DISTRIBUTED AND SCALABLE GRAPH PATTERN MATCHING:
MODELS AND ALGORITHMS

Arash Fard, M. Usman Nisar, John A. Miller, Lakshmish Ramaswamy
Computer Science Department
The University of Georgia
Athens, USA
{ar, nisar, jam, laks}@cs.uga.edu

Abstract
Graph pattern matching is a fundamental operation for many applications, and it is exhaustively studied in its classical forms. Nevertheless, there are newly emerging applications, like analyzing hyperlinks of the web graph and analyzing associations in a social network, that need to process massive graphs in a timely manner. Regarding the extremely large size of these graphs and knowledge they represent, not only new computing platforms are needed, but also old models and algorithms should be revised. In recent years, a few pattern matching models have been introduced that can promise a new avenue for pattern matching research on extremely massive graphs. Moreover, several graph processing frameworks like Pregel have recently sought to harness shared nothing clusters for processing massive graphs through a vertex-centric, Bulk Synchronous Parallel (BSP) programming model. However, developing scalable and efficient BSP-based algorithms for pattern matching is very challenging on these platforms because this problem does not naturally align with a vertex-centric programming paradigm. This paper introduces a new pattern matching model, called tight simulation, which outperforms the previous models in its family in terms of scalability while preserving their important properties. It also presents a novel distributed algorithm based on the vertex-centric programming paradigm for this pattern matching model and several others in the family of graph simulation as well. Our algorithms are fine tuned to consider the challenges of pattern matching on massive data graphs. Furthermore, we present an extensive set of experiments involving massive graphs (millions of vertices and billions of edges) to study the effects of various parameters on the scalability and performance of the proposed algorithms.

Keywords: query graphs; distributed algorithms; graph simulation, pattern matching, big data

1. INTRODUCTION
Graph pattern matching, as an important class of graph queries, seeks to find subgraphs of a data graph that are similar to a given query graph. This problem has been extensively studied over the past several decades; however, the massive scales of modern application domains such as social networks and the World Wide Web have reigned interest in highly scalable graph pattern matching algorithms (Brynielsson, Hogberg, Kaati, Martenson, & Svenson, 2010) (Sun, Wang, Wang, Shao, & Li, 2012).

Subgraph isomorphism is traditionally the most popular model of graph pattern matching. While it returns the strictest matches for graph pattern matching in terms of topology (Gallagher, 2006), the problem is NP-complete (Ullmann, 1976) and thus is impractical for massive graphs. The family of graph simulation algorithms provides a practical alternative to subgraph isomorphism by relaxing its matching conditions. These new models allow matches to be found in polynomial time. Moreover, some researchers (Brynielsson, Hogberg, Kaati, Martenson, & Svenson, 2010) (Fan, Li, Ma, Tang, Wu, & Wu, 2010) (Ma, Cao, Fan, Huai, & Wo, 2011) (Fard, Abdolrashidi, Ramaswamy, & Miller, 2012) believe that graph simulation can be more appropriate than subgraph isomorphism for some modern applications such as social network analysis because it yields matches that are conceptually more intuitive.

Variants of the graph pattern matching paradigm have been proposed that form a spectrum with respect to the stringency of the matching conditions. These models include graph simulation, dual simulation, strong simulation, strict simulation, tight simulation, and subgraph isomorphism, where graph simulation is the least restrictive and strict simulation is the most restrictive (Ma, Cao, Fan, Huai, & Wo, 2011) (Fard, Nisar, Ramaswamy, Miller, & Saltz, 2013). In comparison, tight simulation is conceptually closest to subgraph isomorphism, whereas graph simulation is the farthest. It is also noteworthy that, in prior work in this area, typically the more restrictive the model, the less scalable the corresponding algorithm.

While polynomial time algorithms exist for different varieties of simulations, they still do not scale to the massive graphs that characterize modern applications such as social networks. Therefore, a natural question is whether these algorithms can be parallelized and implemented efficiently on shared nothing clusters. In recent years, several graph processing frameworks have been proposed that enable programmers to easily and efficiently implement distributed graph algorithms on shared nothing clusters. These include Pregel (Malewicz, et al., 2009), Giraph
(Giraph website) and GPS (Salihoglu & Widom, 2012) which are characterized by the BSP programming paradigm. In these platforms, the programmer expresses algorithms in terms of the actions performed by a generic vertex of the graph. In recent years, researchers have designed vertex-centric algorithms for various graph problems. Surprisingly, to the best of our knowledge, there is no other study on vertex-centric approaches for graph simulation models. While there is some recent work on distributed basic graph simulation (Ma, Cao, Huai, & Wo, 2012), the proposed algorithm is not based on the vertex-centric programming paradigm.

Considering the unique benefits offered by vertex-centric programming platforms, this paper is dedicated to exploring the following questions. (1) How well do graph simulation in general, and various simulation models in particular fit into the vertex-centric distributed processing paradigm? (2) If certain simulation models are not suitable for the vertex-centric programming paradigm, what are the major bottlenecks? (3) Can we develop new simulation models that are conceptually similar to the existing ones, but are better suited to the vertex-centric paradigm? (4) How do various graph properties such as the number of vertices and the density of a graph impact the performance and scalability of the distributed graph simulation algorithms?

Towards answering these questions, we have designed vertex-centric distributed algorithms for various graph simulation models. We have comprehensively studied their behaviors through an extensive set of experiments. The main technical contributions of this paper are summarized below.

- We identify the bottlenecks that severely limit the scalability and performance of strong and strict simulation on distributed graph processing platforms. Towards ameliorating these limitations, we propose a new graph simulation model called tight simulation. Tight simulation is vastly more amenable to vertex-centric distribution, and surprisingly, the quality of its result is the same or in some cases better than both strong and strict simulation.
- We propose novel vertex-centric distributed algorithms for all five simulation models. Our algorithms are based upon detailed analytical studies of the models, and they seek to mitigate various performance bottlenecks.
- We also present a detailed experimental study on the benefits and costs of the proposed distributed algorithms for various graph simulation models. The experiments validate that our tight graph simulation model is highly scalable and very well suited for vertex-centric distribution.

The rest of the paper is organized as follows. In section two, we briefly explain some requisite terminology and the main concepts involved in pattern matching models and vertex-centric graph processing. Section three explains strict simulation, a competitive and scalable pattern matching model. Section four introduces tight simulation, a new pattern matching model which is even more stringent and scalable than strict simulation. Section five is dedicated to the explanation of our distributed algorithms. We cover an experimental study about the behavior of our algorithms in section six. We also present some related work in seven, followed by conclusions in section eight.

