A Scalable Parallel Hypergraph Generator (HyGen)

S.M.Shamimul Hasan, Neena Imam, and Ramakrishnan Kannan
Computing and Computational Sciences Directorate
Oak Ridge National Laboratory, Oak Ridge, TN, USA
Email: {hasans, imamn, kannanr}@ornl.gov

ABSTRACT

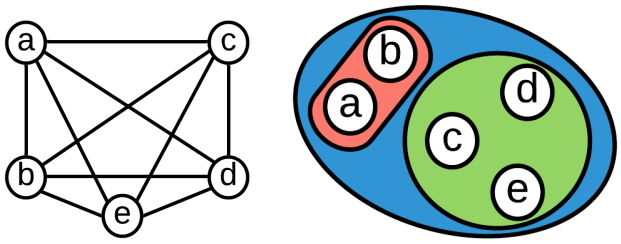
Graphs are extensively used to model real-world complex systems. An edge in a graph can model pairwise relationships. However, multiway relationships (connections between three or more vertices) are common in many complex systems such as cellular process, image segmentation, and circuit design. A graph edge cannot model multiway relationships. A hypergraph, which can connect more than two vertices, is thus a better option to model multiway relationships. A large-scale hypergraph analysis has the potential to find useful insights from a complex system and assist in knowledge discovery. Currently a limited number of hypergraphs exists that are representative of real-world datasets. Moreover, real-world hypergraph datasets are small in size and inadequate to incorporate future needs. A graph generator that can produce large-scale synthetic hypergraphs can solve the above mentioned problems. In this paper, we present a scalable parallel hypergraph generator (HyGen) based on the Message Passing Interface (MPI) standard. To generate hypergraphs, HyGen takes the following parameter values as inputs: i) number of vertices, ii) number of hyperedges, iii) number of clusters, iv) vertex distribution, v) hyperedge distribution, vi) local cluster cardinality, and vii) global cluster cardinality. We have demonstrated that HyGen can generate hypergraphs of various sizes in a scalable fashion. HyGen takes approximately four minutes to generate a hypergraph with 4.8 million vertices, 1.6 million hyperedges, and 800 clusters using 1,024 processes on a leadership class computing platform. Our strong and weak scaling experiments on supercomputers demonstrate that HyGen can quickly create large-scale hypergraphs in a parallel manner, thus providing a useful capability for hypergraph analysis.

KEYWORDS

Hypergraph, MPI, C++, OLCF, Rhea

1 INTRODUCTION

A graph is a powerful mathematical abstraction comprising of a set of objects called vertices and the links that connect some pairs of objects called edges [15]. Real-world complex systems are often represented and analyzed as graphs. This approach has shown tremendous success across many disciplines including biology, economics, engineering, physics, and the social sciences. The advancement of modern technologies has resulted in the explosive growth of graph data [9, 18]. However, one of the limitations is that an edge in a graph, which connects only two vertices, can only model pairwise relationships. In many applications such as cellular processes, social networks, and circuit design, a cluster of vertices represents multilateral relationships. An example is a network with online social communities called folksonomies, in which trilateral interactions occur among consumers, resources, and annotations [23]. In the same way, group interactions between multiple genes are responsible for numerous physiological phenomena in biology disciplines. It is essential to model group interactions (or multilateral relations) of multiple genes to understand diseases and discover cures [13]. As mentioned earlier, graph edge cannot model multilateral relationships (see Figure 1(a)). However, a high-dimensional graph called a hypergraph (see Figure 1(b)) can model complex multilateral relationships [23].



(a) Graph. (b) Hypergraph.

Figure 1: In this figure, we present regular graph and hypergraph examples. Figure 1(a) shows a graph example with 5 vertices and 10 edges. Figure 1(b) provides a hypergraph example with 5 vertices and three hyperedges.

A hypergraph is a collection of vertices and hyperedges (edges in hypergraphs) where a hyperedge can connect any number of vertices. By generalizing regular graphs, hypergraphs showed prominent research benefits in numerous analyses such as clustering, classification, and prediction [23]. Although analyzing the structural properties and dynamics of large-scale complex systems in the form of hypergraphs is crucial for improving our knowledge of complex systems, only a small number of real-world hypergraph datasets is available to aid in this type of analysis. Moreover, real-world

hypergraph datasets are small in size and insufficient to incorporate future requirements. An enormous amount of data collection and processing efforts are required to create a large-scale real-world hypergraph, which is tedious and laborious work. Furthermore, it is difficult to collect real-world datasets because of privacy reasons [17]. One possible solution is hypergraph generators that can quickly produce synthetic but realistic large-scale hypergraphs with millions of vertices and edges. Although several hypergraph analysis software are available, they do not support large-scale hypergraph generation [2, 7, 21]. Hence, we present a scalable and parallel hypergraph generator (HyGen) in this paper. HyGen is implemented in C++ programming language by following object-oriented design principles. We used Message Passing Interface (MPI) standard for our parallel implementation. Our experimental results show that HyGen can create large-scale hypergraphs faster in a parallel manner. We employed the Oak Ridge Leadership Computing Facility's (OLCF) Rhea cluster for our study [5].

