(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 December 2024 (12.12.2024)

WIPO | PCT

(10) International Publication Number
WO 2024/253697 A1

(51) International Patent Classification:
G06F 21/62 (2013.01) *G06F 21/50* (2013.01)

(21) International Application Number:
PCT/US2023/075440

(22) International Filing Date:
28 September 2023 (28.09.2023)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
63/506,549 06 June 2023 (06.06.2023) US

(71) Applicant: **VISA INTERNATIONAL SERVICE ASSOCIATION** [US/US]; PO Box 8999, SAN FRANCISCO, California 94128 (US).

(72) Inventors: **FANG, Zhiyong**; P.O. Box 8999, SAN FRANCISCO, California 94128 (US). **RAGHURAMAN, Srinivasan**; P.O. Box 8999, SAN FRANCISCO, California 94128 (US). **RINDAL, Peter**; P.O. Box 8999, SAN FRANCISCO, California 94128 (US).

(74) Agent: **RACZKOWSKI, David B.** et al.; Kilpatrick, Townsend & Stockton LLP, 1100 Peachtree Street Suite 2800, Atlanta, Georgia 30309 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CV, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IQ, IR, IS, IT, JM, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, MG, MK, MN, MU, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, ST, SV, SY, TH,

WO 2024/253697 A1

(54) Title: PRIVACY-PRESERVING CLIQUE DETECTION

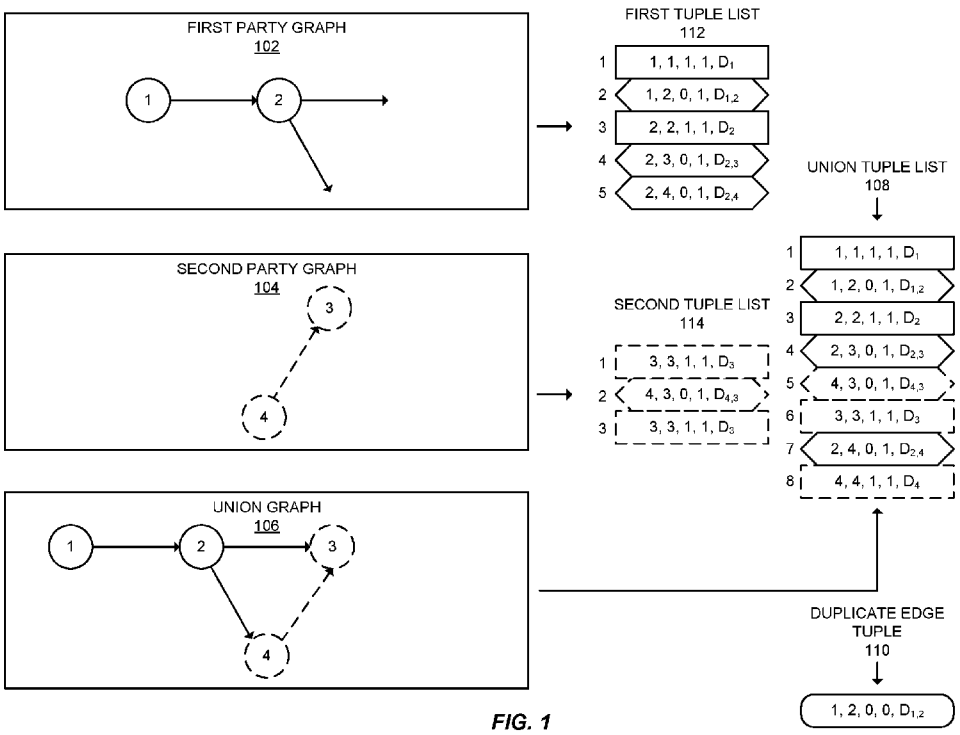

FIG. 1

(57) **Abstract:** Methods and systems can perform efficient, parallel, privacy-preserving graph analysis. One particular application of embodiments is performing private clique detection. Two (or more) parties can each possess private data, which can be represented graphically, and which can be used to construct a private directed union graph corresponding to the union of the parties' data. This private union graph can be analyzed by a multi-party computation network in order to detect cliques in the private union graph. This can be accomplished using oblivious multi-party computation Scatter-Gather-Apply methods in order to preserve the privacy of each party's private data. Also disclosed are several optimization methods that improve the speed and efficiency of parallel, privacy-preserving clique detection methods according to embodiments.

[Continued on next page]

TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) **Designated States** *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, CV, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SC, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, ME, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

— *with international search report (Art. 21(3))*

PRIVACY-PRESERVING CLIQUE DETECTION

CROSS-REFERENCES TO RELATED APPLICATION

[0001] This application is a non-provisional of and claims the benefit of U.S. Provisional Patent Application No. 63/506,549, entitled "PRIVACY-PRESERVING CLIQUE DETECTION ," filed on June 6, 2023, which is herein incorporated by reference in its entirety for all purposes.

BACKGROUND

[0002] A graph generally refers to a collection of nodes (or vertices) connected by edges. Graphs can contain various graph structures, such as subgraphs, which can refer to subsets of nodes and edges from a graph. Many forms of data can be represented by such graphs. For example, social networks can be represented by graphs. In such a social network graph, each person in a group of people can be represented by a vertex and the relationships between those people (e.g., friendships) can be represented by edges connecting two vertices. As another example, a graph could represent a communications network (e.g., a telecommunications network). In such a graph, vertices can represent communications systems (e.g., servers, routers, interchanges, computers, etc.) and edges can represent directed electronic communications between these communications nodes, or channels over which these communications nodes communicate (e.g., telecom lines, wireless transmitter receiver pairs, etc.). Alternatively, edges in such a graph could correspond to directional electronic communications between computers or other devices in the communication network, which can comprise directed transmissions of messages, information, or other data from one computer system to another.

[0003] Patterns or structures in graphs can reveal useful information about the systems modeled by such graphs, or about directional electronic communications between members of such systems. As such, evaluating graphs can be useful for a variety of different applications. For example, in a planned communication network, graph representing the network could be analyzed to identify any inefficient structures in that communication network. Alternatively, in a communication network, a graph representing the network could be analyzed to identify inefficient or illicit directional electronic communications between entities in that communication network. As such, evaluating graphs representing communication systems and directional electronic communications may be useful for

improving the efficiency of those communications system. As another example, graph analysis can be useful for predicting the structure of folded proteins based on their primary structure (which can be represented by a graph).

[0004] In many cases, data that can be modelled graphically may not be held by any single party. Instead, such data may be distributed among a number of parties. This may pose a problem because parties often cannot share their respective data with one another because that data may be sensitive or confidential. For example, two hospitals may be unable to share medical data with one another, as it may contain patient health information. As such, parties are often unable to share the data necessary to construct a corresponding graph. This can in turn prevent graph analysis from being performed on such a graph.

SUMMARY

[0005] Embodiments of the present disclosure relate to efficient privacy-preserving methods of clique detection. A clique can comprise a complete subgraph of a graph, such that each vertex in the graph is connected to each other vertex in the graph by an edge. Cliques are structures that can reveal useful information about the graphs in which those cliques exist, particularly when such graphs represent directional electronic communications between entities (e.g., computer systems) in a communications network. Embodiments enable two or more parties to collectively analyze a secret-shared union graph constructed from a union of the parties' private data using multi-party computation. These methods enable multiple parties to detect cliques across their respective data, without revealing potentially sensitive data to one another, thereby preserving privacy.

[0006] Example embodiments of the present disclosure can have two primary phases. A setup phase can broadly comprise steps that prepare data for later private multi-party clique detection. In a setup phase, each party's computer system (e.g., a first party computer, a second party computer, etc.) can prepare their graphical data by forming tuple lists (e.g., a first tuple list corresponding to a first party and a second tuple list corresponding to a second party) that represents their respective graphical data as lists of tuples, each tuple representing a graph element (e.g., a "vertex tuple" can represent a vertex within a party's subgraph and an "edge tuple" can represent an edge within a party's subgraph). The parties can then use cryptographic techniques such as private set union to generate a secret-shared union tuple list. This union tuple list can represent the union graph produced by combining the parties' collective data. However, because it is in secret-shared form, no party has individual

plaintext access to the data in the secret-shared union tuple list, and as such, no party can access the other parties' private data.

[0007] In a computation phase, a multi-party computation network can perform a multi-party clique detection method. This multi-party clique detection method can be implemented using a "Scatter-Gather-Apply" (SGA) approach. SGA is described in more detail below, but broadly enables the clique detection method to be implemented in three repeating steps: a Scatter step, a Gather step, and an Apply step. One advantage of SGA is that it enables the clique detection method to be performed in parallel by a pool of processors, improving the speed and efficiency of methods according to embodiments.

[0008] This multi-party clique detection method can produce a result that can be transmitted back to the first party computer and the second party computer, or which can be provided to other computers and systems for further processing. For example, if a clique corresponds to a sub-network in a telecommunications network, the clique could be provided to a computer system that would optimize that sub-network in order to remove unnecessary communication channels. In some embodiments, the result of the clique detection process can comprise a plaintext list of these cliques. Alternatively or additionally, the result could comprise some data derived from these detected cliques. As an example, in the context of a communication network, the result could comprise a new communication network configuration. In the context of protein folding, the result could comprise a description of a protein structure.

[0009] One embodiment is directed to a method of performing privacy-preserving detection of one or more cliques in directional electronic communications performed by a multi-party computation network. The multi-party computation network can receive the secret-shared union tuple list from a first party computer and a second party computer. The secret-shared union tuple list can be generated by the first party computer and the second party computer using a first tuple list corresponding to the first party computer, and a second tuple list corresponding to the second party computer. The secret-shared union tuple list can comprise a plurality of secret-shared union tuples corresponding to a representation of a union graph. The secret-shared union tuple list can comprise a plurality of secret-shared vertex tuples representing a plurality of vertices in the union graph, and a plurality of secret-shared edge tuples representing a plurality of edges in the union graph. The multi-party computation network can detect one or more cliques in the secret-shared union tuple list by performing a

multi-party computation on the secret-shared union tuple list. The one or more cliques can comprise one or more complete subgraphs in the union graph. Each complete subgraph of the one or more complete subgraphs can comprise a plurality of subgraph vertex tuples corresponding to a plurality of subgraph vertices in the union graph and a plurality of subgraph edge tuples corresponding to a plurality of subgraph edges in the union graph, such that each subgraph vertex of the plurality of subgraph vertices is connected to each other subgraph vertex of the plurality of subgraph vertices via the plurality of subgraph edges. The multi-party computation network can provide an output corresponding to the one or more cliques to the first party computer and the second party computer, or an additional computer system in response to detecting the one or more cliques.

[0010] Another embodiment is directed to a method of detecting one or more cliques in a union graph corresponding to a secret-shared union tuple list. This method can be performed by a multi-party computation network. The multi-party computation network can receive the secret-shared union tuple list from a first party computer and a second party computer. The secret-shared union tuple list can be generated using a first tuple list corresponding to the first party computer and a second tuple list corresponding to the second party computer. The secret-shared union tuple list can comprise a representation of the union graph. The multi-party computation network can generate a first permutation corresponding to a first ordering and a second permutation corresponding to a second ordering. The first permutation can enable the multi-party computation network to order the secret-shared union tuple list according to the first ordering and the second permutation can enable the multi-party computation network to order the secret-shared union tuple list according to the second ordering. The multi-party computation network can define a set of inputs as a plurality of secret-shared union tuples in the secret-shared union tuple list. The multi-party computation network can execute a parallel private clique detection method, which can comprise a breadth-first or depth-first based clique detection method implemented using an iterative Scatter-Gather-Apply approach. The iterative Scatter-Gather-Apply approach comprising an upward pass, a downward pass, and an Apply step. The upward pass can comprise: (1) dividing the set of inputs among a plurality of processors; (2) processing the set of inputs based on the parallel private graph method and a current ordering of the secret-shared union tuple list using the plurality of processors, thereby producing a set of outputs, wherein the set of outputs comprises less outputs than the set of inputs comprises inputs; (3) defining the set of inputs as the set of outputs; and (4) repeating the upward pass until the set of inputs

comprises a single input. The downward pass can comprise: (5) dividing the set of inputs among the plurality of processors; (6) processing the set of inputs based on the parallel private graph method and the current ordering of the secret-shared union tuple list using the plurality of processors, thereby producing the set of outputs, wherein the set of outputs comprises more outputs than the set of inputs comprises inputs; (7) defining the set of inputs as the set of outputs; and (8) repeating the downward pass until the set of inputs comprises an updated plurality of union tuples in the secret-shared union tuple list. The Apply step can comprise: (9) dividing the updated plurality of secret-shared union tuples among the plurality of processors; (10) applying an apply function to each tuple of the updated plurality of secret-shared union tuples using the plurality of processors, wherein the apply function evaluates and updates a plurality of potential clique lists and a plurality of lists of vertex tuple lists associated with the plurality of secret-shared union tuples; and (11) determining that a terminating condition has not been achieved. If the terminating condition has not been achieved, and if the secret-shared union tuple list is in the first ordering, the multi-party computation network can obliviously shuffle the secret-shared union tuple list into the second ordering using the second permutation, otherwise the multi-party computation network can obliviously shuffle the secret-shared union tuple list into the first ordering using the first permutation. The multi-party computation network can repeat the iterative Scatter-Gather-Apply approach until the terminating condition has been achieved. Afterwards, the multi-party computation network can detect the one or more cliques in the union graph by evaluating the plurality of potential clique lists and/or the plurality of lists of vertex tuple lists, thereby producing a result of the parallel private clique detection method, wherein the result of the parallel private clique detection method comprises a list of the one or more cliques corresponding to the union graph.

[0011] These and other embodiments of the disclosure are described in detail below. For example, some other embodiments are directed to systems, devices, and computer readable media associated with methods described herein.

TERMS

[0012] A "server computer" may refer to a powerful computer or cluster of computers. For example, a server computer can include a large mainframe, a minicomputer cluster, or a group of servers functioning as a unit. In one example, a server computer can include a database server coupled to a web server. A server computer may comprise one or more

computational apparatuses and may use any of a variety of computing structures, arrangements, and compilations for servicing the requests from one or more client computers.

[0013] A "memory" may refer to any suitable device or devices that may store electronic data. A suitable memory may comprise a non-transitory computer readable medium that stores instructions that can be executed by a processor to implement a desired method. Examples of memories include one or more memory chips, disk drives, etc. Such memories may operate using any suitable electrical, optical, and/or magnetic mode of operation.

[0014] A "processor" may refer to any suitable data computation device or devices. A processor may comprise one or more microprocessors working together to accomplish a desired function. The processor may include a CPU that comprises at least one high-speed data processor adequate to execute program components for executing user and/or system generated requests. The CPU may be a microprocessor such as AMD's Athlon, Duron and/or Opteron; IBM and/or Motorola's PowerPC; IBM's and Sony's Cell processor; Intel's Celeron, Itanium, Pentium, Xenon, and/or XScale; and/or the like processor(s).

[0015] An "identifier" may refer to data that can be used to identify something. Examples of identifiers include names and identification numbers. Identifiers can be used to identify things uniquely or relatively. As an example, for a "first list," "second list," and "third list," the terms "first," "second," and "third," may comprise identifiers used to identify the respective lists.

[0016] A "union" may refer to a collection of elements from two or more groups or sets. The union of sets [1, 2, 3] and [3, 4, 5] may comprise the set [1, 2, 3, 4, 5].

[0017] A "disjoint" may refer to all elements from one set that are not included in another set. The disjoint of sets [1, 2, 3] and [3, 4, 5] may comprise the set [1, 2] or the set [4, 5].

[0018] A "graph" may refer to a structure used to represent data. A graph may comprise "vertices" and "edges." In a graph, vertices (usually represented as points) may be connected by edges (usually represented as lines). In a "directed graph" the edges may have a direction, such that they point from one connected vertex to another connected vertex. In directed graphs, edges may be represented by arrows. A "union graph" may refer to a graph comprising the union of two or more other graphs. A "data-augmented graph" may refer to a graph in which vertices and edges may have associated data, such as weights associated with edges or identifiers associated with vertices. A "subgraph" can refer to a graph comprising a

subset of edges and vertices from another graph. An "induced subgraph" can comprise a subgraph "induced" (e.g., produced) by a method or process.

[0019] A "clique" may refer to a complete subgraph, which may comprise a structure within a graph comprising some number of vertices or nodes connected by edges, such that each vertex in the clique is connected to each other vertex in the clique by an edge. A "clique detection method" may refer to a method or function used to detect cliques in a graph.

[0020] "Secret sharing" may refer to techniques used to distribute data (sometimes referred to as a "secret") among a group of participants, such that each participant receives a "share" of the "secret-shared data." Typically, no single party has access to the data, but some group of parties possessing some number of secret shares can collectively reconstruct the data using their respective shares.

[0021] "Multi-party computation" may refer to computations performed by multiple parties, usually using some combination of data belonging to each individual participant. A "secure" multi-party computation may refer to a multi-party computation that does not leak or otherwise reveal the parties' data while the computation is being performed. Secret sharing techniques can be used, in part, to implements secure multi-party computation.

[0022] A "tuple" may refer to a collection of elements (e.g., data values) of some length. For example, a "3-tuple" may comprise the elements [A, 3.2, FALSE]. A tuple may be used to represent some other data or object. For example, a "vertex tuple" may be used to represent a vertex in a graph. Likewise, an "edge tuple" may be used to represent an edge in a graph. A "tuple list" may comprise an ordered list of tuples.

[0023] A "notification" may refer to a message used to notify an entity of something. For example, a "notification of completion" may comprise a message used to notify an entity that something (e.g., a method or function) has been completed.

[0024] A "garbled circuit" or "garbled circuit protocol" may refer to a cryptographic model used to securely evaluate functions. A garbled circuit may comprise an emulation of a Boolean circuit, which when evaluated, performs the function associated with the Boolean circuit without revealing the inputs to the function to the evaluator. Garbled circuits may be used to implement a variety of secure computations, including secure multi-party computations.

[0025] "Private set intersection" may refer to multi-party computation techniques used to compute the intersection of two sets (often belonging to two different parties) without revealing each party's respective set to the other party. "Private set union" may refer to multi-party computation techniques used to compute the union of two sets (often belonging to two different parties) without revealing each party's respective set to the other party.