### 2. BACKGROUND

In this section, we briefly review concepts of BSP-based vertex-centric graph processing and graph pattern matching models.

#### 2.1. BSP vertex-centric Graph Processing

Several distributed computing platforms have been recently designed and implemented with the goal of effectively harnessing shared nothing clusters for processing massive graphs. The most popular platforms among them are based upon BSP-based vertex-centric programming model, like Pregel (Malewicz, et al., 2009), GPS (Salihoglu & Widom, 2012) and Giraph (Giraph website).

Bulk Synchronous Parallel (BSP) was first proposed by Valiant (Valiant, 1990) as a model of computation for parallel processing. In this model, computation is a series of supersteps. Each superstep consists of three ordered stages: (1) concurrent computation, (2) communication, and (3) barrier synchronization. In the vertex-centric programming model, each vertex of the data graph is a computing unit which can be conceptually mapped to a process in the BSP model. At loading time, the data graph is partitioned among several workers. Each worker is a real process which handles the computation for a partition of vertices. Each vertex initially knows only about its own label and its outgoing edges. Vertices can then exchange messages through successive supersteps to learn about their neighborhood or to accomplish a computing task. Within each superstep, vertices are executing the same predefined function. A vertex votes to halt and goes to inactive mode when it believes that it has accomplished its tasks. It remains inactive until it is triggered externally by a message from another vertex. The algorithm terminates when all vertices become inactive.

The vertex-centric approach is a pure message passing model in which programmers focus on a local action for each vertex of the data graph. The usage of the BSP model makes it inherently free of deadlocks. Moreover, it is well-suited for distributed implementation by design and can provide very high scalability.

#### 2.2. Pattern Matching Models

A pattern matching algorithm tries to find all the matches of a given graph, called a query graph, in an existing larger graph, called a data graph. To define it more formally, assume that there is a data graph \( G(V, E, I) \), where \( V \) is the set of vertices, \( E \) is the set of edges, and \( I \) is a function that maps the vertices to their labels. Given a query graph \( Q(V_q, E_q, I_q) \) the task is to find all subgraphs of \( G \) that
match the query $Q$. Here, we assume all vertices are labeled, all edges are directed, and there are no multiple edges. Without loss of generality, we also assume a query graph is a connected graph because the result of pattern matching for a disconnected query graph is equal to the union of the results for its connected components. In this paper, we use pattern and query graph interchangeably.

**Subgraph Isomorphism.** Subgraph isomorphism is the most famous model for pattern matching. It preserves all topological features of the query graph in the result subgraph. However, finding all subgraphs that are isomorphic to a query graph is an NP-hard problem in the general case (Ullmann, 1976). By definition, subgraph isomorphism describes a bijective mapping between a query graph $Q(V_q, E_q, l_q)$ and a subgraph of a data graph $G(V, E, I)$, denoted by $Q \congiso G$.

**Graph Simulation.** Graph simulation, provides a faster algorithm by relaxing some restrictions on matches.

**Definition:** Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, I)$ via graph simulation, denoted by $Q \congsim G$, if there is a binary relation $R \subseteq V_q \times V$ such that (1) if $(u, u') \in R$ , then $l_q(u) = l(I(u'))$ ; (2) $\forall u \in V_q \ , \exists u' \in V: (u, u') \in R$ ; (3) $\forall (u, v) \in E_q, \exists (u', v') \in E: (u, u') \in R \land (v, v') \in R$.

Intuitively, graph simulation only preserves the child relationships of each vertex. The result of pattern matching is a maximum match set of vertices. The **maximum match set**, $R_m \subseteq V_q \times V$, is the biggest relation set between $Q$ and $G$ with respect to $Q \congsim G$. The **result match graph**, $G_r(V_r, E_r, l_r)$, is a subgraph of $G$ that can represent $R_m$ (Ma, Cao, Fan, Huai, & Wo, 2011). By definition, $G_r$ is a subgraph of $G$ which satisfies these conditions: (1) $(u, u') \in R_m \Rightarrow u' \in V$ ; (2) $\forall (u, u'), (v, v') \in R_m[(u, v) \in E_q \Leftrightarrow (u', v') \in E]$. A quadratic time algorithm for graph simulation was first proposed in (Henzinger, Henzinger, & Kopke, 1995) with applications to the refinement and verification of reactive systems. This model and its extensions have been studied especially in recent years (Fan, Li, Ma, Tang, Wu, & Wu, 2010). (Ma, Cao, Huai, & Wo, 2012) because of their new applications in analysis of social networks (Brynielsson, Hogberg, Kaati, Martenson, & Svenson, 2010).

**Dual Simulation.** Dual simulation improves the result of graph simulation by taking into account not only the children of a query node, but also its parents.

**Definition:** Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, I)$ via dual simulation, denoted by $Q \congsim G$, if (1) $Q \congsim G$ with a binary match relation $R_D \subseteq V_q \times V$, and (2) $\forall (u, u') \in R_D[(w, u) \in E_q \Rightarrow \exists w' \in V: (w, w') \in R_D \land (w', u') \in E]$.

In dual simulation similar to graph simulation, we maintain the concept of a maximum match set and a result match graph. Here, the result match graph is also a single graph which might be connected or disconnected. In (Ma, Cao, Fan, Huai, & Wo, 2011) a cubic algorithm for dual simulation is proposed.

**Strong Simulation.** Strong simulation adds a locality property to dual simulation. Shuai Ma et al. (Ma, Cao, Fan, Huai, & Wo, 2011) introduce the concept of a ball to define locality. A ball $b$ in $G(V, E, I)$, denoted by $G[v, r]$, is a subgraph of $G$ that contains all vertices not further than a specified radius $r$ from a center $v \in V$; moreover, the ball contains all edges in $G$ that connect these vertices (i.e., it is an induced connected subgraph). Given two vertices $u$ and $v$ in a connected graph, the **distance** between them is defined as the minimum number of edges in an undirected path which connects them. The **diameter** of a connected graph is then defined as the longest distance between any pair of vertices in the graph.

**Definition:** Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, I)$ via strong simulation, denoted by $Q \congsim G$, if there exists a vertex $v \in V$ such that (1) $Q \congsim G[v, d_q]$ with maximum dual match set $R_D^b$ in ball $b$ where $d_q$ is the diameter of $Q$, and (2) $v$ is member of at least one of the pairs in $R_D^b$. The connected part of the result match graph of each ball with respect to its $R_D^b$ which contains $v$ is called a maximum perfect subgraph of $G$ with respect to $Q$.

