Proceedings of the 2021 Winter Simulation Conference
S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, eds.

ExecutionManager: A SOFTWARE SYSTEM TO CONTROL EXECUTION OF THIRD-PARTY SOFTWARE THAT PERFORMS NETWORK COMPUTATIONS

Henry L. Carscadden
Lucas Machi
Aparna Kishore
Chris J. Kuhlman
Dustin Machi
S. S. Ravi

Biocomplexity Institute and Initiative
University of Virginia
Charlottesville, VA 22904, USA

ABSTRACT

We describe a software system called *ExecutionManager* (abbreviated *EM*) that controls the execution of third-party software (TPS) for analyzing networks. Based on a configuration file that contains a specification for the execution of each TPS, the system launches any number of stand-alone TPS codes, if the projected execution time and the graph size are within user-imposed limits. A system capability is to estimate the running time of a TPS code on a given network through regression analysis, to support execution decision-making by *EM*. We demonstrate the usefulness of *EM* in generating network structure parameters and distributions, and in extracting meta-data information from these results. We evaluate its performance on directed and undirected, simple and multi-edge graphs that range in size over seven orders of magnitude in numbers of edges, up to 1.5 billion edges. The software system is part of a cyberinfrastructure called *net.science* for network science.

1 INTRODUCTION

1.1 Background and Motivation

Network representations of cyber-physical systems abound. This is because a network is a powerful abstraction, where nodes or vertices represent entities such as people, devices, machines, and cities, and edges represent interactions between entities. Thus, there are many classes of real networks, including social networks (e.g., friendship, social media networks), technical networks (e.g., coauthorship, brain networks), and infrastructure networks (e.g., Internet, power grid, cellular and telephone, water, natural gas, road networks); see e.g., (Easley and Kleinberg 2010; Hu et al. 2018). Ahmed et al. (2020) give a short overview of the growth in usage of networks in academic and technical fields.

There is a wide range of simulation systems that use networks as the underlying description of a population and of interactions among its members. These include NetLogo, NDLIB, Repast, Swarm, and Flame, to name a few (Tisue and Wilensky 2004; Collier and North 2012; Rossetti et al. 2018). A small sampling of network-based simulation applications includes virus propagation (Halloran et al. 2008), invasive species migration (McNitt et al. 2019), and evacuation planning (Yang et al. 2019). Structural analysis is another class of network operations, geared to answering fundamental questions such as "Is this road (or trade) network connected?" and "What neurons in the brain have the most connections to other neurons?" (Sporns 2013). There are many other types of computations performed on networks, e.g., time series analyses on brain and seismic networks (Sporns 2013; Grassi et al. 2018).

978-1-6654-3311-2/21/$31.00 ©2021 IEEE

For these reasons and others, our research group and other team members are building a first-of-its-kind, general-purpose open access cyberinfrastructure (CI) for network science called *net.science*. The initial version of *net.science* was released in June 2021 (URL: https://net.science) and will be updated at regular intervals. See Ahmed et al. (2020) for an overview. One of the services it provides is user searching over a large set of networks. Network properties and other metadata are searchable entities. For example, a user analysis within *net.science* may require networks with a giant component, an average degree between 15 and 25, and at least one million nodes. The user composes and submits a query containing this information, and *net.science* returns a collection of networks that meets these requirements. Consequently, structural properties must be computed and available for each network, in order for the query system to function effectively. The need for these computations on, and properties of, networks was the driving force behind the development of *ExecutionManager* (*EM*), a software subsystem in *net.science*.

More specifically, *EM* is a stand-alone controller software that takes as input a sequence of specifications for other (third-party) softwares (TPSs) and executes them in a serial fashion. TPSs are separate source codes, possibly developed by other organizations, that must be executed, often in groups of codes, based on user-specified criteria. For our particular need of metadata generation, after the results for each TPS are generated, *EM* extracts from these output files predefined metadata, which then form a JSON object of metadata for that network.

Our system supports data science for simulation in the following ways. First, networks are used in simulations and characterizing them provides insights into simulation inputs. Second, network properties are often used to explain simulation dynamics on networks, e.g., Kuhlman et al. (2015); hence, these properties aid in understanding simulation outputs.

1.2 Novelty of Our Work

Here, we discuss the novel aspects of the *EM* system. First, *EM* provides a programming language agnostic and application agnostic control system for executing TPSs, i.e., stand-alone C, C++, Java, Julia, Python, etc. codes that run in their own address spaces. The particular use case in this paper is structural analysis computations on networks, but the *EM* system has a broader scope. The current version of *EM* incorporates particular codes from two structural analysis libraries, namely NetworkX (Hagberg et al. 2008) and SNAP (Leskovec and Sosič 2016), to demonstrate its use and illustrate important aspects. Second, constraints can be set on each TPS, *independent of network*, such that the code will not be executed on a particular network unless all the constraints are satisfied. This is useful, for example, if some TPS (e.g., computation of betweenness centrality (Easley and Kleinberg 2010)) does not scale well to large graphs. Third, *EM* can operate outside of *net.science*, on its own, e.g., on laptops and PCs for individual or smaller group use, as well as on high performance computing (HPC) systems for wider use on larger networks.