2 HYGEN ARCHITECTURE

Overview

We developed a hypergraph generator by leveraging the TriData hypergraph generation approach mentioned in [22], where a hypergraph generator creates random hypergraph incidence matrices. The TriData hypergraph generator is conceptually similar to the stochastic block model for graphs [10, 14]. Our proposed hypergraph generator, HyGen, generates non-uniform hypergraphs. In a non-uniform hypergraph, the cardinalities of the hyperedges vary [8]. HyGen was developed in the C++ programming language using object-oriented design principles. In the following, we discuss the architecture of HyGen.

Preliminaries

A hypergraph H can be formally denoted as a tuple $H = (V, E)$, where V is the set of elements, called *vertices*, and E represents non-empty subsets of V, called *hyperedges* [3, 13]. We can employ an incidence matrix to represent the hypergraph. In the following, we present an incidence matrix representation of the hypergraph shown in Figure 1(b).

$$H_{i,j} = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

In the incidence matrix $H_{i,j}$, vertices are represented by matrix rows, while hyperedges are represented by matrix columns. Also, in the incidence matrix $H_{i,j}$, nonzeros (NNZs) in i, j means vertex i belongs to hyperedge j [22]. For example, two 1s in the first column of the incidence matrix $H_{i,j}$ represents a and b vertices of the red cluster in Figure 1(b). The concept of cluster in HyGen is inspired by the research presented in [16, 19].

Hypergraph Generation

Get Input Parameters: The HyGen takes the following input parameters when generating a hypergraph: i) the number of vertices, ii) the number of hyperedges, iii) the number of clusters, iv) the vertex distributions, v) the hyperedge distributions, vi) the local cluster cardinalities, and vii) the global cluster cardinality.

The number of vertices (parameter i) and number of hyperedges (parameter ii) are the maximum number of vertices and hyperedges a generated hypergraph can contain. A hypergraph is a suitable model for cluster analysis. To ensure that a HyGen incidence matrix contains clusters, HyGen takes the number of clusters information as an input (parameter iii). The vertex distribution, hyperedge distribution, local cluster cardinalities, and the global cluster cardinality are used to create clusters in the incidence matrix. The number of vertices in clusters are calculated using the total number of vertices (parameter i) and vertex distributions (parameter iv). Similarly, the hyperedge distributions (parameter v) and the number of hyperedges (parameter ii) are used to compute the number of hyperedges that the clusters contain. The local and global cluster cardinalities determine the density of the local and global clusters. The total number of values available in the vertex distributions (parameter iv), the hyperedge distributions (parameter v), and the local cluster cardinalities (parameter vi) is the same as the total number of clusters (parameter iii). In Figure 2, the "Generate Hypergraph" portion shows an example of a hypergraph incidence matrix, which contains 100 vertices, 100 hyperedges, and five clusters. The first four blocks represent local clusters and the last block depicts a global cluster (read the matrix from left to right). The global cluster contains multiple local clusters. For instance, the blue cluster in Figure 1(b) can be considered as a global cluster.

HyGen requires large input files to generate scalable hypergraphs. Hence, we implemented a program in the C++ language to create input files. This program uses the number of vertices, the number of hyperedges, and the number of clusters to produces an "input.txt" file, which contains the vertex distributions, the hyperedge distributions, the local cluster cardinalities, and the global cluster cardinality. In addition, the input file also contains the number of vertices, the number of hyperedges, and the number of clusters. The vertex distributions, the hyperedge distributions, and the local cluster cardinalities are created using a random number generator in the range 0.05-0.15. We normalized the values

