Special Issue: CCDSC 2022

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

A Survey of Graph Comparison Methods with Applications to Nondeterminism in High-Performance Computing

The International Journal of High Performance Computing Applications
2023, Vol. 0(0) 1–22

DOI: 10.1177/10943420231166610
journals.sagepub.com/home/hpc
SAGE

Sanjukta Bhowmick[1], Patrick Bell[2] and Michela Taufer[2]

Abstract

The convergence of extremely high levels of hardware concurrency and the effective overlap of computation and communication in asynchronous executions has resulted in increasing nondeterminism in High-Performance Computing (HPC) applications. Nondeterminism can manifest at multiple levels: from low-level communication primitives to libraries to application-level functions. No matter its source, nondeterminism can drastically increase the cost of result reproducibility, debugging workflows, testing parallel programs, or ensuring fault-tolerance. Nondeterministic executions of HPC applications can be modeled as event graphs, and the applications' nondeterministic behavior can be understood and, in some cases, mitigated using graph comparison algorithms. However, a connection between graph comparison algorithms and approaches to understanding nondeterminism in HPC still needs to be established. This survey article moves the first steps toward establishing a connection between graph comparison algorithms and nondeterminism in HPC with its three contributions: it provides a survey of different graph comparison algorithms and a timeline for each category's significant works; it discusses how existing graph comparison methods do not fully support properties needed to understand nondeterministic patterns in HPC applications; and it presents the open challenges that should be addressed to leverage the power of graph comparisons for the study of nondeterminism in HPC applications.

Keywords

Graph comparison, graph isomorphism, graph alignment, graph kernels, high-performance computing

Introduction

The convergence of extreme hardware concurrency and the effective overlap of computation and communication in asynchronous executions are resulting in growing nondeterminism in High-Performance Computing (HPC) applications, as illustrated in Figure 1 and presented in Ahn et al. (2013); Gopalakrishnan et al. (2017); Sato et al. (2017); Chapp et al. (2015, 2018, 2021). Nondeterminism can manifest at multiple levels in the software stack: it can manifest in low-level communication primitives (e.g., the inherent nondeterminism of nonblocking matching functions in MPI); it can manifest in libraries (e.g., dynamic load-balancing libraries), as presented in Lusk et al. (2015); or it can display at the application level (e.g., Monte-Carlo simulations). No matter its source, nondeterminism can drastically increase the cost of reproducibility (e.g., in terms of developer time and computational resources), whether or not that reproducibility is desired for scientific outcomes. Moreover, debugging workflows that exhibit nondeterministic bugs, for example, moving from a smaller to a larger scale or from one platform to another, can slow down the transition from development to production in HPC codes. Other scenarios that cannot be ignored are: testing parallel programs to ensure numerical or scientific repetition of results despite, for example, nondeterministic message interleaving or reductions; and providing fault-tolerance where the execution of one or more processes may need to be rolled back and replayed to recover from a fault. Case studies presented in Ahn et al. (2013); Sato et al. (2017); Gioachin et al. (2010); Chiang et al. (2013) and the expert consensus collected in Gopalakrishnan et al. (2017) are indicative of a looming crisis.

[1]University of Northern Texas, Denton, TX, USA
[2]University of Tennessee Knoxville College of Engineering, Knoxville, TN, USA

Corresponding author:
Michela Taufer, EECS, University of Tennessee Knoxville College of Engineering, 401 Min H. Kao Bldg, 1520 Middle Drive, Knoxville, TN 37996, USA.
Email: taufer@utk.edu

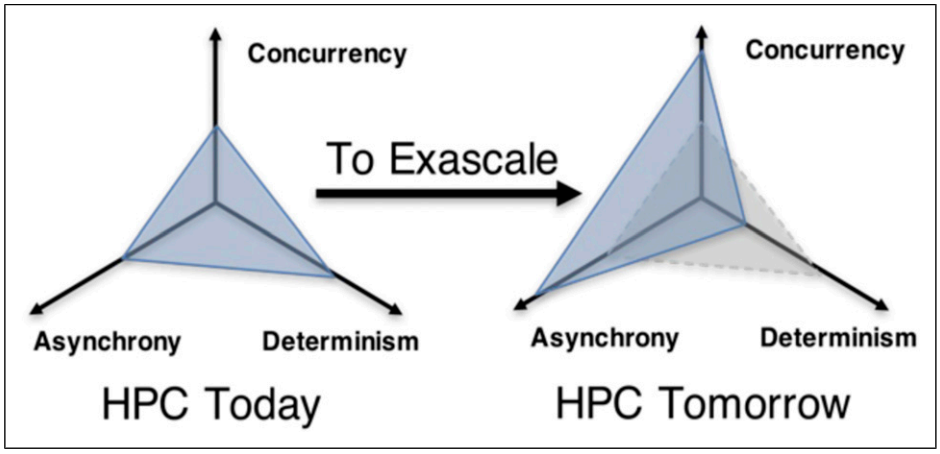

Figure 1. Growing nondeterminism, asynchrony, and concurrency trends in HPC applications. In exascale simulations, with the increase of concurrency and asynchrony of executions, the guarantee of deterministic execution will be lost.

Nondeterministic executions of HPC applications can be modeled as event graphs. Graph comparison algorithms, a fundamental problem in graph theory, can be leveraged to evaluate how "similar" are a set of given graphs. Ultimately, the output of graph comparison algorithms could support users in quantifying variations between multiple executions of the same application, improving the efficiency of checkpointing, detecting subtle differences between distinct nondeterministic communication patterns, and automatically identifying instances of abnormal nondeterministic communication. However, existing graph comparison algorithms do not fully support properties needed to understand nondeterministic patterns in HPC applications, such as sensitivity to different graphs, scalability to input graphs, specialization to HPC systems, and tunability to user specifications.

Our work builds on the rich history of graph comparison algorithms and moves the first steps toward filling this gap between existing graph comparison methods and their use to study nondeterminism in real-world HPC applications. Specifically, this article presents a survey of the state-of-the-art algorithms for graph comparisons, identifying the study of nondeterminism as a research domain still in its infancy. While variations of graph comparison have been extensively used for real-world problems such as protein matching in biology (Ma and Liao, 2020), identifying nondeterminism in HPC simulations (Bell et al., 2021), and social network comparison (Liu et al., 2016); Zhang and Philip, 2015), graph comparison methods have limited deployment in HPC nondeterminism. Furthermore, while there exist several surveys on different aspects of graph comparison, including the latest methods on graph isomorphism (Grohe and Schweitzer, 2020), subgraph isomorphism and mining (Jiang et al., 2013), and graph kernel methods (Kriege et al., 2020); Borgwardt et al., 2020), these articles do not explicitly focus on applying these methods to capture and interpret aspects of nondeterminism in HPC applications. Our goal is to build the missing connection between graph comparison algorithms and the causes of nondeterminism in HPC applications as nondeterministic behaviors rise due to the growth of exascale computation.

We classify the variations of graph comparison methods into three broad categories based on how they are typically used. *Exact graph comparisons* including graph and subgraph isomorphism, are generally designed for solving well-defined theoretical problems, not always related to a specific application. *Graph alignment* involves algorithms that focus on matching vertices of two or more graphs nearly similar in size. Typically the matching criteria depend on the application domain and user specifications. *Quantitative functions for graph comparisons* comprise algorithms that fall between the stringent theoretical work of isomorphism and the user-oriented methods of graph alignment. These include functions that take in properties or embeddings of the graphs and return a real number that quantifies the similarity between the graphs. Many such functions exist, from classifying graphs based on degree distribution to comparing them using graph kernels.

Throughout this article, we discuss the literature on these categories of graph comparison using the Chapp-Taufer format (Chapp et al., 2018) that maps publications based on publication time and categories. A box represents each article; color represents different solutions (i.e., general or specific); and symbols represent algorithms features. Our survey identifies the research gap (and opportunities) still unexplored in graph comparison for studying nondeterministic HPC applications. We define the research directions needed to fill the gap in modeling reproducible executions, ensuring code replicability, and addressing fault-tolerance in graph algorithms.

To summarize, our contributions in this article are threefold. First, we provide a survey of different graph

comparison methods and a timeline for each category's significant works, outlining missing properties that could make existing solutions viable for nondeterminism in HPC. Second, we discuss how graph comparison methods have been applied to challenges in HPC simulations but not for the understanding of nondeterminism. Last, we present the open challenges that should be addressed to fully benefit from graph comparison algorithms to study nondeterministic HPC applications.

Desirable properties of graph comparison for nondeterministic HPC applications

Graph comparison presents an attractive approach to understanding the harmful aspects of nondeterminism in HPC applications. Graph comparison algorithms provide the ability to monitor and record changes in program state over one execution (i.e., the recorded execution) of an application and reproduce those changes, and thus, the application's behavior during a subsequent execution (i.e., the replayed execution). Graph comparison algorithms can be used for comparing execution sequences to identify nondeterminism, for comparing codes and expression DAGs to determine correctness and replicability, and for comparing input graphs to improve fault-tolerance and enable faster computation. In the specific context of HPC applications on exascale systems, graph comparison algorithms must record sufficient and necessary execution events to model the behavior of thousands of interacting processes or threads, each of which may manifest multiple kinds of nondeterministic behavior. To capture the unique properties of execution patterns in nondeterministic HPC applications into event graphs, we need expressive representations of executions; a means of determining the cost of recording events into event graphs; a rigorous notion of dissimilarity between multiple executions of the same nondeterministic application, and metrics for extracting quantitative dissimilarity.

Direct acyclic graphs are an example of general and expressive execution representation where vertices define relevant inter-process or inter-thread communication events. This representation is commonly known as an event graph (Kranzlmüller, 2000) and has seen success as a debugging aid. To capture the unique properties in HPC applications, graph representation has to be enriched with vertices embedding, for example, information about the chain of function calls that terminate in each communication event. This data can link the runtime nondeterminism with its sources in the call graph of the application, which is critical for understanding how changes in application configuration affect runtime nondeterminism, and by extension, the cost of recording events into graphs. Capitalizing on the event graph representation of executions, graph comparison methods address the need for execution dissimilarity metrics to measure the "distance" between executions in the space of all such possible event graphs.

Applications targeting exascale platforms push the envelope of performance. They do not tolerate the performance degradation inherent in heavy-weight tooling, so tools that capture execution patterns, as graph comparison algorithms do, must afford lightweight implementations. More perniciously, graph comparison algorithms' overheads (or costs) must not perturb execution timings such that the very phenomena (e.g., specific simulation trajectories or subtle and intermittent bugs) are meant to be recorded and are prevented from occurring in the first place. Event graphs exist in memory during the graph comparisons and may be persisted to disk, either in chunks during recording or all at once at the end, and necessarily impose some overhead to monitor events and update the representation. Consequently, graph comparisons have to deal with two kinds of overheads: memory overhead and execution time overhead. The memory overhead refers to the size of the in-memory representation, and the execution time overhead refers to the slowdown relative to a monitored execution.

Tools to solve the graph comparison problem in HPC applications require specific properties for the above-listed challenges. A valid graph comparison tool for HPC applications must feature sensitivity to the differences between graphs, scalability to the size of the input graphs, specialization to HPC systems, and tunability to user specifications.

Sensitivity

Refers to whether the quantitative measures of graph similarities or differences are commensurate with the change in the graph. Specifically, small changes in graph structure should result in a small quantitative value and significant changes in larger values. Since the difference is manifested not as the number of edges or vertices changed but by how these elements affect the entire topology, developing sensitive, quantitative metrics for comparing graphs is challenging. The sensitivity should also be an essential parameter when designing scoring functions of vertices. These scoring functions match vertices during graph alignment, and if many vertices have similar scores, the alignment may become incorrect.

Scalability. Refers to how well the graph comparison algorithms handle large graphs. Due to non-localized memory access, it is challenging to design scalable graph algorithms. Among the current state-of-the-art graph comparison algorithms, scalability is achieved if only a few (i.e., one or two) motifs of reasonably small size (order of 10s or vertices) are mined in massive graphs. However, the scalability falls if the motifs' size or diversity increases. In the case of functions for graph comparison, the scalability depends on

the function used. For example, degree distribution is trivially parallelizable and scalable, but graph embedding techniques required for graph kernel methods may not always scale. Identifying small motifs is needed in applications such as comparing critical elements of the expression graphs or compressing graphs for improved checkpointing. However, identifying nondeterminism in event graphs requires comparing the entire graphs, and multiple/extensive motif comparisons or advanced embedding techniques may be needed.