We clarify here that *EM* is *not* a workflow system. See Related Work (Section 2). *EM* has some aspects of one, such as running multiple codes, in a particular order. However, *EM* is specifically designed to be *light-weight* with less overhead than a workflow system, but with fewer features.

1.3 Our Contributions

A. *ExecutionManager* software system. An overview of the software system is provided in Figure 1. The Controller in light blue orchestrates the work done by the system. The input graph (*Graph file* in the figure, in gray) is operated on by each specified code (in magenta) in *net.science* or that resides in the file system (yellow background in figure). This collection of codes is executed (invoked) by the controller, one at a time, based on the input from the *Code specs* (specifications), in gray. Performance data, used for determining whether a code should be run and for storing execution time and graph size after the code is run, are stored in files (the drums represent files and database tables, but currently the system works with data files). Data from the output files of each code (in green) are extracted and operated on, to generate data that are assigned to the graph (metadata, in orange). Additional details are provided in Section 3. In this

paper, we focus on stand-alone codes (magenta) that use SNAP and NetworkX functions, for demonstration purposes. However, there is no restriction on the codes that can be integrated. (Currently, we are using serial codes, per our exemplar libraries; parallel codes will be included in the future.)

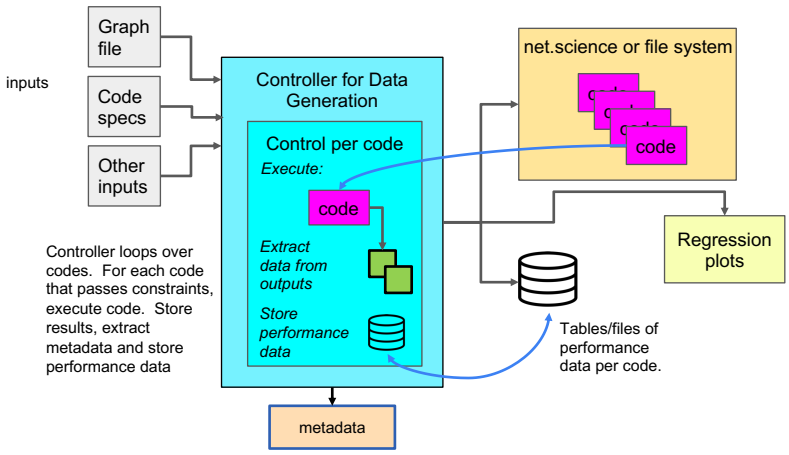


Figure 1: *ExecutionManager* software system diagram. This system (i) controls the execution of other third party software (TPS), labeled *codes*, (ii) monitors the execution of these codes, (iii) extracts and stores performance data on the codes, (iv) makes performance (i.e., execution duration) predictions for future executions of the codes, and (v) extracts user-specified data from the output from the TPSs. Features of the system are overviewed in Section 1 and are detailed in Section 3.

B. Performance evaluation of the system. Twelve networks are analyzed, ranging over five orders of magnitude in numbers of nodes and seven orders of magnitude in numbers of edges, up to a maximum of 1.5 billion edges. Studies were performed on an HPC cluster with the following characteristics: Dell PowerEdge C6420 2.666 GHz hardware nodes, with 384 GB RAM and 40 cores per node. Each core in a node is an Intel Xeon Gold 6148, 2.40 GHz with 1280 KiB L1 cache, 20 MiB L2 cache, and 27 MiB L3 cache. We find the following: (i) execution times for different TPSs on the same network can range over five orders of magnitude; and (ii) for a specific TPS, the execution times across the networks of this study can range over eight orders of magnitude. These observations demonstrate the need to control the execution of each TPS in a *network-independent* fashion, in order to avoid invoking (TPS, network) combinations with excessive execution times and memory requirements. *EM* controls the execution of each TPS based on the following characteristics: (i) predicting the execution time for a current graph by performing a regression on previous executions of the TPS on other graphs; (ii) comparing this predicted time to a user-specified maximum allowable execution time; (iii) comparing the graph size in terms of number of nodes, number of edges, and number of nodes-plus-edges, and comparing these with upper limit values for this TPS; (iv) if any of these comparisons fails (e.g., the predicted execution time is greater than the maximum allowable execution time), then the TPS is not executed on this particular network. These control features are extensible. Networks used in this performance study are presented in Section 4 and results from the study are provided in Section 5.

C. Structural properties comparisons. The system is currently set up to compute and extract 52 metadata properties per network, for subsequent searching within the *net.science* CI. We are expanding this list, which can be done in a straight-forward fashion because *EM* is extensible. Plots of degree distributions and k-core distributions on directed and undirected networks with up to 1.5 billion edges are shown in

Sections 6. We demonstrate how one type of distribution is useful for explaining the other distribution, for several networks.

Paper organization. Section 2 contains related work. Section 3 describes the software modules that constitute the *EM* system. Networks used for performance and case studies are listed in Section 4. Section 5 provides performance evaluation in terms of execution times, and Section 6 contains a case study using *EM*. A summary is in Section 7.

