graph clustering can enhance analytics pipelines. Currently, GPUs are an integral part of HPC systems, especially dense-GPU systems are ubiquitous. Hence, there is a significant effort to port applications on GPUs to extract maximal performance from modern single-node HPC platforms. In general, designing graph applications efficiently on GPUs is challenging, due to relatively low arithmetic computational intensity and arbitrarily high irregular memory accesses. The graph clustering problem is inherently serial, and the current modularity-maximization based parallel graph clustering implementations usually incorporate additional heuristics such as distance-k coloring to improve the overall quality and convergence. In contrast, we introduce a batched method for clustering on GPUs that allow users to achieve the desirable amount of quality without compromising the performance and involving sophisticated algorithms/heuristics [8]. Due to a partition-based design, we can process graphs larger than the combined GPU memory of a node. Despite of increasing the frequency of synchronization through batches (batches are processed in a bulk-synchronous manner among GPUs), the effect on the performance is mild for large graphs. Our implementation is based on the popular Louvain method for graph clustering, and we compare our performance and quality with existing parallel Louvain implementations on CPU and GPU systems. Overall, we observed speedups of about 2–14x/1.3–7.5x on up to 16/8 NVIDIA V100/A100 GPUs and a maximum speedup of 50x/30x as compared to state-of-the-art shared-memory and GPU-based Louvain implementations, as shown in the results below for a variety of input graphs.



Graphs	|V|	|E|	Max speedup over GPUs (A100/V100)	Max speedup over CPU (Grappolo)
uk-2007-05	105.89M	6.6B	2.1×	0.27×
sk-2005	50.63M	3.86B	2.02×	0.27×
com-friendster	65.6M	3.61B	13.75×	3.41×
twitter7	41.65M	2.93B	1.5×	4.55×
it-2004	41.29M	2.27B	3.29×	0.83×
mycielskian20	786.43K	1.35B	3.92×	53.43×
webbase-2001	118.14M	1.98B	3.76×	0.97×
com-orkut	3.07M	234.37M	2.1×	1.31×

Figure 4 GPU partitioning and batched Louvain clustering (left) and performance in terms of speedup compared to multithreaded parallel CPU implementation (right).

4.0 Machine Learning for Predictive Analytics

Our work on Machine Learning spans optimization techniques for elementary machine learning kernels using different programming models, such as sparse matrix-dense matrix multiplication (SpMM) to data parallelism targeting multiple GPUs for spatio-temporal modeling. Latest work also proposes a proxy for predicting molecular properties via Graph Neural Networks. Our goals are to develop proxy applications that can leverage multiple machine learning frameworks (both Tensorflow and PyTorch) and parallelism strategies on broad variety of hardware architectures.

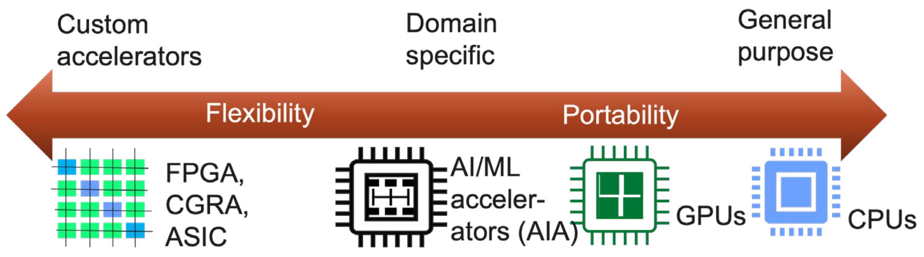


Figure 5 Machine Learning can benefit from broad variety of hardware.

