Chapel HyperGraph Library (CHGL)

Louis Jenkins*, Tanveer Bhuiyan†, Sarah Harun†, Christopher Lightsey†, David Mentgen†,
Sinan Aksoy*, Timothy Stavenger* Marcin Zalewski*, Hugh Medal†, and Cliff Joslyn*
louis.jenkins@pnnl.gov, tb2038@msstate.edu, sh2364@msstate.edu, chrisl@dasi.msstate.edu, dam530@msstate.edu,
sinan.aksoy@pnnl.gov, timothy.stavenger@pnnl.gov, marcin.zalewski@pnnl.gov, hmedal@ise.msstate.edu, cliff.joslyn@pnnl.gov

* Pacific Northwest National Laboratory, Seattle, Washington, USA.
† Mississippi State University, Mississippi State, Mississippi, USA.

***Abstract*—We present the Chapel Hpergraph Library (CHGL), a library for hypergraph computation in the emerging Chapel language. Hypergraphs generalize graphs, where a hypergraph edge can connect any number of vertices. Thus, hypergraphs capture high-order, high-dimensional interactions between multiple entities that are not directly expressible in graphs. CHGL is designed to provide HPC-class computation with high-level abstractions and modern language support for parallel computing on shared memory and distributed memory systems. In this paper we describe the design of CHGL, including first principles, data structures, and algorithms, and we present preliminary performance results based on a graph generation use case. We also discuss ongoing work of codesign with Chapel, which is currently centered on improving performance.**

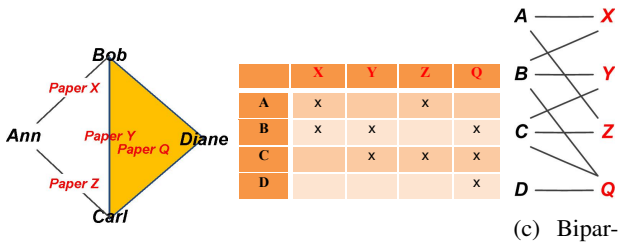
I. Introduction and Background

Many problems in data analytics involve rich interactions amongst multiple entities, for which graph representations are commonly used. High order (high dimensional) interactions, which abound in cyber and social networks, can only be represented in graphs as highly inefficiently coded, "reified" labeled subgraphs. Lacking multi-dimensional relations, it is hard to address questions of "community interaction" in graphs: e.g., how is a collection of entities A connected to another collection B through chains of other communities; where does a particular community stand in relation to other communities in its neighborhood?

Hypergraphs [1] are an extension of the concept of graphs that addresses such concerns. Hypergraphs generalize graphs by allowing edges to connect any number of vertices. More concretely, an undirected hypergraph H is a pair $H = (V, E)$ where V is a set of vertices as in a graph, and $E \subseteq 2^V \setminus \{\emptyset\}$ is a set of **hyperedges**.

Hyperedges capture multi-dimensional relations explicitly, allowing more direct reasoning and algorithms. For example, Fig. 1 shows an author-paper hyper-network involving four authors and four papers. The hypergraph directly represents both the two-author papers as graph edges (2-hyperedges), but also the three-authored paper as a 3-hyperedge (solid triangle). All graphs are hypergraphs, but the proper hypergraph is necessary in this case, as e.g. the $\{B, D\}$ 2-edge can only indicate that Bob and Diane were on *some* paper, but not that that paper also involved Carl. Hypergraphs are directly coded by their incidence matrices (as shown), which, unlike graph incidence matrices, may have other than two entries per column. They are also broadly equivalent to a bipartite graph with $|E| + |V|$ vertices, as shown.

While hypergraphs offer mathematical clarity and support reasoning, they have not received much attention in the software



(a) Hypergraph (b) Incidence matrix (c) Bipartite graph

Fig. 1: Hypergraph, incidence matrix, and bipartite graph representations of the same data.

engineering community at large, and especially in the HPC community. There is a distinct lack of HPC publications and implementations for hypergraphs, as compared to the abundance of work on graphs. In this paper, we introduce our early efforts to address that gap in a modern HPC environment. Specifically, we introduce the **Ch**apel **Hyper**g**r**aph **L**ibrary (CHGL), a library for hypergraph computation in the emerging Chapel programming language [2], [3].