2 RELATED WORK

From individual use to cyberinfrastructures for network science. We motivated the development of *EM*, in part in Section 1, for its use in CIs and other systems that involve operations on graphs. The Science Gateways Community Institute (SGCI) has over 600 gateways (https://catalog.sciencegateways.org/#/home). The XSEDE Science Gateway lists more than 50 gateways (https://www.xsede.org/ecosystem/science-gateways). *EM* may also be useful in at least some of these CIs and gateways. For example, the HRGRN gateway (http://plantgrn.noble.org/hrgrn/) contains biological networks for plant genomics, the UCI Complex Social Science Gateway (CoSSci, http://socscicompute.ss.uci.edu/) uses phylogenetic trees, and the Center for Applied Internet Data Analysis (CAIDA, https://www.caida.org/home/) studies internet networks, among other topics. Structural and other types of analyses on networks are common needs.

We run *EM* on an HPC cluster for large networks and for the *net.science* CI. We also have run the system in a stand-alone fashion on laptops. The system is purposely designed to be light-weight so that it has minimal installation requirements and footprint. Thus, *EM* can support a range of CI sizes, and this is important because CIs range widely in size, as evidenced by the CIs and components presented at the NSF-sponsored meeting on CIs in February, 2020 (https://figshare.com/search?q=NSF-CSSI-2020-Talk&searchMode=1).

Operations on networks: network structural analysis codes. Simulators that use network representations of populations were cited in the Introduction (Section 1), as were selected applications for these simulators. There are several structural analysis libraries, including: Gunrock (Wang et al. 2017), a GPU-based graph analysis software system; graph-tool (Peixoto, Tiago P. 2014); NetworKit (Staudt et al. 2014), an OpenMP-based graph structural analysis library; igraph (Csardi and Nepusz 2006), a Python-wrapped C++ implementation of many methods in NetworkX; NetworkX (Hagberg et al. 2008), a Python-based system; PEGASUS (Kang et al. 2009), an Hadoop-based implementation; and SNAP (Leskovec and Sosič 2016), a C++ implementation with multithreading, along with Snap.py, which is a Python wrapper around SNAP. We have selected initially a subset of methods in NetworkX and SNAP for TPSs in *EM* because of their large user bases and because they are complementary: NetworkX has many methods and SNAP scales to large networks. Various kinds of time series analyses in Python can be accomplished by combining several packages such as pandas, numpy and scipy, and statsmodels.

Workflow management systems. For this discussion, we refer the reader to Liu et al. (2015), da Silva et al. (2017). Workflow management systems (WMSs) are software platforms that execute a collection of (interrelated) computational tasks; typically, there are data dependencies among some subsets of tasks. WMSs also manage data and run tasks on different hardware platforms. Although there are many components of a workflow system, the important ones for our purposes are workflow language and workflow engine. The language is used to specify the tasks of a workflow and their ordering, and the engine carries out these directives. Our system does not have a workflow language; instead, we use a JSON file specification to control the codes or tasks on a per task basis, independent of network. *EM* can run codes with data dependencies, but these dependencies must be specified by a user, through the ordering in which TPSs are executed. Our system uses codes and data that reside in a file system; our *net.science* CI handles management of tasks and data. This enables *EM* to be light-weight. This separation of concerns also enables *EM* to run on its own, outside of any CI, without modification.

3 SYSTEM DESCRIPTION

Table 1: Major inputs to *ExecutionManager*: (i) global inputs and (ii) TPS-specific inputs.

Input Type	Name	Description
Global	Graph file	File containing graph.
Global	Columns in graph file	Column indices that contain node IDs that form edges.
Global	Edge direction	Whether edges in graph are directed or undirected.
Global	Edge multiplicity	Whether the graph is a simple or mullti-graph.
Global	Metadata output file	Output file to write metadata across codes.
Global	Timing data output file	Output file to write code timing data.
Per TPS	Code name	Full path to code's main() method.
Per TPS	Edge directionality	Whether code applies to directed or undirected graphs (or both).
Per TPS	Time limit	Max. permissible code execution duration.
Per TPS	Regression method	Method to correlate code execution duration with graph size.
Per TPS	Min. data points	Min. number of data points to perform a regression.
Per TPS	Node limit	Max. permissible number of nodes in the graph to run code.
Per TPS	Edge limit	Max. permissible number of edges in the graph to run code.
Per TPS	File of performance data	File containing performance data for this code.
Per TPS	Output files and metadata	Metadata to be extracted from each specified output file.

We now provide additional details regarding the *EM* system of Figure 1 (Section 1). Key inputs in the upper left of that figure are shown in more detail in Table 1. Specifically, this table contains the list of inputs for the system, broken down by global inputs (e.g., the graph) and inputs I per TPS (code) c (denoted *Code specs*) in Figure 1.