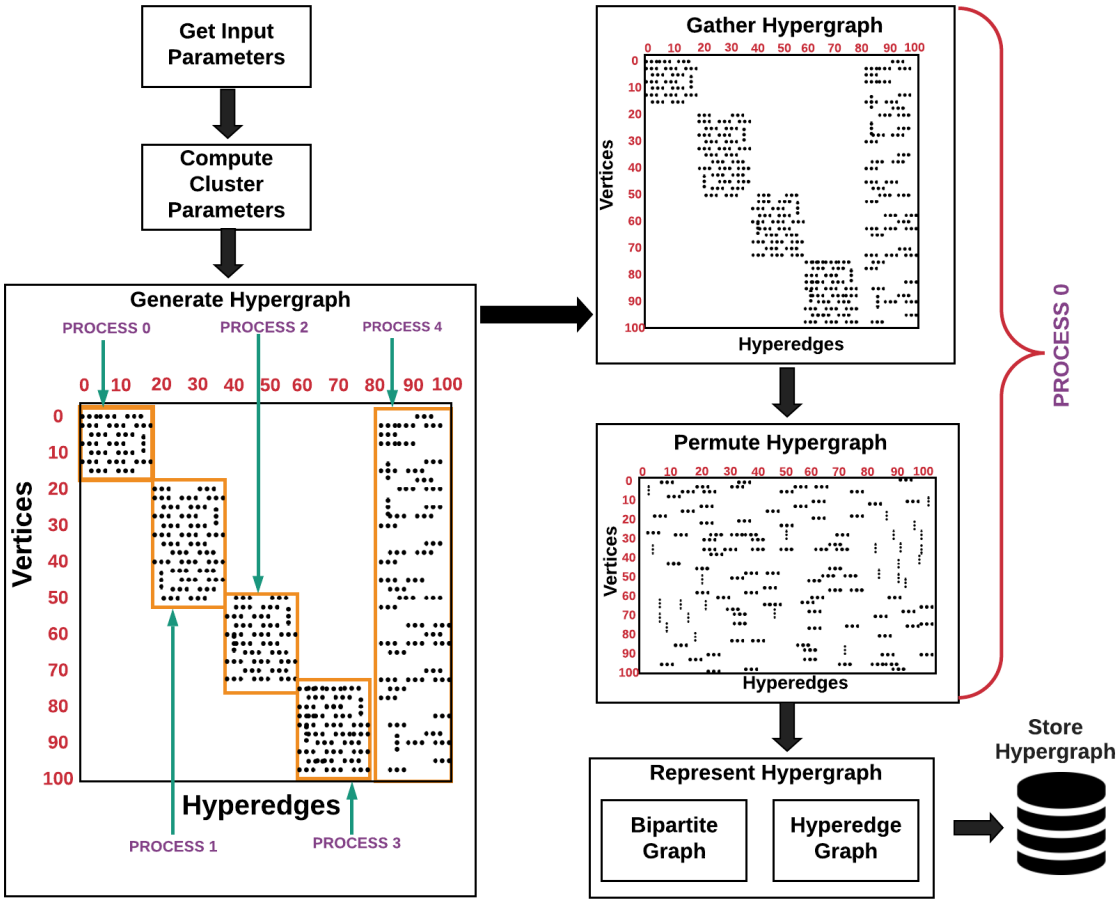


Figure 2: HyGen Architecture.

and checked that the sum of the distribution values is less than or equal to 1. We used 0.05 for the global cluster cardinality. We chose local and global cluster cardinality values in the range 0.05-0.15 because we do not want to make clusters too dense. We implemented HyGen using the MPI standard to generate hypergraph incidence matrices in parallel. The root MPI process (MPI process 0) reads input parameters from the input file.

Compute Cluster Parameters: HyGen generates a hypergraph incidence matrix in a distributed or parallel fashion. One MPI process can generate one complete cluster (the block shown in the "Generate Hypergraph" box in Figure 2) or a portion of a cluster, which depends on the number of High Performance Computing (HPC) resources employed for HyGen execution. After reading the input parameters, HyGen computes each complete cluster's vertices, hyperedges, and density. Next, HyGen calculates the number of partial or complete clusters that each MPI process will generate, the clusters vertices, hyperedges, and density. The incidence matrix to be constructed is partitioned column-wise 1-D across the available processors.

Generate Hypergraph: In the generate hypergraph step, HyGen informs each MPI process of the number of partial or complete clusters that need to be generated, the number of vertices and hyperedges that each cluster contains, and the cluster density. Each MPI process generates a portion of the hypergraph incidence matrix based on the information provided. MPI processes use a Bernoulli random variable with a success probability equal to or less than the cluster density to randomly determine nonzero values in the incidence matrix. In Figure 2, the "Generate Hypergraph" box shows that five MPI processes (processes 0-4) are generating five complete clusters in a parallel manner. Each dot represents a nonzero value in the incidence matrix.

Gather Hypergraph: After the hypergraph generation, the root MPI process (MPI process 0) collects the hypergraph submatrices generated by different MPI processes. Next, the root

MPI process creates the complete hypergraph incidence matrix by combining all the hypergraph submatrices collected from different MPI processes (see "Gather Hypergraph" box in Figure 2).

Permute Hypergraph: In Figure 2, the "Gather Hypergraph" box shows that the hypergraph produced by HyGen nicely contains all the clusters (or groups), which is not realistic. For example, in a streaming scenario (e.g., a network of Twitter followers), a stream of vertices arrives continuously so that all the clusters are not initially available. Hence, to simulate a real-world streaming scenario, we permute the incidence matrix by shuffling the positions of vertices and hyperedges. Now the incidence matrix looks like the "Permute Hypergraph" box in Figure 2. Although the vertices and hyperedges of five clusters are available in the incidence matrix, they are not available together as a group. It is possible to retrieve the clusters from the permuted hypergraph using tensor decomposition methods (see [8, 11]). A detailed discussion of tensor decomposition is outside the scope of this paper.