CHGL provides a highly abstract interface for implementation of hypergraph algorithms. Currently, the primary functionality implemented using CHGL data structures is hypergraph generation. Hypergraph generation serves far-ranging purposes across scientific disciplines. In particular, generative graph models are used for benchmarking, provide null models for algorithm testing, and can create surrogate graphs to protect anonymity of private or restricted data. CHGL provides the Erdős-Rényi, the Chung-Lu (CL) [4], and the Block Two-Level Erdős-Rényi (BTER) [5], [6] generation models.

We chose to implement CHGL in Chapel, both to leverage it's unique features but also to test their applicability in a data-driven, fine-grained application that is notoriously difficult to implement efficiently. Chapel provides high-level built-in concepts and mechanisms for data parallelism, task parallelism, and concurrency. One of the most important features for CHGL are highly abstract array data structure with generic views on data through domains, which provide a common interface to a variety of local and global data layouts, including dense, sparse, and associative data with library-provided distributions such as block or cyclic. Majority of the low-level aspects of that generic view can be controlled by user-defined implementations (in Chapel) in the spirit of *multiresolution* philosophy, where low-level details can be added to an abstract implementation to improve performance. Other important features CHGL relies on are the task and data parallelism mechanisms, such as `forall` loops, synchronous and asynchronous tasks (`begin` and

`cobegin` statements), and locality control through Chapel's data-driven `on` clauses and locale framework. Finally, Chapel also provides a set of modern basic programming features such as type inferencing and generic programming. In section Section II, we describe our experience in using Chapel features to design and implement abstract interfaces, highlighting the usefulness of the Chapel array abstractions. In section Section IV, we evaluate the performance of our code, and in Section V we discuss the ongoing work in CHGL and the codesign and interaction with Chapel.

While CHGL provides the basic data structures and algorithms for hypergraphs, the driving application in this paper is graph generation, which is discussed in Section III. Graph generation serves far-ranging purposes across scientific disciplines. In particular, generative graph models are used for benchmarking, provide null models for algorithm testing, and can create surrogate graphs to protect anonymity of private or restricted data. Throughout these applications, it is ideal to have generators that are easily tunable, require compact inputs, produce graphs with varied structural properties encountered in real data, and—perhaps most importantly—are scalable.

We conduct a scalability study of three hypergraph generators implemented in CHGL: hypergraph Erdős Rényi (ER), hypergraph Chung-Lu (CL) [4], and the recently introduced hypergraph version of the Block Two-Level Erdős Rényi (BTER) model [6]. These models take compact inputs that can be easily sampled from real data or generated artificially, and can be easily tuned to output hypergraph data possessing a wide range of properties, at scale. Furthermore, taken as a suite, each model provides successively more control over hypergraph structure than the previous, providing the flexibility to choose different tiers of structural nuance for the generated data. We use hypergraph statistics implemented in CHGL to evaluate the generative models, providing an end-to-end generation and evaluation framework.

II. CHGL Design and Implementation

The overall goal of CHGL is to provide a high-level modern library of functionality for hypergraphs with HPC performance. CHGL's primary design goals are the following:

- **Genericity**: CHGL is based on the principles of generic programming [7], providing well-defined interfaces independent of particular data structures and generic algorithms defined in terms of these interfaces. Genericity supports interfaces that are minimal, durable, and designed to cover wide classes of data structures, and it provides reusable algorithms that can be written once for many input types.
- **Performance**: CHGL is specifically intended to provide strong performance in modern HPC environments, from multi-core to distributed-memory settings. Specifically, CHGL relies on Chapel abstractions for performance, and the purpose of our effort is in part to explore, test, and contribute to these abstractions in the context of our applications.
- **Usability**: While genericity and simplicity are fundamental, CHGL is also easy to use. Our philosophy is to provide multiple interfaces where the simplest interfaces can be used by beginners, which then can be extended gradually by providing more parameters.

```
class HyperGraph {
// member functions
iter vertices() : vDescType;
iter edges() : eDescType;
proc numVertices : int(64);
proc numEdges : int(64);
iter forEachVertexDegree() : (vDescType, int(64));
iter forEachEdgeDegree() : (eDescType, int(64));
proc vertexDegrees();
proc edgeDegrees();
proc addInclusion(vertex : vDescType,
                  edge : eDescType);
proc hasInclusion(vertex : vDescType,
                  edge : eDescType);
proc inclusions(vertex : vDescType) : eDescType;
proc inclusions(edge : eDescType) : vDescType;
proc numNeighbors(v : vDescType) : int(64);
proc numNeighbors(e : eDescType) : int(64);
proc getVertex(v : vDescType) : MetaData;
proc getEdge(e : eDescType) : MetaData;
...
```