Specialization. Refers to the ability to process information about the heavy structuring and unique metadata relevant to HPC applications but not applications from other domains. For example, in message-passing applications, event graphs must capture communication events in an application's execution into a record, and graph comparison tools must recognize information that unambiguously orders the observed events into the record. One way to do this task is with logical clocks and metadata. Metadata in message-passing applications can be diverse, including processes' rank, tag, and communication event type (e.g., completion of a receive vs. invocation of function).

Tunability. Refers to the ability of users to specify what kind of graph comparison they want, such as which types of vertices to use and which node features to use. Most graph comparison algorithms are designed for general graphs. Graphs generated for HPC applications have unique features. For example, event graphs are very sparse, with the degree bounded by the number of processors. Moreover, in these graphs, many vertices have similar neighbors, giving rise to multiple competing candidates during graph alignment. Complex graphs expressing workflows have labels describing input parameters, platform features, and environment settings. These labels may need to be accounted for during comparison. We posit that existing graph comparison algorithms have to be adapted to the specific questions in HPC for accuracy and tuned to the topology of the graphs for efficiency.

The following four sections survey general graph comparison algorithms and discuss how they miss meeting the required properties.

Exact graph comparison

An exact graph comparison is used to determine whether two graphs are structurally equivalent. An exact graph comparison can be either a full graph isomorphism or a subgraph isomorphism. A full graph isomorphism means one whole graph is isomorphic to another. Subgraph isomorphism is when an induced subgraph is isomorphic to a target graph. Figure 2 shows examples of different types of exact graph comparison. Figure 3 provides an overview of the timeline of critical articles in exact graph comparison. The same articles are discussed in detail in this section.

Graph isomorphism

Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be two graphs. G and H are isomorphic if there exists a pair of bijective functions ϕ_V: $V_G \rightarrow V_H$ and ϕ_E: $E_G \rightarrow E_H$. An example is given in Figure 2(a). These bijections are called the isomorphism between G and H. If G and H are isomorphic, we say that $G \cong H$. The graph isomorphism problem belongs to the NP complexity class but is one of the few ones not known to belong to P or NP-complete subsets. Indeed it is defined to be in its own special complexity class GI (GI for graph isomorphism). In 2016, Babai (2016) proposed a quasi-polynomial time algorithm for solving isomorphism on the general graphs.[1]

One of the earliest articles on graph isomorphism Whitney (1992) introduced the Whitney graph isomorphism theorem. The theorem states that with one exception of the K_3 and $K_{1,3}$[2] two connected graphs are isomorphic if and only if their respective line graphs are isomorphic. The line graph $L(G)$ of a graph G is where the edges of G are the vertices in $L(G)$ and two vertices in $L(G)$ are connected if the corresponding edges in G have a vertex in common.

Much of the early work in graph isomorphism focused on algorithms for generic graphs without considering any unique topological properties. Among these, a popular and fast method for comparing graph isomorphism is iterative refinement using colorings on nodes based on their neighbors and their colors (Morgan, 1965). At each iteration, the colors of the nodes in a graph are updated until the coloring becomes stable (i.e., the colors do not change anymore). Graphs having the same distribution of colors are likely to be isomorphic. Iterative refinement methods are not a perfect test of isomorphism, but they are accurate and fast enough to be used in practice. The Weisfeiler-Leman method is a generalization of the coloring approach, where instead of single vertices, coloring is applied to tuples of vertices (Weisfeiler and Leman, 1968).

The search tree method introduced in McKay (1981) creates a tree that contains different partitions of the nodes of a given graph. The partition tree (or search tree) is used to search and locate automorphisms and isomorphisms of the original graph. Other articles explore the use of probabilistic "Las Vegas" (Monte-Carlo) methods for determining isomorphism between graphs (Babai, 1979). We note that all these methods are based on traversing the graph to find nodes with similar neighbors through deterministic iterative methods or probabilistic algorithms.

Special topology and parallel algorithms. Despite the computational complexity of finding isomorphism in general graphs, isomorphism can be determined in polynomial time for graphs

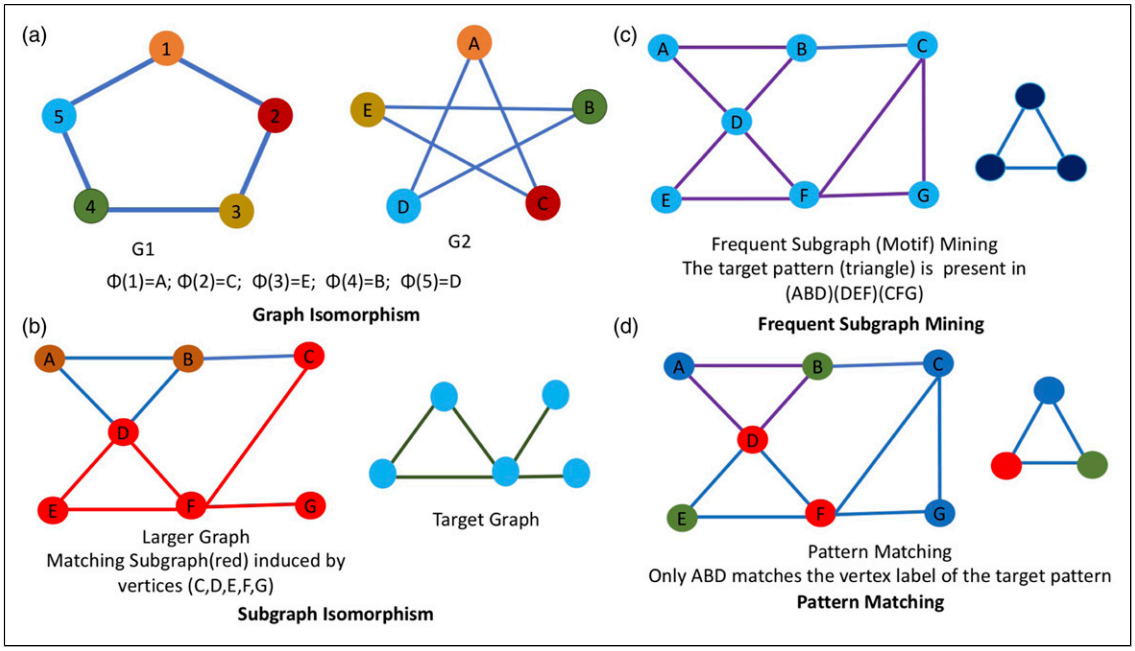

Figure 2. Overview of different types of exact graph comparison.

with special topologies. Among these are isomorphism between planar graphs (Hopcroft and Wong, 1974), bounded degree graphs (Luks, 1982), and tree graphs (Valiente, 2002). An algorithm for isomorphism on directed graphs is given in López-Presa and Fernández Anta (2009). It utilizes a method similar to the search tree algorithm in McKay (1981) to identify automorphisms through partitions of the graphs. Furthermore, a parallel algorithm was developed for graph isomorphism for general graphs Luks (1986), and a parallel algorithm for planar graphs was proposed in Jaja and Kosaraju (1988).

Despite these advances, current graph isomorphism algorithms cannot scale to large graphs even if they are parallel. Popular graph isomorphism software such as NAUTY (McKay and Piperno, 2014) can efficiently handle only graphs of about 100 vertices. We conjecture that this is because although graph isomorphism has been applied historically in computational chemistry (Akutsu and Nagamochi, 2013) and electronic circuits (Abiad et al., 2020), the problem scope is too stringent to scale for large graphs of over a million vertices, as is the current norm. More practical applications focus on subgraph isomorphism and pattern matching, as discussed in the following section.

Subgraph isomorphism and its variations

The subgraph isomorphism problem generalizes the graph isomorphism problem as follows: given a smaller target graph P and a graph G, "Does a subgraph $G_0 \subseteq G$ such that G_0 is isomorphic to P?". An example is given in Figure 2(b). The subgraph isomorphism is NP-complete (Cook, 1971). One of the first subgraph isomorphism algorithms was proposed in Ullmann (1976). A more recent algorithm VF2 (Cordella et al., 2004), solves the general subgraph matching problem by using a search tree with nodes representing possible matchings between the pattern and target graph. This tree is pruned and searched to identify a subgraph isomorphism. Subgraph isomorphism can be solved with polynomial complexity on graphs with particular topology, such as planar graphs in Eppstein (2002).

Due to its use in a wide range of disciplines, including bioinformatics (Bonnici et al., 2013), security applications (Chen and Tsourakakis, 2022), and social sciences (Zhao et al., 2010) many parallel algorithms for solving subgraph isomorphism exist. The "solution-biased" search algorithm (Archibald et al., 2019) randomizes how to search the space of possible isomorphisms. The parallel solution-biased search algorithm uses a combined MPI plus shared memory approach. GraphPI (Shi et al., 2020) proposed a shared memory subgraph matching algorithm. The cuTS tool (Xiang et al., 2021) introduces a GPU-based parallel implementation of subgraph isomorphism and can use multiple GPUs in a distributed memory format. Both these articles focus on identifying an optimal path to search the target graph, thereby reducing the redundancies and the number of options to check. Neural Subgraph Matching tool (Lou et al., 2020) uses neural networks to embed the

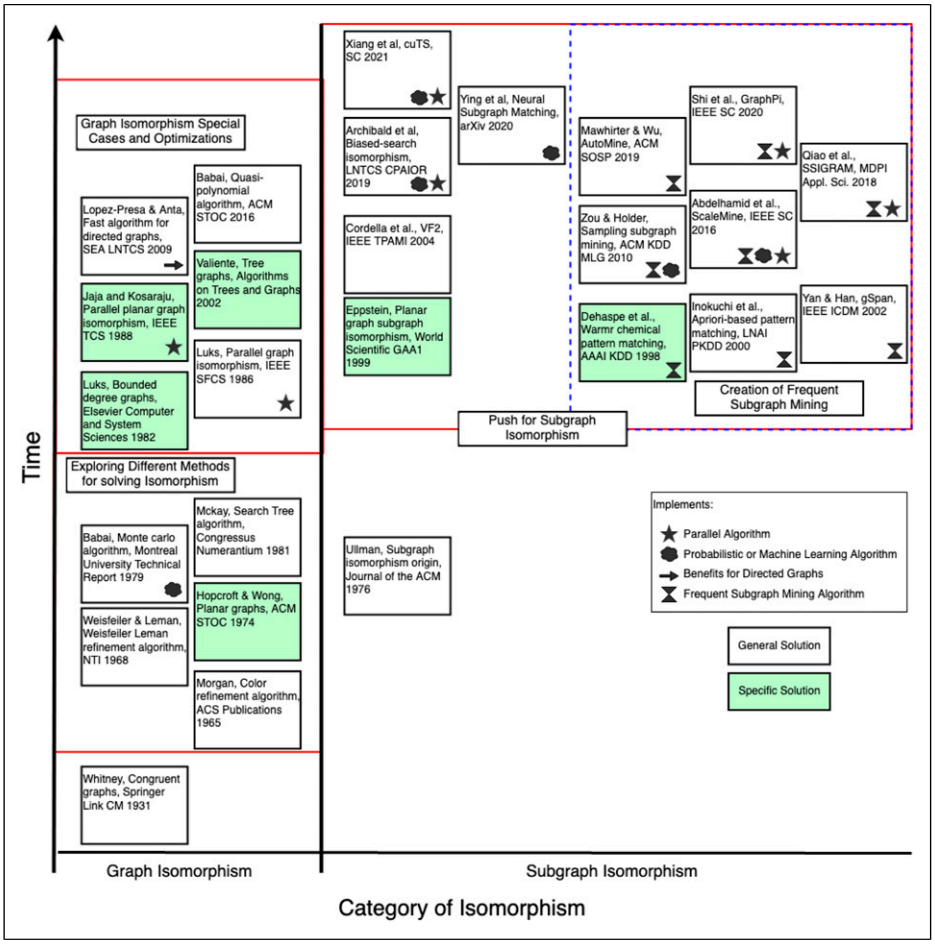

Figure 3. A history of major articles summarizing the literature of exact graph comparison and graph isomorphism.

decomposition of a target graph into a space that orders the decomposition based on features of subgraphs.