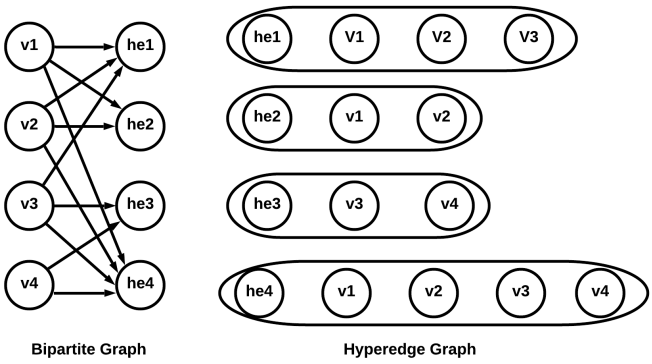


Figure 3: This figure presents hypergraph representation types [13]. In the figure, v1, v2, v3, v4 denotes vertices and he1, he2, he3, and he4 represent hyperedges.

Represent Hypergraph: HyGen represents a hypergraph as a bipartite graph and also as a hyperedge graph. The bipartite representation provides the data in a two-column format. The first value is the vertex and the second is the hyperedge. One instance of bipartite representation provides only partial information about a hyperedge. The hyperedge representation provides a complete hyperedge information at a given time. In the hyperedge representation, the first value denotes the hyperedge and the other values are all the vertices belonging to that hyperedge. A pictorial view of the bipartite and hyperedge representations is presented in Figure 3, where v1, v2, v3, v4 means vertices and he1, he2, he3, and he4 represent hyperedges. We need 11 rows in the incidence matrix to represent Figure 3's bipartite graph. However, only four rows are required to represent Figure 3's hyperedge graph. Users can specify the output format (bipartite or hyperedge) before generating the hypergraph. We developed a C++ iterator to traverse the generated hypergraph. The bipartite representation is commonly used, as it allows us to use a generated hypergraph in many dense and sparse matrix computation tools for further analysis. The hyperedge representation helps us find clusters quickly in a hypergraph.

Store Hypergraph: HyGen stores generated hypergraphs in a persistent storage space. Bipartite graphs are stored as a Matrix Market (MM) exchange format whereas hyperedge graphs are stored in text format.

3 EXPERIMENTAL EVALUATIONS

In this section, a detailed numerical evaluation of HyGen is provided.

HPC Hardware Environment

We employed the Oak Ridge Leadership Computing Facility's (OLCF) Rhea high performance computing cluster for HyGen implementation and experimental evaluation. Rhea is a Linux cluster containing 521 compute nodes and is divided into two partitions. The first partition is called "Rhea Partition," which contains 512 Central Processing Unit (CPU) nodes. Each node contains two 8-core 2.0 GHz Intel Xeon processors with Intel's Hyper-Threading (HT) Technology and 128 GB of main memory. The second one is called "GPU Partition," which contains 9 Graphics Processing Unit (GPU) nodes. Each node has 1 TB of main memory and two NVIDIA K80 GPUs in addition to two 14-core 2.30 GHz Intel Xeon processors with HT Technology. Atlas, a Lustre-based high performance file system at the OLCF is connected to the Rhea HPC resource [5].

HPC Software Environment

HyGen is implemented in the C++ programming language with MPI standard. We used G++ 6.2.0 and Open MPI 3.1.4 for HyGen compilation and execution. Our input file creation program is also implemented in the C++ programming language. We utilized the Slurm batch scheduler available in the OLCF's Rhea HPC resource. We developed a Slurm script that configures and submits jobs to Rhea for parallel execution [6].

Scalability Measurements

Scalability quantitatively measures the performance of a parallel application as the number of processes executing the application are increased. Strong scaling and weak scaling are the two types of scalability measurements. In strong scaling, the problem size is fixed but the number of processes are increased to achieve the speedup. Strong scaling aims

Table 1: This table presents the strong scaling experimental setup, total NNZs generated from each experiment, and output files information.

Experiment Name	Vertices	Hyperedges	Clusters	Compute Node	MPI Processes	Total NNZs	Bipartite Output File Size (in GB)	Hyperedge Output File Size (in GB)
S1	800000	2400000	12000	1	16	208505627	2.80	1.40
S2	800000	2400000	12000	2	32	208481602	2.80	1.40
S3	800000	2400000	12000	3	48	208485963	2.80	1.40
S4	800000	2400000	12000	4	64	208471624	2.80	1.40
S5	800000	2400000	12000	6	96	208480729	2.80	1.40
S6	800000	2400000	12000	8	128	208497776	2.80	1.40
S7	800000	2400000	12000	16	256	208488077	2.80	1.40

Table 2: This table presents the weak scaling1 (the number of vertices < the number of hyperedges) experimental setup, total NNZs generated from each experiment, and output files information.