Fig. 3: Part of the interface of the `HyperGraph` CHGL class.

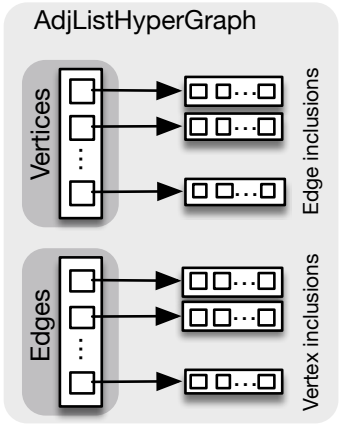


Fig. 2: Adjacency list hypergraph data structure.

The three principles are synergistic. Chapel aims to support both performance and abstraction through its easy-to-use generic programming and compile-time metaprogramming mechanisms. CHGL exploits these mechanisms with intention of pushing the limits of Chapel, feeding requirements and feature requests back to the Chapel language. In that sense, CHGL is meant to both utilize Chapel for its benefit, and also to exist in codesign with Chapel, improving it in the process. In this section, we first provide simple examples of CHGL uses, and then we briefly discuss more advanced elements of CHGL design and implementation.

In CHGL, the creation of hypergraphs is simple and intuitive. For example, creating a medium-sized hypergraph that is cyclicly distributed over a small subset of the *locales*, or compute nodes in the cluster, relies on the Chapel framework for distributions:

```
const numVertices = 1024 * 1024;
const numEdges = 2048 * 1024;
const cyclicDom = new Cyclic(startIdx = 1,
  targetLocales = Locales[4..8]);
var graph = new HyperGraph(numVertices, numEdges,
  cyclicDom);
```

CHGL provides a generic interface that relies on clearly specified hypergraph interfaces that span an open-ended class of hypergraph types that model the interface. Figure 3 lists a portion of the generic hyper graph interface in CHGL (in a simplified form). For example, a modifiable hypergraph

provides a method for including vertices into edges (remember, edges in hypergraphs can contain any number of vertices), using the method `addInclusion`, and vertex degrees of any graph can be iterated over (using Chapel iterators) using the `forEachVertexDegree` method. Figure 2 illustrates the layout of CHGL's main hypergraph data structure, the adjacency list hypergraph `AdjListHyperGraph`. This data structure consists of two outer Chapel arrays, which may be distributed, for vertices and edges and of inner non-distributed arrays for every vertex and edge inclusions. Every inclusion is stored in two directions, both at the included vertex and at the including edge, resulting in storage of the hypergraph and its dual. Thanks to Chapel array capabilities, this data structure supports both static and dynamically growing hypergraphs. Currently, this is the only hypergraph data structure available in CHGL, but it is versatile and widely applicable due to the flexibility of Chapel arrays (e.g., both local and distributed hypergraphs can be represented, different layouts can be used for data, etc.).

A simple task to perform on the graph constructed above is to extract vertex degrees:

```
var vertexDegrees : [graph.verticesDomain()] int;
forall (degree, vertex) in zip(vertexDegrees,
  graph.getVertices()) {
    degree = graph.numNeighbors(vertex);
}
var totalVertexDegrees = + reduce vertexDegrees;
```