Figure 2 summarizes controller activities, which align with Algorithm 1; this algorithm presents a more detailed description of the operations executed by the controller. Per code inputs in the table tell the controller what checks on the graph to perform for each TPS (code) c, often by comparison with features in Table 1. If the checks are successful, processing of this graph with the current code c proceeds; otherwise (e.g., if the graph is too large or the estimated execution duration is too great), code c is not executed and the system advances to the next code. Algorithm 1 provides the steps for composing the command line invocation (CLI), by parsing an intermediate representation (IR) for c, in cases where it exists. Otherwise, the command line invocation is specified explicitly as part of the code information I. A process is forked and code c is invoked. The controller block-waits for c to complete and obtains outputs written to standard out by c. Our codes c, by design, print their execution durations to standard out and the controller captures that output and reads these times, but this information could also be read from an output file. Alternatively, the total execution life of the worker process could be used as an estimate of c's execution time (this duration is recorded). If c completes successfully, the controller writes execution time to that code's performance data file, and the controller iterates through c's output files and extracts data that it needs, according to I. These extracted data, along with possible computations performed by the controller, are the metadata assigned to the input graph, in the form of a JSON file. After all codes c complete, all metadata and timing data are written to files. Table 3 provides a few illustrative TPSs in *net.science* that are invoked by *EM*.

4 NETWORKS

Networks used in our performance and case studies are provided in Table 3. The smaller networks come from (Jure Leskovec and Andrej Krevl 2014); the social networks of city populations are generated with procedures in (Barrett et al. 2009). These networks were selected to cover five orders of magnitude in numbers n of nodes, and seven orders of magnitude in numbers m of edges. Selected properties are also

Algorithm 1: Computational Code Execution and Data Extraction

1 **Input:** (1) Name of the file containing the network $G(V,E)$. (2) Set C of third party codes c to execute against G. (3) Name of the file containing the information I for each TPS (code) c. (4) Name of performance data file for each of the codes $c \in C$. (5) Name of the output file for metadata. (6) Name of the output file for timing data.

2 **Output:** (1) All output data files for each code $c \in C$. (2) Metadata output (one file). (3) Timing data output (one file). (4) Append to each performance data file the execution duration for c on G.

3 **Steps:**

A. Read the file containing the network $G(V,E)$.
B. Read the file I containing the set C of third-party codes to run, and their associated information.
C. **for each** code $c \in C$ **do**:
   1. Perform checks on graph G to determine whether it should run, as follows.
      i. Read performance data file on past execution durations for c on networks of different sizes, and perform regression, of the specified type, on execution duration versus graph size.
      ii. Predict the execution time for c on the current graph G.
      iii. Compare predicted time to the maximum allowable execution time (from input I).
      iv. Compare $n = |V|$ of G to the maximum graph size, in terms of numbers of nodes, from I.
      v. Compare $m = |E|$ of G to the maximum graph size, in terms of numbers of edges, from I.
      vi. Compare the sum of n and m of G to the maximum graph size (in terms of both nodes and edges), from I.
   2. **If** graph passes execution time and size checks **then** proceed. **Else**, **continue** with next code c.
   3. Assemble command line to invoke code c (e.g., method name, command line arguments), as follows.
      i. Get name of intermediate representation (IR) file for c from I.
      ii. Get location of the executable (or interpretable) code from I.
      iii. Read the IR for this code c to assemble command line invocation.
   4. Invoke code c as a separate process in its own address space, as follows.
      i. Invoke code c on graph G by forking a new process for c.
      ii. Block-wait for process c to complete.
      iii. Capture and parse standard out from executing c, to obtain execution time for c and whether code c terminated successfully.
   5. **If** c completed successfully **then**:
      i. Write the execution time of code c to performance data file.
      ii. From I, obtain the output files of interest generated by code c and extract data, as follows. **for each** output file:
         a. Read output file and extract values.
         b. Perform computations on these data as necessary.
         c. Assign resulting values to keys in metadata.
D. Print metadata to metadata output file.
E. Print timing data for controller and third-party codes to file.
F. Return.

provided, generated with *EM*, using the third-party codes in the *net.science* CI (Ahmed et al. 2020) for network science, which contain the structural analysis methods in SNAP (Leskovec and Sosič 2016) and NetworkX (Hagberg et al. 2008).

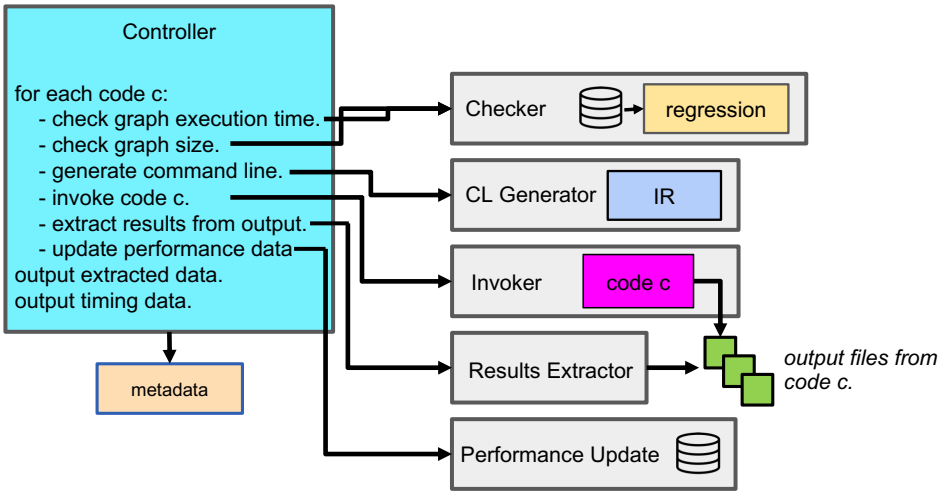


Figure 2: Actions controlled by the *ExecutionManager* software system (blue box in Figure 1). This more parsimonious view is described in Section 3.