Experiment Name	Vertices	Hyperedges	Clusters	Compute Node	MPI Processes	Total NNZs	Bipartite Output File Size (in GB)	Hyperedge Output File Size (in GB)
W1	60000	200000	1000	1	16	2388362	0.03	0.02
W3	200000	600000	3000	2	32	16042887	0.20	0.10
W5	400000	1200000	6000	4	64	56189594	0.74	0.37
W7	800000	2400000	12000	16	256	208488077	2.80	1.40
W9	1600000	4800000	24000	64	1024	786869455	12	5.40

Table 3: This table presents the weak scaling2 (the number of vertices > the number of hyperedges) experimental setup, total NNZs generated from each experiment, and output files information.

Experiment Name	Vertices	Hyperedges	Clusters	Compute Node	MPI Processes	Total NNZs	Bipartite Output File Size (in GB)	Hyperedge Output File Size (in GB)
W2	200000	60000	300	1	16	5437372	0.06	0.03
W4	600000	200000	1000	2	32	23855054	0.30	0.16
W6	1200000	400000	2000	4	64	71276112	0.94	0.48
W8	2400000	800000	4000	16	256	239667974	3.30	1.70
W10	4800000	1600000	8000	64	1024	859300720	13	6.30

to solve a fixed problem faster. In weak scaling, the problem size increases proportionally as the number of processes increase [4]. Weak scaling tries to solve increasingly larger problems. Suppose we have a problem size x, which is executing in the main memory with p processes. In the above scenario, the strong scaling takes $\frac{x}{p}$ memory per process [20]. However, weak scaling takes x memory per process. Strong scaling addresses the question, "How much does parallel execution decrease the run time of a problem?" [1]. Weak scaling addresses the question, "While problem size increases, does the application solve the problem in a feasible time with additional processes" [1, 12].

Experimental Setup

We conducted strong and weak scaling experimentations with our HyGen hypergraph generator. For strong scaling, we generated hypergraphs with 800,000 vertices, 2,400,000 hyperedges, and 12,000 clusters. We used MPI processes 16, 32, 48, 64, 96, 128, and 256 for parallel execution. Our strong scaling input parameters (experiment names, vertices, hyperedges, clusters, compute nodes, and MPI processes) are available in Table 1. We divided weak scaling experimentations into two groups. In the first group, the number of vertices are smaller than the number of hyperedges. We called it "Weak Scaling1." In the second group, the number of vertices are higher than the number of hyperedges, named "Weak Scaling2." Table 2 and 3 present input parameters for "Weak Scaling1" and "Weak Scaling2." We generated five hypergraphs each for "Weak Scaling1" and "Weak Scaling2."

In "Weak Scaling 1," the first hypergraph (experiment name W1) was generated with 60,000 vertices, 200,000 hyperedges, and 1,000 clusters. Sixteen MPI processes were used to conduct the experiment. We used 200,000 vertices, 600,000 hyperedges, 3,000 clusters, and 32 MPI processes for the second experiment (named W3). The third hypergraph (experiment name W5) was created with 400,000 vertices, 1,200,000 hyperedges, and 6,000 clusters, with 64 MPI processes used. We created a hypergraph with 800,000 vertices, 2,400,000 hyperedges, and 2,000 clusters, and used 256 MPI

processes in experiment W7. Finally, the hypergraph generated in experiment W9 includes 24,000 clusters, 4,800,000 hyperedges, and 1,600,000 vertices, with 1,024 MPI processes employed.

In "Weak Scaling 2," the first hypergraph (experiment name W2) contained 200,000 vertices, 60,000 hyperedges, and 300 clusters, with 16 MPI processes executed. The second hypergraph (experiment name W4) contains 600,000 vertices, 200,000 hyperedges, and 1,000 clusters, and was executed with 32 MPI processes. The third hypergraph (experiment name W6) comprised of 1,200,000 vertices, 400,000 hyperedges, and 2,000 clusters. The third hypergraph experiment was conducted with 64 MPI processes. The fourth hypergraph experiment was executed with 256 MPI processes, which generated a hypergraph with 2,400,000 vertices, 800,000 hyperedges, and 4,000 clusters. The fifth hypergraph (experiment name W10) contains 4,800,000 vertices, 1,600,000 hyperedges, and 8000 clusters. We employed 1,024 MPI processes to generate the fifth hypergraph.