Even though the graph is distributed cyclically between different locales of a distributed machine, the vertices can be extracted with a single `forall` loop, which can then be used to query more information from the graph. By using the `zip` operator, we get to request that more than one data source, providing they are of the same size and shape, have their yielded results tupled together. As the `vertexDegrees` array shares the same domain as the hypergraph's vertices, we can obtain both a reference and the descriptor for the respective vertex. Note that the array of vertex degrees gets reduced in a single step, using the Chapel reduce syntax. If only finding the sum of all vertex degrees is desired, a more efficient variant of this algorithm can be produced via Chapel's reduce-intents:

```
var totalVertexDegrees = 0;
forall vertex in graph.vertices() with
  (+ reduce totalVertexDegrees) {
    totalVertexDegrees += graph.numNeighbors(vertex);
}
```

In the above snippet, Chapel is informed that `totalVertexDegrees` is intended as a reduction using the plus operation. Given this hint, Chapel can optimize the global reduction in the same way an advanced programmer could using low-level distributed programming constructs.

Each of the interface functions are error-friendly in that by using Chapel's rich support for generics and overloading of functions and methods, the library uses the `compilerError` and `compilerWarning` to provide user-friendly error messages. For example:

```
// 5% probability
const p = 0.05;
// p * |V| * |E|
const numInclusions = graph.numVertices
                      * graph.numEdges * p;
// Spawn a task on each other locale
coforall loc in Locales do on loc {
  var rng : makeRandomStream(real);
  forall 1 .. numInclusions / numLocales {
    var vertex =
      rng.getNext(0, graph.numVertices() - 1);
    var edge = rng.getNext(0, graph.numEdges() - 1)
    graph.addInclusion(vertex, edge);
  }
}
```

Fig. 4: Simplified Erdős-Rényi algorithm.

```
inline proc numNeighbors(other) {
  compilerError("'numNeighbors(", other.type
      : string, "') is not supported...\n",
    "Require argument of type ", vDescType
      : string, " or ", eDescType : string);
}
```

By making the function inline, the compiler error message will show the location that the user has provided the bad argument. This type of compile-time error checking and custom error messages makes it easy for the user to learn from their mistakes, and even better, where and how to fix it.

In Fig. 4, we show a simplified implementation of the Erdős Rényi CHGL algorithm (see Section III). In this example, probability `p` is the probability that there is an inclusion of a vertex in an edge. Multiplying `p` by the number of vertices and edges gives the expected number of inclusions to be added. The inclusions can be added in an embarassingly parallel way. However, this straightforward implementation needs further steps to become efficient (see Section V).

III. Use Case: Graph Generation

In this work, we choose graph generation for our use case. Given the utility of graph generators, it is unsurprising there are plethora of models and an extensive surrounding literature. In addition to the Stochastic Kronecker Model (SKG) used for the Graph500 supercomputer benchmark [8], the Chung-Lu (CL) [4] and Block Two-Level Erdős Rényi (BTER) [5] models have also received attention for having scalable, parallel implementations [9], [10] and producing graphs with realistic heavy-tailed vertex degree distributions [11], and, in the case of BTER, community structure. In CHGL, we implement three generators: hypergraph Erdős Rényi, hypergraph Chung-Lu, and recently introduced hypergraph BTER model [6]. Below, we briefly describe each of these models:

- **Hypergraph Erdős-Rényi (ER)**. The user specifies three scalar parameters: the number of vertices and hyperedges and the vertex-hyperedge inclusion probability, p, where for each possible vertex-hyperedge pair, the probability that the vertex is assigned to that hyperedge is p. In practice, however, it is too computationally costly to consider all possible inclusions, and several efficient alternatives [12],

[13] have been proposed. In CHGL, we feature both this "naive", as well as a "fast ER" generator in which the desired number of vertex-hyperedge inclusions are determined by sampling the vertices and hyperedges of each inclusion uniformly at random, with replacement. To control for possible duplicate inclusions, CHGL's fast ER generator also includes an optional "coupon-collectors adjustment" (see [10]) that ensures, in expectation, the desired number of unique inclusions are realized in the output graph.