Table 2: Sampling of network codes (TPSs) currently run within the system. Because of the system design, adding additional TPSs is straight-forward.

Code	**Library**	**Description**	**Metadata Properties**
Degree distribution	SNAP	Computes degree distribution for undirected graphs.	Min, max and average degrees.
In-degree distribution (directed graph)	SNAP	Computes in-degree distribution for directed graphs.	Min, max and average in-degrees.
Out-degree distribution (directed graph)	SNAP	Computes out-degree distribution for directed graphs.	Min, max and average out-degrees.
Clustering coefficient per node (treating graph as undirected)	NetworkX	Computes clustering coefficient for each node.	Min, max and average clustering coefficients.

5 PERFORMANCE EVALUATION

All data provided in this section are collected and stored by *EM*. We refer the reader to Section 1.3 ("Our Contributions") for details regarding the hardware platform used to obtain the performance evaluation results in this section and the case study results in Section 6.

5.1 Timing Data Per Third-Party Software and Network

Figure 3 provides timing data for many (TPS, network) pairs. Networks are specified on the x-axis, in order of increasing numbers of edges. TPSs, in these cases, structural analysis codes, are plotted in colors; see the legend. The methods are chosen to show data with different time complexities. For example, the curves for degree distribution and k-core distribution are close to each other since they use very similar algorithms whose running times are linear in the size of the network (i.e., $(n+m)$). At the other extreme, betweenness centrality is an expensive computation: its execution time is proportional to nm. A key result is that these data provide motivation for a system to control the execution of TPSs because their execution times can vary widely; see the figure caption. Figure 4 shows timing data for many TPSs for the MWSU

Table 3: Networks used in performance evaluation and case study. If there are multiple connected components in a graph, we use only the giant component. Here, n and m are numbers of vertices and edges, respectively, in the giant component; d_{ave} and d_{max} are average and maximum degrees; k_{max} is the maximum k-core; c_{ave} is average clustering coefficient; and Δ is graph diameter. All the networks are *simple* except for the MMD network (Miami, FL) which has *multi-edges*. The networks for Miami and Houston are the social contact networks for either a normative Wednesday, or for all days of the week.

Network	**Abbrev.**	**Edge Direction**	n	m	d_{ave}	d_{max}	k_{max}	c_{ave}	Δ
Jazz	Jazz	undirected	198	2,742	27.7	100	29	0.617	6
Ca-Hepth	CAH	undirected	8,638	24,806	5.74	65	31	0.482	18
Ca-Astroph	CAA	undirected	17,903	196,972	22.0	504	56	0.633	14
Epinions	Epin	undirected	75,877	405,739	10.7	3044	67	0.138	14
VA-Beach	VAB	undirected	167,722	3,245,840	38.7	399	380	0.186	7
Miami, FL (Wednesday)	MWSU	undirected	4,894,150	87,077,324	35.6	656	47	0.092	11
Miami, FL	MMD	directed	5,136,141	1,192,987,822	464.5	4342	564	0.022	6
Miami, FL	MSD	directed	5,136,141	1,039,345,260	404.7	4112	346	0.022	6
Miami, FL	MSU	undirected	5,136,141	519,672,630	202.4	2056	173	0.022	6
Houston, TX (Wednesday)	HWSU	undirected	5,841,222	129,310,137	44.3	706	51	0.073	12
Houston, TX	HSD	directed	6,166,985	1,591,054,658	516.0	4186	368	0.021	7
Houston, TX	HSU	undirected	6,166,985	795,527,329	258.0	2093	184	0.021	7

network, one of the Miami social networks. This network is also part of Figure 3, with color coding consistent across the two plots.

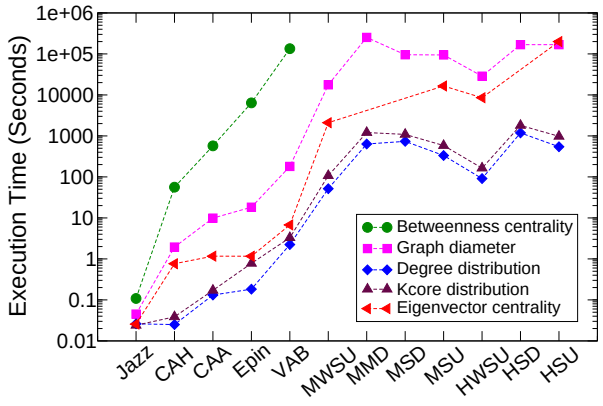


Figure 3: Timing data collected by *EM* for execution of five TPSs, across all of the networks shown on the x-axis. The data show that: (i) there can be five orders of magnitude differences in execution times for different TPSs, for a given network (Epinions); and (ii) for a particular TPS (graph diameter), there can be eight orders of magnitude differences in execution times across networks. Hence it is useful that *EM* provides conditionals on whether a TPS is executed, based on graph size and predicted execution time.