[0026] An "ordering" may refer to a particular order of a group of elements. For example, for the list of elements [A, B, C, D], a first ordering can comprise [B, A, D, C] and a second ordering can comprise [D, C, A, B]. A "permutation" may refer to a way in which a set of elements can be ordered or arranged. A permutation may be used to define an ordering. For example, the permutation [1, 2, 3, 4] may define the ordering [A, B, C, D], while the permutation [4, 3, 2, 1] may define the ordering [D, C, B, A].

[0027] An "oblivious function" may refer to a function that operates on some input, for which the executor of the function (e.g., a multi-party computation network) remains oblivious about the data being operated on. For example, a computer system performing an oblivious sorting operation may sort a list of data elements in ascending or descending order, without learning any information about the data elements being sorted. Likewise, a computer system performing an oblivious shuffling operation may shuffle a list of data elements according to a permutation, without learning any information about the data elements being shuffled.

[0028] A "terminating condition" may refer to a condition under which something (e.g., a function or method) terminates or end. A "halting condition" may refer to a condition under which something (e.g., a function or method) halts. The terms "terminating condition" and "halting condition" may be used somewhat interchangeably.

BRIEF DESCRIPTION OF THE DRAWINGS

[0029] FIG. 1 shows an exemplary graph used to describe some methods according to embodiments.

[0030] FIG. 2 shows a first exemplary multi-party computation network according to some embodiments.

[0031] FIG. 3 shows a second exemplary multi-party computation network according to some embodiments.

[0032] FIG. 4 shows a diagram used to describe garbled circuits.

[0033] FIG. 5 shows a method of privately constructing a secret-shared union tuple list according to some embodiments.

[0034] FIG. 6 shows a method of privately constructing a secret-shared union tuple list using a disjoint garble circuit according to some embodiments.

[0035] FIG. 7 shows a flowchart corresponding to a setup phase of some methods according to embodiments.

[0036] FIG. 8 shows a diagram detailing a process used to generate a secret-shared union tuple list according to some embodiments.

[0037] FIG. 9 shows a diagram detailing a process used to determine permutations corresponding to orderings of a secret-shared tuple list.

[0038] FIG. 10 shows a diagram summarizing a method used to perform graph analysis on a secret-shared union tuple list according to some embodiments.

[0039] FIG. 11 shows a flowchart corresponding to a Scatter-Gather-Apply phase of a method according to embodiments.

[0040] FIG. 12 shows a diagram of an upward pass according to some embodiments.

[0041] FIG. 13 shows a diagram of a downward pass according to some embodiments.

[0042] FIG. 14 shows a diagram of a parallelized shuffling protocol according to some embodiments.

[0043] FIG. 15 shows an exemplary computer system according to some embodiments.

[0044] FIG. 16 shows two depth-first search based clique detection methods according to some embodiments.

[0045] FIG. 17 shows two breadth-first search based clique detection methods according to some embodiments.

[0046] FIG. 18 shows four Scatter-Gather-Apply clique detection methods according to some embodiments.

[0047] FIG. 19 shows a diagram summarizing an arboricity method used to pre-process a union graph, used to improve the efficiency of Scatter-Gather-Apply clique detection methods according to some embodiments.

[0048] FIG. 20 shows a flowchart corresponding to an arboricity method according to some embodiments.

[0049] FIG. 21 shows a diagram detailing a process to determine permutations using secure random shuffling techniques according to some embodiments.

[0050] FIG. 22 shows a diagram detailing the use of virtual vertex tuples in performing Scatter-Gather-Apply clique detection methods according to some embodiments.

[0051] FIG. 23 shows a flowchart of a method of using virtual vertex tuples to management memory according to some embodiments.

[0052] FIG. 24 shows a flowchart summarizing an exemplary method for performing secure multi-party clique detection according to some embodiments.

DETAILED DESCRIPTION

[0053] As summarized above, some embodiments of the present disclosure are directed to methods and systems for performing privacy-preserving detection of one or more cliques in directional electronic communications. Cliques can comprise complete subgraph structures that can reveal useful information about the graphs in which they exist. For example, for a planned communication network, a graph could comprise a set of communications nodes (vertices) connected by edges, which can represent communications channels between those communications nodes. Examples of directional electronic communications include financial transactions and messages (e.g., text or multimedia, such as audio or video), e.g., to identify connected groups of people or computer. Networking messages are another example.

[0054] A clique could comprise a set of communications nodes that are fully connected, and therefore able to directly communicate without an exchange through an intermediate communication node. In some communication systems, the cost of communications channels may be relatively high, and as such, communicating via an exchange may be more efficient than a clique because of the large number of communications channels involved in a clique. As such, detecting cliques in communication system graphs may be useful for improving the efficiency of those communications systems. As another example, in a communications

network, edges could represent directional electronic communications between entities (e.g., computers, users, etc.) in the communications network. Cliques could represent sets of entities that frequently communicate with each other, which could be unusual or demonstrate illicit use of such communications networks.

[0055] In some embodiments, a first party computer and a second party computer, corresponding to a first party and a second party, and each possessing their own respective data, can generate a secret-shared union tuple list, representing a graph representing a union of the parties' respective data (i.e., a "union graph"). The first party computer and second party computer can transmit this secret-shared union tuple list to a multi-party computation network, which can perform a parallel private multi-party computation to detect one or more cliques in the union graph. As summarized above, some methods according to embodiments can involve a setup phase and a computation phase.

[0056] The setup phase can involve two (or more) computers preparing a secret-shared union tuple list representing their collective data. This secret-shared union tuple list can then be provided to a multi-party computation network to perform parallel private clique detection using the secret-shared union tuple list during the computation phase. Because the secret-shared union tuple list is in secret-shared form, neither party, nor the multi-party computation network, has individual access to the secret-shared data, and cannot learn any party's private data. The setup phase can additionally involve the multi-party computation network performing some pre-processing operations on the secret-shared union tuple list, in order to enable more efficient clique detection during the computation phase.

[0057] In more detail, a first party computer associated with a first party and a second party computer associated with a second party, can each represent their respective data (e.g., first party data and second party data) as a first party tuple list and a second party tuple list respectively. Each "tuple" in these tuple lists can comprise a collection of data, and represents a particular graph element (e.g., a vertex or an edge) in the union graph. The first tuple list and the second tuple list can be input into a private set union process in order to construct the secret-shared union tuple list, which can comprise a plurality of secret-shared vertex tuples representing a plurality of vertices in the union graph and a plurality of secret-shared edge tuples representing a plurality of edges in the union graph. In some embodiments, the private set union process can be implemented using a union garbled circuit. This secret-shared union tuple list can be provided to the multi-party computation network.

[0058] Optionally, the multi-party computation network can perform a secure edge reordering process (also referred to as an "arboricity process"). This process is described in more detail in Sections D below and with reference to FIG. 19. Generally, the secure edge redirecting process can involve modifying the direction of edge tuples in the secret shared union tuple list in order to e.g., reduce the effective in-degree and out-degree of vertices, thereby decrease the complexity and memory demand of clique detection methods according to embodiments. Performing this arboricity process can improve the efficiency of clique detection methods according to embodiments.

[0059] Afterwards, the multi-party computation network can perform an edge tuple duplication process to duplicate the secret-shared edge tuples in the secret-shared union tuple list. After performing this edge duplication process, the secret-shared union tuple list can comprise a plurality of secret-shared vertex tuples, a plurality of secret-shared edge tuples, and a plurality of secret-shared duplicate edge tuples. Alternatively, the first party computer and second party computer can duplicate the edge tuples, e.g., by duplicating the edge tuples in their respective tuple lists prior to determining the private set union. As described in more detail below, duplicating edge tuples allows both a Scatter step and a Gather step of a Scatter-Gather Apply (SGA) clique detection process to be performed simultaneously, improving the efficiency of parallel private clique detection methods according to embodiments of the present disclosure.

[0060] Additionally, the multi-party computation network can determine a first permutation and a second permutation corresponding to the secret-shared union tuple list. The first permutation and second permutation can be used by the multi-party computation network to reorder the secret-shared union tuple list into a first ordering and a second ordering respectively. These permutations may be used in the computation phase to oblivious shuffle the secret-shared union tuple list between the first ordering and the second ordering. The purpose of this shuffling is described in greater detail in Section E below. Determining the first permutation and second permutation in advance of the computation phase enables the multi-party computation network to use the first permutation and the second permutation to obliviously shuffle the secret-shared union tuple list during the computation phase, rather than obliviously sort. Because oblivious shuffling is quicker and more efficient than oblivious sorting, determining the first permutation and second permutation improves the speed and efficiency of clique detection methods according to embodiments. Other steps that

improve the efficiency of the later computation phase can involve determining a plurality of "tuple states," which are described in more detail further below.

[0061] After the first party computer, second party computer, and multi-party computation network complete the setup phase, a computation phase can be performed in order to detect one or more cliques in the secret-shared union tuple list. In more detail, a multi-party computation network, comprising a first server computer, a second server computer, and a third server computer, can perform a private, parallel clique detection process on the secret-shared union tuple list using a pool of processors and a Scatter-Gather-Apply (SGA) approach. In broad terms, a clique detection method according to embodiments can be implemented by repeatedly performing a Scatter step, a Gather step, and an Apply step on the secret-shared union tuples in the secret-shared union tuple list. The secret-shared union tuples can be divided among the pool of processors to enable multiple processors to perform these steps in parallel, improving the speed and efficiency of embodiments of the present disclosure. These Scatter, Gather, and Apply steps can be repeated until a terminating condition has been met, at which point the multi-party computation network can produce a result, e.g., a plaintext list of cliques, which can be returned to the first party computer and the second party computer, or to another computer system.

[0062] Embodiments of the present disclosure introduce a variety of optimization methods to SGA clique detection, which are described in more detail in the following sections. One such example is the use of duplicate edge tuples, as introduced above. Generally, the use of duplicate edge tuples enables both the Scatter and Gather steps to be performed as a single combined step, rather than two separate steps, reducing the total number of operations that need to be performed.

[0063] Another example is the use of a first permutation and a second permutation determined during the setup phase. As described in more detail below, ordering the secret-shared union tuple list may improve the speed at which the secret-shared union tuple list can be processed, by enabling the Scatter, Gather, and Apply steps to be performed in "linear scans" across the union tuple list. However, repeatedly sorting the secret-shared union tuple list during the computation phase may be computationally inefficient. By using predetermine permutations instead, the multi-party computation network can shuffle the secret-shared union tuple list using those permutations. Because shuffling is less computationally intensive

than sorting, replacing sorting operations with shuffling operations can improve the speed and efficiency of embodiments of the present disclosure.

[0064] Yet another example of an optimization technique is the use of aggregation trees to perform clique detection methods according to embodiments. Such aggregation trees, and corresponding "upward passes" and "downward passes" are described in more detail further below with reference to FIGs. 12 and 13. In general, aggregation trees provide a framework enabling multiple processors can collectively perform a task, such as performing a step in the SGA framework. Using such aggregation trees, a step in the SGA clique detection method can be broken down into a number of aggregation operations performed on secret-shared union tuples in the secret-shared union tuple list. These aggregation operations and their corresponding secret-shared union tuples can be divided among the pool of processors, enabling the pool of processors to perform the SGA clique detection method efficiently and in parallel.

[0065] Two other optimization techniques described below relate to the management of memory during clique detection. SGA clique detection can require considerable amounts of memory, particularly for larger union graphs, larger cliques, or greater numbers of SGA iterations. Embodiments of the present disclosure can use two memory management techniques to reduce or "bound" the total memory required to perform clique detection. One of these techniques involves evaluating the memory associated with each vertex tuple and each edge tuple in the secret-shared union tuple list during clique detection. If the data stored in association with that vertex tuple or edge tuple exceeds an allocated memory amount, the multi-party computation network can randomly delete data from those associated memory units, in order to limit the total amount of memory used during clique detection. Another technique involves the generation and use of "virtual vertex tuples." By using virtual vertex tuples, the multi-party computation network can limit the total amount of memory that needs to be allocated to any individual vertex tuple or edge tuple in the secret-shared union tuple list, thereby improving the memory efficiency of methods according to embodiments.

[0066] Before describing methods according to embodiments in more detail however, some example graphs, a system model, and some background concepts (which may facilitate a better understanding of embodiments of the present disclosure) are described in more detail below.

I. EXAMPLE GRAPHS AND TUPLE LISTS

[0067] FIG. 1 shows three graphs and three tuple lists that are used as examples throughout the disclosure. These graphs and their corresponding tuple lists are used to explain some method steps according to some embodiments of the present disclosure. The first party graph 102 can correspond to a first party (e.g., a first telecommunications organization) and the second party graph 104, rendered with dashed lines, can correspond to a second party (e.g., a second telecommunication organization). The union graph 106, comprising all the vertices and edges in both the first party graph 102 and the second party graph 104, contains one clique comprising vertices 2, 3, and 4. Such a clique can be detected using parallel private clique detection methods according to embodiments. It should be understood that the example graphs used in FIG. 1 and the other figures have been intentionally simplified for the purpose of explaining methods according to embodiments. In many real-world applications, graphs are often considerably larger, and may (in some cases) not contain any hanging edges, such as the directed edges on vertex 2 of the first party graph 102.

[0068] Each of these graphs may be represented as "tuple lists," e.g., the first party graph 102 may be represented by a first tuple list 112, the second party graph 104 may be represented by a second tuple list 114, and the union graph 106 may be represented by a union tuple list 108. Each "tuple" in each tuple list may correspond to an individual graph element in the tuple list's respective graph. For example, the first tuple list 112 may comprise five tuples, corresponding to the five graph elements in the first party graph 102, i.e., the two vertices and three edges. As described in more detail below, it may be more efficient to perform graph analysis methods such as clique detection on tuple lists, rather than on other representations of graphical data.

[0069] Tuples in tuple lists can comprise "vertex tuples" and "edge tuples", as well as "duplicate edge tuples", described in more detail further below. Throughout the figures, vertex tuples are represented by rectangles with sharp corners, edge tuples are represented by wide hexagons, and duplicate edge tuples (such as duplicate edge tuple 110) are represented by rectangles with rounded corners. As with their corresponding graph elements, tuples corresponding to a first party graph (such as first party graph 102) are usually rendered with solid lines, while tuples corresponding to a second party graph (such as second party graph 104) are usually rendered with dashed lines. A tuple that is a member of a union tuple list may be referred to as a "union tuple."

[0070] Each tuple can comprise a list of data elements that describe the corresponding graph element. Such data elements can define, for example, whether the tuple comprises a vertex tuple or an edge tuple, which two vertices an edge tuple connects, and any other data D associated with a given tuple. This data may be generated, modified, and evaluated (in secret-shared form) during clique detection operations in order to detect cliques in a union graph (such as union graph 106). In the figures, tuples adhere to a tuple format (u, v, *isVertex, isOriginal, data*). In this format, u can comprise an identifier of a vertex (e.g., a numeric identifier), v can comprise an identifier of a connected vertex pointed at by a directed edge, *isVertex* can comprise a bit designating whether the tuple corresponds to a vertex (where *isVertex* = 1 for a vertex tuple and, *isVertex* = 0 for an edge tuple), *isOriginal* can comprise a bit designating whether an edge tuple is a duplicate edge tuple (where *isOriginal*=1 for vertices and original edge tuples, and *isOriginal*=0 for duplicate edge tuples), and *data* (sometimes represented as D_u or $D_{u, v}$) can comprise data associated with the particular vertex or edge tuple. for a vertex tuple, u and v may both comprise the same value u, e.g., (u, u, 1, 1, D_u). By comparison, an original edge tuple may comprise e.g., (u, v, 0, 1, $D_{u,v}$) and a duplicate edge tuple may comprise e.g., (u, v, 0, 0, $D_{u,v}$). In some cases, particularly when describing aggregation trees and "tuple states", vertex tuples may be referred to as "W-tuples" or "white" tuples, original edge tuples may be referred to as "G-tuples" or "gray" tuples, and duplicate edge tuples may be referred to as "Y-tuples" or "yellow" tuples.

[0071] As described above, in some circumstances, the two parties corresponding to the first party graph 102 and the second party graph 104 may want to detect cliques in their union graph 106. However the two parties may not want to reveal their private data to one another. As such, the two parties can use embodiments of the present disclosure to construct the union graph 106 in secret-shared form, enabling a multi-party computation network to efficiently process the union graph 106 to detect any cliques therein.

[0072] The secret-shared form of the union graph 106 could comprise a secret-shared union tuple list, which can comprise a union tuple list (such as union tuple list 108) that has been cryptographically processed. Such a secret-shared union tuple list can comprise secret-shared vertex tuples, secret-shared edge tuples, and in some cases, secret-shared duplicate edge tuples. Secret sharing is described in more detail further below. However, as a broad overview, secret sharing often involves securely generating one or more secret shares that when combined in some manner, reproduce the data that has been secret shared. For

example, multiple secret shares corresponding to a secret-shared vertex tuple can be combined to reproduce that vertex tuple in plaintext form. Such secret shares can be distributed among computers in the multi-party computation network, such that no computer in the multi-party computation network is individually able to recover or reproduce the secret-shared union tuple list. In general terms, by operating on the secret-shared union tuple list, rather than a corresponding tuple list, the multi-party computation network can detect cliques in a union graph (such as union graph 106) corresponding to the secret-shared union tuple list, while still protecting the privacy of the two parties.

[0073] As described below in Section D, the two parties can use a private set union process to combine their respective tuple lists (e.g., first tuple list 112 and second tuple list 114), thereby producing a secret-shared union tuple list. The two parties can provide this secret-shared union tuple list to the multi-party computation network so that the multi-party computation network can perform clique detection. This private set union process can be implemented using a garbled circuit, as described below with reference to FIG. 5.

II. MULTI-PARTY COMPUTATION NETWORK

[0074] Prior to describing methods according to embodiments in more detail, it may be helpful to describe a multi-party computation network that can perform parallel private clique detection methods on a secret-shared union tuple list. FIG. 2 shows a diagram of an exemplary system according to some embodiments. This exemplary system comprises two client computers: a first party computer 202 and a second party computer 204, as well as a multi-party computation network 206 (sometimes referred to as a "secret-sharing network"). In some embodiments, the multi-party computation network 206 may comprise a three-party honest majority semi-honest multi-party computation network, which may collectively execute three-party secret sharing and computation schemes, such as those described by Araki et al. [14].