- **Hypergraph Chung-Lu (CL)**. The user specifies a desired vertex degree and edge cardinality sequence. In this model, which is generalization of ER, the probability a vertex belongs to an edge is proportional to the product of their desired degree and edge cardinality; consequently, each vertex and hyperedge achieves these user-specified values in expectation. As with ER, we include a fast hypergraph CL implementation, as described in [6], in which the vertex and hyperedge of each inclusion is chosen according to a *weighted* random sampling with replacement.
- **Hypergraph Block-Two Level Erdős-Rényi (BTER)**. In addition to the same parameters as Chung-Lu, the user also specifies desired *metamorphosis coefficients* for the hypergraph. As introduced and explained in [6], metamorphosis coefficients are numerical measures of hypergraph community structure shown to be prevalent in real data. The BTER algorithm is designed to output a graph whose degree distributions and metamorphosis coefficients both approximately match the inputs. As the details of the BTER algorithm are complicated, the interested reader may refer to [6] for a more formal description.

Using CHGL, we generated instances of hypergraphs for each of the above models by extracting each model's inputs from real hypergraph data. In particular, we used the well-known "condMat" [14] author-paper network based on preprints posted in the Condensed Matter section of the arXiv e-Print repository. This dataset contains 16,726 authors, 22,016 preprints, and 58,595 author-paper inclusions, and exhibits structurally rich properties, such as heavy tailed degree distributions, and tight-knit community structure.

Figure 5 plots some of CHGL's hypergraph analytics, which we use to compare structural properties of instances of CHGL's generated hypergraphs with those of the real condMat dataset. These plots serve to verify the CHGL implementation generates hypergraphs exhibiting key properties consistent with the abstract ER, CL, and BTER models, and also highlight the breadth of structural nuance achievable through these generators. For instance, while the ER graph possesses about same average edge cardinality and vertex degree as condMat, Fig. 5 (left) shows that the degree distributions are dissimilar, a known shortcoming of ER. In contexts where accurately modeling vertex degree and edge cardinality distributions is important, CL is a more appropriate generator, as confirmed by the closer match in Fig. 5. Nonetheless, Figure 5 (right) shows that CL is not able to reproduce condMat's community structure, as measured by metamorphosis coefficients. However, the same plot shows that BTER generator offers more accurate community structure, which affords the user the option to add another layer of realism to the generated data. In Section IV, we present performance scaling experiments for these generators.

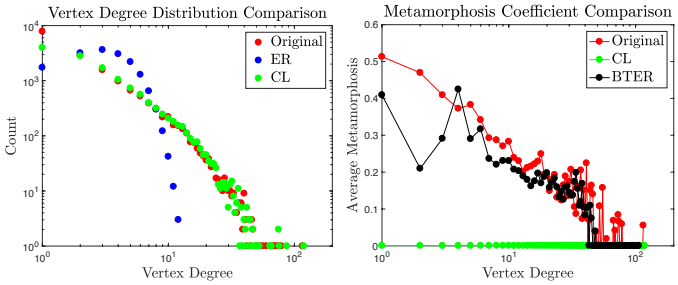
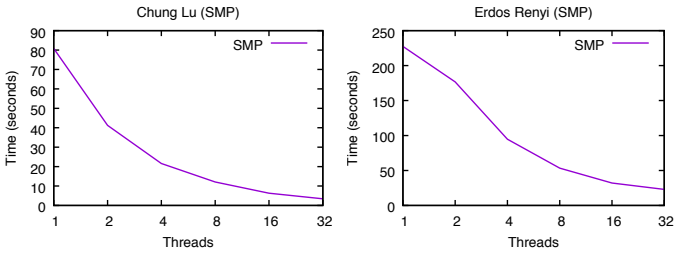
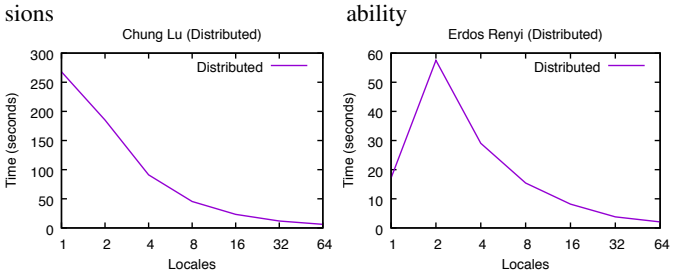


Fig. 5: Structural comparison of ER, CL, and BTER output hypergraphs vs the original condMat dataset.