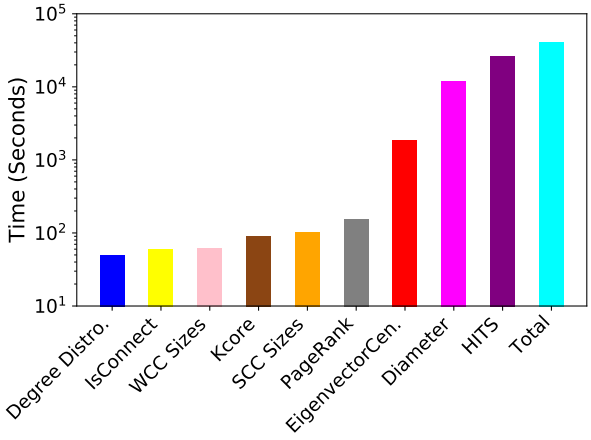
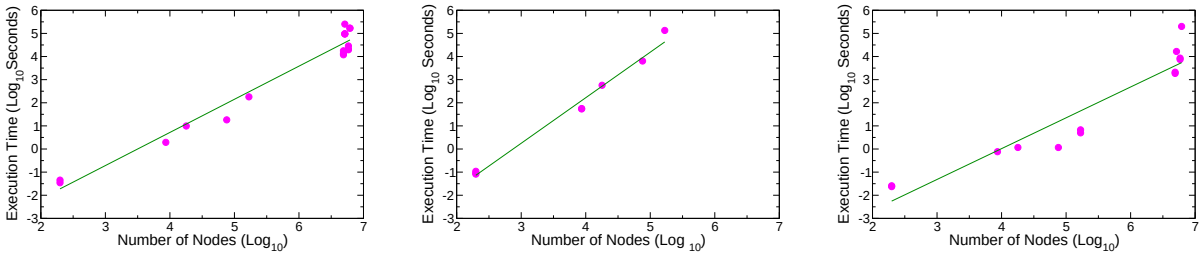


Figure 4: Execution durations for TPSs that compute various structural parameters. These data are generated on the MWSU graph, one of the Miami networks.

5.2 Predicting Execution Duration

Figure 5 provides timing data for executing three TPSs on the networks of this study, as a function of n. The curves in the plots represent regressions to the data. These equations are used to predict the execution duration for the same code c, for different graph sizes. These predicted execution times are then compared to the user-specified limit in execution duration and the code is executed if the predicted time is less than or equal to the specified time limit. Here, we show data only for the networks in Table 3. In reality, the system keeps accumulating data for each TPS as it is run, so the number of data points will increase significantly over time. Nonetheless, the R^2 values in the caption of Figure 5 are acceptable. The slopes of diameter computation and eigenvector centrality are similar since they have nearly the same asymptotic time complexity (namely, that of multiplying two $n \times n$ matrices, where n is the number of nodes).



(a) Graph diameter (b) Betweenness centrality (c) Eigenvector centrality

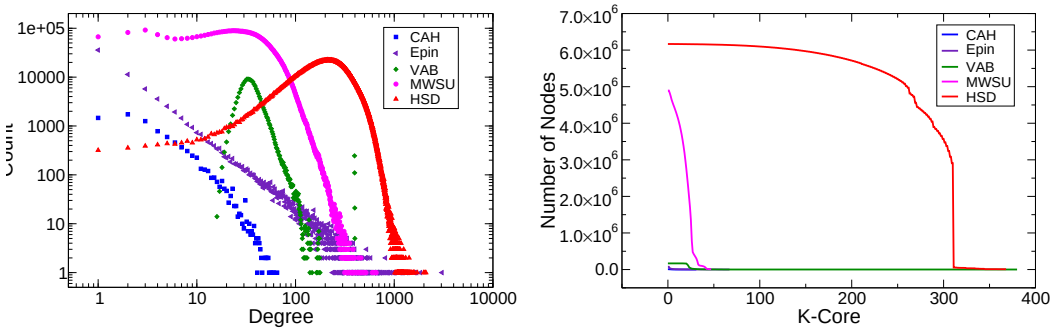
Figure 5: Timing data collected, and regressions performed, by *EM* for execution of three TPSs, in these cases, for: (a) graph diameter, (b) betweenness centrality, and (c) eigenvector centrality. The R^2 values for the fits are, respectively, 0.96, 0.99, and 0.90.