In contrast to the previous types of simulation, strong simulation may have multiple maximum perfect subgraphs (Max-PGs) as its result. As indicated by (Ma, Cao, Fan, Huai, & Wo, 2011), although strong simulation preserves many of the topological characteristics of a pattern graph, its result can be computed in cubic time. Moreover, the number of MaxPGs is bounded by the number of vertices in a data graph, while subgraph isomorphism may have an exponential number of match subgraphs.

**Comparing the result of the models.** Figure 1 provides an example to show the difference in the results of the mentioned pattern matching models. This example is about finding a team of specialists on a social network like LinkedIn, inspired by an example in (Ma, Cao, Huai, & Wo, 2012). A vertex in this example represents a member, and a label represents a member's profession. A directed edge from a member $a$ to a member $b$ indicates that member $a$ has endorsed member $b$. Because the vertices in the query graph for this example have distinct labels, we only specify data graph vertex ID’s for a match rather than specifying the full match relation. Given the pattern and the data graph, the maximum match set for graph simulation consists of all vertices except $\{2,13\}$. It is clear that vertices 1, 3, and 14 are not appropriate matches either. Dual simulation removes these inappropriate vertices; its maximum match set contains all vertices except $\{1,2,3,13,14\}$. However, there are still some vertices that do not provide very meaningful
matches. For example, the cycle \{15,16,17,18,19,20\} creates a very big subgraph match, which is not desirable for such a small pattern. Applying strong simulation then shrinks the result match graph to a more appropriate set of vertices; the result here is the set of vertices \{4,5,6,7,8,9,10,11,12\}. In contrast, there are two isomorphic subgraphs corresponding to these two set of vertices: \{4,6,7,8\} and \{5,6,7,8\}.

**Eccentricity, center, and radius.** The eccentricity of a vertex in a graph is its maximum distance from any other vertex in the graph. The vertices of the graph with the minimum eccentricity are the centers of the graph, and the value of their eccentricity is the radius, \( r_Q \), of the graph. The maximum value of eccentricity equals to the diameter of the graph \( d_Q \). It is proved that \( r_Q \leq d_Q \leq 2r_Q \) (Farber, 1989).

![Figure 1. An example for different models](image1)

**3. STRICT SIMULATION**

Strict simulation is a novel modification of Strong simulation that not only substantially improves its performance, but also maintains a better quality of result because of its revised definition of locality. The locality restriction defined in strong simulation is the main reason for its long computation time because the size of balls can be potentially very big. Here, the size of a ball means the number of vertices that it contains. Proportional to the diameter of a graph, the number of vertices it contains. Furthermore, due to communication overhead during ball creation, the problem is exacerbated in distributed systems where a data graph is partitioned among different nodes. In order to mitigate this overhead, we introduced a pattern matching model, named strict simulation in (Fard, Nisar, Ramaswamy, Miller, & Saltz, 2013). Strict simulation is more scalable and preserves the important properties of strong simulation. It is shown in (Ma, Cao, Fan, Huai, & Wo, 2011), (Fan W., 2012) that: (1) if \( Q \leq_{iso} G \), then \( Q \leq_{sim} G \); (2) if \( Q \leq_{sim} G \), then \( Q \leq_{isim} G \); and (3) if \( Q \leq_{sim} G \), then \( Q \leq_{sim} G \). We show that strict simulation and another new model we introduce in the next section are not only more efficient than strong simulation, but also more stringent; i.e., their results are getting closer to subgraph isomorphism while they become computationally more efficient.

Figure 3.a shows the flowchart of the centralized algorithm for strong simulation. In this algorithm, the match relation of dual simulation, \( R_D \), is computed first. Then, a ball \( G(v, d_Q) \) is created for each vertex \( v \) of the data graph contained in \( R_D \). Members of the ball are selected regardless of their membership in pairs of \( R_D \). At the next step, the dual match relation is projected on each ball to compute the result of strong simulation. Finally, the maximum perfect subgraph (MaxPG) is extracted by constructing the match graph on each ball and finding the connected component containing the center.

In comparison, Figure 3.b illustrates the centralized algorithm for strict simulation. The key difference between them is that in strict simulation the duality condition is enforced before the locality condition; i.e., balls are created from the dual result match graph, \( G_D \), rather than from the original graph. As the balls in strict simulation are often significantly smaller, this seemingly minor difference between the two algorithms has a profoundly positive impact on running time.

There are the same numbers of balls in strict and strong simulation; however, the extracted MaxPG from each ball in strict simulation is always a subgraph of the result in strong simulation on a ball with the same center. Duplicate MaxPGs may be produced from different balls, and in the case of strict simulation a result MaxPG might be subgraph of another one. In a post-processing phase, the duplicate results are filtered, and only the smaller result is kept when it is a subgraph of another result. Any possible isomorphic subgraph match of the query will be preserved in the result of strict simulation.

Figure 2 shows an example that highlights the difference between the results of these two types of simulation. In this example, all vertices except 9 will appear in the dual result match graph. For strong simulation, a ball centered at vertex 2 would contain all the vertices in the data graph. In contrast, because the dual match graph would not contain vertex 9 and some of the edges, the corresponding ball for strict simulation would only contain the set of vertices \{1,2,3,4\}. Therefore, the MaxPG resulting from strict simulation will contain all vertices except 9, while the one resulting from strict simulation will contain only vertices 1, 2, and 3. One can verify that this is the only MaxPG which will result from any ball of strict simulation.

![Figure 2. Comparison between strong and strict simulation](image2)
The formal definition of strict simulation follows.

**Definition:** Pattern $Q(V_q, E_q, I_q)$ matches data graph $G(V, E, I)$ via strict simulation, denoted by $Q \sim_{\text{Sim}} G$, if there exists a vertex $v \in V$ such that (1) $v \in V_D$ where $G_D(V_D, E_D, I_D)$ is the result match graph with respect to $Q \sim_{\text{Sim}} G$; (2) $Q \sim_{\text{Sim}} G[v, d_Q]$ where $G_D[v, d_Q]$ is a ball extracted from $G_D$, and $d_Q$ is the diameter of $Q$; (3) $v$ is a member of the result MaxPG.