(a) Chung Lu (SMP) - 100K Vertices, 200K Edges, 841,088 Inclusions

(b) Erdős-Rényi (SMP) - 100K Vertices, 100K Edges, 1% Probability



(c) Chung Lu (Distributed) - 1M Vertices, 2M Edges, 9,940,947 Inclusions

(d) Erdos Renyi (Distributed) - 100K Vertices, 100K Edges, 1% Probability

Fig. 6: SMP and distributed execution weak scaling for the ER and CL graph generation algorithhms.

IV. Evaluation

Performance benchmarks are performed on Intel Broadwell compute nodes of a Cray-XC50 cluster. For the communication layer, which is the run-time abstraction layer responsible for sending, receiving, and handling active messages, we use Cray's uGNI with 16MB hugepages; we utilize the uGNI communication layer over the GASNet communication layer as it is specialized for Cray's hardware. For the tasking layer, which is the abstraction layer responsible for the creation, scheduling, and management of tasks, which are coroutines that are multiplexed on top of threads, we use qthreads. For the memory management layer, the abstraction layer that is responsible for handling memory management, we use jemalloc. The benchmarks have been tested using Chapel pre-release version 1.18.0, hash 55106c1d2c.

We present a set of benchmarks for the hypergraph generation algorithms: Erdős-Rényi, Chung-Lu, and Block Two-level

Erdős-Rényi (BTER). In each set we provide a brief high-level description of the benchmark, and we display and analyze both shared-memory and distributed performance in all sets except BTER, which is still in its prototype stage and only performs well in a shared-memory environment. In the shared-memory benchmarks, we allocate a single compute node with cores that are a power of 2 up to 32 cores, and so the performance results are be presented in a logarithmic base-2 scale.[1] For distributed memory, we allocate compute nodes, each with 44 cores, by powers of 2 up to 64 nodes, and so the performance results are also be presented in a logarithmic base-2 scale. In both shared-memory and distributed benchmarks we compile the benchmarks with the *–fast* flag, which causes the Chapel compiler to perform all optimization on the Chapel source code, turn off any and all safety run-time checks such as out-of-bounds checking, and runs the generated C code, which Chapel gets compiled down to, with the highest optimization settings for the C compiler used. In the shared-memory benchmarks, we compile with the *–local* flag so that the Chapel compiler never injects run-time checks for whether a memory access is remote before accessing it, which occurs when the compiler is unable to determine such information at compile-time.

A. Erdős-Rényi (ER)

We begin by spawning a task on each logical core on each locale, where each task then computes a random vertex and hyperedge to create an inclusion for. Computing the random vertex and edge is done without involving communication with other tasks or locales, but creating the inclusion may involve communication if either the vertex or edge is hosted in remote memory. Random sampling is performed by using a task-local random number generator and obtaining a random number within a valid range for vertices and hyperedges. The graph being generated contains 100K vertices and 100K hyperedges, and the probability of there being an edge created between any pair of vertex and hyperedge is 1%, hence a total of 100M edges are generated.

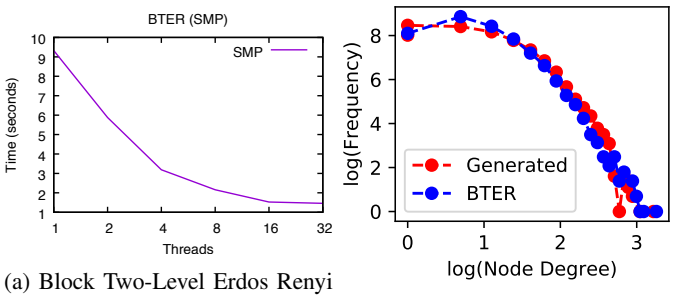
In the shared-memory benchmark which is shown in Fig. 6b, ER scales in a linear fashion as we increase the number of cores by a power of two, as would be expected. In the distributed-memory benchmark which is shown in Fig. 6d, ER shows a rather large increase in overall execution time at 2 locales, which is due to introducing the large but constant overhead of adding communication to the program. As the random sampling of vertices and edges are entirely local and relatively fast, the overhead is more noticeable here. Thanks to aggregation, the execution time decreases as we add more than 2 locales and we pass single-locale execution starting at 8 locales, and at 64 locales the performance is approximately 8x faster.