6 CASE STUDY

Up to this point, all results have focused on the execution times of the various TPSs that are controlled by *EM*. In this section, we provide results that are outputs generated by these TPSs. Figure 6 shows

structural analysis results of two types. Figure 6a depicts degree distributions for five graphs. Two are social networks (CAH and Epin) and these exhibit classic power law degree distributions. The three city social contact networks (VAB, MWSU, and HSD) also have large right tails, but their slopes at higher degrees are much steeper (negative slopes) than those of the two social networks. Also, the three city networks have smaller-degree shapes that have either positive slopes (VAB and HSD) or a plateau (MWSU). This is because in the city networks, individuals predominantly go to school (for school-aged individuals [from grade school through high school]) and adults go to college or work, where they mix with significant numbers of other people. Hence, one does not find relatively large numbers of individuals with very small degrees, as is the case for CAH and Epin.

Figure 6b contains k-core distributions for the same five networks. The HSD red curve is interesting because it has a particular shape: large fractions of nodes for smaller and larger k-core numbers, followed by a steep drop-off where further k-cores (between k-cores of 310 and 368) are very small. This same curve shape is exhibited for VAB in green, although the steep drop-off occurs at a much smaller k-core of about 25. The other large network, MWSU, however, does not have this shape: there is a much more gradual decrease in the fraction of nodes with increasing core number. (Note that both the original plot and the inset show this behavior.) The reason for this difference is found in the degree distribution plot of Figure 6a. For both HSD and VAB, to the left of the peak number of nodes for any degree for each network, the curves decrease, i.e., there are fewer nodes with these lesser degrees. This means that these fewer nodes with lesser degrees, once they are removed from the network in the process of computing successively larger k-cores, cause the numbers of nodes in the remaining cores to decrease very slowly. At some point, here, for a k-core of 310 for HSD, removing greater degree nodes causes the remaining k-core to almost disintegrate. In contrast, the numbers of nodes with degrees less than 30 (which is the degree with the greatest number of nodes for MWSU) remain relatively constant. This means that at smaller cores, larger fractions of nodes are being removed, and therefore the k-core sizes decrease more steadily as the core number increases. In contrast to the three larger networks, the CAH and Epin graphs have concave upward k-core distributions, which is typical for scale-free networks.



(a) degree distributions

(b) k-core distributions

Figure 6: Structural analysis results on many networks, where the structural analysis codes are the TPSs and *EM* controls their execution. (a) Degree distributions for graphs, where for undirected graphs, the standard degree is depicted, and for directed networks, the in-degree distribution is shown. (b) K-core distributions for several graphs. For directed graphs, total degree (=in-degree+out-degree) is used to compute k-cores.

We end this section with three comments. First, it would be virtually impossible to reason about the k-core distributions without the degree distributions; the latter help explain the former. This demonstrates the value of structural analysis data for networks. In this case, one structural parameter is used to explain another. Second, this demonstrates the data science role that *EM* can play, with respect to networks:

controlling the generation of different types of data that can be used to understand networks. Third, the results in Table 3 are metadata extracted from the structural analysis output files, which were executed through *EM*. These demonstrate the utility of the *EM* system.

7 SUMMARY AND FUTURE WORK

We presented *ExecutionManager*, a software system for managing the execution of third-party software (TPS). The motivation for this system, its novelty, and our detailed contributions in this work are explained in Sections 1.1, 1.2, and 1.3, respectively. We evaluated the performance of the system by considering codes from two libraries, namely NetworkX and SNAP. These results point out that the *EM* system scales well to large graphs. We are in the process of extending the system in several ways including the following: (*i*) work with job submission systems (in particular, Slurm), (*ii*) provide plotting functionality for results from codes, (*iii*) do a better job of integrating output from multiple networks and methods for cases of graph collections (e.g., for a series of temporal networks), and (*iv*) improving the predictions of execution durations using more advanced methods, e.g., (Ahmad et al. 2019; Hou et al. 2019).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. We thank our colleagues at NSSAC and Research Computing at The University of Virginia for providing computational resources and technical support. This work has been partially supported by University of Virginia Strategic Investment Fund award number SIF160, NSF Grant OAC-1916805 (CINES), and NSF Grant CMMI-1916670 (CRISP 2.0).

REFERENCES

Ahmad, M., H. Dogan, C. J. Michael, and O. Khan. 2019. "HeteroMap: A Runtime Performance Predictor for Efficient Processing of Graph Analytics on Heterogeneous Multi-Accelerators". In *ISPASS*, 268–281.

Ahmed, N. K., R. A. Alo, C. T. Amelink et al. 2020. "net.science: A Cyberinfrastructure for Sustained Innovation in Network Science and Engineering". In *Gateway Conference*. Oct. 19-23, Virtual.

Barrett, C. L., R. J. Beckman et al. 2009. "Generation and Analysis of Large Synthetic Social Contact Networks". In *Winter Simulation Conference (WSC)*, edited by M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, 1003–1014. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc. Dec. 13-16, Austin, TX, USA.

Collier, N., and M. North. 2012. "Parallel Agent-Based Simulation with Repast for High Performance Computing". *Simulation: Transactions of the Society for Modeling and Simulation International* 89(10):1215–1235.

Csardi, G., and T. Nepusz. 2006. "The igraph Software Package for Complex Network Research". *InterJournal, Complex Systems* 1695:1–9.

da Silva, R. F., R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman. 2017. "A Characterization of Workflow Management Systems for Extreme-Scale Applications". *Future Generation Computer Systems* 75:228–238.