A MaxPG for strict simulation is defined the same as a MaxPG for strong simulation. Using the definition and the properties of dual simulation presented in (Ma, Cao, Fan, Huai, & Wo, 2011), (Fan W., 2012), properties of strict simulation can be proved as follows.

**Theorem 1:** For any query graph $Q$ and data graph $G$ such that $Q \sim_{\text{Sim}} G$, there exists a unique set of maximum perfect subgraphs for $Q$ and $G$.

Proof: It is proved that the result of dual simulation is unique. Therefore, the balls extracted from its result match graph and consequently their result after applying dual filter would be unique. □

<table>
<thead>
<tr>
<th>Q and G as inputs</th>
<th>Q and G as inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply dual simulation on G to find DualSim match set</td>
<td>Apply dual simulation on G to find DualSim match set</td>
</tr>
<tr>
<td>Create a ball centered at each matching node via DualSim in G</td>
<td>Create a ball centered at each matching node via DualSim in G</td>
</tr>
<tr>
<td>For each ball, extract the connected component containing the center</td>
<td>For each ball, extract the connected component containing the center</td>
</tr>
<tr>
<td>(a) Strong Simulation</td>
<td>(b) Strict Simulation</td>
</tr>
<tr>
<td>Extracting MaxPG &amp; Post-processing</td>
<td>Extracting MaxPG &amp; Post-processing</td>
</tr>
<tr>
<td>(c) Tight Simulation</td>
<td>(c) Tight Simulation</td>
</tr>
</tbody>
</table>

Figure 3. Flowcharts of strong, strict, and tight simulation

**Proposition 1:** (1) If $Q \sim_{\text{Sim}} G$, then $Q \sim_{\text{Sim}} G$. (2) if $Q \sim_{\text{Sim}} G$, then $Q \sim_{\text{Sim}} G$.

Proof: (1) Any subgraph isomorphic match in $G$ is also dual match to $Q$; therefore, it will appear in the result of dual match graph. Clearly, a ball with radius $d_Q$ (diameter of $Q$), and centered on a vertex of this subgraph will contain its whole vertices. The subgraph later will be also in the result of dual filter on the ball. (2) $G_D$ is a subgraph of $G$, and the distance between any pair of vertices in $G_D$ is smaller than their distance in $G$. Therefore, any ball in strict simulation will be a subgraph of the corresponding ball in strong simulation with the same center. Consequently, when there is a subgraph result in the ball of strict simulation after applying dual filter, it will be also preserved in the result of strict simulation. □

**Proposition 2:** The number of maximum perfect subgraphs produced by $Q \sim_{\text{Sim}} G$ is bounded by the number of vertices in $G$.

Proof: $G_D$ is a subgraph of $G$, and the number of balls in strict simulation equals to the number of vertices in $G_D$. Moreover, not more than one MaxPG can be produced from each ball. Therefore, their total number is bounded by the number of vertices in $G$. □

Theorem 1 ensures that strict simulation yields a unique solution for any query/data graph pair. Proposition 1 indicates that strict simulation is a more stringent notion than strong simulation, though still looser than subgraph isomorphism. Proposition 2 gives an upper bound on the number of possible matches for a query graph in a data graph. It should be noted that the number of matches in subgraph isomorphism can be exponential to the number of vertices in $G$. The following theorem also shows that the asymptotic time complexity of strict simulation is the same as that of strong and dual simulation.

**Theorem 2:** For any query graph $Q$ and data graph $G$, the time complexity for finding all maximum perfect subgraphs with respect to strict simulation is cubic.

Proof: It is proved that time complexity for finding $G_D$ is cubic. Moreover, it is proved that time complexity of strong simulation is cubic. Regarding the fact that each ball in strict simulation is a subgraph of its corresponding ball in strong simulation, time complexity of strict simulation is also cubic. □

4. Tight Simulation

Strict simulation reduces the computation time of strong simulation by decreasing the size of the balls, but the number of balls remains the same. The number of balls can be big when the number of vertices left after of dual simulation is big which exacerbates the performance. Moreover, it is still desirable, both in terms of computation time and the quality of the result, to shrink the size of the balls. In this section, we introduce a novel pattern matching model, named *tight simulation*, which not only decreases the size of the balls further in comparison to strict simulation, but also reduces the number of balls. This will make the search for the pattern faster and the resulting subgraphs more stringent.

A summary of the centralized algorithm for tight simulation is compared to strict simulation in Figure 3.c. The main difference between these two is on the phases of preprocessing the query graph, and ball creation. In the phase of preprocessing, a single vertex, $u \in Q$, is chosen as a candidate match to the center of a potential ball on the data graph. The appropriate radius of the ball is also calculated in this phase. Then, in the phase of ball creation, only those vertices of the data graph which are in the dual match set of $u$ will be picked as the center of balls. The precalculated radius of such a ball is always between the radius and the diameter of the query graph. It should be noticed...
that only the vertices in the result of dual match graph will be used for ball creation, similar to strict simulation.

We introduce multiple selectivity criteria for finding the candidate vertex and radius out of the query graph. A vertex \( u \in Q \) with the minimum eccentricity (a center of \( Q \)) which has the highest ratio of degree to label frequency (in \( Q \)) will be picked as the candidate vertex. Selecting one of the centers of \( Q \) as the candidate vertex makes it also possible to select the candidate radius of balls equal to the radius of \( Q \). It is the tightest ball which also preserves all the subgraph isomorphic matches of the query graph in the result. Among the potential vertex candidates those with the highest degree and lowest label frequency present higher selectivity condition. In the case that there are several vertices with same selectivity score, one of them will be selected randomly.

In a centralized algorithm, it is possible to postpone selecting the candidate vertex and radius until the end of dual simulation phase and right before ball creation. One may consider that a vertex \( u \in Q \) with the smallest dual match set in \( G \) would be the best vertex candidate. In this case, the candidate radius would be equal to the eccentricity of \( u \) in \( Q \). Although this choice might be a good option in a centralized algorithm, it is not viable for a distributed algorithms based on vertex-centric framework. In such a distributed environment every vertex of \( G \) will learn if it is a member of a dual match set to a few vertices in \( Q \), but no global view of this set would be available. Proposed vertex-centric distributed algorithms will be explained in the next section.