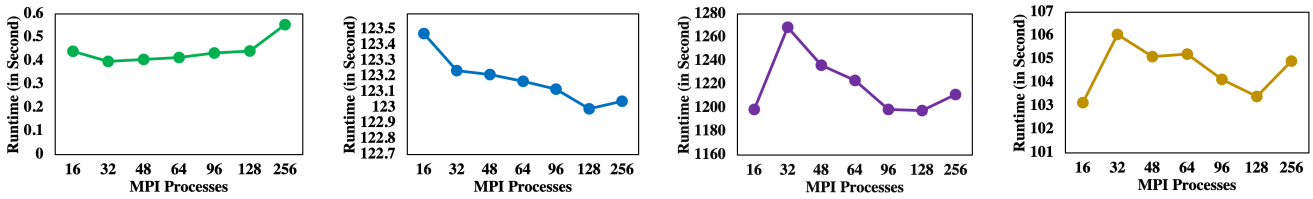
Results

We conducted seven strong scaling experiments (experiment names: S1-S7). Each experiment generated more than 208 million NNZs. The total number of NNZs generated by each experiment is provided in Table 1 (see "Total NNZs" column). We present strong scaling experimental results in Figure 4. Figure 4(a) shows strong scaling communication times. The first experiment (S1) with 16 MPI processes took 0.44 second as a communication time. The second experiment (S2) took a little less time (0.40 second) than the first experiment. The third (S3), fourth (S4), fifth (S5), and sixth (S6) experiments took more communication times compared to the second experiment. The third experiment took $\approx$ 0.41 second, the fourth experiment took 0.41 second, the fifth experiment took 0.43 second, and the sixth experiment took 0.44 second. The last experiment (S7) with 256 MPI processes had the highest communication time of 0.55 second. Although communication time available in Figure 4(a) vary somewhat, the differences are not statistically significant.

We report strong scaling computation times in Figure 4(b), which shows that computation times decreases as more resources (MPI processes) are used for execution. The first experiment (S1) with 16 MPI processes had the highest computation time (123.47 seconds). The sixth experiment (S6), with 128 MPI processes, took the lowest computation time (122.99 seconds). The second experiment, with 32 MPI processes took, 123.24 seconds. The third experiment, with 48 MPI processes, took 123.21 seconds. The fourth experiment, with 64 MPI processes, needed 123.17 seconds. The fifth experiment, including 96 MPI processes, had 123.12 seconds. Finally, the seventh experiment, with 256 MPI processes, took 123.04 seconds.

We present strong scaling IO time in Figure 4(c) and 4(d). Here, IO time means hypergraph file writing time. Figure 4(c) displays the bipartite IO time. The figure shows that the second experiment (S2), with 32 MPI processes, took the highest bipartite IO time of 1,268.82 seconds. The sixth experiment (S6), with 128 MPI processes, took the lowest bipartite IO time of 1,197.81 seconds. Figure 4(c) shows that the difference between the first (S1) and second (S2) experiments' bipartite IO times is large. Next, for the third (S3), fourth (S4), fifth (S5), and sixth (S6) experiments, the bipartite IO time decreases compared to the second experiment. We observe a slight bipartite IO time increase in the seventh (S7) experiment. Figure 4(d) shows that all the experiments' (S1-S7) hyperedge IO times are significantly lower than the bipartite IO time mentioned in Figure 4(c). This is because hyperedge representation contains a lower number of rows in the incidence matrix compared to the bipartite representation. As in Figure 4(c), Figure 4(d) also shows that the second experiment (S2), with 32 MPI processes, took the highest hyperedge IO time (106.06 seconds) and that the sixth experiment (S6), with 128 MPI processes, took the lowest hyperedge IO time (103.41 seconds). Both Figure 4(c) and Figure 4(d) show the disparities among the experiments' (S1-S7) runtime. Currently, we are investigating this time discrepancy issue. We believe by applying load balancing techniques, we will be able to improve the strong scaling results in the future. All the strong scaling experiments' (S1-S7) bipartite output files are $\approx$ 2.80 GB in size (see Table 1, the "Bipartite Output File Size (in GB)" column) and hyperedge output files are $\approx$ 1.40 GB in size (see Table 1, the "Hyperedge Output File Size (in GB)" column).

We report the weak scaling1 experimentation results in Figure 5, which shows that the communication 5(a), computation 5(b), bipartite IO 5(c), and hyperedge IO 5(d) times increased across the experiments (W1, W3, W5, W7, and W9). Figure 5(a) shows that the first experiment (W1), with 16 MPI processes, took the lowest communication time of 0.012 second and that the fifth experiment (W9), with 256 MPI processes, took the highest time of 2.76 seconds. Although we observed an increase across all the experiments' communication times, most of the experiments' (W1, W3, and W5) communication times were less than $\frac{1}{2}$ second, the fourth experiment's (W7) communication time was a little over $\frac{1}{2}$ second, and the last experiment's (W9) communication time was less than 3 seconds. Figure 5(b) illustrates that the first experiment (W1), with 16 MPI processes, took the minimal computation time of 0.78 second and that the last experiment (W9), with 1,024 MPI processes, took the maximal computation time of 240.78 seconds. The first three experiments' (W1, W3, and W5) computation times were less than a minute. The fourth (W7) experiment's computation time was slightly