Frequent subgraph mining. The subgraph isomorphism can be further extended to frequent subgraph mining (FSM) problem, which asks whether there is a set of subgraphs G_0, G_1, ..., $G_n \subseteq G$ such that G_0, G_1, ..., G_n are each isomorphic to P. An example is given in Figure 2(c).

Subgraph isomorphism algorithms influenced frequent subgraph miners because each possible matching subgraph in the target graph must be tested for isomorphism. FSM algorithms were motivated by subgraph miners focusing on mining for real-world subgraphs, such as the Warmr algorithm (Dehaspe et al., 1998), which mined for specific chemical substructures.

Early approaches to solving the FSM problem in general graphs include the Apriori-based method (Inokuchi et al., 2000) and the gSpan method (Yan and Han, 2002). Apriori-based solutions to FSM generate "candidate matches," or possible isomorphisms, and then test the candidate match for isomorphism. gSpan and similar methods instead organize possible matches into a search tree and search that space of options for subgraph isomorphisms. More recently, AutoMine (Mawhirter and Wu, 2019) has generated code for efficient mining for subgraphs by adjusting the code to work well on the patterns specified by the user.

Probabilistic methods focus on sampling possible subgraph matches (Zou and Holder, 2010) or identifying graph statistics based on highly frequent subgraphs, like with ScaleMine (Abdelhamid et al., 2016). The tool SSIGRAM (Qiao et al., 2018) utilizes machine learning functionality in Spark to calculate subgraph isomorphism. This allows it to integrate with other machine learning workflows. ScaleMine and SSIGRAM also utilize parallelism to improve the scalability of their code. The study of scalable FSM continues using parallelism in tools like GraphPi and cuTs. SSIGRAM is implemented across 20 compute nodes using distributed memory parallelism and uses one core per node. ScaleMine, GraphPi, and cuTs execute at a higher scale combining distributed memory and shared memory parallelism over graphs of a million vertices.

Motif finding is another crucial aspect of FSM (Yan and Han, 2002); ?); Slota and Madduri, 2013). The goal is to find topologically well-defined motifs such as triangles and cliques (Gianinazzi et al., 2021) and k-truss (Kabir and Madduri, 2017) to identify the properties of the graph. Counting triangles are well studied as it estimates whether it is community-like. Recent parallel algorithms for triangle counting on massive graphs include (Kolda et al., 2014) using MapReduce (Wolf et al., 2017), linear algebra primitives on Kokkos Kernels, and dynamic load balancing on distributed memory systems (Arifuzzaman et al., 2015).

Pattern matching. A further extension to subgraph isomorphism is pattern matching, where the goal is to find a topologically equivalent subgraph and match the vertex labels. An example is given in Figure 2(d). The term pattern matching is overloaded and, in certain publications, is treated interchangeably with subgraph isomorphism. Here we use pattern matching to refer to algorithms matching structure and labels.

Pattern matching is increasingly becoming critical with the rise of knowledge graphs, where the vertices and edges contain essential information. Knowledge graphs are typically formed of heterogeneous vertices of different types, such as relations between co-authors and the articles they publish in social networks, connections between various diseases and compounds in medical networks, or relations between words in a text in natural language processing. Apart from sequential algorithms (von Der Malsburg, 1988), efficient parallel algorithms also exist that scale to million node graphs (Reza et al., 2018). A survey of pattern matching for large graphs with attributes is given in Bouhenni et al. (2021).

Graph alignment

In contrast to exact graph comparison, graph alignment provides matching between vertices of graphs with high correspondence. The *network (or graph) alignment* (NA) problem focuses on matching vertex pairs as follows; Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be graphs. G and H are aligned if there exists a function ϕ_A: $V_G \rightarrow V_H$ such that $\phi_A(v_G) = v_H$ for a pair of vertices $v_G \in V_G$ and $v_H \in V_H$, that have similar properties, such as similar local substructures or common node labels. Network alignment uses variations of subgraph isomorphism to compare two graphs which may be of different sizes Kuchaiev et al. (2010); Khan et al. (2012). Each approach to solving the graph alignment problem uses different similarity definitions based on factors including computational scalability, domain knowledge, and accuracy metrics. Unlike exact graph comparison tools, graph alignments do not need to be perfectly matched. Graph alignment is used to compare graphs known to be nearly, but not exactly, similar. Figure 4 provides an overview of the timeline of critical articles in graph alignment. The same articles are discussed in detail in this section.

The *global network alignment* problem (GNA) is satisfied if every vertex $v_G \in V_G$ and $v_H \in V_H$ are aligned to a vertex in the opposing graph. The local network alignment problem (LNA) requires only a subset of the vertices to be aligned. A one-to-one alignment occurs when the alignment function ϕ_A is bijective. This means that each node in one input graph is aligned to exactly one node from the other. A many-to-many alignment is when the function is not bijective. This means that each node in one input graph can be aligned with multiple nodes from the other. Figure 5 visualizes how different one-to-one and many-to-many alignments can be.

A graph alignment software tool can be a pairwise or multiple network aligner. A pairwise network alignment (PNA) exists when the tool creates alignments between exactly two graphs. A multiple network alignment (MNA) exists when the tool simultaneously creates alignments between three or more graphs. A comparison of PNA and MNA methods is given in the survey (Vijayan et al., 2020). Figure 6 visualizes the difference between multiple and pairwise alignments.

Pairwise network alignment

Early graph alignment tools were designed to align nodes across protein networks. PathBLAST (Kelley et al., 2004) introduced the concept by matching sequences (or paths) of proteins with similar sequences in more extensive protein networks. NetworkBLAST (Kalaev et al., 2008) expanded on this idea by matching complex structures of proteins with similar structures in more extensive protein networks. This expansion allowed aligning graphs more generally than just as protein networks. PathBLAST and NetworkBLAST only match a pattern with a subgraph of a larger graph. This inexact matching of subgraphs is called a local network alignment (LNA). Isorank introduced the concept of a global network alignment (GNA), or the alignment of whole graphs to compare entire graph structures. The recent survey (Meng et al., 2016b) discusses the differences between LNA and GNA methods and reconciles them by introducing methods for comparing tools across both LNA and GNA.

Both NetworkBLAST and Isorank utilize a scoring and alignment system called two-stage alignment. The first stage of two-stage alignment extracts a matrix of similarities between pairs of nodes by applying a measure of commonality (similarity score) to the data. The second stage calculates the best alignment matches by applying an optimization method (alignment method) to the similarity matrix data. Every alignment method can use a different similarity scores and optimization metric. For example,

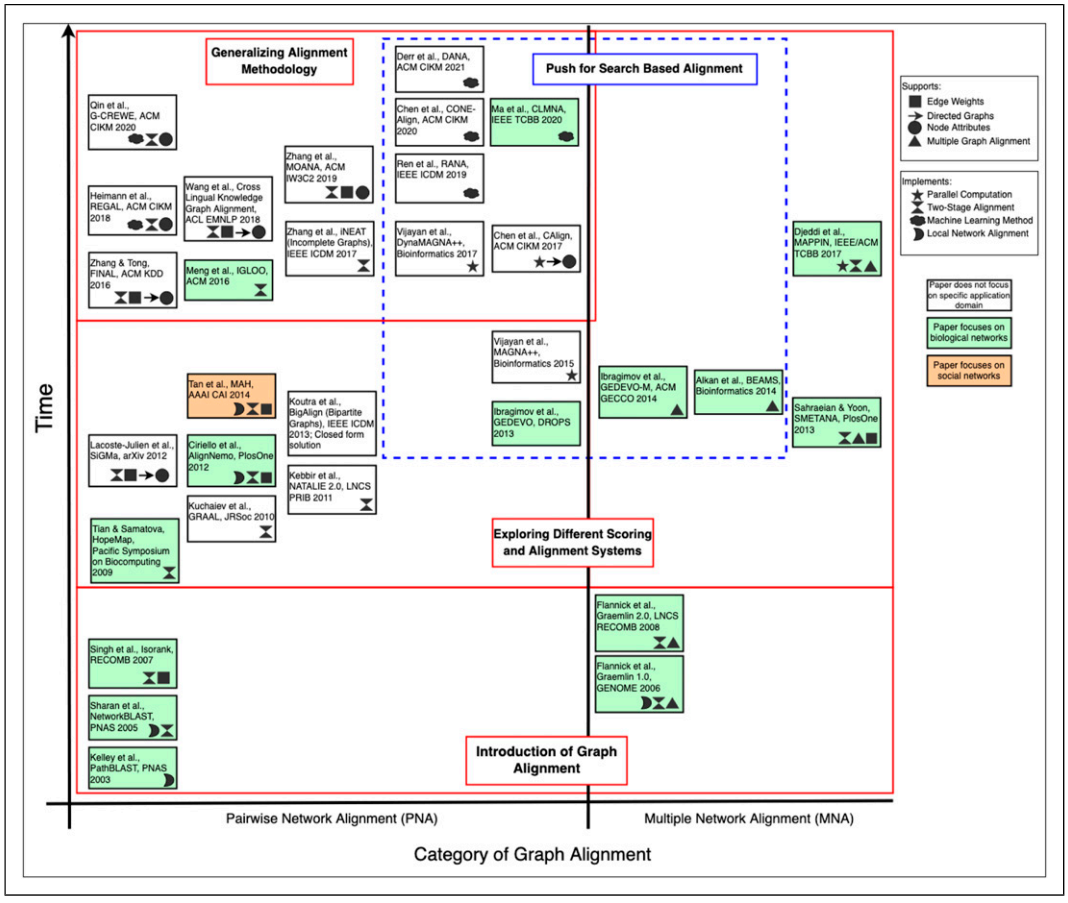

Figure 4. A history of major articles summarizing the literature of graph alignment.

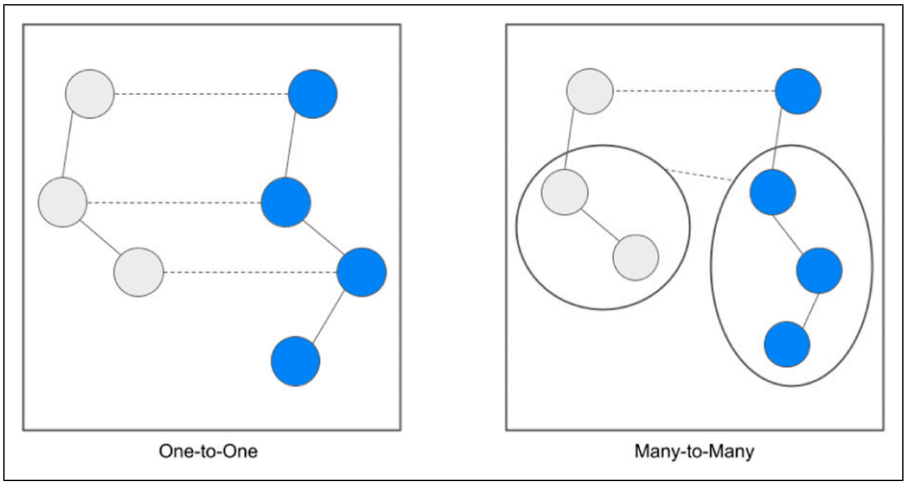

Figure 5. The left subfigure displays a one-to-one alignment. Each node in one graph is matched to exactly one node in the other graph. The right subfigure displays a many-to-many alignment. Each node in one graph can be matched to multiple nodes in the other graph.

similarity for Isorank means similar local substructures for each node and similar BLAST scores for matched nodes. Figure 7 visualizes this framework for two-stage aligners. Most pairwise network aligners produce a one-to-one alignment. HopeMap (Tian and Samatova, 2013) is a pairwise aligner that makes a many-to-many alignment.

Many network aligners inspired by PathBLAST and Isorank provide the option to utilize node-label information

about proteins to improve accuracy while aligning protein networks. GRAAL (GRAph ALigner) (Kuchaiev et al., 2010) measures the similarity between nodes of graphs by calculating local substructures called graphlets for each node. Graph nodes are similar when they have similar graphlets. GRAAL can be used across any graph structure because it is not built around the possible use of protein information.

NATALIE 2.0 (El-Kebir et al. 2011) is based on optimizing a Lagrangian relaxation of a maximum weight matching problem. SiGMa (Lacoste-Julien et al., 2013) utilizes information about edge direction and generic node labels within input graphs to improve the accuracy of results on graphs with edge direction and node labels. BigAlign (Koutra et al., 2013a) introduces a closed-form solution to the alignment problem when aligning bipartite graphs. All these examples are for global network alignment.