The results of tight simulation are subgraphs of the corresponding results of strict simulation while they always contain all the subgraph isomorphic matches. Therefore, the results of tight simulation are closer to subgraph isomorphism in comparison to strict simulation. It should be noted that the post-processing phase explained in the last section will be applied to the results of strong, strict, and tight simulation in the same way. Figure 4 shows an example which displays the difference between the results of tight versus strict and strong simulation. In this example, all the vertices shown in the data graph will remain in the dual match graph. Clearly, the vertex labeled \( B \) in the pattern is its center; hence, will be picked as the candidate vertex. Therefore, vertices \{2,4,6,10,12,14\} will be picked as the center of balls with radius \( r_Q = 1 \) in tight simulation. Only the ball centered at 2 can result a MaxPG which contains these vertices \{1,2,3\}. In contrast in strict and strong simulation, a ball with radius \( d_Q = 2 \) will be created for any vertex of the data graph. One can verify that for the ball created on vertex 1, strict simulation results a MaxPG containing \{1,2,3,4,5,6,7,8\}, and strong simulation results a MaxPG containing all the vertices.

We formally define tight simulation as follows.

**Definition:** A Pattern \( Q(V_q, E_q, l_q) \) matches data graph \( G(V,E,I) \) via tight simulation, denoted by \( Q \subseteq^{T}_{\text{sim}} G \), if there are vertices \( u \in Q \) and \( u' \in G \) such that (1) \( u \) is a center of \( Q \) with highest defined selectivity; (2) \((u,u') \in R_D \) where \( R_D \) is dual relation set between \( Q \) and \( G \); (3) \( Q = Q^{D}_{\text{sim}} G_{D}[u',r_Q] \) where \( G_D(V_D,E_D,l_D) \) is a ball extracted from \( G_D(V_D,E_D,l_D) \) which is the result match graph with respect to \( Q \); (4) \( u' \) is a member of the result MaxPG.

![Figure 4. Comparing tight vs. strict and strong simulation](image)

The criterion for selectivity of \( u \) in \( Q \) is the ratio of its degree to its label frequency. The definition of MaxPG is also similar to its definition for strong and strict simulation. Similar to strict simulation, we can assert and prove the properties of tight simulation as follows.

**Theorem 3:** For any query graph \( Q \) and data graph \( G \) such that \( Q \subseteq^{T}_{\text{sim}} G \), there exists a unique set of maximum perfect subgraphs for \( Q \) and \( G \).

**Proof:** It is proved that the result of dual simulation is unique. The candidate vertex selected from the query is also unique when the query and the selectivity criteria are fixed. Therefore, the balls created in tight simulation and their result after dual filter will be also unique.

**Proposition 3:** (1) If \( Q \subseteq^{T}_{\text{iso}} G \), then \( Q \subseteq^{T}_{\text{sim}} G \). (2) if \( Q \subseteq^{T}_{\text{sim}} G \), then \( Q \subseteq^{T}_{\text{iso}} G \).

**Proof:** (1) Any subgraph isomorphic match in \( G \) is also dual match to \( Q \); therefore, it will appear in the result of dual match graph. The dual match to candidate vertex of \( Q \) is one of the vertices of the isomorphic match and will be selected as the center of a ball with radius \( r_Q \). As the candidate vertex was the center of \( Q \), the isomorphic match will be entirely enclosed in the ball and therefore will appear in the result of tight simulation. (2) When there is a subgraph result in the ball of strict simulation, it will also appear as a part of the result of strict simulation because there is a corresponding ball in strict simulation created on \( G_D \) with a bigger radius. In other words, a ball in tight simulation is always a subgraph of its corresponding ball in strict simulation.

**Proposition 4:** The number of maximum perfect subgraphs produced by \( Q \subseteq^{T}_{\text{sim}} G \) is bounded by the number of vertices in \( G \).
Proof: The results of tight simulation are subset of the results of strict simulation. The number of maximum perfect subgraphs produced by $Q \leq_{\text{tsim}} G$ is bounded by the number of vertices in $G$; hence, it is the same for $Q \leq_{\text{tsim}} G$. □

**Theorem 4:** For any query graph $Q$ and data graph $G$, the time complexity for finding all maximum perfect subgraphs with respect to tight simulation is cubic.

Proof: The time complexity for finding the center and the radius of $(V_q, E_q, l_q)$ is $\Theta(|V_q|^3)$. The procedure of tight simulation is similar to strict simulation, but it deals with a smaller number of balls which are most likely smaller than the corresponding balls in strict simulation; therefore, its time complexity must be smaller as well. Taking into account that the time complexity of dual simulation phase is cubic, we can conclude that it is the same for tight simulation. □

5. DISTRIBUTED PATTERN MATCHING

In contrast to the usual graph programming models, an algorithm in the vertex-centric programming model should be designed from the perspective of each vertex of a graph.

In this section, we present distributed algorithms for different types of graph simulation based vertex-centric programming model.

5.1. Distributed Graph Simulation

<table>
<thead>
<tr>
<th>Superstep 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Set match flag true when a vertex in the query has the same label</td>
</tr>
<tr>
<td>• Make a local matchSet of potential match vertices</td>
</tr>
<tr>
<td>• Ask children about their status</td>
</tr>
<tr>
<td>• Otherwise vote to halt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Superstep 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• If received a message while the flag is true reply back with matchSet</td>
</tr>
<tr>
<td>• Otherwise vote to halt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Superstep 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• If match flag is true evaluate the members of matchSet</td>
</tr>
<tr>
<td>• In the case of any removal from matchSet, inform parents and set match flag accordingly</td>
</tr>
<tr>
<td>• Otherwise vote to halt</td>
</tr>
<tr>
<td>• Otherwise vote to halt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Superstep 4 and beyond:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• If there is any incoming removal message reevaluate matchSet</td>
</tr>
<tr>
<td>• In the case of any removal from matchSet, inform parents and set match flag accordingly</td>
</tr>
<tr>
<td>• Otherwise vote to halt</td>
</tr>
</tbody>
</table>

Figure 5 shows a summary of our algorithm for a vertex in distributed graph simulation. Initially, we distribute the query graph among all workers. The cost of this distribution is negligible because the size of query is small and the total number of workers is limited to the number of processing elements in the system. Different tasks in different supersteps are distinguished using an if-else ladder. Moreover, the BSP framework ensures that all vertices are always at the same superstep.