[0075] The multi-party computation network 206 can comprise a first server computer 208 (sometimes referred to as a "first computer"), a second server computer 210 (sometimes referred to as a "second computer"), and a third server computer 212 (sometimes referred to as a "third computer"). These server computers may each comprise one or more processors and one or more non-transitory computer readable media coupled to those processors. Any processors available to the multi-party computation network 206 for performing methods according to embodiments may collectively be referred to as a "pool of processors," and the

memory resources available to the multi-party computation network 206 may be referred to as a "shared memory." Because clique detection methods according to embodiments can be performed in a highly parallel manner, the multi-party computation network 206 may possess a large pool of processors in order to take advantage of this parallelism. In some embodiments, each of the server computers may manage or orchestrate their own respective computing clusters, which may provide the processors in the pool of processors. It should be understood that methods according to embodiments can conceivably be executed with other forms of multi-party computation networks 206, including multi-party computation networks comprising two computer systems or comprising more than three computer systems, and as such, FIG. 2 is intended only as a non-limiting example of a single possible configuration.

[0076] The computers of FIG. 2 may communicate with one another via a communication network, which can take any suitable form, and may include any one and/or the combination of the following: a direct interconnection; the Internet; a Local Area Network (LAN); a Metropolitan Area Network (MAN); an Operating Missions as Nodes on the Internet (OMNI); a secured custom connection; a Wide Area Network (WAN); a wireless network (e.g., employing protocols such as, but not limited to a Wireless Application Protocol (WAP), I-mode, and/or the like); and/or the like. Messages between computers and devices may be transmitted using a secure communications protocol, such as, but not limited to, File Transfer Protocol (FTP); HyperText Transfer Protocol (HTTP); Secure HypterText Transfer Protocol (HTTPS); Secure Socket Layer (SSL), ISO (e.g., ISO 8583) and/or the like.

[0077] The first party computer 202 can correspond to a first party (e.g., a data owner) that possesses first party data that can be evaluated during a parallel private clique detection process. The second party computer 204 can similarly correspond to a second party that possesses second party data that can be evaluated for this purpose. The first party computer 202 and second party computer 204 can communicate to generate secret-shared union tuple list, as summarized above and described in more detail further below. This secret-shared union tuple list can be provided to the multi-party computation network 206 in order for the multi-party computation network to perform clique detection. As an example, the first party computer 202 and second party computer 204 could each distribute secret shares corresponding to the secret-shared union tuple list to the first server computer 208, the second server computer 210, and the third server computer 212, thereby providing the secret-shared union tuple list to the multi-party computation network 206. Such secret shares can be

generated using any appropriate secret-sharing technique, such as the three-party secret-sharing technique of Araki [14].

[0078] As summarized above, after receiving the secret-shared union tuple list, the computers in the multi-party computation network 206 can communicate with one another in order to collectively perform a multi-party clique detection process on the secret-shared union tuple list. This computation can be performed by the first server computer 208, the second server computer 210, and the third server computer 212 using a three-party honest majority semi-honest multi-party implementation of a clique detection method. Upon completing this clique detection method, the multi-party computation network 206 can produce a result which can be provided the first party computer 202 and second party computer 204. Such a result could comprise, e.g., a plaintext list of cliques, or alternatively some data or information derived from the plaintext list of cliques. For example, in the context of protein structure prediction, the output could comprise a description of the structure of a protein based on any detected cliques.

[0079] In some embodiments, the multi-party computation network 206 can further process the output of the clique detection process or alternatively can transfer the output to another computer system (e.g., a computer system other than the first party computer 202 or the second party computer 204) for processing. Such processing may depend on the context or purpose of clique detection. As an example, in the context of communications network optimization, the multi-party computation network may determine or replace detected cliques with more optimized (e.g., possessing less communication channels) subgraphs. Alternatively, the multi-party computation network 206 may transmit the output of the clique detection process to another computer system to perform this optimization.

[0080] In some embodiments, the first party computer and second party computer may be members of the multi-party computation network, such that the first party computer is the first server computer and the second party computer is the second server computer. FIG. 3 shows an alternative system model according to these embodiments. In this system, the first party computer 304 and second party computer 306 can replace the first server computer and second server computer respectively. In such embodiments, the first party computer 304 and the second party computer 306 can generate a secret-shared union tuple list based on their respective data, then perform a three-party honest majority semi-honest multi-party computation (with the third server computer 308) in order to detect one or more cliques in the

secret-shared union tuple list. Afterwards, a result, such as a plaintext list of one or more cliques may then be output to the first party computer 304 and second party computer 306.

[0081] Having introduced embodiments of the present disclosure, described the example graph, and introduced the system models, the rest of the detailed description is organized as follows: Section C describes some background concepts, including technical details that may facilitate a better understanding of the setup phase and computation phase. Additionally, Section C describes some difference between embodiments and conventional graph analysis or clique detection techniques. Section D describes operations, processes, and other steps associated the setup phase. Section E describes operations associated with the computation phase, including a parallel private Scatter-Gather-Apply implementation of a clique detection method. Section F describes some metrics and techniques that can be used to evaluate the performance of embodiments of the present disclosure. Section G describes a computer system according to some embodiments, and Section H provides a list of references.

III. BACKGROUND CONCEPTS

[0082] The sections below generally describe some concepts related to embodiments of the present disclosure, such as graphs, oblivious functions, multi-party computation, clique detection techniques, Scatter-Gather-Apply, etc. Understanding such concepts may facilitate a better understanding of embodiments of the present disclosure. However, before describing these background concepts in more detail, a description of some notation may be helpful. When referring to individual elements of a collection (e.g., a party of a plurality of parties, a tuple of a plurality of tuples), an element $i \pm 1$ can refer to the element following element i (+) or the previous element (-) preceding element i, with wrap around when applicable. For example, for a set of three parties (party 1, party 2, and party 3), party 3 + 1 can refer to party 1 and party 1 – 1 can refer to party 3. Let κ refer to the computational security parameter and λ refer to the statistical security parameter. Some embodiments of the present disclosure use $\kappa = 128$ and $\lambda = 40$ as the computational security parameters and statistical security parameter respectively.

A. Data Augmented Directed Graph

[0083] As described above, some embodiments of the present disclosure can be used to perform parallel private (e.g., oblivious) clique detection using a multi-party computation network. Such clique detection may be performed on a secret-shared union tuple list, which

may represent a union graph. Such a union graph can comprise a "data augmented directed graph." As review, a "directed graph" $G(V, E)$ can comprise a collection of vertices (or "nodes") V that are connected by directed edges E, which can comprise edges that point away from one vertex connected to that edge toward another vertex connected to that edge. A data-augmented directed graph $G(V, E, D)$ can comprise a directed graph $G(V, E)$ comprising vertices V and edges E, as well as data corresponding to each vertex and each edge $D \in (\{0,1\})^{|V|+|E|}$. There is a large variety of data that can be used to augment a directed graph. As an example, for a data augmented directed graph corresponding to a telecommunications network, data associated with a vertex could comprise, for example, an identifier used to identify a relay, server, or router, while data associated with an edge could comprise, for example, an identifier of a communication channel (such as a cable or telephone wire) connecting two communications node in the telecommunication network. In embodiments of the present disclosure, data used to augment a directed graph can comprise one or more potential cliques stored in a plurality of potential clique lists associated with secret-shared vertex tuples, edge tuples, and duplicate edge tuples. As described in more detail further below, by iteratively updating and evaluating these potential clique lists (e.g., determining whether a given potential clique is actually a clique or is not a clique), the multiparty computation network can detect one or more cliques in the union graph.

[0084] For any given vertex $v \in V$ and any edge $e \in E$ in a data augmented directed graph $G(V, E, D)$, "v.data" and "e.data" may be used to refer to the data associated with that vertex and that edge respectively. Alternatively or additionally, an expression such as "$v.X$" or "$e.X$" may be used to refer to a data element X associated with a vertex or edge. As a brief example, in FIG. 18 $v.Ts$ may refer to a "vertex tuple potential clique list," and $v.Ss$ may refer to a "first list of vertex tuple lists", while $e.Ts$ and $e.Ss$ may refer to an "edge tuple potential clique list" and a "second list of vertex tuple lists" respectively.

B. Oblivious Functions and Memory Access

[0085] The term "oblivious" has different meaning in different cryptographic contexts. In a general sense, an action or process, performed on some data elements, is oblivious if that action or process does not reveal any information about those data elements. For example, when performing an oblivious sorting process, a party can sort a list of encrypted or secret shared data without revealing any information about that data (e.g., the relative "rank" or position of particular data elements). As another example, in an oblivious data transfer, a

receiving party can receive data (e.g., a message) from a sending party, without the sending party knowing what data it transmitted. The term "insecure" is generally used to describe non-oblivious processes. For example, an insecure sorting process can refer to standard storing techniques, such as quicksort or bubble sort.

[0086] For ease of exposition, many of the oblivious methods or processes described herein are described from the perspective of their insecure counterparts. It is broadly assumed that a practitioner of methods according to embodiments has the ability to produce oblivious version of insecure methods or processes. Garbled circuits (described in more detail below), are one technique that can be used to implement oblivious methods or processes from their insecure variants, e.g., by designing a Boolean circuit to implement the insecure method, then "garbling" that circuit to produce a garbled circuit that can be used to perform a corresponding oblivious methods.

[0087] However, for the sake of completeness, some considerations relating to the design and implementation of parallel oblivious methods are described in some detail below. Generally, when multiple processors are performing some method (in parallel) on some data, that data may be stored in a shared memory, which may be either real or virtual. As an example, each processor may store some amount of secret-shared data in local memory, but the processors may transmit or otherwise share this data with one another in order to perform an oblivious method. In such a case, a virtual memory array may refer to the collective local memory associated with the processors. Generally, for oblivious multi-party computations involving shared memory arrays, it is often insufficient to perform secure multi-party computation using conventional techniques, because accesses to the shared memory array (e.g., via read and write operations) may reveal information about the distribution of the data being operated on. For example, sorting operations may reveal the relative sorting rank of data elements based on accesses to the shared memory.

[0088] Such oblivious functions can be performed by multiple processors performing such functions in parallel. Consider N processors that make oblivious accesses to a shared memory array. Let a parallel method (e.g., a clique detection method) execute in T parallel steps. Then, in every step, processors $i \in [N]$ make access to some shared memory location $\text{addr}_{t,i}$. The trace $\text{Tr}(G)$ of the method is the ordered tuple that consists of all memory locations accessed by all processors:

$$Tr(G) = (addr_{t,i})_{t\in[T],i\in[N]}$$

[0089] A parallel graph processing method is oblivious, if for any input data-augmented graphs $G = (V, E, D)$ and $G' = (V', E', D')$ with $|V| + 2|E| = |V'| + 2|E'|$ and $|d| = |d'|$ for $d \in D$ and $d' \in D$:

$$Tr(G) = Tr(G')$$

[0090] In some embodiments of the present disclosure, the parallel oblivious methods can be deterministic. In such cases, the traces for both graphs can be identical rather than identically distributed. Note that for any given graph $G(V, E, D)$, some methods according to embodiments can reveal the total number of vertices and edges in the union graph, i.e., $|V| + |E|$.

[0091] Generally, in embodiments of the present disclosure, when oblivious "Scatter," "Gather," and "Apply" steps are performed on a secret-shared union tuple list, those oblivious steps may not reveal any information about the secret-shared union tuple list. For example, if the Apply step is oblivious, observing memory access operations performed during the Apply step should not enable the observer to determine which secret-shared tuples comprise vertex tuples or edge tuples.

[0092] In embodiments of the present disclosure, a plurality of processors, or multiple pluralities of processors (e.g., each plurality of processors being associated with a different computer system in a multi-party computation network) may collectively and privately perform clique detection on a private union graph. In order to perform this function, these processors may have to collectively access this secret-shared data, represented by a secret-shared union tuple list. Such secret-shared data may be stored in a shared memory, which may be either real or virtual. As an example, each processor may store some amount of secret-shared data (e.g., in the form of secret-shared tuples) in local memory (e.g., RAM), but the processors may transmit or otherwise share this secret-shared data with one another in order to perform multi-party computation functions. In such a case, a virtual memory array may comprise the collective local memory associated with the processors.

C. Multi-Party Computation

[0093] The phrase "multi-party computation" (MPC) is sometimes used as a shorthand for "secure multi-party computation", which can refer to computations collectively performed by multiple entities (e.g., computer systems), that do not involve the computer systems revealing

their respective data (e.g., inputs to the computation) to one another. In embodiments, a multi-party computation network can perform clique detection using MPC on a secret-shared union tuple list, without each computer revealing their respective secret shares to one another.

[0094] There are a variety of ways in which multi-party computations can be classified, e.g., based on the number of parties and based on factors that influence the security of such computations. For example, a "three-party honest majority semi-honest multi-party computation protocol" can refer to a multi-party computation protocol performed by three parties, of which it is assumed that at least a majority of the parties are "honest" and the remaining party is "semi-honest." In MPC, an honest party generally refers to a party that performs the MPC correctly and does not attempt to learn any additional information about each party's input or otherwise subvert the MPC. A semi-honest party, by contrast, may perform the MPC correctly, but may also attempt to learn information about the other party's inputs, e.g., by evaluating messages or other data received from those parties.

[0095] In some embodiments of the present disclosure, the multi-party computation network can perform a three-party honest majority semi-honest multi-party computation protocol. This is in contrast to most convention MPC methods, which often use a two-party garbled circuit protocol. As described in more detail below, the use of a three-party honest majority semi-honest MPC protocol enables the use of an efficient three-party, honest majority, semi-honest oblivious shuffling protocol, which is an improvement over oblivious sorting. In order to perform this three-party honest majority semi-honest multi-party computation protocol, some embodiments of the present disclosure can use the replicated secret sharing technique of Araki, et al. [14]. Using this replicated secret-sharing technique, a secret value $x \in \mathbb{Z}_{2^k}$ (e.g., a union tuple list, or individual tuples in a union tuple list) can be shared by sampling three random values $x_1, x_2, x_3 \in \mathbb{Z}_{2^k}$ such that $x = x_1 + x_2 + x_3$. These shares can be distributed as pairs $\{(x_1, x_2), (x_2, x_3), (x_1, x_3)\}$, where each multi-party computation participant i (e.g., the first server computer 208, the second server computer 210, and the third server computer 212 from FIG. 2) holds the i^{th} pair. This secret sharing protocol can also be denoted $[\![x]\!]^A$, and is resilient against one corrupt participant, as any two of the three parties have sufficient information to reconstruct the value x. In embodiments of the present disclosure, three random values or vectors x_1, x_2, x_3 can be sampled that represent the secret-shared union tuple list (or individual secret-shared union tuples in the secret-shared union tuple list). These values can be distributed among computers in a multi-party

computation network, enabling the multi-party computation network to securely and obliviously detect one or more cliques in the secret-shared union tuple list.

[0096] Embodiments of the present disclosure can use multi-party computation software libraries in order to implement multi-party computation. These libraries can include the ABY^3 library [11], which is implemented in C++ and provides support for replicated secret sharing. ABY^3 uses the Boost C++ library for networking among parties. Additive secret sharing can be implemented on top of ABY^3 to provide extra functionality. ABY^3 can be used to implement three-party honest majority semi-honest secure multi-party computation. ABY^3 uses the libOTe library [12] and provides C++ classes for composing circuit libraries. ABY^3 additionally uses cryptoTools [13] which supports MPI-like non-blocking send and blocking receive operations. Processes in ABY^3 are identified by their unique identifiers. Oblivious shuffling techniques such as those described in [2] can be implemented in ABY^3.

D. Depth-First and Breadth-First Search

[0097] Clique detection can be implemented using graph or tree searching techniques such as depth-first search and breadth-first search. While there are a variety of other methods that can be used to perform clique detection, breadth-first search and depth-first search based methods may be more efficient than these other methods. As an example, cliques can be detected using a naive brute force method that can involve selecting each subgraph of a particular size within a graph, then evaluating that subgraph to determine if it is a clique. However, the number of subgraphs in a graph generally grows exponentially as the graph grows. As a result, for large graphs, such methods may be less efficient than breadth-first search or depth-first search. Further, methods other than breadth-first search may be difficult to parallelize, making such methods impractical for large graphs in computationally intense oblivious contexts.

[0098] For a connected graph (e.g., a graph for which there is a path connecting any vertex and any other vertex), either depth-first search or breadth-first search can be used to traverse the entire graph. However, the order in which vertices are visited in depth-first search and breadth-first search may be different. In executing a breadth-first or depth first search in a hypothetical graph, a particular vertex can be selected as a root vertex or root node. This root vertex may be connected to one or more other vertices by edges. The "distance" between the root vertex and another vertex in the graph may be equal to the number of edges that are traversed in order to move from that root vertex to the other vertex. Hence, the graph may

contain one or more other vertices that are at a distance of one from the root vertex, one or more other vertices that are at a distance of two from the root vertex, etc.

[0099] In general terms, in a breadth-first search, the entity (e.g., a computer system) performing the breadth-first search may start at the root vertex, then visit each vertex of distance one from the root vertex, then visit each vertex of distance two from the root vertex, then distance three, etc. By contrast, in a depth-first search, the entity performing the depth-first search may start at the root vertex, then visit a connected vertex at a distance one away from the root vertex, then visit a connected vertex at distance one from that vertex (and at distance two from the root vertex), etc., until the entity has exhausted an entire "path" from the root vertex. The entity can then return to the root vertex and visit a different vertex, repeating this process until the entire graph has been traversed.

[0100] FIG. 16 shows two methods corresponding to a sequential insecure (e.g., non-oblivious) implementation of a k-clique detection method based on depth-first search (DFS). Such a clique detection method can be used to identify cliques of size k or less in an input graph. In method one, an output list of cliques D can be initialized along with a counter variable *idx*. For each vertex v among a set of vertices in a graph V, a computer system can initialize a potential clique T with the vertex v and initialize a vertex list S with the outgoing neighbors of v (i.e., the vertices connected to v by directed edges pointing away from v and toward those vertices). For each vertex v, the computer system can then call the recursive DFS method (method 2) for this vertex v.