B. Chung-Lu (CL)

We begin by spawning a task on each logical core on each locale, where each task then computes a random vertex and hyperedge to create an inclusion for. However unlike ER, CL performs a weighted random sampling from a table that is formed by taking the prefix sum (`scan` in Chapel) of the normalized desired vertex degree and edge cardinality sequences. The current algorithm is naive and is heavy-weight, effectively being a linear operation per random sample. Each task still computes the weighted random samples without communication; the random number is generated using a task-local random number generation as it does in ER, and the probability table is duplicated on each locale, hence only datasets that can fit on an individual locale can be sampled. Both the shared-memory and the distributed benchmarks sample the same LiveJournal[15] dataset, but we sample 100K vertices and 200K hyperedges to create 841,088 edges for shared-memory, and 1M vertices and 2M hyperedges to create 9,940,947 edges for distributed.

In the shared-memory benchmark which is shown in Fig. 6a, CL scales linearly as expected. In the distributed memory benchmark, which is shown in Fig. 6c, CL scales linearly including at two locales due to the overhead of the naive weighted random sampling being significantly higher than the cost of communication.

[1]It has been observed that using a non power of 2 does not show much of any variation in performance.



(a) Block Two-Level Erdos Renyi (SMP) - CondMat x5

(b) Quality of BTER generator.

C. Block Two-Level Erdős-Rényi (BTER)

We begin by sorting the desired vertex degree sequence and edge cardinality sequences in parallel, then scan both for the index of the first vertex and edge that have a desired degree or cardinality greater than 1. Then we compute affinity block information from metamorphosis coefficients and degree of the vertex and edge, and then create the affinity block using ER on isolated sub-hypergraphs using this information. After all affinity blocks are created, we set the new desired vertex degree and edge cardinality sequences as the difference between the desired final degrees and the actual degrees generated by ER on affinity blocks. Afterwards, we use CL to create the remaining inclusions. We use the CondMat dataset, but scale it up 5 times by appending the same vertex degree and edge cardinality sequences concatenating it with itself.

In this shared-memory benchmark, which is shown in Fig. 7a, BTER scales linearly just as ER and CL does, which is expected as it is built on top of them. Furthermore, in Fig. 7b we compare the graph generated by our BTER generator (“BTER”) with a known, high-quality generator [6] (“Generated”). The plot compares degree distribution between the two generators. The distribution produced by our generator closely matches the high-quality generator.

V. Discussion

In Fig. 4, we can not only elegantly obtain and query information about the graph, but we can also easily dispatch distributed computations by utilizing Chapel's language constructs. However, just because it is easy to write correct code does not mean that it is easy to write code that performs well. In particular, in Chapel, classes instances are allocated on the heap, and it should be noted that updates are not propagated globally, nor do class instances migrate from one locale to another. That is to say, the class instance is owned by the locale it was allocated on, and any access from any other locale is inherently remote, hence there is a hefty performance penalty from all of the implicit PUT/GET operations. To counteract this, Chapel implements a process called 'privatization' that allocates a local copy of the class instance on each locale, where all accesses can be redirected toward their local copy. While this process was originally intended for arrays and distributions, we exploit this process to ensure that our data structure scales. By making use of Chapel's `forwards` construct, we can forward any and all access from the hypergraph class to its respective privatized class instance, just like for Chapel's arrays. As Chapel's 'forall' loops pass data by reference by default, arrays have a specific compiler optimization called 'remote-value forwarding' where all usage of an array is forwarded to its privatized instance; since this is currently hard-coded for Chapel's arrays and domains, CHGL currently requires users to emulate this by declaring the forall-intent of the graph as `in`.

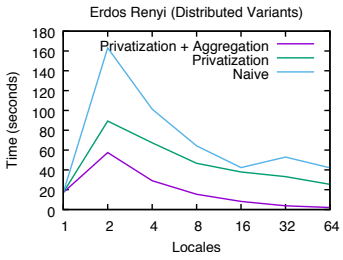


Fig. 8: Optimized ER performance.