A boolean flag, named *match*, is defined for each vertex in $G$ in order to track if it matches a vertex in $Q$. It is initially assumed that the vertex is not a match. Then at the first superstep, the *match* flag becomes true if its label matches the label of a vertex in $Q$. In this case, a local match set, named *matchSet*, is created to keep track of its potential matches in $Q$. Each vertex, then, learns about the *matchSets* of its children during the first three supersteps and keeps them in a local list for later evaluation of graph simulation conditions. Any match is removed from the local *matchSet* if it does not satisfy the simulation conditions. The vertex should also inform its parents about any changes in its *matchSet*. Consequently, any vertex that receives changes in its children’s *matchSets* reflects those changes in its list of match children and reevaluates its own *matchSet*. The algorithm can terminate after the third superstep if no vertex removes any match from its *matchSet*. This procedure will continue in superstep four and beyond until there is no change. To guarantee the termination of the algorithm, any active vertex with no incoming message votes to halt after the third superstep. At the end, the local *matchSet* of each vertex contains the correct and complete set of matches between that vertex and the vertices of the query graph.

Figure 6 displays an example for distributed graph simulation. Here, all the vertices of the data graph labeled $a$, $b$, and $c$ make their match flag true at the first superstep, and then vertices 1, 2, and 5 send messages to their children. At the second superstep only vertices 5, 6, and 7 will reply back to their parents. At the third superstep, vertices 1, 5, 6, 7, and 8 can successfully validate their *matchSet*, but vertex 2 makes its flag false, because it receives no message from any child. Therefore, vertex 2 sends a removal message to vertex 1. This message will be received by vertex 1 at superstep four. It will successfully reevaluate its match set, and the algorithm will finish at superstep five when every vertex has voted to halt (there is no further communication).
number of members of all matchSets is finite; therefore, the algorithm will terminate eventually in a finite time when all the matchSets become empty in the worst case. □

Lemma 2: At the end of the proposed algorithm, the matchSet of each vertex contains the correct and complete set of matches for that vertex.

Proof: At the first superstep, each vertex creates its matchSet from any vertex in Q with the same label. Hence, any potential match initially becomes a member of this set. The set is filtered during the next supersteps, and it is expected that it will contain only correct matches at the end. In other words, the completeness condition of the set is satisfied at the first superstep, and we should only prove the correctness of its members when the algorithm terminates.

BSP computational model ensures that all vertices are synchronized at the beginning of each superstep. Having this property in mind, the set of supersteps 4 and beyond in the proposed algorithm (Figure 5) is very similar to a while loop. Therefore, this lemma can be proved using loop invariant theorem (Hoare, 1969). Here, the invariant is the validation of each matchSet with respect to the local list of match children. At the end of the third superstep, each vertex has a matchSet which its members are validated based on the information gathered from the matchSet of its children. The guard condition for iterating through supersteps 4 and beyond is receiving at least one removal message. The invariant condition is true at the beginning of each superstep, and will be also true at the end of the superstep because the vertices that have received any removal message will update their list of match children accordingly and reevaluate their matchSets. According to lemma 1, the guard condition will become false after a finite number of iterations. Existence of no removal message means that all vertices have satisfied the children condition. Therefore, the invariant is true after termination; i.e., each member of a matchSet is a correct match. □

Figure 7 demonstrates the number of supersteps in the worst case. One can verify that if the length of path 1,2,3,4 is increased to \( L_p \) in such a way that the label pattern of vertices 3 and 4 are repeated, the algorithm will need \( L_p + 1 \) supersteps to terminate. In general, the minimum number of supersteps is 3, and its upper bound is \( O(|E|) \).

![Figure 7. An example for the number of supersteps](image)

5.2. Distributed Dual Simulation

The distributed algorithm for dual simulation is a smart modification of the distributed algorithm proposed for graph simulation in the previous subsection. Indeed, we extend the algorithm to check parent relationship as well. Therefore, each vertex also needs to keep track of the matchSets of its parents.

At the first superstep each vertex sends not only its ID, but also its label to its children. At the second superstep a vertex can infer the matchSets of its parents from the received labels and store them. Having this initial list at the second superstep allows each vertex to verify the parent relationships for each of the candidate matches in its matchSet. Very similar to the idea explained in the previous subsection for child relationships, removals from matchSet caused by evaluating the parent relationship must be reported to the children. The rest of the algorithm remains similar to the algorithm for graph simulation, with a few small modifications to consider the evaluation of a vertex with respect to its parent relationships.

The proof of correctness for this algorithm is very similar to the proof of correctness for the graph simulation algorithm. Similarly, the upper bound on the number of required supersteps for the distributed algorithm of dual simulation is \( O(|E|) \).

5.3. Distributed strong, strict, tight simulation

The algorithms for distributed strong, strict, and tight simulation are built on top of the algorithm for distributed dual simulation. In the case of strong and strict simulation, each vertex that has successfully passed the filter of dual simulation will create a ball around itself. Recalling their definitions, each ball in strong simulation is an induced connected subgraph of the data graph; whereas, each ball in strict simulation is an induced connected subgraph of the dual match graph. In the case of tight simulation, only those vertices that find themselves a dual match of the candidate vertex of Q will create a ball around itself.

The distributed algorithms that we have designed and implemented for strong and strict simulation follow the flowcharts of Figure 3, but in a distributed fashion. The first step for applying dual simulation is the same as the distributed algorithm for dual simulation. At the end of the dual simulation phase, each vertex of the data graph has a matchSet that contains the IDs of vertices of Q that match to that vertex. Any vertex with a non-empty matchSet which is qualified to make a ball centered at itself, finds the member of its ball in a breadth-first search (BFS) fashion. Ball creation phase takes \( 2(R - 1) \) supersteps where \( R \) is the selected radius for the ball. For strong and strict simulation \( R = d_Q \), while it is \( R = r_Q \) for tight simulation. Moreover, all the vertices in the neighborhood are considered as members of a ball in strong simulation. In contrast for strict and tight simulation, only vertices with the match flag set to true will answer the request for neighborhood information. In the latter case, while the center vertex of the ball receives neighborhood information, it adds a vertex to the ball only if there is a corresponding edge in the pattern graph. Eventually, the center vertex of each ball will perform the rest of the computation on that ball in a sequential fashion.
Because vertices are distributed among workers, this phase can be considered an embarrassingly parallel workload.

6. EXPERIMENTAL STUDY

This section is dedicated to experimental study which aims to evaluate the new pattern matching models and the proposed distributed algorithms. Regarding the distributed algorithms, the study attempts to learn about their bottlenecks and design trade-offs with respect to the properties of their inputs. We implemented our distributed algorithms on the GPS platform (Salihoglu & Widom, 2012), which is similar to Google’s proprietary Pregel system.