Sparse deep neural networks have gained increasing attention recently in achieving speedups on inference with reduced memory footprints. Real-world applications often must deal with sparse data and irregularities in the computations, yet a wide variety of Deep Neural Network (DNN) tasks remain dense without exploiting the advantages of sparsity in networks. Recent works presented in MIT/IEEE/Amazon GraphChallenge have demonstrated significant speedups and various techniques. Still, we find that there is limited investigation of the impact of various Python and C/C++ based programming models to explore new opportunities for the general cases. In our work [9], we provide first-of-its-kind extensive quantitative evaluations of various contemporary GPGPU programming models such as CUPY, CUDA CUSPARSE, and OPENMP in the context of Sparse Deep Neural Network (SPDNN) implementations (derived from the Graph Challenge reference serial code) on single and multiple GPUs from NVIDIA DGX-A100 40GB/80GB platforms.

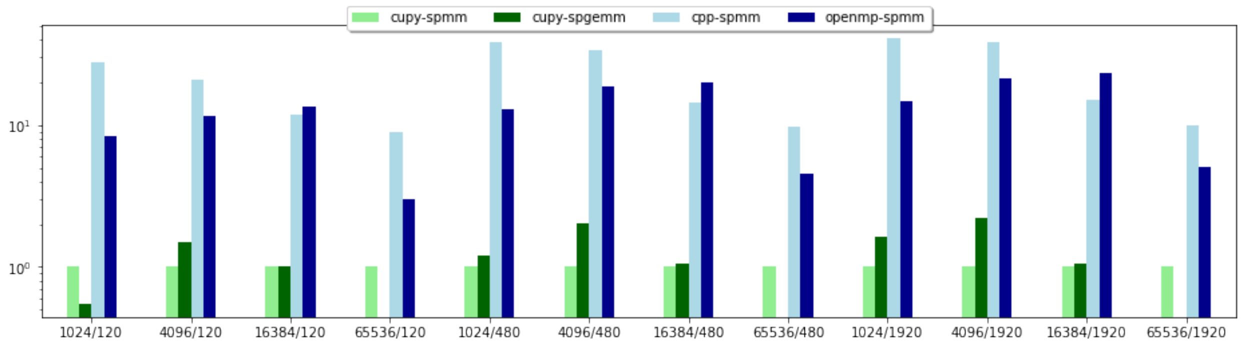


Figure 6 OpenMP-offload based SpDNN exhibits 18x speedup compared to NVIDIA CuPy on a single A100 GPU, depicting tremendous promise for traditional HPC programming models.

Dependencies between layers makes SpDNN difficult to scale across GPUs (limited parallelism), but for larger number of neurons, a trivial 1D partitioning of features can lead to a sudden performance spike (for CuPy) until a limited number of GPUs, due to reduction in the data volume per GPU (alluding to sub-optimal implementation of CuPy).

For the next work, we discuss time-series prediction, a common application scenario for several domains, which exhibit a lack of representative benchmarks and proxies. To facilitate the co-design of next generation hardware architectures, it is critical to characterize the workloads of deep learning (DL) applications and assess their computational patterns on different levels of the execution stack. Time series prediction is one such DL application heavily used in areas that include critical decision making: ensuring power grid resiliency, climate forecasting, transportation infrastructure optimization, stock market prediction, etc. Unlike cross-sectional data (e.g., images), time-series data is inherently sequential, posing challenges to parallelization in the context of deep learning. In this work [10], we developed a proxy application for deep learning-based time-series application that uses spatio-temporal data from a dynamical system for model training and inference. We study the performance profiles of the associated computational patterns for both training and inference on four different levels: models (Long short-term Memory and Convolutional Neural Network), DL frameworks (Tensorflow and PyTorch), datatypes (FP64, mixed-precision), and single-node dense GPU platforms (Nvidia DGX A100 and DGX-2 V100). Overall, our findings indicate that, in the context of multiple variants of our time-series prediction proxy application, computational profiles of Tensorflow and PyTorch mostly exhibit divergent overheads across GPU platforms. Our studies also demonstrate that associated data movement, transformation and combination can take more than 50% of the overall execution times.