[0101] In method 2, the computer system can evaluate if the counter variable *idx* is greater than or equal to k. If this is true, then method 2 has been completed for a particular vertex tuple. Such a condition can be referred to as a terminating condition. If the counter variable *idx* is greater than or equal to k, then the computer system has determined that the potential clique T is a true clique, and can include the potential clique T in the output clique list D. The computer system can return to method 1 and advance to a new vertex v.

[0102] Otherwise, if the terminating condition has not been achieved, the completer system can evaluate if the vertex list S contains no vertices. If this is the case, then the computer system can determine that the vertex v is not a member of a clique. The computer system can return to method 1 and advance to a new vertex v, without including potential clique T in clique list D.

[0103] Otherwise, the computer system can traverse through all of the elements in vertex list *S* (e.g., vertex *v'*), generating a new potential clique *T'* and a new vertex list *S'* and perform method two again on this new vertex *v'*, new potential clique *T'*, and new vertex list *S'* until either a clique has been detected (which can then be added to clique list *D*), or the computer system has exhausted all the vertices in vertex list *S*.

[0104] FIG. 17 shows two methods corresponding to a sequential insecure (e.g., non-oblivious) implementation of a *k*-clique detection method based on breadth-first search (BFS). In method 3, a computer system can initialize a clique list *D*, and for each vertex tuple *v* among a set of vertex tuples in a graph *V*, the computer system can initialize a potential clique *T*, a vertex list *S*, and perform method 4 for each of these vertex tuples *v*.

[0105] In method 4, the computer system can initialize a list of vertices *vs*, a list of potential cliques *Ts*, and a list of vertex lists *Ss*, as well as a second list of vertices *vs'*, a second list of potential cliques *Ts'*, and a second list of vertex lists *Ss'*. The computer system can perform an iterative process for a counting variable *idx* from *idx*=1 to *idx*=*k*. If *idx* is equal to *k*, a terminating condition has been achieved, and the computer system can determine that the potential cliques *T* in the list of potential cliques *Ts* are valid cliques. In such a case, the computer system can include each of the potential cliques in *Ts* in the output clique list *D*.

[0106] Otherwise, the computer system can iterate through each vertex *v*, potential clique *T*, and vertex list *S* in the list of vertices *vs*, the list of potential cliques *Ts*, and the list of vertex lists *Ss*. If a vertex list *S* is not empty, the computer system can evaluate each vertex *v'* in that vertex list *S*. The computer system can add that vertex *v'* to a corresponding list of vertices *vs'* and to a potential clique list *Ts'* (along with potential clique *T*). Further, the computer system can determine the intersection between a vertex list *S* and the outgoing neighbors of *v'*, in order to determine if *v'* is potentially a member of a clique. As indicated at indicator 1702, this intersection can be added to a list of vertex lists *Ss'*. After completing a round of this iterative process for each *v*, *T*, and *S* in *vs*, *Ts*, and *Ss*, the computer system can initialize a new set of *vs*, *Ts*, and *Ss*, and a new set of *vs'*, *Ts'*, and *Ss'*, increment *idx*, and repeat this process. By repeating method 4 for each vertex *v* in the set of vertices *V*, it is possible to detect all cliques of size *k* or less in an input graph.

[0107] It should be understood that the methods described above with reference to FIGs. 16 and 17 are serial, insecure *k*-clique detection methods using breadth-first search and depth first search. However, embodiments of the present disclosure are directed to parallel

oblivious clique detection methods implemented using a Scatter-Gather-Apply (SGA) framework. As such, the description above is intended only to facilitate an understanding of the general process of clique detection using breadth-first and depth-first search. While either breadth-first search or depth-first search can be used to implement clique detection methods according to embodiments of the present disclosure, in some embodiments, breadth-first search may be preferable because it can be more efficiently parallelized. By contrast, it is considerably more difficult to efficiently parallelize depth-first search, and as such, depth-first searches typically take more time to complete, particularly in oblivious or secure implementations.

E. Scatter-Gather-Apply (SGA)

[0108] As described above, some embodiments of the present disclosure use Scatter-Gather-Apply (SGA) technique, paradigms, or frameworks to perform methods according to embodiments. SGA may also be referred to as "Gather Scatter" or other similar terms and may also be referred to as the Pregel and GraphLab [4]-[6] programming paradigms. SGA is highly efficient for parallel computations performed on graphical data. In such paradigms, parallel graph processing methods can be performed in a series of Scatter, Gather, and Apply steps, performed repeatedly and in sequence. SGA steps are usually performed on a per graph element basis and are often described as if those elements themselves (e.g., the vertices and edges) are performing those steps rather than the computer system or device actually performing the SGA steps. In each iteration of an SGA method, each vertex "scatters" data associated with that vertex along outgoing edges, then "gathers" and aggregates data from other vertices along incoming edges, then "applies" some function to that data, in order to perform the graph analysis methods (e.g., clique detection) being implemented using SGA.

[0109] Because SGA methods are performed on a per-element basis, such methods can be efficiently parallelized by dividing those elements among a pool of processors. In each step of a parallel SGA implementation, each processor can perform the appropriate step operations (e.g., step operations associated with scattering during the Scatter step) to the graph elements assigned to that processor. In some cases, if a sufficient number of processors are available to the multi-party computation network, each processor could conceivably be assigned a single graph element, enabling highly parallel processing.

1. Scatter

[0110] During the Scatter step, each vertex in a graph can propagate data to its neighboring edges and updates the edge's data based on this propagated data. More specifically, Scatter takes a user-defined function f_s: $\{0,1\}^* \rightarrow \{0,1\}^*$, and updates the data (*e.data*) associated with each directed edge *e(u, v)* in a manner consistent with the following pseudocode:

Scatter(*G(V, E, D)*, f_s, *b*)
For each *e(u, v)* in *E* If *b* = "in" $e.data := f_s(e.data, v.data)$ else $e.data := f_s(e.data, u.data)$

[0111] In the above pseudocode, *G(V, E, D)* is the directed graph, f_s is a scatter function, *b* is a control bit indicating the scattering direction (i.e., with or against the directed edges), e(u, v) is a directed edge pointing from vertex *u* to vertex *v*, *e.data* is the data associated with edge e(u, v), *u.data* is the data associated with vertex *u*, and *v.data* is the data associated with vertex *v*. In summary, if the control bit is set to "in," each edge *e(u, v)* updates its data by applying the scatter function f_s to its data and the data associated with vertex *v* (i.e., the vertex being pointed to by the directed edge). If the control bit is set to "out," (or simply set to anything other than "in"), each edge *e(u, v)* updates its data by applying the scatter function f_s to its data and the data associated with the vertex *u* (i.e., the vertex that is not being pointed to by the directed edge). Notably, the Scatter step is applied to each vertex individually and as a result, vertices (or representations of vertices, such as secret-shared vertex tuples) can be divided among a pool of processors that can perform the Scatter step collectively and in parallel.

[0112] The scatter function f_s is typically user-defined and depends on the particular parallel private graph analysis method being implemented. As such, a different scatter function f_s may be used for performing clique detection than, for example, determining a minimum spanning tree or performing graph-based matrix factorization. Scatter functions according to embodiments are described in more detail further below. As a general summary, data associated with secret-shared vertex tuples can be scattered to secret-shared edge tuples that represent outgoing edges connected to those vertices. Such data can comprise lists of potential cliques, which can themselves comprise identifiers of vertices that may be members of those cliques.

2. Gather

[0113] During the Gather step, each vertex can aggregate data that is received from incoming edges. This aggregated data can be stored in association with the vertex. More specifically, in the Gather step, a binary aggregation operator $\oplus: \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$ is used to updates the data *v.data* associated with each vertex $v \in V$ in a manner consistent with the following pseudocode:

```
Gather(G(V, E, D), ⊕, b)
    for each v in V
        if b = "in"
            v.data = v.data ||⊕_{∀e(u,v)∈E} e.data
        else
            v.data = v.data ||⊕_{∀e(v,u)∈E} e.data
```

[0114] In this pseudocode, *G(V, E, D)* is the directed graph, $\oplus$ is an aggregation (or "gather") function, *b* is a control bit indicating the gathering direction (i.e., with or against the directed edges), *v* is a vertex, *V* is the set of all vertices, *v.data* is the data associated with vertex *v*, *e.data* is the data associated with an edge *e(u, v)* (or *e(v, u)*, depending on the value of the control bit *b* described below) connected with vertex *v*, and || indicates the concatenation operation. In summary, if the control bit is set to "in," each vertex *v* updates its data *v.data* by aggregating its data *v.data* and all data associated with incoming edges *e(u, v)* using the aggregation function $\oplus$. If the control bit is set to "out," (or anything other than "in"), each vertex *v* updates its data *v.data* by aggregating its data *v.data* and all data associated with outgoing edges *e(v, u)* using the aggregation function $\oplus$. Notably the Gather step is applied to each vertex *v* individually, meaning that vertices (or representations of vertices, such as vertex tuples) can be divided among a pool of processors that can perform the Gather step collectively in parallel.

[0115] The aggregation function $\oplus$ is typically user-defined and depends on the particular parallel-private graph analysis method being implemented. As such, a different aggregation function $\oplus$ may be used for performing clique detection than, for example, determining a minimum spanning tree or performing graph-based matrix factorization. Aggregation functions and gather steps according to embodiments of the present disclosure are described in more detail further below. In general however, the gather step can involve gather data corresponding to secret-shared incoming edge tuples and aggregating that data in association

with secret-shared vertex tuples connected to those secret-shared edge tuples. Such data can comprise potential clique lists and lists of vertex identifiers, which can be used (e.g., during the Apply step) to determine if potential cliques in those potential clique lists are valid.

3. Apply

[0116] During the Apply step, data associated with vertices can be updated, e.g., based on data aggregated during the previous Gather step. Formally, the Apply step can involve performing an Apply function f_a: $\{0,1\}^* \rightarrow \{0,1\}^*$ in a manner consistent with the following pseudocode:

Apply($G(V, E, D)$, f_A)
for each v in V $v.data := f_a(v.data)$

[0117] In this pseudocode, *v.data* is the data associated with a vertex *v*, *V* is the set of all vertices, and f_A is the apply function. In summary, during the Apply step, the apply function f_A is applied to the data associated with each vertex *v*. Notably, the Apply step can be applied to each vertex individually, meaning that vertices (or representations of vertices, such as vertex tuples) can be divided among a pool of processors that can perform the Apply step collectively in parallel.

[0118] The apply function f_A is typically user-defined and depends on the particular method being implemented. As such, a different apply function f_a may be used for performing clique detection, than, for example, determining a minimum spanning tree or performing graph-based matrix factorization. Apply steps and apply functions according to embodiments are described in more detail further below. In general however, the Apply function can involve evaluating data associated with each secret-shared vertex tuple in the secret-shared union tuple list. Such data can comprise potential clique lists and sets of vertex identifiers. By evaluating this data, the multi-party computation network can determine if these potential cliques in the potential clique list are valid or invalid. If a potential clique is determined to be invalid, it can be removed from its corresponding potential clique list. Once the SGA clique detection method is complete (e.g., some terminating condition has been achieved), the multi-party computation network can evaluate the potential cliques in the potential clique lists. Any remaining potential cliques (i.e., those that were not removed during the SGA clique detection method) can comprise valid cliques. A list of these one or more cliques can be output to the first party computer and second party computer, as described above.

F. Linear Scan

[0119] In brief, a linear scan broadly refers to a method of processing some array of data (e.g., a list), by iterating through each element in that array of data successively. Performing data processing methods as linear scans can be useful, because the time complexity of a linear scan of an array of data is linearly proportional to the length of the array of data. Because many (naive) implementations of methods or processes have "worse than linear" time complexity, determining ways to implement such methods or processes using linear scans may improve the speed and efficiency of such methods or processes. Further in some cases, it may be possible to parallelize a linear scan by dividing the array into a number of sub-arrays and dividing those sub-arrays among a number of processors. Each processor can perform its own linear scan on its respective sub-array, resulting in sub-linear performance of the linear scan over the entire array. If there are a sufficient number of processors, parallelized linear scans can be completed in logarithmic time.

[0120] Some methods generally cannot be performed using linear scans alone. Because linear scans generally involve iterating through the array and operating on its elements sequentially, methods that involve operations on non-consecutive elements of the array may not be able to be performed using linear scans. In some cases, it may be possible to perform linear scans by first sorting the array such that relevant array elements are sequential. For example, in a hypothetical unsorted array, data from element 7 may be needed to operate on element 4. By sorting the array such that element 7 precedes element 4, this update process can be performed in a linear scan.

[0121] In broad terms, as described in more detail below, embodiments of the present disclosure can use oblivious sorting and shuffling methods to order a secret-shared union tuple list, such that the secret-shared union tuple list can be processed using linear scanning techniques. For example, prior to a Scatter step, the secret-shared union tuple list can be ordered such that each outgoing edge tuple is proceeded by the vertex tuple representing a vertex connected to that outgoing edge. As such, when performing the Scatter step, the multi-party computation network can perform a linear scan along the secret-shared union tuple list, as each edge tuple sequentially follows its corresponding vertex tuple.

G. *Sorting Versus Shuffling*

[0122] Embodiments of the present disclosure can make use of oblivious sorting and oblivious shuffling methods in order to perform parallel private clique detection. As such, sorting, shuffling, and permutations are described in brief detail below. Additionally some edge prefix definitions and some more specific shuffling details are provided below.

[0123] In general, sorting involves ordering the elements of an array (e.g., a list) in some order according to some criteria. As an example, a list of numbers can be sorted in ascending or descending order using a sorting method such as quicksort. Obliviously sorting involves ordering elements of an array in some order according to some criteria, without the sorter learning anything about the elements or their ordering (e.g., without learning which element is the largest element).

[0124] By contrast, shuffling usually involves ordering the elements in an array either randomly (in the case of a "random shuffle") or according to some permutation, which may refer to some data that defines some ordering (i.e., a specific permutation) of the elements in that array. A permutation can comprise, for example, a list of numbers corresponding to particular locations in an array, which can be used to assign elements in the array to those locations. Such a list of numbers may also be referred to as an index. For example, the array [A, B, C, D] can be shuffled using the permutation [2, 1, 4, 3] to produce a shuffled array [B, A, D, C]. Similar to an oblivious sorting operation, an oblivious shuffling operation may involve ordering the elements of an array in some order according to a permutation (or ordered randomly), without the shuffler learning anything about the elements or their ordering.

[0125] Shuffling is generally a linear time operation, as it can involve iterating through the array and the corresponding permutation and positioning each element in the array based on the permutation. By contrast, sorting is usually worse than linear, as it involves comparing elements against each other based on some predefined criteria in order to determine their eventual ordering. Further, oblivious processes are typically slower (i.e., have worse time complexity) than their corresponding non-oblivious processes, and as a result, oblivious sorting operations can be considerably slower than oblivious shuffling operations.

[0126] As described in more detail below, in broad terms, some methods according to embodiments involve the multi-party computation network determining permutations

corresponding to different orderings of the secret-shared union tuple list during the setup phase. These permutations can be cached and used during the computation phase to obliviously shuffle the secret-shared union tuple list into two different orderings, which (in broad terms) enable SGA clique detection methods according to embodiments to be performed in efficient parallelized linear scans. Determining these permutations during the setup phase enables the multi-party computation network to use fast oblivious shuffling methods, as opposed to slow oblivious sorting methods, thereby improving the speed and efficiency of some embodiments of the present disclosure.

1. Edge Prefix Definitions

[0127] Some tuple prefix and suffix definitions may be useful in better understanding embodiments of the present disclosure.

[0128] **Definition 1** (Longest Edge Prefix)**.** For a tuple $j \in \{1, 2, \dots, N\}$, the longest edge prefix before j, denoted $LEP[1, j]$, is defined to be the longest consecutive sequence of G-tuples before j, not including j. Note that when the j^{th} tuple is a Y-tuple, LEP can be empty because the prefix of a Y-tuple often starts with either a Y-tuple or a W-tuple.

[0129] **Definition 2** (Longest Edge Suffix)**.** For a tuple $j \in \{1, 2, \dots, N\}$, the longest edge suffix after j, denoted $LES[1, j]$, is defined to be the longest consecutive sequence of Y-tuples after j, not including j. Note that when the j^{th} tuple is a G-tuple, LES can be empty because the suffix of a G-tuple often starts with either a G-tuple or a W-tuple.

[0130] Let $1 \leq i \leq j \leq N$, the notation $LEP[i, j]$ is used to denote the longest consecutive sequence of G-tuples before tuple j, constrained to the subarray $G[i, \dots, j)$ (where the index i is inclusive, and index j is exclusive). Similarly, the notation $LEP(j, k]$ is used to denote the longest sequence of Y-tuples after j constrained to the subarray $G[j, k)$ (where the index j is exclusive, and index k is inclusive).

[0131] **Definition 3** (Longest Prefix Sum)**.** Let $1 \leq i \leq j \leq N$, the notation $LPS[i, j)$ is used to denote the aggregation (with respect to the $\oplus$ operator) of $LEP[i, j)$.

[0132] **Definition 4** (Longest Suffix Distributed)**.** Let $1 \leq i \leq j \leq N$, the notation $LSD[i, j)$ is used to denote the process of writing a value v among the tuples of $LES[i, j)$.

[0133] Abusing notation, $LPS[i, j)$ can be treated as an alias for $LPS[1, j)$ if $j < 1$. Similarly, $LSD(j, k]$ can be treated as an alias for $LSD(j, N]$ if $k > N$.

2. Additional Shuffling Details

[0134] Some embodiments of the present disclosure can use shuffling techniques described by Chida et al. [2]. This shuffling method can involve steps in which the computers in the multi-party computation network can reveal permuted secret shares to each other. Let P be the number of processors associated with each computer in the multi-party computation network. Let G be the corresponding secret-shared list. If the elements of the secret-shared list are divided evenly among the P processors, each of the P processors can be responsible for computing a permutation of G / P tuples in the secret-shared tuple list.

[0135] Briefly, the oblivious shuffling process can function as follows. Inputs can comprise secret shares of a key and a value. Without loss of generality, assume that each item consists of a single key and a single value. Let ℓ be the bit length of the keys. Let $\bar{k}$ and $\bar{v}$ be the set of keys and associated values that are to be sorted.