The parameters for data graphs are the number of vertices, denoted by $|V|$, the density of the graph, denoted by parameter $\alpha$, where $|E| = |V|^\alpha$, and the number of distinct labels, denoted by $l$. In all experiments, $l = 200$, unless it is mentioned explicitly. The parameters for queries are also the number of vertices, denoted by $|V_q|$. Another parameter in the experiments is the number of workers denoted by $k$.

6.1. Experimental setting

We used both real world and synthesized datasets in our experiments. In terms of real world datasets, we used uk-2002 with 18,520,486 vertices and 298,113,762 edges; ljournal-2008 with 5,363,260 vertices and 79,023,142 edges; and amazon-2008 which has 735,323 vertices and 5,158,388 edges (Datasets) (Boldi & Vigna, 2004).

We used graph-tool (graph-tool website) to synthesize small and medium size randomly generated data graphs, but because of its memory limits we also implemented our own graph generator to synthesize large semi-randomly generated graphs. The input parameters of our graph generator are the number of vertices, the average number of outgoing edges $deg_O$, and the number of distinct labels. It picks a random integer between 0 and $2deg_O$ as the number of outgoing edges for every vertex. Then, for each outgoing edge, the endpoint of the edge is randomly selected. The label of each vertex is also a randomly picked integer number between 1 and $l$.

To generate a pattern graph, we randomly extract a connected subgraph from a given dataset. Our query generator has two input parameters: the number of vertices and the desired average number of outgoing edges. Unless mentioned otherwise, the average number of outgoing edges is set to such a value that $\alpha = 1.2$.

The experiments were conducted using GPS on a cluster of 8 machines. Each one has 128GB DDR3 RAM, two 2GHz Intel Xeon E5-2620 CPUs, each with 6 cores. The intra-connection network is 1Gb Ethernet. One of the machines plays the role of master.
6.2. Experimental results

The results of the experiments are categorized in four groups. It should be mentioned that distributed strong simulation is so slow on large datasets that we could run it only on fairly small datasets to be compared with strict and tight simulation. It is also noteworthy that presenting running time or speedup of different models in the same chart is only for studying their scalability. In other words, when the quality of pattern matching increases from graph simulation to dual and then strong simulation, the running time also increases. Strict and tight simulation are exceptions; i.e., we observe decrease in running time when the quality increases from strong to strict and then tight simulation.

We also performed a set of experiments to compare the quality of the results of strong, strict, and tight simulation. For comparison, we measured a few parameters in their set of subgraph results including: the number of subgraph results, their total number of distinct vertices, their total number of distinct edges, and the average and standard deviation of their diameters. We found that the number of subgraph results is increasing and their diameters are decreasing while we change the model from strong to strict and from strict to tight simulation. However, our experiments have yet shown no significant difference in either the total number of distinct vertices or the total number of distinct edges.

**Experiment 1 - Running time and Speedup.** We examine the running time and speedup of the proposed algorithms to study their performance and scalability (Figure 8). It can be observed that the running time of tight simulation is always less than the running time of strict simulation.

In the experiment displayed in Figure 8.a, we could not run the test on a single machine because of memory limits. Therefore, we extrapolate its running time.

As expected, the BSP model scales very well on bigger datasets. Moreover, all types of simulation exhibit a filtering behavior, meaning that they start by processing a large set and then refine it; this causes light workload at the final supersteps.

**Experiment 2- Impact of pattern.** Figure 9 shows that running time increases as the size of the query becomes bigger, which is not surprising. The behavior remains similar across datasets with different numbers of vertices. The running times of strict and tight simulations are similar for small patterns because the difference in the overhead of ball-creation phase is negligible. However, it can be observed that the difference between their running-time increases with increase in the size of pattern.

**Experiment 3- Impact of dataset.** In the first experiment of this group (Figure 10.a), we compare the running time of different pattern matching models with
respect to the number of vertices in the data graph. As expected, the running time increases with growth in the size of data graph. The ratio between the running times of graph simulation and dual simulation shows the difference between their computational complexities. The increasing difference between the running times of strict and dual simulation can be explained by the fact that the number of balls increases proportionally to the number of vertices in the result of dual simulation. However, the rate of increase in difference of tight and strict simulation is very smaller because even in the case of tight simulation all the active vertices after dual-filtering phase will contribute to the ball creation although the number of balls is smaller as well. Figure 10.c shows the total number of supersteps for the same set of experiments. It is not surprising that tight simulation needs less number of supersteps to terminate because the radius of its ball is smaller. The stable number of supersteps indicates the scalability of the algorithms with respect to the number of vertices in data graph. That is, increase in the size of the data graph mostly increases the local computation of the workers not their communication.

Figure 10.b shows the impact of the density of a data graph on running time. An increase in running time is to be expected. The unchanged cost of ball creation (difference between dual, strict and tight simulation) reveals a good feature of strict and tight simulation; because the density of the data graph does not have a big impact on the density of the dual match graph, the balls do not necessarily increase in size as the density of the data graph increases. The total number of supersteps for these experiments is reported in Figure 10.d. The small changes in the number of supersteps indicate the scalability of the algorithms with respect to the density of data graph.

Experiment 4- Comparison of strong, strict, and tight simulation. The difference in behavior between the algorithms for strong, strict, and tight simulation can be seen in Figure 11. Because of the high cost of ball creation in distributed strong simulation, it was only possible to test it on fairly small datasets.

Figure 11.a and Figure 11.b compare the running time of the three algorithms on two different datasets and a range of query sizes. The differences between strong simulation and the other two are huge on both datasets, though they are small for $|V_q| \leq 10$. The running time of tight simulation is always slightly better than strict simulation.

Figure 11.c shows the total size of the communication in the system per superstep for strong, strict, and tight simulation. The chart shows three phases for the procedure. The first phase is the dual simulation phase that occurs before superstep 6. The ball creation occurs between supersteps 6 and 19 for strong and strict simulation, while it finishes at superstep 13 for tight simulation. Supersteps 20 and 21 in strong and strict algorithms correspond to

---

**Figure 11. Comparison of strong, strict, and tight simulation for real-world and synthesized datasets ($l = 200; k = 14$)**

\[ a) \text{Running time, synthesized} (|V| = 1e6, \alpha = 1.2) \]

\[ b) \text{Running time, amazon-2008} (|V| \approx 7.3e5, |E| \approx 5.1e6) \]

\[ c) \text{Communication size, amazon-2008} \]

\[ d) \text{Number of active vertices, amazon-2008} \]
processing balls and terminating the algorithms. This phase occurs in supersteps 14 and 15 of tight simulation. It is clear that, there is no communication in these supersteps.