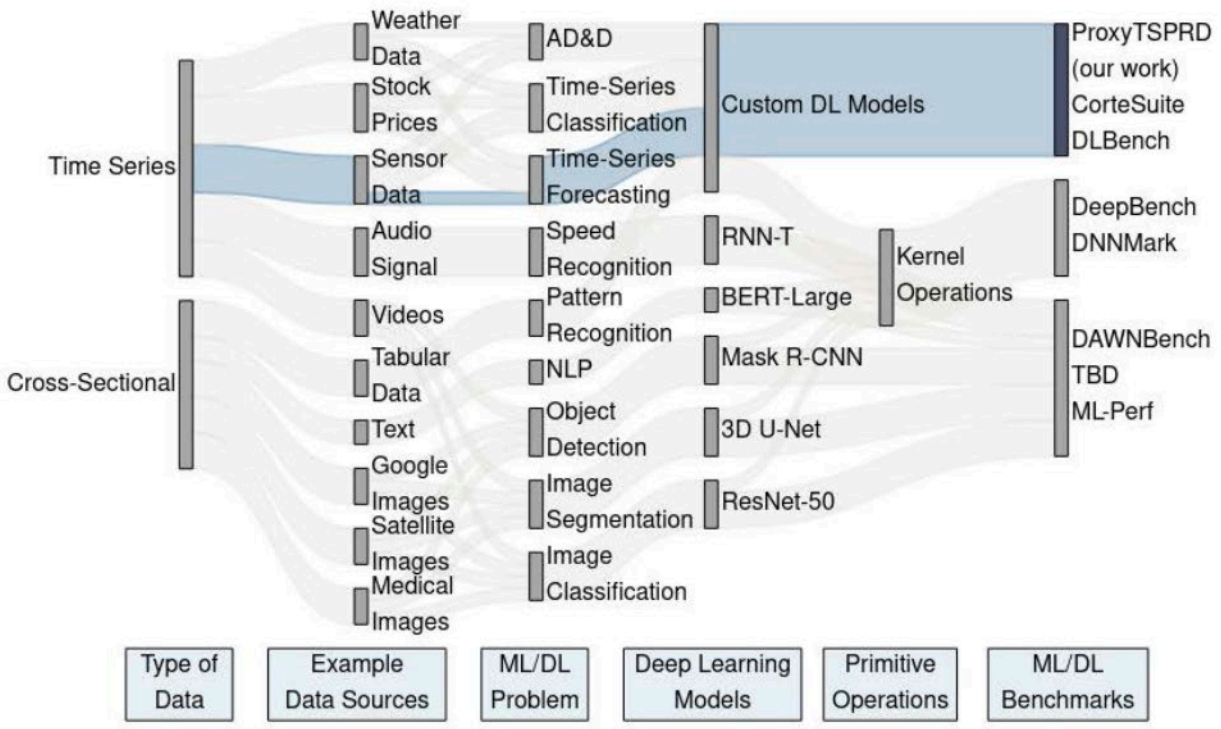


Figure 7 Machine Learning is data-centric, areas which does not have standard models (like time-series prediction) lack representative proxies and benchmarks.

Data / Models / #RDUs / Training Time (s)		1	2	4	8
Grid	LSTM	190	129	65	34 (31)
	CNN	94	63	41	28 (32)
Climate	LSTM	622	397	222	118 (136)
	CNN	454	311	194	126 (135)

Figure 8 Strong scaling on Sambanova RDUs (compared to NVIDIA A100 on 8 RDUs).

Our proxy application demonstrates scalable performance on Sambanova Data-Flow architecture known as Reconfigurable Dataflow Unit (RDU), in comparison with NVIDIA A100 GPU, providing a template for contemporary spatio-temporal models. Further optimizations are possible via synthesizing specific kernels through compiler-based High-Level Synthesis (HLS) tools, targeting custom accelerators such as FPGAs.