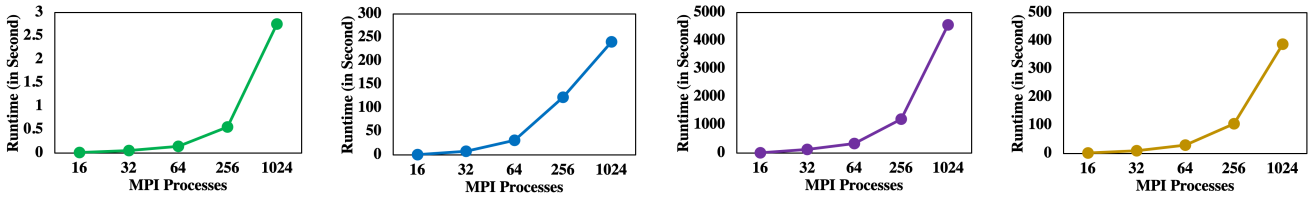
(a) Strong scaling: Communication time.

(b) Strong scaling: Computation time.

(c) Strong scaling: Bipartite IO time.

(d) Strong scaling: Hyperedge IO time.

Figure 4: Strong scaling experimentation results.



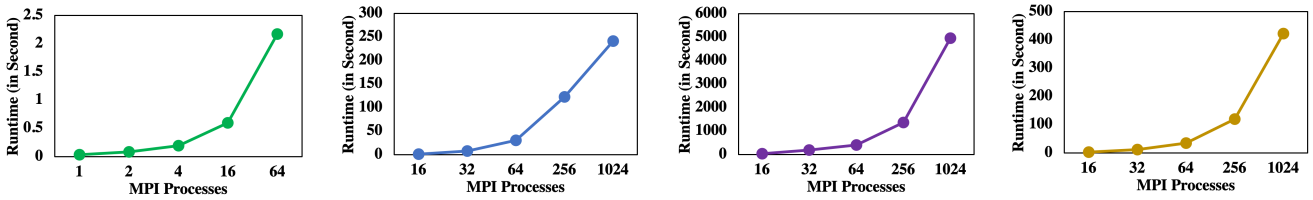
(a) Weak scaling1: Communication time

(b) Weak scaling1: Computation time

(c) Weak scaling1: Bipartite IO time

(d) Weak scaling1: Hyperedge IO time

Figure 5: Weak scaling1 experimentation results.



(a) Weak scaling2: Communication time

(b) Weak scaling2: Computation time

(c) Weak scaling2: Bipartite IO time

(d) Weak scaling2: Hyperedge IO time

Figure 6: Weak scaling2 experimentation results.

more than 2 minutes, and the final experiment's computation time was a little over 4 minutes. Figure 5(c) demonstrates that 16 MPI processes (W1) took the lowest bipartite IO time of 14.42 seconds and that 1,024 MPI processes (W9) took highest bipartite IO time of 4,559.14 seconds. We observed that weak scaling1's hyperedge IO times are significantly lower than the bipartite IO times. Figure 5(d) shows that the first experiment (W1), with 16 MPI processes, took the lowest hyperedge IO time (1.62 seconds) and fifth experiment (W9) with 1,024 MPI processes took highest hyperedge IO time (388.49 seconds). All the experiments' hyperedge IO times were less than 7 minutes. We provide the weak scaling1 experiments' (W1, W3, W5, W7, and W9) NNZs as well as bipartite output file size and hyperedge output file size information in Table 2. The largest hypergraph generated from weak scaling1 experiments contains 786,869,455 NNZs.

Similar to the weak scaling1 experiments, the weak scaling2 experiments' communication 6(a), computation 6(b), bipartite IO 6(c), and hyperedge IO 6(d) times are increased across the experiments (W2, W4, W6, W8, and W10). The experiment W10 took the maximal communication time of 2.17 seconds. Most of the experiments' (W2, W4, W6, and W8) communication times were less than 1 second (see Figure 6(a)). Also, the experiment W10 took the highest computation time of 241.12 seconds. Most of the experiments' (W2, W4, W6, and W8) computation times were less than 3 minutes (see Figure 6(b)). Figure 6(c) shows that the first experiment (W2), with 16 MPI processes, took the lowest bipartite IO time (32.24 seconds) and that the fifth experiment (W10), with 256 MPI processes, took the highest bipartite IO time (4,963.60 seconds). The experiment with 1,024 MPI processes (W10) had the highest hyperedge IO time, which

was 421.70 seconds. Most of the weak scaling2 experiments' (W2, W4, W6, and W8) hyperedge IO times were less than 2 minutes.