AlignNemo (Ciriello et al., 2012) introduces a local network aligner that aligns networks faster and more accurately than its predecessors. A survey of local and global network alignment tools in biology is provided in Guzzi and Milenković (2018). Network alignment has also been used in specific domains other than biology. MAH (Tan et al., 2014) creates local alignments of social networks. GEDEVO (Ibragimov et al., 2013) uses an existing measure of graph difference, GED (Sanfeliu and Fu, 1983), to create an alignment. MAGNA++ (Vijayan et al., 2015) aligns graphs by matching the topological information of both the nodes and the edges of graphs. GEDEVO and MAGNA++ utilize a method different from two-stage alignment, called search-based alignment. Search-based alignment methods optimize alignment while calculating the commonality score between nodes. This is opposed to two-stage aligners, which separate those tasks into different stages. Figure 8 visualizes how search-based aligners work. Unlike two-stage aligners,

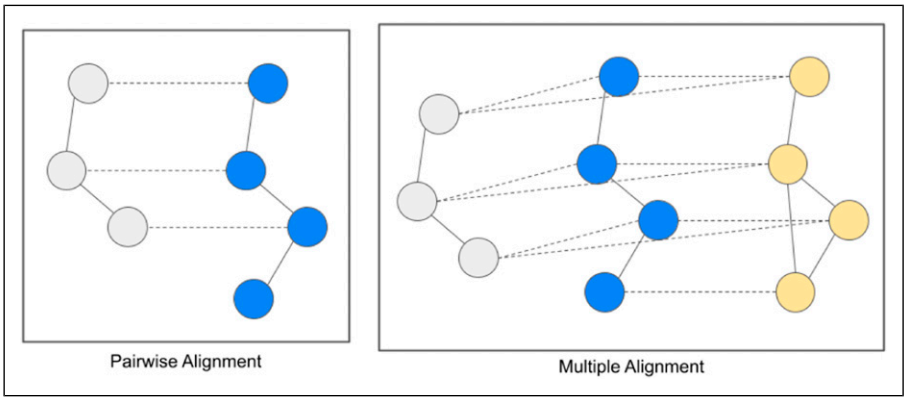

Figure 6. The left subfigure displays a pairwise network alignment. Exactly two graphs are aligned. The right subfigure displays a multiple network alignment. More than two graphs are aligned at once. In practice, many MNA tools can simultaneously align an arbitrary number of graphs.

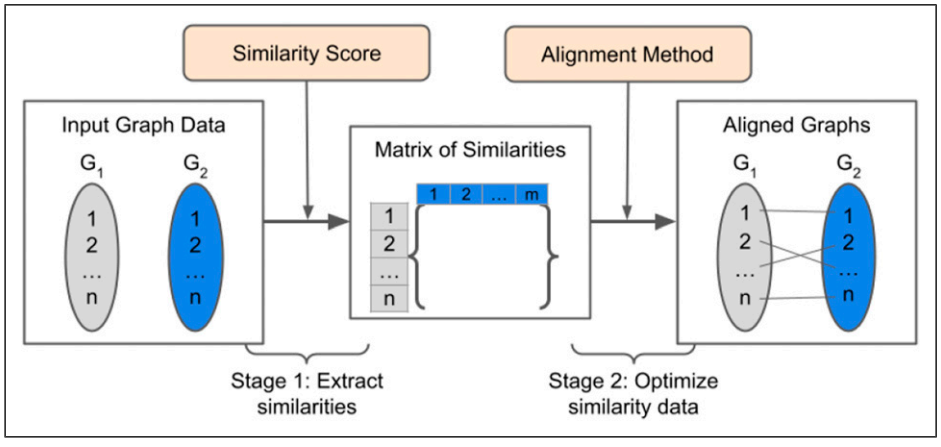

Figure 7. An overview of how two-stage network aligners work. In Stage 1, a similarity score is applied to input graph data to extract the similarities between nodes of the graphs. In Stage 2, an optimization algorithm (alignment method) is applied to the matrix of similarities to determine which nodes are most similar and should be aligned.

visualized in Figure 7, search-based aligners utilize an alignment method that directly applies a similarity score while searching over the optimization space.

Generalizing alignment methodology. Recently, pairwise network aligners are generalizing alignments to allow users to incorporate more problem-specific information into their alignments. The FINAL (Zhang and Tong, 2016) graph aligner models graphs as structures with node attributes and edge attributes in matrices. The attribute generalization allows FINAL to utilize any graph node data and edge data to improve the accuracy of alignments. FINAL also allows the user to define prior alignment data in a matrix, which lets the algorithm adjust when the alignment of some parts of a graph is already known. The alignment tool IGLOO (Meng et al., 2016a) generalizes the construction of a global network alignment by creating a temporary local network alignment between two networks and extending it until the entire graphs are aligned. iNEAT (Zhang et al., 2017) generalizes the alignment methodology to align graphs with incomplete information.

In the cross-lingual knowledge graph alignment article (Wang et al., 2018), graphs are modeled as knowledge graphs. This representation allows the graph alignment to use graph data to improve the alignments. MOANA (Zhang et al., 2019) builds alignments by comparing the alignment quality of the input graph's substructures across many granularities of the input networks.

Generalization of Alignment. Researchers are also generalizing search-based network alignment. DynaMAGNA++ (Vijayan et al., 2017) aligns dynamic graphs, which generalizes to graphs that can evolve with time. CAlign (Chen et al., 2017) generalizes by allowing the user to decide whether to make a one-to-one or a many-to-many alignment.

Using Machine Learning. REGAL (Heimann et al., 2018) utilizes machine learning to learn the features of each node in the input graphs. It then compares the graphs based on these learned features. G-CREWE (Qin et al., 2020) uses graph convolutional networks to learn features to embed nodes into. The learned features and embedding compress the graph into a smaller one. This smaller graph is aligned as a representative of the original graph to make the alignment easier to compute. Additional machine learning-based methods for aligning graphs are introduced as search-based methods. RANA (Ren et al., 2019) uses an adversarial learning approach to embed two graphs into a space for alignment. CONE-Align (Chen et al., 2020) embeds graphs into a space to match the neighborhoods of nodes in that space. These neighborhoods are then used to match individual nodes. CLMNA (Ma et al., 2020) simultaneously learns three different alignments (or proximity) between the nodes of two graphs and then combines the information from these three proximities to form a final alignment. DANA (Derr et al., 2021) uses adversarial learning like RANA to embed two graphs into a space of features, and it uses a nearest-neighbor approach to align the embedded graphs.

Multiple network alignment. Multiple network alignment was initially motivated to solve local network alignment similar to pairwise network alignment. Graemlin 1.0 (Flannick et al., 2006) was designed to solve the alignment problem across multiple graphs simultaneously. The authors next utilized Graemlin 1.0 to create Graemlin 2.0 (Flannick et al., 2008), a global network alignment that is also a multiple network alignment. SMETANA (Sahraeian and Yoon, 2013) utilizes random walk probability methods to estimate the similarity across nodes of different graphs. GEDEVO-M (Ibragimov et al., 2013) extends the existing pairwise network alignment method, GEDEVO (Ibragimov et al., 2013), to allow it to align multiple graphs at once without a conflicting alignment. GEDEVO-M and BEAMS

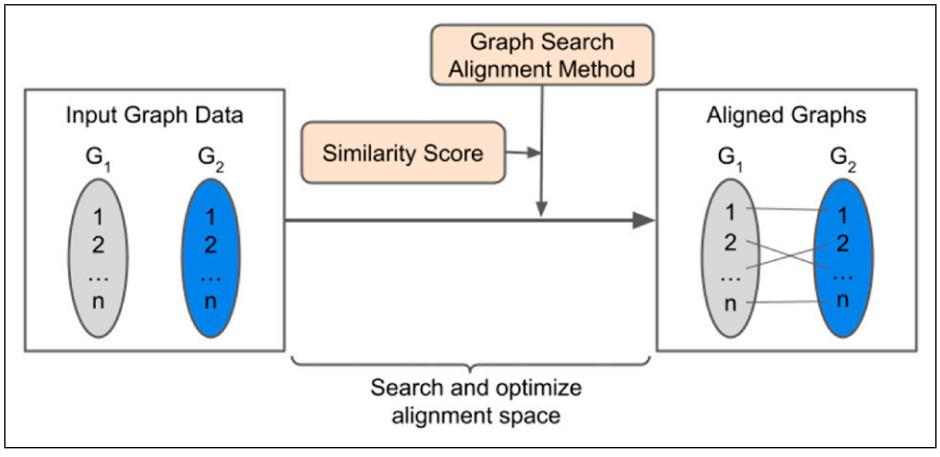

Figure 8. Overview of search-based methods for network alignment. An optimization algorithm (alignment method) leverages a similarity score to search the space of possible alignments. This optimal alignment from the search is selected as the network alignment.

(Alkan and Erten, 2014) introduce the existence of search-based multiple network aligners. MAPPIN (Djeddi et al., 2018) utilizes shared memory parallelism to improve the runtimes of calculating the similarities and alignments between graphs.

Alignment versus isomorphism

Because network alignments calculate mappings between the vertices of input graphs based on commonalities, an alignment has the potential to identify an isomorphism between the graphs. Both graph isomorphism and global network alignment calculate a mapping between whole graphs. Both subgraph isomorphism and local network alignment calculate mappings between subgraphs of the input graphs. Graph isomorphisms and network alignments, however, are not the same. A graph isomorphism constitutes two bijective functions that match both the vertices and the edges of the input graphs. A graph alignment only requires a mapping between the vertices, and the alignment mapping need not be bijective unless the alignment is one-to-one.

Quantitative functions for graph comparison

Graph and subgraph isomorphism produce a one or zero output; either the isomorphism exists or does not exist. Frequent subgraph matching and graph alignment extend the output to several patterns or vertices matched. More fine-grained quantitative metrics for graph comparisons exist and go under the designation of quantitative function methods because they use graph metrics. Graph metrics quantify the correspondence between two graphs $G \in \mathcal{G}$ and $H \in \mathcal{G}$ by the output of a function $S : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}$. Many variations exist, including edit distances, graph kernels, and statistical measures such as degree distribution. A graph metric can be an unknown node correspondence (UNC) metric or a known node correspondence (KNC) metric (Koutra et al., 2013b). A UNC metric measures the correspondence between graphs without prior knowledge of the correspondence between the graphs. A KNC metric uses prior knowledge about the correspondence between graphs during the metric calculation. A graph metric can be a similarity, a distance, or a kernel metric. A graph similarity metric measures how much is in common between two graphs. A graph kernel measures how much is in common between the embeddings of two graphs into a feature space. Figure 9 displays the history of the graph metrics in literature. The articles are discussed in detail in this section.

Graph similarities and distances

Early in graph similarities and distances development, the graph edit distance (GED) (Sanfeliu and Fu, 1983) introduced a way to measure the difference between two graphs. Graph edit distance measures the number of changes required to transform from one graph to another. Other metrics of similarity include vertex/edge overlap (VEO), and Vertex/edge vector similarity (VS) (Papadimitriou et al., 2010). VEO measures similarity between graphs as sharing common vertices and edges. VS instead measures similarity between graphs as sharing similar vertex and edge weights. The degree distribution of the graph is typically used to classify large graphs. A graph distance can also be formed from the degree distribution statistic (Malod-Dognin et al., 2014) by measuring the difference between two graphs as the difference between the degree distributions of the graphs.

DeltaCon (Koutra et al., 2013b) measures the similarity between two graphs based on their familiar neighbors up to a given distance. This method is similar to the iterative method of graph isomorphism and can scale to graphs of over one million nodes. NetDis (Ali et al., 2014) uses graphlets as representations of graph nodes to rapidly compare the two graphs' differences. Using graphlets provides the ability to compare large graphs based on comparing many small structures.

The directed graphlet correlation distance (DGCD) (Sarajlić et al., 2016) uses directed graphlets to represent graph nodes. This allows measuring the distance between directed graphs using the information encoded in graphlets. The family of tractable graph distances (FTD) (Bento and Ioannidis, 2018) introduces a graph distance that takes a different arbitrary distance between graph nodes and creates a new distance metric by combining it with the difference between the adjacency matrices of the two graphs.