Easley, D., and J. Kleinberg. 2010. *Networks, Crowds and Markets: Reasoning About a Highly Connected World*. New York, NY: Cambridge University Press.

Grassi, F., A. Loukas, N. Perraudin, and B. Ricaud. 2018. "A Time-Vertex Signal Processing Framework: Scalable Processing and Meaningful Representations for Time-Series on Graphs". *IEEE Transactions on Signal Processing* 66:817–829.

Hagberg, A. A., D. A. Schult, and P. J. Swart. 2008. "Exploring Network Structure, Dynamics, and Function using NetworkX". In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, 11–15. Aug. 19-24, Pasadena, CA, USA.

Halloran, M., N. Ferguson, I. L. S. Eubank, D. Cummings, B. Lewis, S. Xu, C. Fraser, A. Vullikanti, T. Germann, D. Wagener, R. Beckman, K. Kadau, C. Barrett, C. Macken, D. Burke, and P. Cooley. 2008. "Modeling Targeted Layered Containment of an Influenza Pandemic in the United States". *PNAS* 105(12):4639–4644.

Hou, Z., S. Zhao, C. Yin, Y. Wang, J. Gu, and X. Zhou. 2019. "Machine Learning Based Performance Analysis and Prediction of Jobs on a HPC Cluster". In *PDCAT*, 247–252. Dec. 5-7, Gold Coast, Australia.

Hu, F., A. Mostashari, and J. Xie. 2018. *Socio-Technical Networks: Science and Engineering Design*. Boca Raton, FL: CRC Press.

Kang, U., C. E. Tsourakakis, and C. Faloutsos. 2009. "PEGASUS: A Peta-Scale Graph Mining System–Implementation and Observations". In *IEEE International Conference on Data Mining (ICDM)*, 229–238. Dec. 6-9, Miami Beach, FL, USA.

Kuhlman, C. J., V. S. A. Kumar et al. 2015. "Inhibiting Diffusion of Complex Contagions in Social Networks: Theoretical and Experimental Results". *DMKD*:423–465.

Jure Leskovec and Andrej Krevl 2014, June. "SNAP Datasets: Stanford Large Network Dataset Collection". http://snap.stanford.edu/data. Accessed Oct. 15, 2020.

Leskovec, J., and R. Sosič. 2016. "SNAP: A General-Purpose Network Analysis and Graph-Mining Library". *ACM Transactions on Intelligent Systems and Technology (TIST)* 8(1):1.

Liu, J., E. Pacitti, P. Valduriez, and M. Mattoso. 2015. "A Survey of Data-Intensive Scientific Workflow Management". *J Grid Computing* 13:457–493.

McNitt, J., Y. Y. Chungbaek, H. Mortveit, M. Marathe, M. R. Campos, N. Desneux, T. Brevault, R. M. R, and A. Adiga. 2019. "Assessing the Multi-Pathway Threat From an Invasive Agricultural Pest: Tuta Absoluta in Asia". *Royal Society B* 286:20191159–1–20191159–9.

Peixoto, Tiago P. 2014. "The graph-tool python library". Software available at https://figshare.com/articles/dataset/graph_tool/1164194. Accessed Jan. 10, 2021.

Rossetti, G. et al. 2018. "NDlib: a Python Library to Model and Analyze Diffusion Processes Over Complex Networks". *International Journal of Data Science and Analytics* 5:61–79.

Sporns, O. 2013. "Structure and Function of Complex Brain Networks". *Dialogues in Clinical Neuroscience* 15:247–262.

Staudt, C., A. Sazonovs, and H. Meyerhenke. 2014. "NetworKit: A Tool Suite for Large-scale Complex Network Analysis". *Network Science* 4(4):508–530.

Tisue, S., and U. Wilensky. 2004. "NetLogo: Design and Implementation of a Multi-Agent Modeling Environment". In *SwarmFest Conference*.

Wang, Y., Y. Pan et al. 2017. "Gunrock: GPU Graph Analytics". *ACM Trans. Parallel Comput.* 4(1).

Yang, Y., L. Mao, and S. S. Metcalf. 2019. "Diffusion of Hurricane Evacuation Behavior Through a Home-Workplace Social Network: A Spatially Explicit Agent-Based Simulation Model". *Computers, Environment and Urban Systems* 74:13–22.

AUTHOR BIOGRAPHIES

HENRY CARSCADDEN is a student in the Computer Science Department at the University of Virginia (UVA). His email address is hlc5v@virginia.edu.

LUCAS MACHI is an undergraduate student at New River Community College and is a member of the Biocomplexity Institute and Initiative (BII) at UVA. His email address is lhm4v@virginia.edu.

APARNA KISHORE is a student in the Computer Science Department at UVA. Her email address is ak8mj@virginia.edu.

CHRIS J. KUHLMAN is faculty at the BII of UVA. His email address is cjk8gx@virginia.edu.

DUSTIN MACHI is a Senior Software Architect at the BII of UVA. His email addresses is dm8qs@virginia.edu.

S. S. RAVI is faculty at the BII of UVA. His email address is ssr6nh@virginia.edu.