Graph Neural Networks (GNN) are neural networks for processing data represented as graphs, with applications in diverse science domains, such as chemistry, particle physics, molecular biology, etc. Molecules represented as graphs can be applied to various GNN models for predicting molecular properties. Hence, leveraging GNN techniques in quantum chemistry is crucial for molecular modeling and drug design. However, the complexity of scientific data and lack of expertise create challenges in selecting an accurate machine learning model and parameters with reasonable turn-around times in the training and prediction phases, especially for large molecules. Our work provides case studies for prioritizing accurate GNN models for density calculations using adaptive techniques carried out on contemporary GPU-based platforms. We have developed a prototypical application pipeline with state-of-the-art libraries that replicate quantum chemistry pipelines and thermo-dynamics of biochemical reactions to compare different models, considering the structure of molecules and properties into graphs. We compare the resulting molecular predictions of our prototypical GNN framework with ab initio techniques for DFT calculations using the NWChem computational chemistry package. Unlike the proxies discussed, quality assessment is important in this case, which can indicate the level of success one could expect modeling myriad molecules.

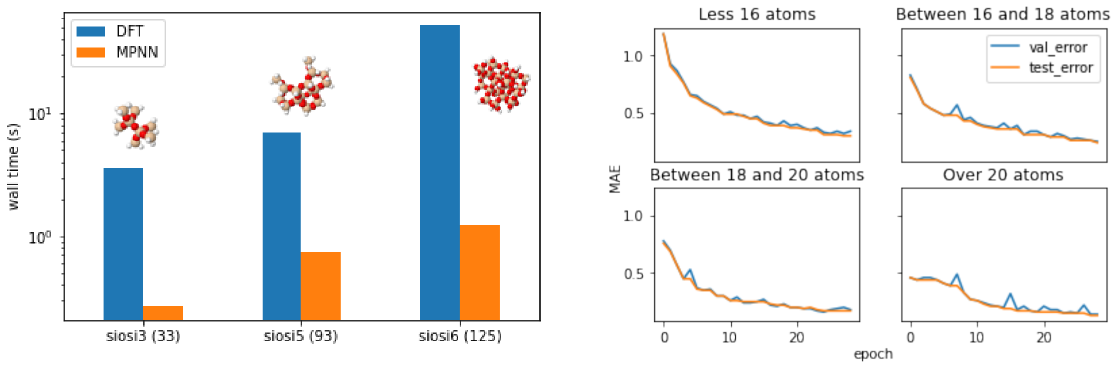


Figure 9 GNN-based MPNN vs. NWChem DFT performance for Zeolite fragments (left) and QM9 Mean Average Error (MAE) per atoms indicating good model accuracy (right)