Another critical graph similarity measure is based on their spectral properties R. et al. (2018). The spectrum of a graph is the eigenvalue distribution of either their adjacency or Laplacian matrices and provides insights into the graph's topology. In Wilson and Zhu (2008), the authors compare the edit distance and graph spectra. Since graph spectrum can change significantly with a small change in graphs, this metric is useful for detecting small changes. However, the spectrum is not unique to a graph. Thus zero difference does not necessarily mean that the graphs are isomorphic.

Graph kernels

The kernel trick is a method for binary classification. Consider a set of points in an input space that is challenging to classify using a support vector machine, that is, partition using a hyperplane. However, if these points were embedded in a different feature space, they could be partitioned

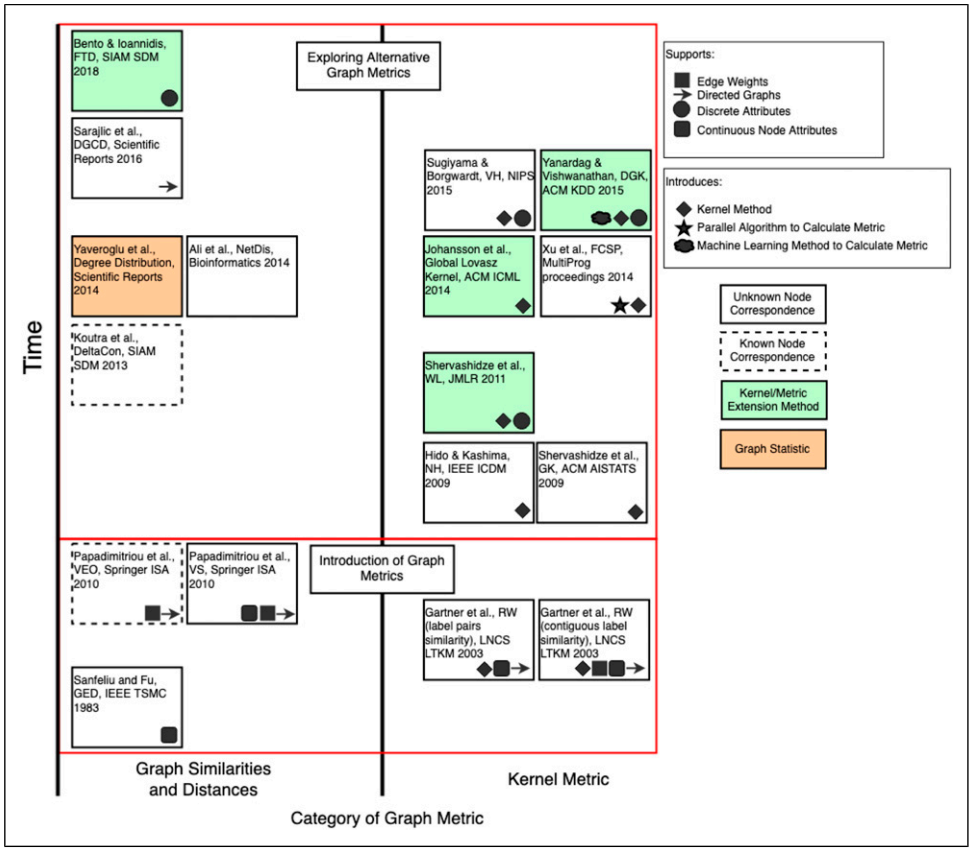

Figure 9. A history of major articles summarizing the literature of quantitative functions for graph comparison.

more easily. Graph kernel functions apply this kernel trick to compare graphs. The kernel function takes the vectors in the original input space, maps them to a feature space, and returns the dot product of the vectors in the feature space.

Formally, given two graphs G, and G' and an graph embedding ϕ, a graph kernel is defined as $K(G, G') = \langle \phi(G), \phi(G') \rangle$ where $\phi(G)$ denotes the *embedding* of G in a specific kind of vector space. Thus, the kernel similarity $K(G, G')$ is the value of the inner product of the embeddings of G and G' in that vector space. Graph kernel methods focus on designing embeddings that can appropriately quantify the difference between the graphs.

The random walk (RW) (Gärtner et al., 2003) graph kernel introduces the label pairs and the contiguous label random walk. The label pairs kernel measures the similarity of graphs based on the similarity of the labels that exist between paths through the graphs. The label pairs kernel only measures the similarity between graphs with discrete node labels. The contiguous labels kernel compares the input graphs based on the similarity of their labels within the space of all possible labeled paths. The contiguous label kernel measures the similarity between graphs with continuous labels by comparing the similarity between paths with continuous labels.

The neighborhood hash (NH) kernel (Hido and Kashima, 2009) iteratively hashes the labels of a node's neighbors into the node's label so that the kernel can compare graphs based on these hashed labels. It can hash the labels and calculate the kernel in linear time complexity. The graphlet kernel (GK) (Shervashidze et al., 2009) compares graphs based on the similarity of their graphlets. The Weisfeiler-Lehman graph kernels (Weisfeiler and Leman, 1968); Shervashidze et al., 2011) is a set of kernels that use an iterative node-label refinement like the NH kernel but instead apply a separate arbitrary kernel at each iteration and combine the results of those calculations for a final measure of similarity. Thus it has the advantages of both iterative refinement and the base kernel used.

The fast computation of shortest path (FCSP) kernel (Xu et al., 2014) uses parallelism on both the CPU and GPU to speed up the existing shortest path kernel (Borgwardt and Kriegel, 2005) calculation. FCSP uses shared memory parallelism for both the CPU and GPU parallelism. The global Lovasz kernel (Johansson et al., 2014) compares graphs based on the similarity of the Lovasz number Lovàsz (1979) of the graphs. Recently, the vertex histogram (VH) kernel (Sugiyama and Borgwardt, 2015) compares graphs based on the frequency of their node labels. The deep graph

kernel (DGK) (Yanardag and Vishwanathan, 2015) uses machine learning to learn the input graphs' subgraphs and compare them based on their similarity subgraphs.

In contrast to graph/subgraph isomorphism or graph alignment methods, graph comparison functions provide a single actual number as output, providing a quantitative measure of the similarity between two graphs. Based on the functions used, these metrics can differ considerably.

A rare case of graph kernel application to nondeterminism

ANACIN-X (Chapp et al., 2021) is a rare case of application of graph kernel for identifying the percentage and sources of communication nondeterminism. The software framework models parallel executions as directed graphs and leverage graph kernels to characterize run-to-run variations in inter-process communication. ANACIN-X demonstrates the potential of graph kernel similarity as a proxy for nondeterminism by showing that these kernels can quantify the type and degree of nondeterminism present in communication patterns. To demonstrate the framework's ability to link and quantify runtime nondeterminism to root sources, Chapp and coworkers showed results for an adaptive mesh refinement application, where our framework automatically quantifies the impact of function calls on nondeterminism, and a Monte-Carlo application, where our framework automatically quantifies the effect of parameter configurations on nondeterminism.

Limits of general graph comparison methods

For a realistic deployment of graph comparison methods for the study of nondeterminism in HPC applications, we have to capitalize on the methods' sensitivity, scalability, specialization, and tunability. Our literature survey outlines how existing comparison tools commonly pursue one or several of the features but none of the existing tools pursues all of the four features simultaneously. Existing algorithms primarily compare graphs based on their commonality rather than their difference. Commonality refers to how much is in common across graphs and is distinct from similarity quantifying the commonality between graphs as a single value (e.g., graph kernels). Sensitivity costs for recording changes in event graphs depend on the application configuration. Scalability cannot be achieved via sampling because every node in an HPC graph is important. The memory overhead associated with comparison tools is considered a major obstacle to scalability. There are no current tools or methods that are specialized to HPC domain knowledge and metadata because graph comparison has not been well studied with regard to its applications in HPC. Existing forms of graph comparison often use pre-defined nodes, features, or comparison types because otherwise, the comparison is difficult to tune. These challenges are exacerbated when the nondeterminism of the application induces nondeterminism in the overhead imposed by the tool. When tool overheads become nondeterministic, the tradeoff between tool utility and tool overhead becomes infeasible for the HPC user to consider. Unfortunately, the user may opt not to use the tool rather than deal with unpredictable overheads. To make the problem worse, application inputs and system configurations influence the degree of nondeterminism in the executions.

Properties such as sensitivity, scalability, specialization, and tunability highlight the open challenges for graph comparison algorithms to understand nondeterministic patterns in HPC applications at exascale. A collaboration between the graph algorithm community and the HPC community can ultimately lead to transformative techniques to address HPC's robustness (e.g., dealing with rare bugs) and reproducibility (e.g., dealing with different application results) issues.

Existing graph algorithms in HPC and unaddressed needs in nondeterminism

The HPC community has recently deployed graph comparison algorithms for optimizing HPC simulations and network analysis, record-and-reply of HPC applications in production HPC environments, and, in a few cases, studying individual aspects of nondeterminism in HPC executions. There are still important unaddressed research needs to fill the gap between graph comparison algorithms and nondeterminism in HPC.

Graph methods for HPC optimizations

There is a rich history of graph algorithms in optimizing HPC simulations to study properties of complex HPC systems, with the goal of (a) finding significant subgraphs in networks (e.g., by using community detection and network motifs) and (b) developing metrics of comparison across networks, in the context of identifying nondeterminism (e.g., network alignment).

Graph theory is used to optimize the computational resources in HPC simulations. Graph partitioning algorithms (Buluç et al., 2016) and software such as METIS (Karypis and Kumar, 1998) and Zoltan (Boman et al., 2012) are extensively used to reduce communication and balance workload across processes. Graph algorithms are used in sparse matrix solvers, from lowering the extra memory costs of fill-in (Bui and Jones, 1993) to compiler optimization by analyzing dependency graphs (Cheshmi et al., 2017) and reducing communication costs on large parallel systems

(Mohiyuddin et al., 2009). Graph algorithms are also used in different aspects of automatic differentiation, a method for computing more accurate derivatives using the chain rule. Minimizing vertex coloring reduces the memory requirement for computing the Jacobian matrix (first-order derivatives) (Gebremedhin et al., 2005). Finding the line of symmetry in directed acyclic graphs reduces the computation costs of the Hessian matrix (second-order derivative) (Bhowmick and Hovland, 2008).

Software solutions for nondeterministic executions

In recent years, increasing awareness of the need to manage nondeterminism in parallel applications (Gopalakrishnan et al., 2017) has given rise to a few software solutions. Three important solutions most closely address the problem: PopMine (Seo et al., 2011), SABALAN (Alimadadi et al., 2018), and ANACIN-X (Chapp et al., 2021). PopMine Seo et al. (2011) is a tool that analyzes traces of message orders in MPI applications, also represented by an event graph-like structure to determine a minimal DAG that triggers a given bug. PopMine does not search for a message order that triggers a specific bug. SABALAN (Alimadadi et al., 2018) is a tool that analyzes trace data, albeit not from MPI applications, and finds hierarchies of motifs that can potentially be linked in bug manifestation. SABALAN cannot compare communication patterns across multiple executions and searches for possibly nondeterministic execution motifs within a single run. ANACIN-X enables a generalized notion of anomaly detection for nondeterministic communication patterns and thus allows localization of nondeterminism that impacts scientific correctness (e.g., through the interaction between nondeterministic communication and floating-point non-associativity) rather than being restricted to bugs that cause crashes or hang. However, it supports only point-to-point MPI communications.

In production HPC environments, record-and-replay tools allow users to record a nondeterministic application's execution and then replay it exactly, thus enabling the reproducibility of nondeterministic bugs (Chapp et al., 2018). State-of-the-art record-and-replay tools such as ReMPI (Sato et al., 2015) target production-scale runs and prioritize scalability in terms of runtime and record size. Other record-and-replay tools target hybrid MPI + OpenMP executions (Budanur et al., 2012), MPI applications using one-sided communication (Qian et al., 2016b,a), replay of isolated subgroups of processes (Xue et al., 2009), and probabilistic replay (Park et al., 2009). In addition, tools such as NINJA (Sato et al., 2017) are used in conjunction with record-and-replay tools to improve the chances of capturing nondeterministic bugs. These record-and-replay tools do not focus on localizing nondeterminism and mapping it back to previously unknown root causes, but rather they aim at determinizing executions.