4 CONCLUSION AND FUTURE WORK

In this paper, we discussed a scalable parallel hypergraph generation framework called HyGen. First, we provided a brief hypergraph background in the "Introduction" section. Next, we presented a detailed HyGen architecture discussion in the "HyGen Architecture" section, which explains HyGen parameters as well as hypergraph generation, permutation, and representations. Finally, we presented our experimental results in the "Experimental Evaluation" section. Only a limited number of parallel hypergraph generators are available to this date and none can match the scalability features of HyGen. We believe HyGen will be useful for understanding the interesting properties of very large-scale hypergraphs. In the future, we will implement a machine learning-based hypergraph generator, that will learn the structures of real-world hypergraphs to generate realistic massive-scale hypergraphs.

ACKNOWLEDGEMENTS

Support for this work was provided by the United States Department of Defense. We used resources of the Computational Research and Development Programs and the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

REFERENCES

[1] 2020. High Performance Computing for Science and Engineering. https://www.cse-lab.ethz.ch/wp-content/uploads/2018/11/amdahl_gustafson.pdf. [Online; accessed 2020-04-03].

[2] 2020. HyperX. https://github.com/jinhuang/hyperx. [Online; accessed 2020-04-03].

[3] 2020. Introduction to Hypergraphs. https://pt.slideshare.net/IsmaelAAliShilani/introduction-to-hypergraphs/23. [Online; accessed 2020-04-03].

[4] 2020. Performance Scaling. https://www.archer.ac.uk/training/course-material/2016/12/mpi_scaling_manc/Slides/Scaling.pdf. [Online; accessed 2020-04-03].

[5] 2020. Rhea Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/olcf-resources/compute-systems/rhea/. [Online; accessed 2020-04-01].

[6] 2020. Rhea User Guide. https://docs.olcf.ornl.gov/systems/rhea_user_guide.html. [Online; accessed 2020-04-17].

[7] 2020. SAGE - Hypergraph Generators. http://doc.sagemath.org/html/en/reference/graphs/sage/graphs/hypergraph_generators.html. [Online; accessed 2020-04-03].

[8] Anirban Banerjee, Arnab Char, and Bibhash Mondal. 2017. Spectra of general hypergraphs. *Linear Algebra Appl.* 518 (2017), 14–30.

[9] Hasanuzzaman Bhuiyan, Maleq Khan, and Madhav Marathe. 2017. Efficient algorithms for assortative edge switch in large labeled networks. In *Proceedings of the 25th High Performance Computing Symposium*. Society for Computer Simulation International, 2.

[10] Stephen E Fienberg and Stanley S Wasserman. 1981. Categorical data analysis of single sociometric relations. *Sociological methodology* 12 (1981), 156–192.

[11] Debarghya Ghoshdastidar and Ambedkar Dukkipati. 2015. A provable generalized tensor spectral method for uniform hypergraph partitioning. In *International Conference on Machine Learning*. 400–409.

[12] SM Shamimul Hasan, Drew Schmidt, Ramakrishnan Kannan, and Neena Imam. 2019. A Scalable Graph Analytics Framework for Programming with Big Data in R (pbdR). In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 4783–4792.

[13] Benjamin Heintz and Abhishek Chandra. 2014. Beyond graphs: toward scalable hypergraph analysis systems. *ACM SIGMETRICS Performance Evaluation Review* 41, 4 (2014), 94–97.

[14] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. 1983. Stochastic blockmodels: First steps. *Social networks* 5, 2 (1983), 109–137.

[15] Héctor J Fraire Huacuja and Norberto Castillo-García. 2016. Optimization of the Vertex Separation Problem with genetic algorithms. In *Handbook of Research on Military, Aeronautical, and Maritime Logistics and Operations*. IGI Global, 13–31.

[16] Tamara G Kolda, Ali Pinar, Todd Plantenga, and Comandur Seshadhri. 2014. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing* 36, 5 (2014), C424–C452.

[17] Kiran Lakkaraju and Gita Sukthankar. 2014. *Synthetic Generators to Simulate Social Networks.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[18] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[19] Mark EJ Newman. 2006. Modularity and community structure in networks. *Proceedings of the national academy of sciences* 103, 23 (2006), 8577–8582.

[20] David A Patterson and John L Hennessy. 2016. *Computer Organization and Design ARM Edition: The Hardware Software Interface.* Morgan kaufmann.

[21] Brenda Praggastis, Dustin Arendt, Cliff Joslyn, Emilie Purvine, Sinan Aksoy, and Kyle Monson. [n. d.]. pnnl/HyperNetX. ([n. d.]).

[22] Michael M Wolf, Alicia M Klinvex, and Daniel M Dunlavy. 2016. Advantages to modeling relational data using hypergraphs versus graphs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[23] Songyang Zhang, Zhi Ding, and Shuguang Cui. 2019. Introducing Hypergraph Signal Processing: Theoretical Foundation and Practical Applications. (2019).