[0136] Some sorting protocols implement a stable sort, in which the order of elements is rearranged based on the keys. A relative order is maintained for items with equal keys. In other words, the protocol outputs $\bar{v}' = (v_1, v_2, \ldots, v_n)$ that satisfies the following conditions. Let σ be the permutation that satisfies $\bar{v}' = \sigma\bar{v}$, and $\bar{k}' = \sigma\bar{k}$. It holds that $k_i \leq k_{i+1}$, and if $k_i = k_j$, then $\sigma^{-1}(i) < \sigma^{-1}(j)$ only when $i < j$.

[0137] The sorting protocol of Chida et al. [2] implements a variant of radix sort by combining a sequence of ℓ permutations. Each of these permutations is a re-arrangement of the keys based on a specific bit of the key. Note that if all these permutations are applied one after another, then the resulting order of the keys will be the sorted order. However, the construction of Chida et al., does not apply these permutations directly. Instead, it computes the composition of the permutation and then applies the composed permutations all at once. By doing so [2] improves the total communication cost of the sorting protocol compared to the construction of [7] which applies the permutation for every bit. Embodiments of the present disclosure can adopt the approach by Chida [2] et al., first composing permutations corresponding to all the bits and then apply these permutations only once at the end.

[0138] These methods generate the abovementioned permutations in a preserving manner. Each of these protocols adopts an approach due to Bogdanov et al. [7] which takes in a secret share comprising the i^{th} bit of the key and generates a permutation σ whose inverse sorts the

key as per the i^{th} bit. Some embodiments repeatedly compute these permutations for each bit of the key.

H. Garbled Circuits

[0139] Several steps in methods according to embodiments can be implemented using garbled circuits, including privately determining a secret-shared union tuple list. A garbled circuit can comprise a cryptographic protocol that enables two-party (or more) secure computation. Two parties can use a garbled circuit to evaluate a function on their private inputs. For example, a first party and a second party, possessing a first set and a second set, can use a garbled circuit to determine the intersection of their two sets, without requiring either party to reveal their set to the other party. Such an application of garbled circuit can be referred to as "circuit-PSI."

[0140] A garbled circuit is so-called because functions evaluated using garbled circuits can be described as Boolean circuits. After a Boolean circuit is designed, it can then be "garbled," enabling it to be evaluated in encrypted form. This garbling mechanism is what enables the function to be evaluated without either party revealing their (encrypted) private inputs to one another.

[0141] A Boolean circuit generally comprises a collection of Boolean gates connected by wires. Often, in cryptographic contexts, Boolean circuits are models, and thus the wires and gates do not exist as physical objects. Typically, a Boolean circuit is evaluated by processors or other computer systems in order to determine the output of the Boolean circuit based on its inputs.

[0142] A Boolean gate typically comprises one or more inputs and an output. "Signals," comprising the Boolean values {0, 1} (or {FALSE, TRUE}), are carried by the wires to the inputs of the Boolean gate. The Boolean gate produces an output (also a Boolean value), which is carried by a wire through the rest of the circuit. As an example, a two input Boolean "AND" gate produces a Boolean value of 1 if both of its inputs are 1 and produces a Boolean value of 0 otherwise. The relationship of the inputs and outputs for a Boolean gate can be defined by "truth table," a table that relates every combination of Boolean valued inputs with their respective Boolean output.

[0143] Wires and Boolean gates can be combined to produce a wide variety of Boolean circuits implementing useful functions. For example, addition of two variables can be

implemented using a ripple-carry adder circuit, and multiplication can be implemented using a Wallace tree or a Dadda multiplier. Comparatively complex functions such as determining the set intersection or outputting cliques in a graph can also be implemented using Boolean circuits. Provided that some care is taken to avoid leaking private information (e.g., based on the frequency of read and write operations on specific memory addresses, as described above), garbled circuits can even be used to implement oblivious protocols.

1. Garbled Circuit Generation

[0144] In broad terms, a Boolean circuit is "garbled" by replacing each value associated with each truth table corresponding to each gate in the Boolean circuit with randomly generated "labels," then using the input labels to encrypt the output label. The process used to generate a garbled gate is summarized in FIG. 4. FIG. 4 shows an AND gate 402, comprising two inputs: input A and input B, along with an output C. The truth table 404 for this AND gate is shown.

[0145] A "garbler" (e.g., one of the two parties) can replace each Boolean value in the truth table 404 with a randomly generated label, producing a labeled table 406. In labeled table 406, the label associated with a Boolean value of 0 for input A is "X_0^A," and the label associated with a Boolean value of 1 for input A is "X_1^A". A similar labelling scheme is used for the labels for input B and output C.

[0146] The garbler can then encrypt each output label using a known cryptosystem and the two corresponding input labels as cryptographic keys. For example, the label X_1^C can be encrypted using labels X_1^A and X_1^B. This process can be repeated for every row in the table, resulting in a garbled table 408. Although not shown in FIG. 4, the rows of the garbled table may be shuffled or otherwise randomized, in order to prevent an observer from determining any correspondence between labels and their associated values based on the row order.

[0147] To evaluate the garbled gate, an "evaluator" (e.g., the other of the two parties) can attempt to decrypt each row in the table using the labels corresponding to their respective inputs. For example, if the evaluator's inputs correspond to A=0, and B=1, the evaluator can attempt to decrypt each row in the garbled table 408 using the input labels X_0^A and X_1^B. Because only one of these rows corresponds to an output label encrypted with those input labels, the evaluator will (generally) only succeed at decrypting that respective row and receiving the corresponding output label (X_0^C).

[0148] For a garbled circuit comprising multiple gates (e.g., circuit 410), this process could be performed for each gate sequentially, eventually resulting in a set of labels corresponding to the output of the function evaluated by the garbled circuit. The garbler can then convert these labels back into their respective Boolean values, which can be interpreted as the output of the function. As an example, for a garbled circuit that determines a private set intersection, these Boolean values could correspond to the intersection set of the two input sets.

I. Bounding Exponential Growth

[0149] Detecting cliques in a graph is generally a difficult problem that can require a large amount of memory. The number of edges in a clique scales exponentially with the number of vertices. For example, a clique containing five vertices comprises ten edges, while a clique containing ten vertices has 45 edges. During SGA clique detection, the amount of data that has to be stored in order to track potential cliques can grow exponentially during successive rounds or iterations, which can be problematic. Because dynamically allocating memory can leak or otherwise reveal information about the structure of the graph being analyzed, it is sometime preferable to pre-allocate memory. However, the exact amount of memory to pre-allocate can be difficult or impossible to determine, because it can depend on the structure of the graph, which (in order to preserve privacy) cannot be directly determined by any participant in the multi-party computation. As a result, one potential memory allocation strategy is "worst-case allocation", where an amount of memory allocated in association with each vertex and edge is proportional to the most data that could potentially be accumulated during the clique detection process. However, because of the exponential nature of the amount of data stored to detect potential cliques, this worst case memory allocation may be infeasible.

[0150] As such, efficient memory management can improve the efficiency of clique detection methods. In embodiments of the present disclosure the multi-party computation network can perform some methods in order to bound the exponential memory requirement growth during clique detection, or otherwise limit the total amount of memory required to perform clique detection. While the growth may still be exponential, the "base" at which the memory requirement grows can be reduced, decreasing the total amount of memory required to perform clique detection. In broad terms, and as described in more detail further below, embodiments of the present disclosure can make use of an edge redirection process

(sometimes referred to as an arboricity process), to reduce the amount of data scattered and gathered during clique detection, and thereby reduce the total amount of data that is stored during clique detection. Other techniques involve generating and using virtual vertex tuples, in order to avoid large volumes of data being stored in association with any given vertex tuple, thereby reducing the "worst case" amount of memory allocated to a given vertex tuple. Other techniques, such as limiting the total amount of data stored in association with a given vertex tuple by using random deletion operations are also described.

IV. SETUP PHASE

[0151] The setup phase broadly comprises steps performed prior to the computation phase, e.g., steps that enable the multi-party computation network to perform parallel private graph processing on a secret-shared union tuple list. During the setup phase, a first party computer and second party computer can generate a secret-shared union tuple list comprising a plurality of secret-shared vertex tuples and a plurality of secret-shared edge tuples. In some embodiments, the secret-shared union tuple list can additionally comprise a plurality of duplicate edge tuples. The setup phase can also involve the generation of these duplicate edge tuples. As described in more detail further below, the use of duplicate edge tuples can improve the efficiency of clique detection by enabling both a Scatter step and a Gather step to be performed in a single parallelized linear scan.

[0152] The setup phase can additional involve an arboricity method used to redirect directed edges in the secret-shared union tuple list, prior to performing clique detection on the secret-shared union tuple list. This arboricity method (also referred to as an edge redirection method) can reduce the amount of data scattered and gathered during clique detection. As a result, the arboricity method can bound the memory growth of clique detection methods according to embodiments, decreasing the amount of memory that is needed to perform methods according to embodiments, and thereby improving their efficiency.

[0153] Additionally, the setup phase can comprise determining a first permutation and a second permutation, which can be used to obliviously shuffle the secret-shared union tuple list into a first ordering and a second ordering. As described further below, these orderings may enable the multi-party computation network to perform SGA clique detection using parallelized linear scans. Further, determining the first permutation and second permutation in the setup phase enables the use of oblivious shuffling operations instead of slow oblivious

sorting operations, thereby improving the speed and efficiency of methods according to embodiments.

A. Pre-union processing

[0154] Pre-union processing generally refers to steps in the setup phase that can be performed by the first party computer and the second party computer prior to generating the secret-shared union tuple list. The step of pre-union processing is described with reference to steps 702-706 in FIG. 7.

[0155] At step 702 the first party computer and second party computer can pre-process their respective data used to generate the secret-shared union tuple list. The first party computer and second party computer can remove any irrelevant information from this data, such as information that is not needed as part of clique detection. Removing this irrelevant information can decrease the size of the first party data and second party data, thereby decreasing the communication cost associated with generating the secret-shared union tuple list.

[0156] Additionally, the first party computer and second party computer can pre-process their respective data by removing any data elements (e.g., tuples) corresponding to vertices with zero in-degree or zero out-degree (e.g., vertices with no incoming directed edges and/or vertices with no outgoing directed edges). Such vertices cannot be part of cliques, and therefore do not need to be analyzed in a multi-party clique detection process. Further, the first party computer and second party computer can optionally pre-process their data by individually detecting any local cliques in their respective data. Each party can detect local cliques without needing to perform secure multi-party computation, and hence such cliques can be detected prior to generating the secret-shared union tuple list.

[0157] At step 704, the first party computer and second party computer can convert their respective data into a first tuple list and second tuple list respectively. The first tuple list and second tuple list can be combined in a tuple list unionization process (described further below) to generate the secret-shared union tuple list. The first party computer and second party computer can use any appropriate data processing technique in order to generate the first party tuple list and second party tuple list.

[0158] At optional step 706, the first party computer and second party computer can duplicate edge tuples in the first tuple list and second tuple list. The first party computer and

second party computer can use any appropriate memory management techniques to generate these duplicate edge tuples. For example, if the first tuple list and second tuple lists are stored by the first party computer and second party computer as vectors of data, the first party computer and second party computer can iterate through their respective tuple lists, and when they encounter an edge tuple, the first party computer and second party computer can copy the data associated with that edge tuple, then append that data (as a duplicate edge tuple) to the end of the vector storing the first tuple list or second tuple list. As an alternative, the multi-party computation network can generate a plurality of secret-shared duplicate edge tuples (e.g., at step 714) after receiving the secret-shared union tuple list from the first party computer and second party computer, hence step 706 is optional. As described in more detail further below, the duplicate edge tuples can be used in order to combine the Scatter and Gather steps of an SGA parallel private graph analysis method into a single Scatter-Gather step, thereby improving the speed and efficiency of embodiments of the present disclosure.

B. Tuple List Unionization

[0159] At step 708, the first party computer and second party computer can generate the secret-shared union tuple list. The first party computer and second party computer can generate the secret-shared union tuple list using a private set union process, which can be implemented (for example) using secret-shared multi-party computation. As an alternative, the first party computer and second party computer can implement the private set union using a private set union garbled circuit protocol. Such a private set union garbled circuit protocol can be configured to produce a plurality of secret-shared disjoint tuples based on the first tuple list and the second tuple list, then combine the plurality of secret shared disjoint tuples with either the first tuple list (e.g., according to the formula $L_1 \cup L_2 = L_2 \backslash L_1 + L_1$) or with the second tuple list (e.g., according to the formula $L_1 \cup L_2 = L_1 \backslash L_2 + L_2$), thereby generating the secret-shared union tuple list. This private union garbled circuit protocol can comprise a modified private set intersection garbled circuit protocol, similar to the circuit-PSI framework described in [8]. At the end of step 708, the first party computer and second party computer can learn secret shares of the secret-shared union tuple list, which they can then provide to the multi-party computation network, enabling the multi-party computation network to perform clique detection on the secret-shared union tuple list.

[0160] Step 708 and processes for generating the secret-shared union tuple list can be better understood with reference to FIGs. 5 and 6. FIG. 5 illustrates how a first party computer 502

and a second party computer 504 can use garbled circuits to generate a secret-shared union tuple list 520. This secret-shared union tuple list can be provided to a multi-party computation network 510, which can evaluate the secret shared union tuple list using multi-party clique detection process 512 in order to return a list of cliques 522 to the first party computer 502 and the second party computer 504. Alternatively or additionally, the list of cliques can be further processed (e.g., by the multi-party computation network or by another computer or device). Any results of this further processing can be transmitted to the first party computer 502 and the second party computer 504, instead of or in addition to the list of cliques 522.

[0161] The first party computer 502 and the second party computer 504 can perform a private union computation 524 in order to generate the secret-shared union tuple list 520. In some embodiments, the private union computation 524 can be implemented using a private set union implemented using a secret-shared multi-party computation. In other embodiments (as depicted in FIG. 6) the private union computation 524 can be implemented using a disjoint garbled circuit 606 and a union computation sub-process 608. The first party computer 502 and the second party computer 504 can use the disjoint garbled circuit 606, as well as the union computation process 608 to generate the secret-shared union tuple list 520. In FIG. 6, it is assumed that the disjoint garbled circuit 606 has been previously generated (e.g., by one of the parties acting as a garbler or by a trusted third party).

[0162] When a garbled circuit is used (e.g., as depicted in FIG. 6), the first party computer 502 and the second party computer 504 can use their respective tuple lists to generate data representative of the first tuple list 514 and data representative of the second tuple list 516. Such data can comprise first part tuple list labels and second party tuple list labels used as an input to the disjoint garbled circuit 606. This disjoint garbled circuit can be used to determine the disjoint of the two parties' lists (e.g., all the labels corresponding to the first tuple list that are not contained in the second tuple list, or all the labels corresponding to the second tuple list that are not contained in the first tuple list). In some embodiments, this disjoint garbled circuit 606 can comprise a modified private set intersection (PSI) garbled circuit protocol, which can be configured to produce a plurality of secret shared disjoint tuples (or labels) 518 based on the first party tuple list labels and the second party tuple list labels.

[0163] A PSI garbled circuit protocol can generally involve collecting all the labels corresponding to the input sets (e.g., the first party tuple list labels 514 and the second party tuple list labels 516) then using a garbled circuit to only reveal the labels that are present in both input sets, thereby determining the intersection of the two sets. A circuit-PSI protocol can be modified to reveal the labels that are present in one input set (e.g., the first party tuple list labels 514) and are not present in the other (e.g., the second party tuple list labels 516). In this way, the modified circuit-PSI protocol can be used to produce a disjoint garbled circuit 506, which can produce the disjoint of the first party tuple list labels 514 and the second party tuple list labels 516. These disjoint labels 518 (i.e., either $L_1 \backslash L_2$ or $L_2 \backslash L_1$) can be used to compute the secret-shared union tuple list 520, as described below.

[0164] The specific configuration of such a garbled circuit (e.g., the number and organization of gates) are not described herein. Generally, such garbled circuits require a large number of gates (e.g., on the order of tens or hundreds of thousands) which make them difficult to illustrate and describe with figures.

[0165] If a garbled circuit is used, as depicted in FIG. 6, the disjoint labels 518, as well as the first party tuple list labels or the second party tuple list labels can be used as the input to a union computation process 608. The union computation process 608 can generate the union $L_1 \cup L_2$ by securely implementing the formula $L_1 \cup L_2 = L_1 \backslash L_2 + L_2$ or $L_1 \cup L_2 = L_2 \backslash L_1 + L_1$ depending on how the disjoint labels 518 were generated. As such, the union computation process 608 can combine the plurality of secret-shared disjoint tuples (or disjoint labels 518) with either the first party tuple list labels 514 or the second party tuple list labels 516 to generate the secret-shared union tuple list 520 i.e., $L_1 \cup L_2$ 520.

[0166] In some embodiments, because the label sets $L_1 \backslash L_2$ and L_2 or $L_2 \backslash L_1$ and L_1 are disjoint (i.e., do not contain any common elements), the union computation process 608 can comprise a concatenation operation, e.g., by concatenating a label list corresponding to $L_1 \backslash L_2$ and the second party tuple list labels L_2 516, or by concatenating a label list corresponding to $L_2 \backslash L_1$ and the first party tuple list labels L_1 514.

[0167] The secret-shared union tuple list $L_1 \cup L_2$ 520 can be secret-shared among computers or devices in a multi-party computation network 510 (e.g., multi-party computation network 206 from FIG. 2), which can use the secret-shared union tuple list 520 to perform a multi-party clique detection process 512. This multi-party clique detection process 512 can comprise, for example, a three-party honest majority semi-honest

computation using the techniques described in [14], and can involve an SGA implementation of a clique detection process, such as a clique detection process based on depth-first or breadth-first search (described further below) in order to produce a plaintext list of cliques 522 (or another appropriate representation). This plaintext list of cliques 522 can be returned by the multi-party computation network to the first party computer 502 and the second party computer 504, or alternatively can be transmitted to another computer system, or alternatively can be further processed by the multi-party computation network 510.