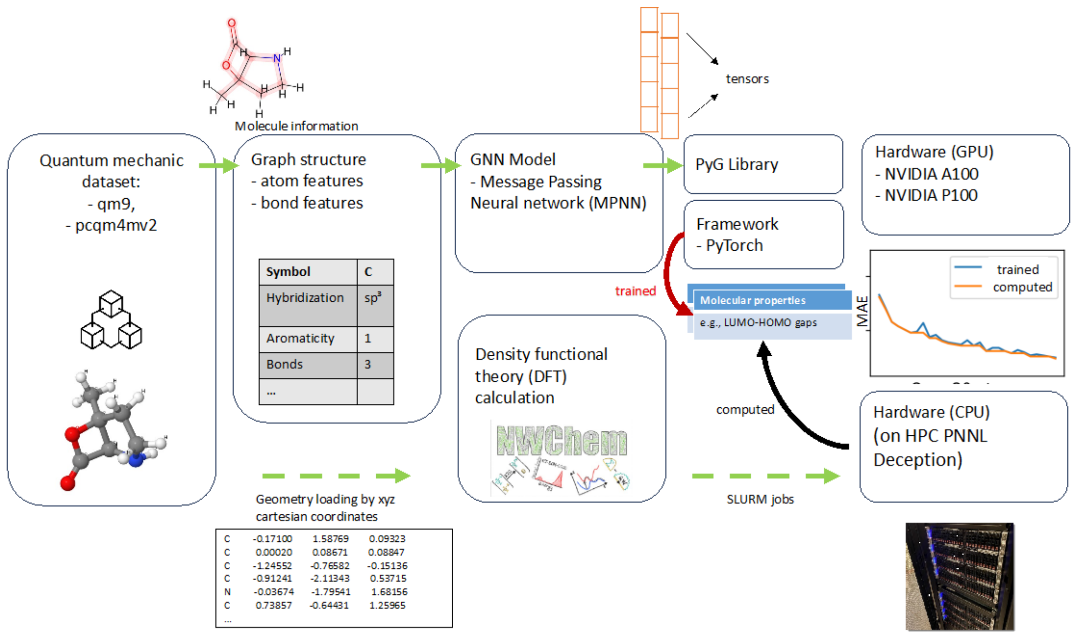

Molecule information
Quantum mechanic dataset:
- qm9,
- pcqm4mv2
Graph structure
- atom features
- bond features
Symbol C
Hybridization sp3
Aromaticity 1
Bonds 3
...
GNN Model
- Message Passing Neural network (MPNN)
tensors
PyG Library
Framework
- PyTorch
Hardware (GPU)
- NVIDIA A100
- NVIDIA P100
trained
Molecular properties
e.g., LUMO-HOMO gaps
MAE
trained
computed
computed
Density functional theory (DFT) calculation
NWChem
Hardware (CPU) (on HPC PNNL Deception)
Geometry loading by xyz cartesian coordinates
C -0.17100 1.58769 0.09323
C 0.00020 0.08671 0.08847
C -1.24552 -0.76582 -0.15136
C -0.91241 -2.11343 0.53715
N -0.03674 -1.79541 1.68156
C 0.73857 -0.64431 1.25965
SLURM jobs

Figure 10 Overview of the proxy setup for GNN-based molecular property prediction.

Preliminary results show that our PyTorch Geometric based GNN method (leveraging Message Passing Neural Network, MPNN) can predict the molecular properties by reducing the computation time significantly without any feature engineering (directly learns their own features from the molecular graphs) and returns relatively stable MAE for molecules less than 20 atoms, while depicting slight fluctuations for cases with higher number of atoms.

5.0 References

1. Cook, Jeanine, Hal Finkel, Christoph Junghams, Peter McCorquodale, Robert Pavel, and David F. Richards. *Proxy app prospectus for ECP application development projects*. No. LLNL-TR-740859. Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), 2017.

2. Ghosh, Sayan, Clara Alsobrooks, Martin Rüfenacht, Anthony Skjellum, Purushotham V. Bangalore, and Andrew Lumsdaine. "Towards modern C++ language support for MPI." In *2021 Workshop on Exascale MPI (ExaMPI)*, pp. 27-35. IEEE, 2021.

3. Ghosh, Sayan, Yanfei Guo, Pavan Balaji, and Assefaw H. Gebremedhin. "RMACXX: An Efficient High-Level C++ Interface over MPI-3 RMA." In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 143-155. IEEE, 2021.

4. Bulgac, Aurel, and Kenneth J. Roche. "Time-dependent density functional theory applied to superfluid nuclei." In *Journal of Physics: Conference Series*, vol. 125, no. 1, p. 012064. IOP Publishing, 2008.

5. Ghosh, Sayan, Nathan R. Tallent, and Mahantesh Halappanavar. "Characterizing performance of graph neighborhood communication patterns." *IEEE Transactions on Parallel and Distributed Systems* 33, no. 4 (2021): 915-928.

6. Ghosh, Sayan, Nathan R. Tallent, Marco Minutoli, Mahantesh Halappanavar, Ramesh Peri, and Ananth Kalyanaraman. "Single-node partitioned-memory for huge graph analytics: cost and performance trade-offs." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-14. 2021.

7. Ghosh, Sayan. "Improved distributed-memory triangle counting by exploiting the graph structure." In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1-6. IEEE, 2022.

8. Chou, Han-Yi, and Sayan Ghosh. "Batched Graph Community Detection on GPUs." In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 172-184. 2022.

9. Lee, Hyungro, Milan Jain, and Sayan Ghosh. "Sparse Deep Neural Network Inference Using Different Programming Models." In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1-6. IEEE, 2022.

10. Jain, Milan, Sayan Ghosh, and Sai Pushpak Nandanoori. "Workload characterization of a time-series prediction system for spatio-temporal data." In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, pp. 159-168. 2022.