Four critical unaddressed needs

Our study of the current literature points out how manuscripts on graph comparison methods target essential aspects in scientific domains in general and HPC in particular but miss to address four critical needs of HPC communities dealing with nondeterministic applications: debugging and testing, fault-tolerance, resilience, and reproducibility; data center administration; and training.

The *first need* is for agnostic tools that can be applied to a broad range of applications across four communities (i.e., HPC application developers, HPC researchers, data center administrators, and educators) rather than being of interest to a single one dealing with a single programming model, application, comparison technique, or platform.

The *second need* is the scale at which the communities must address recording overheads without the need to run the applications on real platforms. Current graphs representing executions have grown in complexity; however, the scale at which the associated graph recording can run strongly depends on desired fidelity. Recording and replaying execution traces on over 5K processes is feasible for reproducibility studies but is still a barrier for debugging, particularly for long-running applications.

The *third need* is identifying frequently occurring subgraphs representing nondeterministic executions motifs and quantifying dissimilarities across the motifs observed within execution or across independent executions of the same HPC application. The challenge here is that motifs are based not only on the graph structure but also on annotations of the vertices and edges; these annotations describe function calls and their nondeterministic resource usage components (i.e., CPU, memory, and power) that lead to recording overheads.

A *fourth need* is to integrate power usage in a graph recording an execution in addition to time and other resource information. Haider and coworkers (Haidar et al., 2017) showed substantial value in supporting power monitoring in conjunction with performance counter monitoring. Currently, no framework exists for modeling nondeterminism and recording power metrics gearing toward energy-efficient nondeterministic applications.

Graph comparison and nondeterministic applications: opportunities and challenges

The rise of exascale computation has introduced new challenges to HPC, particularly in ensuring the reproducibility and robustness of computation in nondeterministic HPC applications. We define three instances where graph comparison techniques can be used for the reproducibility of executions, replicability of code, and fault-tolerance in graph algorithms.

Reproducibility of executions

As the number of processors increases, the asynchronous executions also increase, leading to nondeterminism in the execution. While nondeterminism is an inevitable byproduct of concurrency, nondeterminism can also hamper the reproducibility of executions. One method of studying reproducibility at runtime is by studying the event graphs of the executions of the same simulation code at multiple instances. Using event graphs to study simulations has been proposed by Kranzlmüller (Kranzlmüller, 2000). However, the event graphs described in Kranzlmüller's article are not feasible for studying simulations on exascale systems because debugging through visualization cannot scale to orders of millions of events. Moreover, event graphs in Kranzlmüller's work store all forms of nondeterminism regardless of the type of reproducibility under study, making it challenging to identify commonly occurring patterns in the graphs.

We address these deficiencies by modeling event graphs to represent the continuity or the chain of dependencies across the execution rather than simply listing each dependency separately on a time scale. We combine multiple events into one vertex as required by the aspect of nondeterminism under study. Our event graphs will form a directed network, where each weakly connected component represents a system of interdependent events. Consider a set of consecutive send functions from a single process without any receive function interleaving the associated send operations as an example of how events can be merged. Because there is no receive in between, the order in which they send messages is executed does not affect the final result. Thus, if our reproducibility concern is only obtaining the same result at each run, we can combine events representing a series of consecutive sends from a single process. Note that the techniques for combining events can be tailored to the reproducibility criteria (e.g., debugging the code), ensuring the reproducibility of results. Thus merging events gives us a flexible tool where we can create an event graph to represent exclusively the nondeterminism aspect being studied.

In ANACIN-X (Chapp et al., 2021), we demonstrate that quantitative comparison using graph kernels can indeed be an appropriate proxy for nondeterminism and can further be used to identify the functions of the root cause of nondeterminism. ANACIN-X takes as input execution traces for a nondeterministic synthetic application using the DUMPI tracing library Wilke and transforms them into event graphs. The event graphs are subsequently compared using the Weisfeiler-Lehman graph kernel. Figure 10 shows this comparison for the miniAMR applications presented in Chapp et al. (2021).

Replicability of code

Another challenge arising from exascale computations is ensuring the replicability of code. Optimized execution of large-scale simulations on heterogeneous systems necessitates software-hardware co-design. Ensuring correctness and replicability across the different code versions for the same simulation is essential. Graphs, particularly DAGs (Cosnard et al., 2004), are regularly used to represent codes at various levels, and as schedulers in heterogeneous architectures (Bosilca et al., 2011) The source code can be modeled as a graph representing each function as a vertex and dependencies between the functions as edges. Moreover, compiler optimization and automatic code generation regularly use expression graphs in their algorithms (Herholz et al., 2022); Kramer et al., 1992) Comparing the DAGS for the different refactored codes is an orthogonal method to identify whether the code is replicable after various optimizations.

This technique is already being used in software engineering to trace the evolution of software (Meyer et al., 2014); Qu et al., 2021). In a graph theory context, it has been observed that the essential functions are retained in the innermost k-cores of the graph. For HPC codes, the problem is more involved as the code will include lower-level operations, and it is not sufficient to only test whether functions are retained. In addition to the core-periphery analysis proposed in Meyer et al. (2014), we would also need to identify subgraphs that ensure certain operations are executed correctly and ensure their subgraphs or their equivalent exist in each of the expression graphs.

Fault-tolerance in graph algorithms

Despite the rise of parallel algorithms for graphs and fault-tolerance for large HPC simulations (Benoit et al., 2022); Nicolae et al., 2021), fault-tolerance for graph algorithms is in its infancy. Checkpointing is an essential component of fault-tolerance, which involves storing partially computed data between iterations. The challenge in keeping information about graphs is that every vertex is associated with unique data. For example, when adding single source shortest paths, each vertex will likely contain a partially calculated distance. Similarly, most community detection algorithms require multiple iterations over which the vertices can change communities. As the graphs can extend to millions of nodes, storing the information for every vertex is very expensive, and algorithms are needed to identify where the change occurred between two consecutive checkpointing steps.

Thus the checkpointing strategy has to be aware of the graph topology and the underlying algorithm. One approach would be to find frequently occurring subgraphs in the graph following similar computing patterns. For example, vertices belonging to a clique are likely to be in the same community, and thus, all the vertices in the same clique can be stored as one data point. Indeed using subgraph isomorphism to compress the graph can reduce the memory

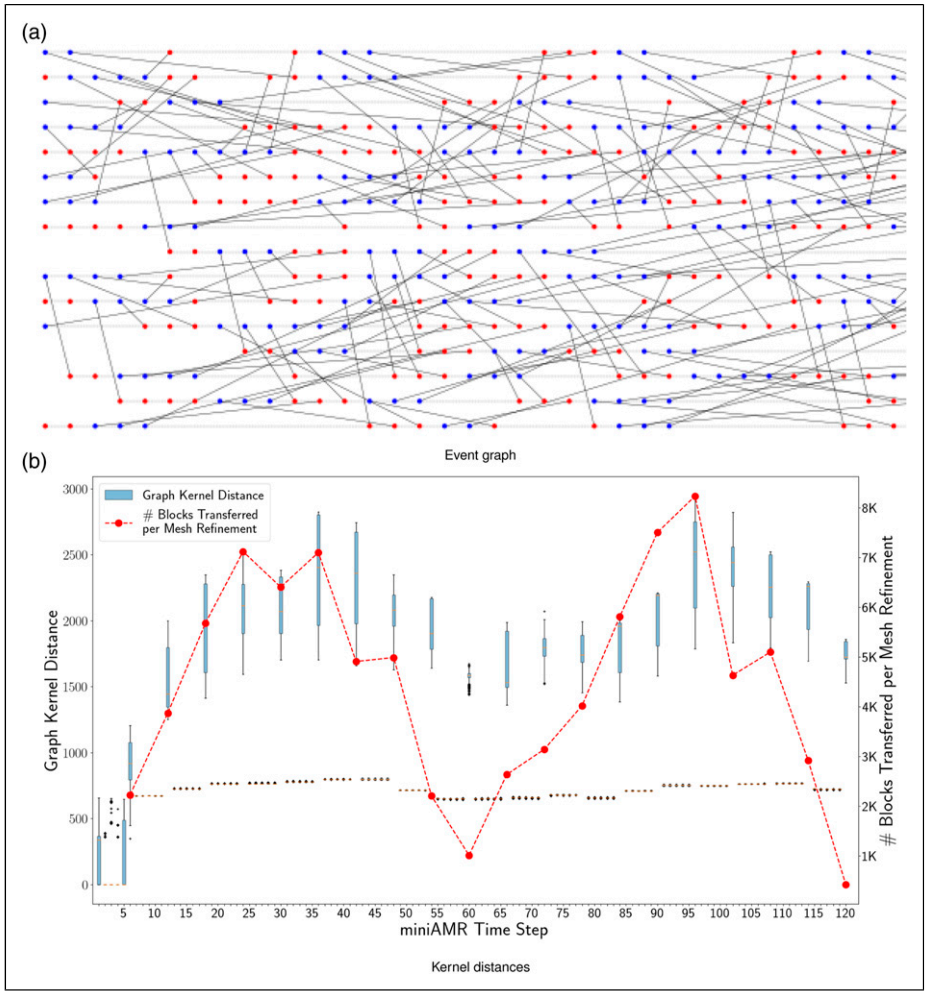

Figure 10. Event graph for a miniAMR simulation on 16 vertices (a) and co-relation between graph kernel distance and the work done at each time step (b) from Chapp et al. (2021).

requirements for checkpointing. If the checkpointing information can be retained even after the execution is complete, we can leverage this knowledge to predict the partial computation of a graph with a similar structure. For example, a graph G whose communities have been computed and selected checkpoints are stored at some of the iterations. If a graph H has a similar structure to G, then we can use the partial results of G as the starting iteration of H for faster convergence.

Conclusion

Leveraging graph comparison methods for understanding nondeterminism in HPC applications can impact four communities that are driving the exascale transition: HPC application developers can identify unintended sources of nondeterminism and manage necessary nondeterminism (e.g., associated with asynchronous executions of computation and communication), especially in the context of debugging and testing; the HPC researchers can tackle fault-tolerance, resilience, and reproducibility at exascale associated with nondeterministic patterns in application executions; data center administrators can develop, integrate, and manage scheduling policies and resources management of nondeterministic applications on the HPC machines; and educators can reach out to their students and postgraduates to promote HPC in general and exascale research in particular, without needing to access high-end, expensive computers. This article presents the state of art-of-the-art graph algorithms and outlines directions in which the existing algorithms should be designed and deployed to advance the need of these communities when solving the pending issues associated with nondeterministic HPC applications.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the United States National Science Foundation (Grant #1900888 and Grant #1900765).

ORCID iD

Michela Taufer https://orcid.org/0000-0002-0031-6377

Notes

1. The claim of the proof was debated, and the new proof after correction is available online but yet to be published in a journal.
2. K3 is a complete graph with three vertices (i.e., a triangle). K1,3 is a bipartite graph with one set containing one vertex and the other containing three vertices.

References

Abdelhamid E, Abdelaziz I, Kalnis P, et al. (2016) Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In: Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC). IEEE, pp. 716–727.

Abiad A, Grigoriev A and Niemzok S (2020) Printed circuit boards isomorphism: An experimental study. *Computers and Industrial Engineering* 148: 106715.

Ahn DH, Lee GL, Gopalakrishnan G, et al. (2013) Overcoming extreme-scale reproducibility challenges through a unified, targeted, and multilevel toolset. In: Proceedings of the 1st International Workshop on Software Engineering for High-Performance Computing in Computational Science and Engineering (SE-HPCCSE). ACM, pp. 41–44.

Akutsu T and Nagamochi H (2013) Comparison and enumeration of chemical graphs. *Computational and Structural Biotechnology Journal* 5(6): e201302004.

Ali W, Rito T, Reinert G, et al. (2014) Alignment-free protein interaction network comparison. *Bioinformatics* 30(17): i430–i437.

Alimadadi S, Mesbah A and Pattabiraman K (2018) Inferring hierarchical motifs from execution traces. In: Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, pp. 776–787.

Alkan F and Erten C (2014) BEAMS: backbone extraction and merge strategy for the global many-to-many alignment of multiple PPI networks. *Bioinformatics* 30(4): 531–539.

Archibald B, Dunlop F, Hoffmann R, et al. (2019) Sequential and parallel solution-biased search for subgraph algorithms. In: Proceedings of the International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. Springer, pp. 20–38.