There is a difference in communication at the first superstep because in strong simulation every vertex needs to learn about its neighborhood regardless of its matching status. Strict and tight algorithms perform exactly the same in the first phase. The exponential increase of communication size in strong simulation during creating balls is because of the involvement of all vertices in that process. The jagged shape of communication is because of our BFS-style algorithm for discovering balls, which contains requests at one superstep and responses at the next. Expectedly, the communication size of tight simulation is less than strict during the second phase.

Figure 11.d displays the difference between the numbers of active vertices in the three different types of simulation. Although the number of balls is smaller in tight in comparison to strict simulation, it has the same number of active vertices in the phase of ball creation because all the vertices of dual-match result graph stay active in this phase.

After the dual simulation phase, though the number of active vertices in strict and tight simulation does not experience a significant change, the number of active vertices in strong simulation increases exponentially because inactive vertices become active again during ball creation.

7. RELATED WORK

We surveyed related work in two categories: pattern matching models, and distributed algorithms for pattern matching on massive graphs.

Many matching models find similarity between graphs based on exact structural matching like subgraph isomorphism and its approximate algorithms. On the other hand, some matching models take into account the meaning of attributes of vertices and edges and their relationship in addition to graph structure. We are interested in the recent type which is also called semantic graph matching (Gallagher, 2006).

A few graph pattern matching models introduced during recent years to find a proper semantic graph match like p-homomorphism (Fan, Li, Ma, Wang, & Wu, 2011) and bounded simulation (Fan, Li, Ma, Tang, Wu, & Wu, 2010). Ma et al., in (Ma, Cao, Huai, & Wo, 2011), talk about the idea of a distributed algorithm in abstract, but it is not based on vertex-centric and thus varies greatly from our approach. There are also very interesting theoretical investigations of different types of simulation presented in (Fan W., 2012), but it never comes close to a distributed algorithm for strong simulation.

In (Ma, Cao, Huai, & Wo, 2012) a distributed algorithm only for basic graph simulation is proposed. They have implemented and tested their algorithm on a cluster of 16 machines; however, neither the algorithm nor its implementation is adapted to a widely-used cluster computing platform, but it is implemented all in Python. To best of our knowledge, our algorithms are the first distributed algorithms for dual and strong simulation.

There are also some proposed approaches based on indexing to increase efficiency of answering graph pattern matching queries; for example (Jin & Yang, 2011) or (Cheng, Yu, Ding, Yu, & Wang, 2008). However, creating and storing indexes for massive data graphs may not be feasible.

In (Sun, Wang, Wang, Shao, & Li, 2012) a distributed algorithm is introduced for finding isomorphic subgraph matches in a huge data graph. They deploy graphs on Trinity (Shao, Wang, & Li, 2012) which is a commercial distributed memory cloud environment. To avoid the construction time and the storage capacity for complicated indices, they only use a simple string index which maps node labels to node IDs. Conceptually similar to our approach, their main idea is based on graph exploration, but they cannot avoid expensive joins because of the intrinsic limits of subgraph isomorphism.

Our new simulation model, tight simulation, is a novel improvement of strict simulation which was introduced in (Fard, Nisar, Ramaswamy, Miller, & Saltz, 2013). Its distributed algorithm is also a modification of the distributed algorithm for strict simulation. The experiments for other distributed algorithms are complementary to the previous experiments presented in (Nisar, Fard, & Miller, 2013).

8. CONCLUSIONS

This paper discusses a spectrum of graph pattern matching models, and compares them in terms of the quality of the result and scalability. Also, a new model, named tight simulation, is introduced which is more scalable than the previous models in its family and yields higher quality results. Moreover, novel vertex-centric distributed algorithms for five simulation models are presented. The implementation of the distributed algorithms follow a vertex-centric BSP approach which has been successful in other areas of graph analytics, but has not yet been completely explored in the area of graph pattern matching. The detailed experimental study shows the benefits and costs of the proposed distributed algorithms for various graph simulation models. Our new technique for pattern matching, tight simulation, has been shown to scale better than previous simulation models in a vertex-centric framework.

In the future, we will explore the efficiency of similar techniques for graph pattern matching when the edges of the query and data graph are also labeled. We will also adapt algorithms to use incremental graph pattern matching techniques.

9. ACKNOWLEDGMENT

The authors would like to thank Semih Salihoglu, the developer of GPS from Stanford University, for his valuable
advice regarding the installation and usage of GPS. We would like also to thank Yang Cao, one of the authors of (Ma, Cao, Huai, & Wo, 2012), for sending us his Python code about distributed graph simulation. We are also grateful to Ayushi Jain for her programming efforts.

10. REFERENCES


Proceedings of the 28th ACM symposium on Principles of distributed computing (pp. 135-146). ACM.


Authors

Arash Fard is a PhD candidate in the Department of Computer Science at The University of Georgia. His research interests include distributed computing and big data analytics, especially for graph algorithms. He received a Khwarizmi International Award in 2008 for applied research on Cluster Computing.

M. Usman Nisar recently received his M.S. in Computer Science from the University of Georgia. He is interested in big data analytics and distributed computing, and planning to pursue his PhD in the same area.

John A. Miller is a Professor of Computer Science at the University of Georgia. His research interests include database systems, simulation, Web services and bioinformatics. Dr. Miller received the B.S. degree in Applied Mathematics from Northwestern University in 1980 and the M.S. and Ph.D. in Information and Computer Science from the Georgia Institute of Technology in 1982 and 1986, respectively. During his undergraduate education, he worked as a programmer at the Princeton Plasma Physics Laboratory. He is has been an Associate Editor for the ACM TOMACS, IEEE TSMC and SCS SIMULATION as well as an Editorial Board Member for the JOS and International Journal of SPM.

Dr. Lakshmish Ramaswamy is an Associate Professor in the Dept. of Computer Science at the University of
Georgia. There, he directs the Data Intensive Pervasive Systems Lab. Dr. Ramaswamy's research interests include Data Intensive Systems, Pervasive Systems, and Distributed Computing. He has published extensively in these areas. Dr. Ramaswamy has received the best paper awards at the CoopIS-2010 and WWW-2004 conferences and the Pat Goldberg best paper award in 2005. He obtained a PhD from the Georgia Institute of Technology. His research is supported by grants from the National Science Foundation. Dr. Ramaswamy is a member of the IEEE.