[0168] It should be understood that the description of the systems of FIGs. 5 and 6 is intended as an example and is not intended to be limiting. There are a number of apparent variations on this system. For example, the union computation process 608 could be implemented using a garbled circuit, and the disjoint garbled circuit 606 and the union computation process 608 could then be implemented as a single garbled circuit system. As another example, the multi-party computation network 510 could evaluate the disjoint garbled circuit 606 and execute the union computation process 608 rather than the first party computer 502 and the second party computer 504 evaluating and executing these processes. FIGs. 5 and 6 implicitly assumes a system model similar to the system model depicted in FIG. 2 (as opposed to the model depicted in FIG. 3), although either model is valid.

[0169] The process of tuple duplication (step 706) and generating the union tuple list (step 708) is described in more detail with reference to FIG. 8, which visually summarizes the setup steps described above. Referring to FIG. 8, at step 808, the first party computer and the second party computer can represent the first party graph 802 and the second party graph 804 as a first party tuple list 810 and a second party tuple list 812 respectively. As described above, each tuple in the tuple lists may correspond to a graph element (e.g., a vertex or edge) in the corresponding graph. Each party can use any appropriate means to convert their respective graph into a representative tuple list. In some embodiments, each party may already represent their respective graphs as a tuple list. As such, step 808 may be optional.

[0170] At step 814, each party can duplicate each of the edge tuples in the first party tuple list 810 and second party tuple list 812, thereby generating one or more duplicate first edge tuples 816 and one or more duplicate second edge tuples 818. As described above, edge tuple duplication enables the "Scatter" and "Gather" steps of an SGA parallel private graph analysis method to be combined into a single "Scatter-Gather" step. This halves the number

of oblivious shuffling operations that need to be performed, and consequently improves the speed and efficiency of the iterative scatter-gather approach.

[0171] Afterwards, at step 820, the first party can combine the first party tuple list 810 and the one or more duplicated first edge tuples 816 to generate an expanded first party tuple list 822. Likewise, the second party can combine the second party tuple list 812 and the one or more duplicated second edge tuples 818 to generate an expanded second party tuple list 824.

[0172] At step 826, the first party and the second party can generate a secret-shared union tuple list 828 using a private set union protocol, which can be implemented either using secret-shared multi-party computation or using garbled circuits, such as the modified circuit-PSI protocol described above with reference to FIG. 6. The secret-shared union tuple list 828 can comprise a tuple list corresponding to the union graph 806 and include the duplicate edge tuples. The secret-shared union tuple list 828 can be generated according to a set equation such as $A \cup B = A \backslash B + B$, where A represents the expanded first party tuple list 822 and B represents the expanded second party tuple list 824. As an alternative, the secret-shared union tuple list 828 can be generated using the set equation $A \cup B = B \backslash A + A$, or any other appropriate set formulation.

[0173] It is not strictly necessary for the first party and second party to generate the duplicated first party edge tuples and the duplicated second party edge tuples prior to determining the union of their respective tuple lists. Instead, the edge tuples can be duplicated and included in the union tuple list after the union tuple list has been determined, using any appropriate private data duplication method.

[0174] Returning to FIG. 7, at step 710, the first party computer and second party computer can transmit the secret-shared union tuple list to the multi-party computation network, such that the multi-party computation network receives the secret-shared union tuple list. As described above, by this point, the secret-shared union tuple list can comprise a plurality of secret-shared vertex tuples representing a plurality of vertices in a union graph, a plurality of secret-shared edge tuples representing a plurality of edges in the union graph, and optionally a plurality of secret-shared duplicate edge tuples representing the plurality of edges in the union graph. This transmission can comprise, for example, the first party computer and second party computer transmitting the secret shares corresponding to the secret-shared union tuple list to the computer systems that make up the multi-party computation network (e.g., a first server computer, a second server computer, and a third server computer).

C. Edge Direction Reorientation

[0175] At step 712, the multi-party computation network can perform an arboricity method to redirect the edges in the secret-shared union graph (e.g., by modifying secret-shared edge tuples in the secret-shared union tuple list). This arboricity method can reduce the effective in-degree and out-degree of the vertices in the union graph, which can reduce the amount of data transmitted and stored during SGA clique detection, thereby improving the efficiency of methods according to embodiments. This arboricity method is summarized with reference to FIG. 19 and described in more detail with reference to FIG. 20.

[0176] Referring now to FIG. 19, which shows a union graph 1902 comprising 9 vertices (A-I) connected by directed edges. Generally, the arboricity method involves identifying vertices in the union graph 1902 that have a low degree and removing them, thereby producing an induced subgraph 1904. Any edges connected to those vertices can also be removed, which may change the degree of remaining vertices in the induced subgraph 1904. These low degree vertices can comprise, e.g., the bottom 10% (or any other appropriate percentage) of vertices based on degree. In FIG. 19, vertices A and G, both with only a single directed edge (and therefore a degree of one), have been removed from the union graph 1902 to produce the induced subgraph 1904. Vertices removed from the union graph 1902 can be recorded for later edge redirection.

[0177] The process of identifying, removing, and recording low degree vertices (and their associated edges) can be performed repeatedly until the induced subgraph comprises no vertices. In FIG. 19, vertices B and E are removed from induced subgraph 1904, producing induced subgraph 1906. Subsequently, vertices C and H were removed from induced subgraph 1906, producing induced subgraph 1908. Finally, vertices D, F, and I were removed from induced subgraph 1908, at which point the induced subgraph comprises no vertices.

[0178] The order in which the vertices were removed from the induced subgraph or the union graph 1902 can be used to rank the vertices. In general, high ranking vertices can comprise vertices removed later during the arboricity method, while vertices with lower rank can comprise vertices removed earlier during the arboricity method. In the case of a tie, i.e., if two vertices were removed from the union graph 1902 or the induced subgraph in the same iteration, the relative rank of these vertices can be determined based on their degree in the union graph 1902. For example, while vertices D, F, and I were all removed from induced

subgraph 1908 at the same time, vertex F has higher degree than vertices D or I in the union graph 1902, and consequently vertex F can be assigned a higher rank than vertices D or I.

[0179] After the vertices are ranked, the edges in the union graph can be redirected based on the ranks of the vertices. Each edge can be directed such that it points from a tuple of lower rank to a tuple of higher rank. The resulting updated union graph 1910 is shown in FIG. 19. As described above during SGA methods (including clique detection), data from vertices is scattered along edges and gathered and aggregated by other vertices. As a consequence, vertices with multiple in-going edges can generally accumulate data, while vertices with multiple outgoing edges distribute data to multiple other vertices. As such, typically data "accumulates" across the whole graph over multiple SGA rounds, as both the data stored in association with any given vertex, and the data distributed by any given vertex can increase on average. The rate at which this data accumulates is generally related to the structure of the graph. As an example, a vertex with both high in-degree and high out-degree generally accumulates larger amount of data, and transmits that data to other vertices, greatly increasing the total amount of data that needs to be stored in association with the graph during SGA processing. Further, structures such as loops can lead to data accumulation, particularly when those loops have additional edges directed into and out of the loop. For example, data in the F-C-D loop of union graph 1902 may not only accumulate in this loop, but may also grow due to data gathered into the loop on H-F edge, and may be scattered to other vertices via the C-B and F-I edges.

[0180] But by changing the direction of edges in the graph using arboricity methods, the total amount of data accumulated during SGA methods can be reduced. Loops can be eliminating by directing edges from vertices of higher lower rank to vertices of higher rank. Further, the updated union graph 1910 is effectively given an "overall direction," originating generally at vertices A and G toward vertex F. F, the vertex with the highest in-degree and by far the highest "effective in-degree" (e.g., based on the total number of vertices that directly or indirectly scatter data to F) has zero out-degree, meaning that while vertex F may accumulate large amounts of data during gather steps, it does not scatter any of that data to other vertices, greatly reducing the total amount of data scattered during SGA clique detection.

[0181] An arboricity or edge redirection method according to embodiments is described with reference to the flowchart of FIG. 20. Such an arboricity methods can be performed

obliviously on secret-shared tuples in the secret-shared union tuple list, using any appropriate oblivious multi-party computation techniques. Further, this arboricity method can be performed by a multi-party computation network prior to detecting one or more cliques in the secret-shared union tuple list by performing a parallel oblivious SGA clique detection process.

[0182] At step 2002, the multi-party computation network can optionally initialize an induced secret-shared union tuple list. This induced secret-shared union tuple list can correspond to an induced subgraph of the union graph. The multi-party computation network can initialize the induced secret-shared union tuple list by generating an induced secret-shared union tuple list that is initially equivalent to the secret-shared union tuple list (e.g., by copying the secret-shared union tuple list). As such, the induced subgraph can be initially equivalent to the union graph.

[0183] At step 2004, the multi-party computation network can perform an oblivious iterative process on the induced secret-shared union tuple list until the induced secret-shared union tuple list comprises zero secret-shared vertex tuples. This oblivious iterative process can comprise steps 2006 and 2008. At step 2006, the multi-party computation network can identify and record one or more low degree vertex tuples by evaluating a degree of each secret-shared vertex tuple in the induced secret-shared union tuple list. In broad terms, the multi-party computation network can (obliviously) determine an identifier of each secret-shared vertex tuple, then iterate through the list to count the number of secret-shared edge tuples that correspond to that identifier, thereby determining the degree of each secret-shared vertex tuple. The multi-party computation network can then identify the low degree vertex tuples by comparing the degrees of each secret-shared vertex tuple, e.g., identifying the bottom 25% of vertex tuples (in terms of degree) as low degree vertex tuples. Such low degree vertex tuples (or identifiers corresponding to those low degree vertex tuples) can be recorded, e.g., in an array or other data structure.

[0184] At step 2008, the multi-party computation network can update the induced secret-shared union tuple list by removing the one or more low degree vertex tuple and one or more associated edge tuples from the induced secret-shared union tuple lists. The one or more associated edge tuples can comprise secret-shared edge tuples that represent edges of the vertices represented by the one or more low degree vertex tuples.

[0185] This oblivious iterative process (steps 2004-2008) can be repeated on the induced secret-shared union tuple list, progressively recording and removing low degree vertex tuples from the induced secret-shared union tuple list until it comprises zero secret-shared vertex tuples. Afterwards, the order in which the low degree vertex tuples were removed from the induced secret-shared vertex tuple list can be used to redirect the edge tuples in the secret-shared union tuple list, as described below with reference to step 2010-2016.

[0186] At step 2010, the multi-party computation network can assign a rank to each secret-shared vertex tuple in the secret-shared union tuple list. As described above with reference to FIG. 19, such ranks can be based on the order in which that secret-shared vertex tuple was identified and recorded as a low degree vertex tuple, and further based on a degree of that secret-shared vertex tuple.

[0187] These determined ranks can be used to redirect the edge tuples in the secret-shared union tuple list. For each edge tuple in the secret-shared union tuple list, the multi-party computation network can perform steps 2012-2016 in order to modify the secret-shared edge tuples such that they represent a redirection of the edges in the secret-shared union tuple list. At step 2012, the multi-party computation network can determine a first rank of a first secret-shared vertex tuple associated with the secret-shared edge tuple. Likewise, at step 2014, the multi-party computation network can determine a second rank of a second secret-shared vertex tuple associated with that secret-shared edge tuple. The multi-party computation network can determine these ranks, e.g., by performing some form of oblivious search to identify the first secret-shared vertex tuple and the second-secret shared vertex tuple using, e.g., identifier data associated with the secret-shared edge tuple.

[0188] Afterwards, at step 2016, the multi-party computation network can modify the secret-shared edge tuple based on a comparison of the first rank and the second rank, such that the secret-shared edge tuple points from the secret-shared vertex tuple of lower rank toward the secret-shared vertex tuple of higher rank. In more detail, the multi-party computation network can determine if the first rank is greater than or equal to the second rank. If the first rank is greater than or equal to the second rank, the multi-party computation network can modify the secret-shared edge tuple such that it indicates that a directed edge in the union graph (corresponding to the secret-shared edge tuple) points from a second vertex in the union graph (corresponding to the second secret-shared vertex tuple) toward a first vertex in the union graph (corresponding to the first secret shared vertex tuple).

[0189] Alternatively, the multi-party computation network can determine if the second rank is greater than the first rank. If the second rank is greater than the first rank, the multi-party computation network can modify the secret-shared edge tuple such that it indicates that the directed edge in the union graph (corresponding to the secret-shared edge tuple) points from the first vertex in the union graph (corresponding to the first secret-shared vertex tuple) toward a second vertex in the union graph (corresponding to the second secret-shared vertex tuple). In this way, the multi-party computation network can redirect the secret-shared edge tuples in the secret-shared union tuple list.

[0190] Referring back to FIG. 7, at step 714, if the edge tuples were not duplicated at step 706, the multi-party computation network can generate a plurality of secret-shared duplicate edge tuples by duplicating the plurality of secret-shared edge tuples. The multi-party computation network can perform this step using any appropriate oblivious data duplication technique. Afterwards, the multi-party computation network can include the plurality of secret-shared duplicate edge tuples in the secret shared union tuple list, such that the secret-shared union tuple list comprises the plurality of secret-shared duplicate edge tuples representing the plurality of edges in the union graph, in addition to the plurality of secret-shared vertex tuples and the plurality of secret-shared edge tuples.

D. Sort and Determine Permutations

[0191] At step 716, the multi-party computation network can determine a first permutation corresponding to a first ordering and a second permutation corresponding to a second ordering. The first permutation can enable the multi-party computation network to order the secret-shared union tuple list according to the first ordering, e.g., prior to a combined Scatter-Gather step in the computation phase. Likewise, the second permutation can enable the multi-party computation network to order the secret-shared union tuple list according to the second ordering prior to the combined Scatter-Gather step. Determining the first permutation and the second permutation during the setup phase may enable the multi-party computation network to perform oblivious shuffling operations during clique detection, rather than slower oblivious sorting operations, thereby improving the speed and efficiency of methods according to embodiments.

[0192] Step 716 can be implemented using any appropriate oblivious sorting methods or techniques, such as those disclosed by Chida et al. [2]. Chida et al. implements oblivious radix sorting of secret-shared keys. Secret-shared tuples in the secret-shared union tuple list

can be sorted using a key-based sorting scheme. For a W-tuple (a vertex tuple) i, the value $i \parallel N$ can be used as its key. For a Y-tuple (a duplicate edge tuple) (i, j) the value $i \parallel (j + N)$ as its key, and for a G-tuple (an original edge tuple) (i, j) the value $j \parallel (N - i)$ can be used as its key. In each case, N refers to the total number of tuples in the tuple list. Note that the order of the G-tuples can be inverted (i.e., $N - i$). The tuple list can be sorted per their respective keys.

[0193] In the first ordering, each secret-shared vertex tuple of the plurality of secret-shared vertex tuples in the secret-shared union tuple list can be preceded by one or more corresponding secret-shared edge tuples of the plurality of secret-shared edge tuples, and followed by one or more corresponding secret-shared duplicate edge tuples of the plurality of secret-shared duplicate edge tuples. Using the "W, G, Y" tuple notation described above, a string representation of the first ordering is: (G*WY*)*, where * is the Kleene operator. This means, broadly, that the list can take the form of any number of original edge tuples (including zero) followed by a vertex tuple, followed by any number of duplicate edge tuples, and this pattern can be repeated any number of times. As an example, the secret-shared union tuple list in the first ordering 906 in FIG. 9 can be represented as the string "WGYWYYGWYGGW."

[0194] In the second ordering, each secret-shared vertex tuple of the plurality of secret-shared vertex tuples in the secret-shared union tuple list is preceded by one or more corresponding secret-shared duplicate edge tuples of the plurality of secret-shared duplicate edge tuples and followed by one or more corresponding secret-shared edge tuples of the plurality of secret-shared edge tuples. Using the "W, G, Y" tuple notation, a string representation of the second ordering is (Y*WG*)*. This means that the secret-shared union tuple list can take the form of any number of duplicate edge tuples (including zero) followed by a vertex tuple, followed by any number of original edge tuples, and this pattern can be repeated any number of times. As an example, the secret-shared union tuple list in the second ordering 912 in FIG. 9 can be represented by the string "WGYWYYGWGYYG."

[0195] In other words, in the first permutation and the second permutation, the positions of the secret-shared edge tuples and the secret-shared duplicate edge tuples in the secret-shared union tuple list may be swapped. As briefly mentioned further above, the use of duplicate edge tuples, the first permutation, and the second permutation may improve the speed and

efficiency of embodiments of the present disclosure by enabling the combination of a Scatter step and a Gather step into a single linear scan Scatter-Gather step.

[0196] Generally, when a Scatter step is performed using a linear scan, it may be preferable for vertex tuples to be followed by their corresponding outgoing edge tuples (i.e., edge tuples corresponding to outgoing edges of corresponding vertices), as "scattered" data from that vertex tuple can be immediately applied to the relevant edge tuples. Likewise, when a Gather step is performed using a linear scan, it may be preferable for vertex tuples to be preceded by corresponding incoming edge tuples (i.e., edge tuples corresponding to incoming edges of corresponding vertices), as "gathered" data from those incoming edge tuples can be immediately applied to relevant vertex tuples. As such, in a non-duplicated secret-shared union tuple list (i.e., a secret-shared union tuple list that does not comprise secret-shared duplicate edge tuples), it is possible to perform the SGA clique detection by sorting (or shuffling) the secret-shared union tuple list such that outgoing edge tuples follow vertex tuples, performing the Scatter step, then sorting (or shuffling) the secret-shared union tuple list such that incoming edge tuples precede vertex tuples, and performing the Gather step. Afterwards the Apply step can be performed, and this process can be repeated, alternating between these two orderings until SGA clique detection has been completed. In such a case, three linear scans are performed corresponding to the Scatter, Gather, and Apply steps.

[0197] By contrast, in iterative SGA methods according to embodiment using duplicate edge tuples, the Scatter and Gather steps can be performed in a single Scatter-Gather linear scan. In the first ordering, original edge tuples precede vertex tuples, and those vertex tuples are followed by duplicate edge tuples. As such, data can be gathered from original edge tuples to vertex tuples and scattered from vertex tuples to duplicate edge tuples. Afterwards, an Apply step linear scan can be performed to execute the apply function on each vertex tuple. The secret-shared union tuple list can be obliviously shuffled to the second ordering, in which duplicate edge tuples precede vertex tuples, and those vertex tuples are followed by original edge tuples. As such, data can be gathered from duplicate edge tuples to vertex tuples and scattered from vertex tuples to original edge tuples. Afterwards an Apply step linear scan can be performed to execute the apply function on each vertex tuple. This process can be repeated, alternating between the two orderings, until SGA clique detection has been completed. This technique effectively enables both a Scatter step and a Gather step to be performed in a single linear scan, effectively reducing the total number of linear scans that

need to be performed in each iteration from three to two, thereby improving the efficiency of methods according to embodiments.