Arifuzzaman S, Khan M and Marathe M (2015) A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. In: Proceedings of the IEEE International Conference on Big Data (Big Data). IEEE, pp. 1839–1847.

Babai L (1979) Monte-Carlo algorithms in graph isomorphism testing. Université Tde Montréal Technical Report. *DMS* (79-10).

Babai L (2016) Graph isomorphism in quasipolynomial time. In: Proceedings of the forty-eighth annual ACM symposium on Theory of Computing. ACM, pp. 684–697.

Bell P, Suarez K, Chapp D, et al. (2021) ANACIN-X: A software framework for studying non-determinism in MPI applications. *Software Impacts* 10, 100151.

Benoit A, Perotin L, Robert Y, et al. (2022) Checkpointing workflows à la young/daly is not good enough. *ACM Transactions on Parallel Computing* 9(4): 1–25.

Bento J and Ioannidis S (2018) A family of tractable graph distances. In: Proceedings of the 2018 SIAM International Conference on Data Mining (SDM). SIAM, pp. 333–341.

Bhowmick S and Hovland PD (2008) A polynomial-time algorithm for detecting directed axial symmetry in hessian computational graphs. Advances in Automatic Differentiation. Springer, pp. 91–102.

Boman EG, Çatalyürek ÜV, Chevalier C, et al. (2012) The zoltan and isorropia parallel toolkits for combinatorial scientific computing: partitioning, ordering and coloring. *Scientific Programming* 20(2): 129–150.

Bonnici V, Giugno R, Pulvirenti A, et al. (2013) A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics* 14: S13.

Borgwardt K, Ghisu E, Llinares-López F, et al. (2020) Graph kernels: State-of-the-art and future challenges. *Foundations and Trends® in Machine Learning* 13(5–6): 531–712.

Borgwardt KM and Kriegel HP (2005) Shortest-path kernels on graphs. In: Proceedings of the 5th IEEE International Conference on Data Mining (ICDM). IEEE.

Bosilca G, Bouteiller A, Danalis A, et al. (2011) DAGuE: A Generic Distributed DAG engine for high-performance computing. In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD. IEEE, pp. 1151–1158.

Bouhenni S, Yahiaoui S, Nouali-Taboudjemat N, et al. (2021) A survey on distributed graph pattern matching in massive graphs. *ACM Computing Surveys* 54(2): 1–35.

Budanur S, Mueller F and Gamblin T (2012) Memory trace compression and replay for SPMD systems using extended PRSDs. *The Computer Journal* 55(2): 206–217.

Bui TN and Jones C (1993) *A Heuristic for Reducing Fill-In in Sparse Matrix Factorization*. U.S. Department of Energy Office of Scientific and Technical Information.

Buluç A, Meyerhenke H, Safro I, et al. (2016) *Recent Advances in Graph Partitioning*. Springer International Publishing, pp. 117–158.

Chapp D, Johnston T and Taufer M (2015) On the need for reproducible numerical accuracy through intelligent runtime selection of reduction algorithms at the extreme scale. In:

Proceedings of the 2015 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp. 166–175.

Chapp D, Sato K, Ahn D, et al. (2018) Record-and-replay techniques for HPC Systems: a survey. *Supercomputing Frontiers and Innovations* 5(1).

Chapp D, Tan N, Bhowmick S, et al. (2021) Identifying degree and sources of non-determinism in MPI applications via graph kernels. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 32(12): 2936–2952.

Chen T and Tsourakakis C (2022) AntiBenford subgraphs: unsupervised anomaly detection in financial networks. In: Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD). ACM, pp. 2762–2770.

Chen X, Heimann M, Vahedian F, et al. (2020) CONE-Align: consistent network alignment with proximity-preserving node embedding. In: Proceedings of the 29th ACM International Conference on Information and Knowledge Management(ICDM). ACM, pp. 1985–1988.

Chen Z, Yu X, Song B, et al. (2017) Community-based network alignment for large attributed network. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management. CIKM. ACM, pp. 587–596.

Cheshmi K, Kamil S, Strout MM, et al. (2017) Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In: Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC). ACM.

Chiang WF, Gopalakrishnan G, Rakamarić Z, et al. (2013) Determinism and reproducibility in large-scale HPC Systems. In: Proceedings of the 4th Workshop on Determinism and Correctness in Parallel Programming (WoDet). IEEE.

Ciriello G, Mina M, Guzzi PH, et al. (2012) AlignNemo: a local network alignment method to integrate homology and topology. *PLOS ONE* 7(6): e38107.

Cook SA (1971) The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC). ACM, pp. 151–158.

Cordella LP, Foggia P, Sansone C, et al. (2004) A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26(10): 1367–1372.

Cosnard M, Jeannot E and Yang T (2004) Compact DAG representation and its symbolic scheduling. *Journal of Parallel and Distributed Computing* 64(8): 921–935.

Dehaspe L, Toivonen H and King RD (1998) Finding frequent substructures in chemical compounds. In: Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining. IEEE.

Derr T, Karimi H, Liu X, et al. (2021) Deep adversarial network alignment. In: Proceedings of the 30th ACM International Conference on Information and Knowledge Management(CIKM). ACM, pp. 352–361.

Djeddi WE, Yahia SB and Nguifo EM (2018) A novel computational approach for global alignment for multiple biological networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 15(6): 2060–2066.

El-Kebir M, Heringa J and Klau GW (2011) Lagrangian relaxation applied to sparse global network alignment. In: Proceedings of the IAPR International Conference on Pattern Recognition in Bioinformatics. Springer, pp. 225–236.

Eppstein D (2002) Subgraph isomorphism in planar graphs and related problems. Graph Algorithms and Applications I. World Scientific, pp. 283–309.

Flannick J, Novak A, Do CB, et al. (2008) Automatic parameter learning for multiple network alignment. In: Proceedings of the Annual International Conference on Research in Computational Molecular Biology. Springer, pp. 214–231.

Flannick J, Novak A, Srinivasan BS, et al. (2006) Graemlin: general and robust alignment of multiple large interaction networks. *Genome Research* 16(9): 1169–1181.

Gärtner T, Flach P and Wrobel S (2003) On graph kernels: Hardness results and efficient alternatives. Learning Theory and Kernel Machines. Springer, pp. 129–143.

Gebremedhin AH, Manne F and Pothen A (2005) What color Is your jacobian? graph coloring for computing derivatives. *SIAM Review* 47(4): 629–705.

Gianinazzi L, Besta M, Schaffner Y, et al. (2021) Parallel algorithms for finding large cliques in sparse graphs. In: Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). ACM, pp. 243–253.

Gioachin F, Zheng G and Kalé LV (2010) Robust non-intrusive record-replay with processor extraction. In: Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD). ACM, pp. 9–19.

Gopalakrishnan G, Hovland PD, Iancu C, et al. (2017) *Report of the HPC Correctness Summit*, 2017, Washington, DC. *CoRR* abs/1705. 07478.

Grohe M and Schweitzer P (2020) The graph isomorphism problem. *Communications of the ACM* 63(11): 128–134.

Guzzi PH and Milenković T (2018) Survey of local and global biological network alignment: the need to reconcile the two sides of the same coin. *Briefings in Bioinformatics* 19(3): 472–481.

Haidar A, Jagode H, YarKhan A, et al. (2017) Power-aware computing: Measurement, control, and performance analysis for Intel Xeon Phi. In: Proceedings of the 2017 IEEE High-Performance Extreme Computing Conference (HPEC). IEEE, pp. 1–7.

Heimann M, Shen H, Safavi T, et al. (2018) Regal: Representation learning-based graph alignment. In: Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM). ACM, pp. 117–126.

Herholz P, Tang X, Schneider T, et al. (2022) Sparsity-specific code optimization using expression trees. *ACM Transactions on Graphics* 41(5): 1–19.

Hido S and Kashima H (2009) A linear-time graph kernel. Proceedings of the Ninth IEEE International Conference on Data Mining(ICDM). IEEE, pp. 179–188.

Hopcroft JE and Wong JK (1974) Linear time algorithm for isomorphism of planar graphs (preliminary report). In: Proceedings of the 6th Annual ACM Symposium on Theory of Computing. ACM, pp. 172–184.

Ibragimov R, Martens J, Guo J, et al. (2013) Nabeeco: biological network alignment with bee colony optimization algorithm. In: Proceedings of the 15th annual conference companion on Genetic and evolutionary computation, pp. 43–44.

Inokuchi A, Washio T and Motoda H (2000) An apriori-based algorithm for mining frequent substructures from graph data. In: Proceedings of the European conference on Principles of Data Mining and Knowledge Discovery. Springer, pp. 13–23.

Jaja J and Kosaraju SR (1988) Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems* 35(3): 304–311.

Jiang C, Coenen F and Zito M (2013) A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review* 28(1): 75–105.

Johansson F, Jethava V, Dubhashi D, et al. (2014) Global graph kernels using geometric embeddings. In: Proceedings of the International Conference on Machine Learning (ICML). PMLR, pp. 694–702.

Kabir H and Madduri K (2017) Parallel k-truss decomposition on multicore systems. In: Proceedings of the IEEE High-Performance Extreme Computing Conference (HPEC). IEEE, pp. 1–7.

Kalaev M, Smoot M, Ideker T, et al. (2008) NetworkBLAST: comparative analysis of protein networks. *Bioinformatics* 24(4): 594–596.

Karypis G and Kumar V (1998) A fast and high-quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20(1): 359–392.

Kelley BP, Yuan B, Lewitter F, et al. (2004) PathBLAST: a tool for alignment of protein interaction networks. *Nucleic Acids Research* 32(suppl_2): 83–88.

Khan AM, Gleich DF, Pothen A, et al. (2012) A multithreaded algorithm for network alignment via approximate matching. In: Proceedings of the ACM/IEEE International Conference on High-Performance Computing, Networking, Storage and Analysis (SC). IEEE, pp. 1–11.

Kolda TG, Pinar A, Plantenga T, et al. (2014) Counting triangles in massive graphs with mapreduce. *SIAM Journal on Scientific Computing* 36(5): S48–S77.

Koutra D, Tong H and Lubensky D (2013a) Big-align: Fast bipartite graph alignment. In: Proceedings of the 13th International Conference on Data Mining (ICDM). IEEE, pp. 389–398.

Koutra D, Vogelstein JT and Faloutsos C (2013b) Deltacon: a principled massive-graph similarity function. In: Proceedings of the 2013 SIAM International Conference on Data Mining(SDM). SIAM, pp. 162–170.

Kramer R, Gupta R and Soffa M (1992) The combining DAG: a technique for parallel data flow analysis. In: Proceedings Sixth International Parallel Processing Symposium, pp. 652–655.

Kranzlmüller D (2000) *Event Graph Analysis for Debugging Massively Parallel Programs*.

Kriege NM, Johansson FD and Morris C (2020) A survey on graph kernels. *Applied Network Science* 5(1): 6–42.

Kuchaiev O, Milenković T, Memišević V, et al. (2010) Topological network alignment uncovers biological function and phylogeny. *Journal of the Royal Society Interface* 7(50): 1341–1354.

Lacoste-Julien S, Palla K, Davies A, et al. (2013) Sigma: Simple greedy matching for aligning large knowledge bases. In: Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp. 572–580.

Liu L, Cheung WK, Li X, et al. (2016) Aligning users across social networks using network embedding Ijcai, pp. 1774–1780.

López-Presa JL and Fernández Anta A (2009) Fast algorithm for graph isomorphism testing. In: Proceedings of the International Symposium on Experimental Algorithms(SEA). Springer, pp. 221–232.

Lou Z, You J, Wen C, et al. (2020) *Neural Subgraph Matching*. arXiv preprint arXiv:2007.03092.

Lovàsz L (1979) On the Shannon capacity of a graph. *IEEE Transactions on Information Theory* 25(1): 1–7.

Luks EM (1982) Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences* 25(1): 42–65.

Luks EM (1986) Parallel algorithms for permutation groups and graph isomorphism. In: Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS). IEEE, pp. 292–302.

Lusk R, Pieper S, Butler R, et al. (2015) Asynchronous dynamic load balancing. *Programming Models for Parallel Computing*. Scientific and Engineering Computation Series, pp. 186–203.

Ma CY and Liao CS (2020) A review of protein-protein interaction network alignment: From pathway comparison to global alignment. *Computational and Structural Biotechnology Journal* 18: 2647–2656.