By making use of privatization, we can then make use of aggregation, where we keep relatively-small 'destination buffers', where we aggregate pairs of vertices and edges, for each locale. This significantly speeds up performance by eliminating communication that would occur on each call to `addInclusion`, and instead perform them in one big bulk task, via `addInclusionBuffered` which requires a call to `flushBuffers` when finished. Usage of aggregation in an efficient ER can be seen below, and Fig. 8 shows the difference in performance between three levels of optimization of ER.

```
// 5% probability
const p = 0.05;
// p * |V| * |E|
const numInclusions = graph.numVertices *
  graph.numEdges * p;
// Spawn a task on each other locale
coforall loc in Locales with (in graph) do on loc {
  var rng : makeRandomStream(real);
  forall 1 .. numInclusions / numLocales {
    var vertex = rng.getNext(0,
      graph.numVertices - 1);
    var edge = rng.getNext(0, graph.numEdges - 1)
    // Aggregates (vertex, edge) pairs
    graph.addInclusionBuffered(vertex, edge);
  }
}
// Flushes pending aggregation (vertex, edge) pairs
graph.flushBuffers();
```

In this prototype, the types of `vertex` and `edge` are left out for the Chapel compiler to figure out from the call context. The only requirement is that the type of `vertex` is convertible to the `vDescType` and the type of `edge` to `eDescType`, where both descriptor types are provided as associated types by every hypergraph. Conversions from an integral type to a `vDescType` or `eDescType` is performed with an implicit bounds check unless the user specifies the *–fast* flag.

VI. Conclusions and Future Work

We preset CHGL, a Chapel library for hypergraph computation. Initially, the main driving application for CHGL is hypergraph generation using the Erdős-Rényi, Chung-Lu, and BTER models. CHGL strives to provide highly abstract interfaces in the spirit of generic programming and to leverage Chapel for modern parallel programming constructs, with the guiding principle of achieving HPC-class performance.

Currently, CHGL is a prototype. Our efforts are concentrated on developing the abstract interfaces, adding generic algorithms, and on improving performance. We are actively interacting with the Chapel community reporting problems, requesting features, and investigating design paradigms. We plan to release CHGL as open-source software before HPEC 2018.

References

[1] C. Berge, *Hypergraphs: Combinatorics of Finite Sets*. Elsevier, 1989.

[2] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[3] "The chapel parallel programming language," https://chapel-lang.org/, Jul. 2018.

[4] W. Aiello, F. Chung, and L. Lu, "A Random Graph Model for Power Law Graphs," *Experimental Mathematics*, vol. 10, no. 1, pp. 53–66, 2001.

[5] C. Seshadhri, T. G. Kolda, and A. Pinar, "Community Structure and Scale-Free Collections of Erdős-Rényi Graphs," *Physical Review E*, vol. 85, no. 5, may 2012.

[6] S. G. Aksoy, T. G. Kolda, and A. Pinar, "Measuring and Modeling Bipartite Graphs With Community Structure," *Journal of Complex Networks*, vol. 5, no. 4, pp. 581–603, mar 2017.

[7] D. Musser and A. Stepanov, "Generic Programming," in *Proc. International Symposium on Symbolic and Algebraic Computation (ISSAC)*, ser. LNCS, vol. 358. Springer, 1988, pp. 13–25.

[8] "Graph 500 steering committee," specification available at https://graph500.org/, 2018.

[9] M. Alam and M. Khan, "Parallel algorithms for generating random networks with given degree sequences," *International Journal of Parallel Programming*, vol. 45, no. 1, pp. 109–127, oct 2015.

[10] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C424–C452, jan 2014.

[11] A. Pinar, C. Seshadhri, and T. G. Kolda, "The similarity between stochastic kronecker and chung-lu graph models," in *Proceedings of the 2012 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, apr 2012, pp. 1071–1082.

[12] J. C. Miller and A. Hagberg, "Efficient generation of networks with given expected degrees," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 115–126.

[13] M. Winlaw, H. DeSterck, and G. Sanders, "An in-depth analysis of the chung-lu model," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2015.

[14] M. E. J. Newman, "The structure of scientific collaboration networks," *Proc. Natl. Acad. Sci.*, vol. 98, no. 2, pp. 404–409, jan 2001.

[15] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.