[0198] Two methods of determining the first permutation and second permutation are described below with reference to FIGs. 9 and 21. Referring to FIG. 9, at step 904, a multi-party computation network can obliviously sort the secret-shared union tuple list 902 into the first ordering 906. From the first ordering 906, the multi-party computation network can determine a first permutation 908 that can be used to obliviously shuffle the secret-shared union tuple list 902 into the first ordering during the computation phase. The first permutation 908 may be stored by the multi-party computation network in secret-shared form. Likewise, at step 910, the multi-party computation network can obliviously sort the secret-shared union tuple list 902 into the second ordering 912. From the second ordering, the multi-party computation network can determine a second permutation 914 that can be used to obliviously shuffle the secret-shared union tuple list 902 into the second ordering 912 during the computation phase. The second permutation 914 may be stored by the multi-party computation network in secret-shared form.

[0199] However, because oblivious sorting operations may be computationally expensive, it may be preferable to limit the number of oblivious sorting operations performed by the multi-party computation network. As such, an alternative way of determining the first permutation and second permutation using a single oblivious sorting operation (rather than two) and secure random shuffling operations is describe with reference to FIG. 21. As part of determining the first and second permutation, the multi-party computation network can obliviously sort the secret-shared union tuple list 2102 into the first ordering. The multi-party computation network can then perform a secure random shuffle on the secret-shared union tuple list while the secret-shared union tuple list is in the first ordering (i.e., 2104), thereby generating a randomly shuffled secret-shared union tuple list 2106, a shuffle permutation Π, and an inverse shuffle permutation Π^{-1}. The shuffle permutation can be used to perform the secure random shuffle, and the inverse shuffle permutation can be used to "reverse" the secure random shuffle, i.e., produce the secret-shared union tuple list in the first ordering 2104 given the randomly shuffled secret-shared union tuple list 2106. The multi-party computation network can use secure random shuffling methods such as those described in [12].

[0200] Afterwards, the multi-party computation network can reveal a plurality of edge tuple identities corresponding to the plurality of secret-shared edge tuples, and a plurality of duplicate edge tuple identities corresponding to the plurality of secret-shared duplicate edge tuples. The multi-party computation network can reveal these edge tuple identities and duplicate edge tuple identities to a plurality of computers in the multi-party computation network (e.g., the first server computer 208, the second server computer 210, and the third server computer 212 from FIG. 2). The computers in the multi-party computation network can perform an oblivious process to reveal the plurality of edge tuple identities and the plurality of duplicate edge tuple identities. Revealing the edge tuple identities and duplicate edge tuple identities can effectively reveal the locations of the edge tuples and duplicate edge tuples within the randomly shuffled secret-shared union tuple list with revealed edges 2108.

[0201] The multi-party computation network can determine an intermediate permutation ϕ based on the plurality of edge tuple identities and the plurality of duplicate edge tuple identities. Applying the intermediate permutation ϕ to the randomly shuffled secret-shared union tuple list 2106 can swap a plurality of edge tuples positions and a plurality of corresponding duplicate edge tuple positions in the randomly shuffled secret-shared union tuple list. In other words, the intermediate permutation can effectively swap the locations of each secret-shared edge tuple and a corresponding secret-shared duplicate edge tuple. As an example, the multi-party computation network can iterate through the secret-shared union tuple list with revealed edges 2108, and identify each edge tuple and duplicate edge tuple using the plurality of edge tuple identities and the plurality of duplicate edge tuple identities, then determine an intermediate permutation ϕ that swaps each secret-shared edge tuple and its corresponding secret-shared duplicate edge tuple. Because the intermediate permutation ϕ directly swaps pairs of corresponding secret-shared edge tuple and secret-shared duplicate edge tuples, the intermediate permutation ϕ may be equivalent to its inverse intermediate permutation ϕ^{-1}

[0202] By applying the shuffle permutation, the intermediate permutation, and the inverse shuffle permutation in sequence, it is possible to reorder the secret-shared union tuple list in the first ordering 2104 into the secret-shared union tuple list in the second ordering 2112. Likewise, by applying the inverse shuffle permutation, the intermediate permutation, and the shuffle permutation in sequence, it is possible to reorder the secret-shared union tuple list in the second ordering 2112 into the secret-shared union tuple list in the first ordering 2104.

[0203] As such, the multi-party computation network can then determine the first permutation 2114 based on the shuffle permutation, the inverse shuffle permutation, and the intermediate permutation, e.g., by sequentially combining the inverse shuffle permutation, the intermediate permutation, and the shuffle permutation. The multi-party computation network can likewise determine the second permutation based on the shuffle permutation, the inverse shuffle permutation and the intermediate permutation, e.g., by sequentially combining the shuffle permutation, the intermediate permutation, and the inverse shuffle permutation.

E. Prepare Processor Operations using Tuple States

[0204] Returning to FIG. 7, at step 718, the multi-party computation network can determine a plurality of tuple states corresponding to the secret-shared tuples in the secret-shared union tuple list. These tuple states can be used to generate operational instructions for a pool of processors (associated with the multi-party computation network) that can process the secret-shared union tuple list during the computation phase. As a result of determining these tuple states prior to the computation phase, the processors do not need to do so during each operation of the computation phase, thereby reducing the number of operations performed and increasing the overall speed and efficiency of the multi-party graph analysis method.

[0205] A tuple state may indicate whether a corresponding secret-shared tuple comprises a vertex tuple (sometimes referred to as a W tuple), an original edge tuple (G tuple) or a duplicate edge tuple (Y tuple). Typically, when the tuples are in secret-shared form, the tuple state of a particular tuple cannot be readily determined without performing some operation or protocol (e.g., a garbled circuit protocol) to determine these tuple states. This has some implications for the multi-party computation process, particularly the combined Scatter-Gather step, as described in Section E below.

[0206] In general terms, during the computation phase, the secret-shared union tuples in the secret-shared union tuple list can be divided among a pool of processors, such that each processor receives, e.g., two secret-shared tuples. Each processor then performs some operation based on the tuples they received as part of performing the parallel private clique detection process. Such operations can depend on the context on the tuple state (e.g., W, G, or Y) of those tuples, as described below in the section on aggregation trees. For example, if a processor receives a W tuple and a G tuple, the receiving processor may scatter the data from the W tuple to the G tuple as part of the combined Scatter-Gather step. As an

alternative example, if a processor receives a Y tuple and a W tuple, then the receiving processor can gather the data from the Y tuple to the W tuple. Dividing the secret-shared tuples among the pool of processors can enable methods according to embodiments performed in parallel, decreasing the total execution time.

[0207] However, it is not necessary for the processors to determine the state of the tuples during the computation phase. Because the first ordering and second ordering are known (via the first permutation and second permutation), each operation that each processor performs during the computation phase can be determined in advance based on these orderings and the tuple states, provided that the processors are assigned inputs according to a defined pattern (e.g., a first processor receives the first two secret-shared tuples in the union tuple list, a second processor receives the second two tuples in the union tuple list, etc.).

[0208] Broadly, instead of each processor receiving its respective tuples, determining the tuple state corresponding to those two tuples (during the computation phase), then performing an operation based on the determined tuple states, each processor can instead be assigned an operation in advance, based on the tuple states determined during the setup phase at step 720. This both reduces the number of operations performed in the computation phase and prevents any information about the underlying union graph from being leaked during the computation phase.

V. COMPUTATION PHASE

[0209] Having completed the setup phase, the multi-party computation network can perform the computation phase. During this phase, the multi-party computation network can detect one or more cliques in the secret-shared union tuple list by performing a multi-party computation on the secret-shared union tuple list. In some embodiments, the multi-party computation network can use a three-party honest majority semi-honest multi-party computation (MPC) protocol. This is in contrast to some conventional oblivious graph evaluation methods, which often use a two-party garbled circuit MPC protocol. The use of a three-party honest majority semi-honest MPC protocol enables the use of an efficient three-party honest majority, semi-honest oblivious shuffling protocol, which are typically more computationally efficient than conventional oblivious sorting protocols.

[0210] An exemplary method corresponding to the computation phase is summarized with reference to FIG. 24. At step 2402, the multi-party computation network can receive a secret-

shared union tuple list from a first party computer and a second party computer (e.g., as described above with respect to the setup phase). As described above, the secret-shared union tuple list can be generated using a first tuple list corresponding to the first party computer and a second tuple list corresponding to the second party computer. As described above, the secret-shared union tuple list can comprise a plurality of secret-shared union tuples corresponding to a representation of a union graph. As described above, the secret-shared union tuple list can comprise a plurality of secret-shared vertex tuples representing a plurality of vertices in the union graph and a plurality of secret-shared edge tuples representing a plurality of edges in the union graph.

[0211] At step 2404, the multi-party computation network can detect one or more cliques in the secret-shared union tuple list by performing a multi-party computation on the secret-shared union tuple list. As described above, the one or more cliques can comprise one or more complete subgraphs of the union graph. Each complete subgraph of the one or more complete subgraphs can comprise a plurality of subgraph vertex tuples corresponding to a plurality of subgraph vertices in the union graph (e.g., vertices within the complete subgraphs), and a plurality of subgraph edge tuples corresponding to a plurality of subgraph edges in the union graph (e.g., edges within the complete subgraphs). Each subgraph vertex of the plurality of subgraph vertices can be connected to each other subgraph vertex of the plurality of subgraph vertices via the plurality of subgraph edges.

[0212] At step 2406, the multi-party computation network can provide an output corresponding to the one or more cliques in response to detecting the one or more cliques to the first party computer, the second party computer, or an additional computer system.

[0213] In some embodiments, the multi-party computation network can detect the one or more cliques in the secret-shared union tuple list by performing a private Scatter-Gather-Apply implementation of a depth-first or breadth-first search method until a terminating condition has been achieved. In these embodiments, the computation phase can involve performing this private Scatter-Gather-Apply (SGA) implementation of the depth-first or breadth-first search method. As such, an SGA implementation of a breadth-first search clique detection method is summarized below with reference to FIG. 10, and the individual steps (e.g., the Scatter, Gather, and Apply steps) an exemplary breadth-first search SGA clique detection method is described with reference to FIG. 18.

[0214] Referring to FIG. 10, at step 1004, the multi-party computation network can obliviously shuffle the secret-shared union tuple list into the first ordering 1006 using the first permutation. Afterwards, at step 1008, the multi-party computation network can perform a combined Scatter-Gather step on the secret-shared union tuple list, then perform an Apply step on the secret-shared union tuple list. The Scatter-Gather step may be based on the ordering of the secret-shared union tuple list (e.g., the first ordering versus the second ordering), and is generally illustrated by the curved arrows in FIG. 10. In step 1008, each vertex tuples 1, 4, and 8 can scatter data to subsequent duplicate edge tuples 2, 5, 6, and 9. Likewise, each vertex tuple can gather data from preceding original edge tuples 3, 7, 10, and 11, completing both the Scatter and Gather step in a single linear scan of the secret-shared union tuple list. Afterwards, a real Apply function can be applied to each vertex tuple (e.g., checking if the data at any vertex tuple is indicative of a clique), and a dummy Apply function can be applied to each edge tuple, thereby preserving obliviousness.

[0215] At step 1010, the multi-party computation network can obliviously shuffle the secret-shared union tuple list into the second ordering 1012 using the second permutation. Afterwards, at step 1014, the multi-party computation network can perform a combined Scatter-Gather step on the secret-shared union tuple list, then perform an Apply step on the secret-shared union tuple list. Like in step 1008, the Scatter-Gather step at step 1014 may be based on the ordering of the secret-shared union tuple list and is generally illustrated by the curved arrows in FIG. 10. In step 1014, vertex tuples 1, 4, and 8 can scatter data to subsequent original edge tuples (now tuples 2, 5, 6, and 9 due to the second ordering) and gather data from preceding duplicate edge tuples (now tuples 3, 7, 10, and 11), completing both the Scatter and Gather steps in a single linear scan of the secret-shared union tuple list. Afterwards, a real Apply function can be applied to each vertex tuple, and a dummy Apply function can be applied to each edge tuple, thereby preserving obliviousness.

[0216] Steps 1005, 1008, 1010, and 1014 can be repeated until a terminating condition has been achieved. The multi-party computation network can check if the terminating condition has been achieved during the Apply step, or at any other appropriate time. The terminating condition can comprise, for example, the detection of one or more cliques, or the expiration of a predetermined number of clique detection rounds.

[0217] A Scatter step, Gather step, and Apply step for an SGA clique detection method are described in more detail with reference to FIG. 18. Generally, in order to perform SGA

clique detection, the multi-party computation network can store a vertex tuple potential clique list (*v.Ts* in FIG. 18) in association with each vertex tuple, and an edge tuple potential clique list (*e.Ts* in FIG. 18) in association with each edge tuple. These potential clique lists can be used to track potential cliques in the union graph represented by the secret-shared union tuple list. By iteratively adding vertices to these potential cliques, verifying the validity of the potential cliques, and removing invalid cliques from the potential clique list, the multi-party computation network can detect one or more cliques in the secret-shared union tuple list. The multi-party computation network can also store a first list of vertex tuple lists (*v.Ss* in FIG. 18) in association with each secret-shared vertex tuple, and a second list of vertex tuple lists (*e.Ss* in FIG. 18) in association with each secret-shared edge tuple (and each secret-shared duplicate edge tuple). These lists of vertex tuple lists can be used to store identifiers corresponding to vertex tuples, which can correspond to outgoing neighbors of a particular vertex tuple *v*, in order to evaluate the validity of the potential cliques in the potential clique lists. As such, the multi-party computation network can thereby store a plurality of vertex tuple potential clique lists (*v.Ts*), a plurality of first lists of vertex tuple lists (*v.Ss*), a plurality of edge tuple potential clique lists (*e.Ts*), and a plurality of second lists of vertex tuple lists (*e.Ss*).

[0218] As such, in some embodiments, prior to performing the Scatter step, Gather step, and Apply steps described below, the multi-party computation network can allocate a vertex tuple memory unit for each secret-shared vertex tuple of the plurality of secret-shared vertex tuples. The multi-party computation network can use this vertex tuple memory unit to store the vertex tuple potential clique list and the first list of vertex tuple lists corresponding to the secret-shared vertex tuple. The multi-party computation network can thereby allocate a plurality of vertex tuple memory units. Likewise, the multi-party computation network can allocate an edge tuple memory unit for each secret-shared edge tuple of the plurality of secret-shared edge tuples. The multi-party computation network can use this edge tuple memory unit to store the edge tuple potential clique list and the second list of vertex tuple lists corresponding to the secret-shared edge tuple. The multi-party computation network can thereby allocate a plurality of edge tuple memory units. Such vertex tuple memory units and edge tuple memory units can be allocated in a shared memory available to the pool of processors in the multi-party computation network.

[0219] Method 5 of FIG. 18 describes a Scatter step corresponding to an SGA clique detection method. The multi-party computation network can perform this Scatter step by

scattering, for each secret-shared vertex tuple of the plurality of secret-shared vertex tuples, the vertex tuple potential clique list and the first list of vertex tuple lists to a plurality of secret-shared outgoing edge tuples corresponding to the secret-shared vertex tuple, thereby updating the plurality of edge tuple potential clique lists and the plurality of second lists of vertex tuple lists.

[0220] In other words, each vertex v can scatter a vertex tuple potential clique list $v.Ts$ and a first list of vertex tuple lists $v.Ss$ to each outgoing edge e. The multi-party computation network can update an edge tuple potential clique list $e.Ts$ and a second list of vertex tuple lists $e.Ss$ based on the vertex tuple potential clique list $v.Ts$ and the first list of vertex tuple lists $v.Ss$, effectively by directly assigning the data from $v.Ts$ to $e.Ts$ and from $v.Ss$ to $e.Ss$. Initially, $v.Ts$ can comprise a single potential clique T comprising the vertex v, and $v.Ss$ can comprise a single list of vertex tuples S comprising the outgoing neighbors of v. However, in each successive round, the data stored in $v.Ts$ and $v.Ss$ may accumulate as the vertex v gathers data from incoming edges (as described in more detail below).

[0221] Afterwards, the multi-party computation network can perform a Gather step by updating the plurality of vertex tuple potential clique lists and the plurality of first lists of vertex tuple lists based on the plurality of edge tuple potential clique lists and the plurality of second lists of vertex tuple lists. A Gather step according to some methods according to embodiments is depicted in Method 7 of FIG. 18. In this Gather step, each vertex v can initially clear its vertex tuple potential clique list $v.Ts$ and its first list of vertex tuple lists $v.Ss$. Afterwards, the vertex can aggregate one or more edge tuple potential clique lists $e.Ts$ received from one or more incoming edges e by determining the union of those one or more edge tuple potential clique lists $e.Ts$. Likewise, the vertex v can aggregate one or more second lists of vertex tuple lists $e.Ss$ by constructing the union of those second lists of vertex tuple lists $e.Ss$.

[0222] Initially, each edge tuple potential clique list $e.Ts$ can comprise a single potential clique T comprising a vertex v' that was scattered to a corresponding edge in the previous Scatter step. Likewise, each second list of vertex tuple lists $e.Ss$ can comprise a single list of vertex tuples S corresponding to the outgoing neighbors of the vertex v'. However, in each successive round, as vertices gather data from incoming edges, the potential clique lists and lists of vertex tuple lists associated with each vertex may accumulate and be scattered to

outgoing edges, resulting in each vertex receiving more data form incoming edges in the subsequent Gather step.

[0223] Afterwards, the multi-party computation network can perform an Apply step by performing a series of steps for each secret-shared vertex tuple v of the plurality of vertex tuples, and for each vertex tuple potential clique T in the vertex tuple potential clique list $v.Ts$ corresponding to that secret-shared vertex tuple v. An Apply step according to some embodiments is depicted in Method 8 of FIG. 18. In the Apply step, the multi-party computation network can update the vertex tuple potential clique T based on an identity of the secret-shared vertex tuple. As depicted in FIG. 18, the multi-party computation network can update the vertex tuple potential clique T by adding the secret-shared vertex tuple v to the vertex tuple potential clique T, i.e., by determining a union $T = T \cup \{v\}$.