Ma L, Wang S, Lin Q, et al. (2021) Multi-neighborhood learning for global alignment in biological networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 18: 2598–2611.

Yaveroğlu ÖN, Malod-Dognin N, Davis D, et al. (2014) Revealing the hidden language of complex networks. *Scientific Reports* 4, 4547.

Mawhirter D and Wu B (2019) Automine: harmonizing high-level abstraction and high performance for graph mining. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 509–523.

McKay BD (1981) *Practical Graph Isomorphism*. Department of Computer Science, Vanderbilt University Tennessee.

McKay BD and Piperno A (2014) Practical graph isomorphism, II. *Journal of Symbolic Computation* 60: 94–112.

Meng L, Crawford J, Striegel A, et al. (2016a) *IGLOO: Integrating Global and Local Biological Network Alignment*. arXiv preprint arXiv:1604.06111.

Meng L, Striegel A and Milenković T (2016b) Local versus global biological network alignment. *Bioinformatics* 32(20): 3155–3164.

Meyer P, Siy HP and Bhowmick S (2014) Identifying important classes of large software systems through k-Core decomposition. *Advances in Complex Systems* 17(7–8): 1550004.

Mohiyuddin M, Hoemmen M, Demmel J, et al. (2009) Minimizing communication in sparse matrix solvers. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC). ACM.

Morgan HL (1965) The generation of a unique machine description for chemical structures-a technique developed at chemical abstracts service. *Journal of Chemical Documentation* 5(2): 107–113.

Nicolae B, Moody AT, Kosinovsky G, et al. (2021) *VELOC: VEry Low Overhead Checkpointing in the Age of Exascale*. *ArXiv* abs/2103.02131.

Papadimitriou P, Dasdan A and Garcia-Molina H (2010) Web graph similarity for anomaly detection. *Journal of Internet Services and Applications* 1(1): 19–30.

Park S, Zhou Y, XiongYin ZW, et al. (2009) PRES: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP). ACM, pp. 177–192.

Qian X, Sen K, Hargrove P, et al. (2016a) OPR: deterministic group replay for one-sided communication. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM, p. 47. 1–47:2.

Qian X, Sen K, Hargrove P, et al. (2016b) SReplay: deterministic sub-group replay for one-sided communication. In: Proceedings of the 2016 International Conference on Supercomputing (ICS). ACM, p. 17. 1–17:13.

Qiao F, Zhang X, Li P, et al. (2018) A parallel approach for frequent subgraph mining in a single large graph using spark. *Applied Sciences* 8(2): 230.

Qin KK, Salim FD, Ren Y, et al. (2020) G-CREWE: graph compression with embedding for network alignment. In: Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM). ACM, pp. 1255–1264.

Qu Y, Zheng Q, Chi J, et al. (2021) Using K-core decomposition on class dependency networks to improve bug prediction model's practical performance. *IEEE Transactions on Software Engineering* 47(2): 348–366.

Ren J, Zhou Y, Jin R, et al. (2019) Dual adversarial learning based network alignment. In: 2019 IEEE International Conference on Data Mining (ICDM). IEEE, pp. 1288–1293.

Reza T, Ripeanu M, Tripoul N, et al. (2018) prunejuice: pruning trillion-edge graphs to a precise pattern-matching solution. In: Proceedings of the IEEE/ACM International Conference for High-Performance Computing, Networking, Storage and Analysis (SC), pp. 265–281.

Sahraeian SME and Yoon BJ (2013) SMETANA: accurate and scalable algorithm for probabilistic alignment of large-scale biological networks. *PLOS ONE* 8(7): e67995.

Sanfeliu A and Fu KS (1983) A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics* 3: 353–362.

Sarajlić A, Malod-Dognin N, Yaveroğlu ÖN, et al. (2016) Graphlet-based characterization of directed networks. *Scientific Reports* 6: 35098.

Sato K, Ahn DH, Laguna I, et al. (2015) Clock delta compression for scalable order-replay of non-deterministic parallel applications. In: Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC). ACM, p. 62. 1–62:12.

Sato K, Ahn DH, Laguna I, et al. (2017) Noise injection techniques to expose subtle and unintended message races. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM, pp. 89–101.

Seo E, Khan MMH, Mohapatra P, et al. (2011) Exposing complex bug-triggering conditions in distributed systems via graph mining. In: Proceedings of the International Conference on Parallel Processing (ICPP). IEEE, pp. 186–195.

Shervashidze N, Schweitzer P, Van Leeuwen EJ, et al. (2011) Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research* 12(9).

Shervashidze N, Vishwanathan S, Petri T, et al. (2009) Efficient graphlet kernels for large graph comparison. In: *Artificial Intelligence and Statistics*, pp. 488–495.

Shi T, Zhai M, Xu Y, et al. (2020) GraphPi: high-performance graph pattern matching through effective redundancy elimination. In: Proceedings of the ACM/IEEE International Conference for High-Performance Computing, Networking, Storage and Analysis (SC). IEEE.

Slota GM and Madduri K (2013) Fast approximate subgraph counting and enumeration. In: 2013 42nd International Conference on Parallel Processing (ICPP). IEEE, pp. 210–219.

Sugiyama M and Borgwardt K (2015) Halting in random walk kernels. *Advances in Neural Information Processing Systems* 28: 1639–1647.

Tan S, Guan Z, Cai D, et al. (2014) Mapping users across networks by manifold alignment on hypergraph. In: Proceedings of the AAAI Conference on Artificial Intelligence.

Tian W and Samatova NF (2013) Global alignment of pairwise protein interaction networks for maximal common conserved patterns. *International Journal of Genomics* 2013: 670623–671847.

Ullmann JR (1976) An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23(1): 31–42.

Valiente G (2002) Tree isomorphism. In: *Algorithms on Trees and Graphs*. Springer, pp. 151–251.

Vijayan V, Critchlow D and Milenković T (2017) Alignment of dynamic networks. *Bioinformatics* 33(14): 180–189.

Vijayan V, Gu S, Krebs E, et al. (2020) *Pairwise versus Multiple Network Alignment*. IEEE Access.

Vijayan V, Saraph V and Milenković T (2015) Magna++: Maximizing accuracy in global network alignment via both node and edge conservation. *Bioinformatics* 31(14): 2409–2411.

von der Malsburg C (1988) Pattern recognition by labeled graph matching. *Neural Networks* 1(2): 141–148.

Wang Z, Lv Q, Lan X, et al. (2018) Cross-lingual knowledge graph alignment via graph convolutional networks. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pp. 349–357.

Weisfeiler B and Leman A (1968) The reduction of a graph to canonical form and the algebra which appears therein. *Nauchno-Technicheskaya Informatsia* 2(9): 12–16.

Whitney H (1992) Congruent graphs and the connectivity of graphs. In: *Hassler Whitney Collected Papers*. Springer, pp. 61–79.

Wilke J (n. d.) *Structural Simulation Toolkit (SST) DUMPI Trace Library*. https://github.com/sstsimulator/sst-dumpi

Wilson RC and Zhu P (2008) A study of graph spectra for comparing graphs and trees. *Pattern Recognition* 41(9): 2833–2841.

Wolf MM, Deveci M, Berry JW, et al. (2017) Fast linear algebra-based triangle counting with KokkosKernels. In: Proceedings of the IEEE High-Performance Extreme Computing Conference (HPEC). IEEE, pp. 1–7.

Xiang L, Khan A, Serra E, et al. (2021) cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In: Proceedings of the ACM/IEEE International Conference for High-Performance Computing, Networking, Storage and Analysis (SC). IEEE, pp. 1–14.

Xu L, Wang W, Alvarez M, et al. (2014) Parallelization of shortest path graph kernels on multi-core cpus and gpus. In: Proceedings of the Programmability Issues for Heterogeneous Multicores (MultiProg).

Xue R, Liu X, Wu M, et al. (2009) MPIWiz: subgroup reproducible replay of MPI applications. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM, pp. 251–260.

Yan X and Han J (2002) gspan: graph-based substructure pattern mining. In: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM). IEEE, pp. 721–724.

Yanardag P and Vishwanathan S (2015) Deep graph kernels. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1365–1374.

Zhang J and Philip SY (2015) Multiple anonymized social networks alignment. In: Proceedings of the IEEE International Conference on Data Mining. IEEE, pp. 599–608.

Zhang S and Tong H (2016) Final: Fast attributed network alignment. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 1345–1354.

Zhang S, Tong H, Maciejewski R, et al. (2019) Multilevel network alignment. In: The World Wide Web Conference, pp. 2344–2354.

Zhang S, Tong H, Tang J, et al. (2017) iNEAT: incomplete network alignment. In: Proceedings of the 2017 IEEE International Conference on Data Mining(ICDM). IEEE, pp. 1189–1194.

Zhao Z, Khan M, Kumar VSA, et al. (2010) Subgraph enumeration in large social contact networks using parallel color coding and streaming. In: Proceedings of the 39th IEEE International Conference on Parallel Processing (ICPP). IEEE, pp. 594–603.

Zou R and Holder LB (2010) Frequent subgraph mining on a single large graph using sampling techniques. In: Proceedings of the 8th Workshop on Mining and Learning with Graphs, pp. 171–178.

Author biographies

Sanjukta Bhowmick is an Associate Professor at the Computer Science and Engineering Department at the University of North Texas. Her research interests focus on studying the properties of large, dynamic complex networks and using high-performance computing to analyze them. She is particularly interested in understanding how noise affects network analysis, and developing uncertainty quantification for network analysis. Dr Bhowmick received her PhD in Computer Science from Pennsylvania State University, University Park in 2004.

Nick Bell is a research scientist working with the Global Computing Lab team under Dr Michela Taufer's supervision. Bell's research focuses on graph algorithms and combinatorial optimization. He earned his M.A. in Computational and Applied Mathematics at Rice University in Fall of 2018.

Michela Taufer is an ACM Distinguished Scientist and holds the Jack Dongarra professorship in high-performance computing with the Department of Electrical Engineering and Computer Science, University of Tennessee Knoxville. Her research interests include high-performance computing, volunteer computing, scientific applications, scheduling and reproducibility challenges, and in situ data analytics. Taufer received her PhD in Computer Science from the Swiss Federal Institute of Technology (ETH) in 2002.

Appendix

Graph notation and terminology

We provide a brief overview of the common notations and terminology to support this article.

A graph G is defined by a pair of sets (V, E) and an incidence function $\psi_G: E \rightarrow V \times V$, where V is the set of vertices (or nodes), E is the set of edges between the nodes, and for any $e \in E$, there exists a pair of vertices $u \in V$ and $v \in V$ such that $\psi_G(e) = \{u, v\}$. The edge e is said to join vertices u and v. If edge e joins vertices u and v, we often denote this by $e = \{u, v\}$.

An undirected graph is where for an edge $e = \{u, v\} \in E$, the pair $\{u, v\} = e = \{v, u\}$. G is directed if the vertices of an edge pair are not interchangeable. G is cyclic if there exists a path of n edges $e_0 = \{u_0, u_1\}, e_1 = \{u_1, u_2\}, \ldots, e_n = \{u_n, u_0\}$ such that no edge appears twice in the path. G is acyclic if no such path exists. G is a weighted graph if it is combined with a set of weights $W_G \subseteq \mathbb{R}$ and weight function $w_G: E \rightarrow W_G$ that assigns a number w to each edge. It is unweighted if it has no such function. G is a labeled graph if it is combined with a label set L_G and a label function $l_G: V \rightarrow L_G$ that assigns a label to l to each vertex. Label sets will vary based on the problem or application.

If two vertices u and v are connected by an edge, then u is the neighbor of v and v is the neighbor of u. In an undirected graph, the degree of a vertex the number of its neighbors. In a directed graph, the number of neighbors pointing to a vertex is its indegree and the number of neighbors pointing away is its outdegree.

A subgraph $G_0 = (V_0, E_0) \subseteq G$ is a graph such that $V_0 \subseteq V$ and $E_0 \subseteq E$. The k-core is a maximal subgraph G_{k-core} such that each node in G_{k-core} has a degree at least k.

The adjacency matrix, A, of an unweighted graph G is as follows; $A[i, j] = 1$ if there exists an edge between vertices i and j in G. The Laplacian matrix, L, of the graph G is; $L[i, i]$ is the degree of i and $L[i, j] = -1$, if there is an edge between vertices i and j in G, and $i \neq j$.