[0224] The multi-party computation network can additionally update a corresponding first list of vertex tuples S based on one or more secret-shared outgoing neighbor vertex tuples of the secret-shared vertex tuple v. As depicted in FIG. 18, the multi-party computation network can determine an intersection of the corresponding first list of vertex tuples S and the outgoing neighbor vertex tuples of the secret-shared vertex tuple v, and reassign the first list of vertex tuples S to this intersection, i.e., $S = S \cap Outgoing\_Neighbors(v)$.

[0225] Further, during the Apply step, the multi-party computation network can evaluate the corresponding first list of vertex tuples S. If the corresponding first list of vertex tuples S contains no secret-shared vertex tuples (i.e., $S = \emptyset$), then the multi-party computation network can remove the vertex tuple potential clique T and the corresponding first list of vertex tuples S from the vertex tuple potential clique list $v.Ts$ and the first list of vertex tuple lists $v.Ss$. In broad terms, the first list of vertex tuples S can contain outgoing neighbors of vertex tuples in the vertex tuple potential clique T. If the outgoing neighbors of vertex tuple v are not in the first list of vertex tuples (i.e., the intersection of S and the outgoing neighbors is the empty set $\emptyset$), then some number of edges are missing from the potential clique T, meaning that potential clique T is not a valid clique, and can therefore be removed from the vertex tuple potential clique list $v.Ts$.

[0226] The multi-party computation network can repeat the iterative SGA process (depicted in Method 8 of FIG. 18) until a terminating condition has been met. As depicted in Method 8, the terminating condition can comprise a predetermined number of iterations k,

which may be achieved once the multi-party computation network has performed the predetermined number of iterations *k*.

[0227] The multi-party computation network can detect one or more cliques in response to achieving the terminating condition by evaluating (for each secret-shared vertex tuple *v*) each vertex tuple potential clique *T* and/or each first vertex tuple list *S* in the vertex tuple potential clique list *v.Ts* and the first list of vertex tuple lists *v.Ss* corresponding to that secret-shared vertex tuple *v*. Because invalid cliques are removed from the vertex tuple potential clique list *v.Ts* during the Apply step, any cliques remaining in the vertex tuple potential clique list *v.Ts* after completion of the iterative SGA process can comprise valid cliques, and therefore the multi-party computation can detect the one or more cliques based on these vertex tuple potential cliques. In some embodiments, the multi-party computation network can output a result comprising a list of the one or more cliques, e.g., to a first party computer and a second party computer, or alternatively can provide the output to another computer system, or can further process the list of the one or more cliques.

[0228] During SGA processing, the Scatter, Gather, and Apply steps can risk leaking information about the structure of the union graph represented by the secret-shared union tuple list, as some steps (e.g., the Apply step) are only performed on vertices. As such, if a read or write operation is made to a shared memory element, a participant in the multi-party computation network can potentially determine that the shared memory element corresponds to a vertex tuple. However, some embodiments of the present disclosure can use "dummy" operations, such as a "dummy Apply step" in order to preserve obliviousness. Rather than performing the Apply step on each secret-shared vertex tuple in the secret-shared union tuple list, the multi-party computation network can perform an Apply step on each secret-shared vertex tuple and a dummy Apply step on each secret-shared edge tuple and each secret-shared duplicate edge tuple. Consequently, memory accesses or other operations performed during the Apply step (real or "dummy") do not reveal any information about whether those operations are performed on a vertex tuple or an edge tuple, and consequently do not leak any information about the tuples in the secret-shared union tuple list.

A. Memory Management Optimizations

1. Memory Management Problem

[0229] In general, during SGA processes, the amount of data gathered at a particular vertex depends on the number of incoming edges connected to that vertex, and further depends on the structure of the subgraphs connected to that vertex by those incoming edges. For example, a vertex may be connected to only two other vertices by incoming edges, however those vertices may have hundreds of incoming edges connecting them to other vertices. As such, even though the vertex only has two incoming edges, it may accumulate large amounts of data during rounds of an iterative SGA process.

[0230] It can be difficult to predict how much data will be accumulated by any given vertex during an SGA process, as the amount of data accumulated is greatly dependent on the structure of the entire graph. This is made even more difficult during oblivious SGA processing (e.g., oblivious clique detection according to embodiments), as the graph is represented in a secret-shared form. Further, during MPC oblivious processing, care should be taken regarding accesses to shared memory (e.g., read or write operations), as such operations can reveal information about the underlying structure of the graph, which may violate the privacy of the parties participating in the oblivious MPC. As such, dynamic memory allocation risks revealing information about the underlying structure of the graph. If a memory element (e.g., an array) associated with a particular vertex is frequently being resized to accommodate greater amounts of accumulated data, then that may reveal to the participants that the corresponding vertex has a large number of incoming edges. Likewise, if a memory element corresponding to a particular vertex remains small during the entire SGA process, it can reveal that the corresponding vertex is gathering small amounts of data during SGA rounds, and therefore may have few (or zero) incoming edges.

[0231] As a result, "worst-case" memory allocation can be used to avoid dynamically resizing memory elements during oblivious SGA processing, and thereby preserve obliviousness. In short, large vertex tuple memory units and edge tuple memory units can be allocated to each vertex and edge in a graph, such that even under the "worst possible" structure of that graph (i.e., some hypothetical structure that maximizes the amount of data aggregated at each vertex), no secret-shared tuple will accumulate more data than can be stored in its corresponding memory element. However, the amount of memory that needs to be allocated to each tuple in the worst-case scenario may be unreasonably large, and it may

be infeasible to allocate that much memory for each secret-shared tuple, particularly in a graph comprising a large number of vertices and edges.

2. Random Deletion

[0232] Some embodiments use a random deletion strategy in order to reduce the necessary size of vertex tuple memory units and edge tuple memory units allocated to the secret-shared vertex tuples and secret-shared edge tuples. During oblivious SGA clique detection, the multi-party computation network can evaluate each of the vertex tuple memory units of the plurality of vertex tuple memory units. The multi-party computation network can identify any exceeded vertex tuple memory units of the plurality of vertex tuple memory units. An exceeded vertex tuple memory unit can contain more data than a threshold amount of data corresponding to the exceeded vertex tuple memory unit. Notably, this threshold amount of data does not need to be equal to the total amount of memory allocated to a vertex tuple memory unit. For example, if 1 MB of memory is allocated to a vertex tuple memory unit, that vertex tuple memory unit could be an exceeded vertex tuple memory unit if, e.g., 0.95 MB of memory is in use. That is, an exceeded vertex tuple memory unit can comprise a vertex tuple memory unit that is at risk of exceeding the total amount of memory allocated to that vertex tuple memory unit. In this example, the threshold amount of data corresponding to the exceeded vertex tuple memory unit could comprise 0.95 MB of data. Thus, the exceeded vertex tuple memory unit can contains more data than a threshold amount of data corresponding to the exceeded vertex tuple memory unit.

[0233] The multi-party computation network can randomly delete one or more vertex tuple potential cliques and one or more corresponding first vertex tuple lists from an exceeded vertex tuple potential clique list and an exceeded first list of vertex tuple lists corresponding to the exceeded vertex tuple memory unit. The "exceeded" vertex tuple potential clique list and the "exceeded" first list of vertex tuple lists may themselves not be "exceeded", instead, the adjective "exceeded" is intended to denote their correspondence to the exceeded vertex tuple memory units. Randomly deleting one or more vertex tuple potential cliques and one or more corresponding first vertex tuple lists can free up memory in the exceeded vertex tuple memory unit, enabling SGA processing to be performed using lower memory allocation volumes than what would typically be used in worst-case memory allocation scenarios.

[0234] The multi-party computation network can likewise evaluate each of the edge tuple memory units of the plurality of edge tuple memory units. The multi-party computation

network can identify any exceeded edge tuple memory units of the plurality of edge tuple memory units. An exceeded edge tuple memory unit can contain more data than a threshold amount of data corresponding to the exceeded edge tuple memory unit. Notably, this threshold amount of data does not need to be equivalent to the total amount of memory allocated to an edge tuple memory unit. For example, if 100 GB of memory is allocated to an edge tuple memory unit, that edge tuple memory unit could be an exceeded edge tuple memory unit if, e.g., 85 GB of memory is in use. That is, an exceeded edge tuple memory unit can comprise an edge tuple memory unit that is at risk of exceeding the total amount of memory allocated to that edge tuple memory unit. In this example, the threshold amount of data corresponding to the exceeded edge tuple memory unit could comprise 85 GB of data. Thus, the exceeded edge tuple memory unit can contain more data than a threshold amount of data corresponding to the exceeded edge tuple memory unit.

[0235] The multi-party computation network can randomly delete one or more edge tuple potential cliques and one or more corresponding second vertex tuple lists from an exceeded edge tuple potential clique list and an exceeded second list of vertex tuple lists corresponding to the exceeded edge tuple memory unit. As with the exceeded vertex tuple potential clique list and the exceeded first list of vertex tuple lists, the "exceeded" edge tuple potential clique list and the "exceeded" second list of vertex tuple lists may themselves not be "exceeded", instead, the adjective "exceeded" is intended to denote their correspondence to the exceeded edge tuple memory units. Randomly deleting one or more edge tuple potential cliques and one or more corresponding second vertex tuple lists can free up memory in the exceeded edge tuple memory unit, enabling SGA processing to be performed using lowed memory allocation volumes than what would typically be used in worst-case memory allocation scenarios.

[0236] Randomly deleting vertex tuple potential clique, edge tuple potential cliques, first vertex lists, and second vertex lists in this manner can cause the multi-party computation network to probabilistically fail to detect some cliques in the union graph, and therefore creates a tradeoff between memory allocation and clique detection rates.

3. Virtual Vertex Tuples

[0237] An alternative memory management technique according to embodiments involves the generation and use of virtual vertex tuples. In broad terms, virtual vertices can be introduced to a graph to reduce the indegree of each vertex in the graph. For each vertex in a

graph, virtual vertices can be generated based on the number of incoming edges that are connected to that vertex. These virtual vertices can receive data from the incoming edges in place of their corresponding vertex. As a result, each (non-virtual vertex) can effectively have an in-degree of zero, as any data it typically would have gathered is instead "gathered" to a virtual vertex. Further, each virtual vertex effectively has an in-degree of one, as that virtual vertex was generated to correspond to a particular incoming edge. While a graph comprising virtual vertex tuples may collectively require the same total amount of memory as a corresponding graph without virtual vertex tuples during SGA processing, because the amount of data stored in association with any vertex tuple or virtual vertex tuple is predictable, considerably less memory can be allocated in association with each vertex tuple and virtual vertex tuple than in the worst-cast memory allocation described above. As a result, using virtual vertex tuples, as in embodiments of the present disclosure, greatly improves the memory efficiency of methods according to embodiments, while still maintaining obliviousness.

[0238] A method for generating virtual vertex tuples is summarized with reference to FIG. 22, and is described in more detail with reference to the flowchart of FIG. 23. FIG. 22 shows a subgraph 2202 comprising vertices 2204, 2206, 2208, 2214, 2216. Vertex 2204 has two incoming edges 2210 and 2212, and two outgoing edges 2218 and 2220. As such, vertex 2204 may gather data from both vertices 2206 and 2208 and scatter that data to vertices 2214 and 2216. As such, vertex 2204 and subgraph 2202 may contribute to the accumulation of data in the graph in successive SGA rounds.

[0239] Instead of gathering from edges 2210 and 2212 to vertex 2204, the multi-party computation network can generate two virtual vertices 2224 and 2226, which can receive scattered data from vertices 2206 and 2208. Vertex 2204 and virtual vertices 2224 and 2226 collectively gather and store the same amount of data as 2204 alone in absence of the virtual vertices 2224 and 2226 (e.g., as depicted in subgraph 2202). However, when virtual vertices are used, the amount of data stored in association with vertex 2204 alone does not accumulate. The vertex 2204 and virtual vertices 2224 and 2226 can collectively scatter their data to vertices that would normally receive data from vertex 2204 alone, i.e., vertices 2224 and 2230. If necessary, new edges, such as edges 2260 and 2262 can be generated for this purpose.

[0240] This general process can be performed for each vertex with incoming edges in the subgraph and in the graph as a whole. As depicted in subgraph 2232, virtual vertices 2234 and 2236 can be generated corresponding to vertex 2214 and 2216 respectively. Further, the graph can be simplified to remove any redundant edges or vertices. For example, because virtual vertices 2224 and 2226 scatter data directly to other virtual vertices (i.e., virtual vertices 2234 and 2236), virtual vertices 2224 and 2226 can be eliminated, and vertices 2206 and 2208 can be directed connected to virtual vertices 2240-2246, e.g., via edges 2248-2254 as depicted in subgraph 2238.

[0241] FIG. 23 shows a flowchart of a method for using virtual vertex tuples according to some embodiments, in these embodiments, the multi-party computation network can use the method depicted in FIG. 23 in order to update the plurality of vertex tuple potential clique lists and the plurality of first lists of vertex tuple lists based on the plurality of edge tuple potential clique lists and the plurality of second lists of vertex tuple list, e.g., as part of the Gather step described above.

[0242] At step 2302, the multi-party computation network can generate one or more secret-shared virtual vertex tuples for each secret-shared vertex tuple of the plurality of secret-shared vertex tuples in the union tuple list, corresponding to that secret-shared vertex tuple, and further corresponding to one or more secret-shared incoming edge tuples. In other words, for a given vertex tuple, a virtual vertex tuple can be generated corresponding to each edge tuple representing an incoming edge of that vertex.

[0243] Each secret-shared virtual vertex tuple of the one or more secret-shared virtual vertex tuples can be associated with a first virtual vertex tuple potential clique list and a virtual list of vertex tuple lists, similarly to how each secret-shared vertex tuple can be associated with a vertex tuple potential clique list and a first list of vertex tuple lists. Each virtual vertex tuple potential clique list and each virtual list of vertex tuple lists can be equivalent to a corresponding incoming edge tuple potential clique list and a corresponding second list of vertex tuple lists respectively. In effect, the data from the incoming edge tuples (e.g., the incoming edge tuple potential clique lists and the second lists of vertex tuple lists) can be copied the secret-shared virtual vertex tuples.

[0244] At step 2304, the multi-party computation network can, for each secret-shared virtual vertex tuple, include the one or more secret-shared virtual vertex tuples in the secret-shared union tuple list and in the plurality of secret-shared vertex tuples. In this way, the

multi-party computation network can include a plurality of secret-shared virtual vertex tuples (corresponding to the plurality of secret-shared vertex tuples collectively) in the secret-shared union tuple list and in the plurality of secret-shared vertex tuples.

[0245] At step 2306, the multi-party computation network can remove one or more secret-shared edge tuples from the secret-shared union tuple list. These one or more secret-shared edge tuples can correspond to redundant edges. Further, at step 2308, the multi-party computation network can generate and include one or more new secret-shared edge tuples in the secret-shared union tuple list based on the plurality of secret-shared vertex tuples and the plurality of secret-shared virtual vertex tuples. For example, as depicted in FIG. 22, redundant edges 2210 and 2212 were removed from subgraph 2222 when virtual vertices 2224 and 2226 were added, and additional edges 2256 and 2258 were added to connect vertices 2206 and 2208 to virtual vertices 2228 and 2230. Likewise, new edges 2260 and 2262 were added to connect virtual vertices 2223 and 2230 to vertices 2214 and 2216. Similarly, the multi-party computation network can remove and add secret-shared edge tuples in the secret-shared union tuple list based on the plurality of secret-shared vertex tuples and the plurality of secret-shared virtual vertex tuples. This removing and adding edges can also be performed as part of simplifying the secret-shared union tuple list, as described above with reference to subgraph 2238.

B. SGA Using Aggregation Trees

[0246] The multi-party computation network can perform some clique detection methods according to embodiments using "aggregation trees," as described below. In general, aggregation trees are model for aggregating or otherwise processing data, which can be used to parallelize data processing operations, including linear scan based operations. As such, the multi-party computation network can use aggregation trees to parallelize the process of oblivious clique detection, as described in more detail below. In some embodiments, the multi-party computation network can perform an "upward pass" and "downward pass" in order to perform a combined Scatter-Gather step on a secret-shared union tuple list using an aggregation tree.

1. Summary of Upward Pass

[0247] FIG. 12 shows a visualization of an upward pass performed on a secret-shared union tuple list 1202 using an aggregation tree model. Each set of two consecutive tuples can

comprise inputs that are "aggregated" in some manner to produce an output, which can comprise a unit of data that can be referred to as a "cell" or an "output cell." In FIG. 12, six cells 1204 are shown, each with two persistent storage elements and two ephemeral storage elements, shown in FIG. 12 as four subdivisions in each cell 1204. The manner in which data from input tuples is aggregated into cells can be in accordance with a step of "propagation rules," which are defined in the upward pass propagation rules table presented further below. By following these propagation rules, the multi-party computation network can implement an SGA clique detection method using aggregation trees, upward passes, and downward passes. As one example, tuples 7 and 8 (counting from the top of FIG. 12) comprise an original edge tuple corresponding to the edge between vertices 2 and 3 and a vertex tuple corresponding to vertex 3. As such, the output cell 1212 corresponding to tuples 7 and 8 can be generated such that it reflects a gather operation, in which vertex tuple 8 gathers data from edge tuple 7. As such, one persistent storage element in output cell 1212 comprises a vertex tuple, in which the data corresponding to the input vertex tuple and edge tuple has been combined according to a gather function $\oplus$. This is consistent with the second propagation rule in the upward pass propagation rules table below (i.e., the rule presented at Reference ID (REF ID) 104). Such a gather function can comprise the clique detection gather function described above, e.g., a function used to determine the unions of potential clique lists and lists of vertex tuple lists.

[0248] The upward pass can be repeated three more times until the final cell output 1210 is produced, which can comprise the root cell of an implicitly constructed binary tree. However, instead of using the secret-shared tuple list 1202 as inputs to produce a set of output cells 1204, the multi-party computation network can use the set of output cells 1204 as inputs to produce a new set of output cells 1206, which can then be used as inputs to produce cell output 1208, and so on, until the final cell output 1210 is produce. This cell output 1210 can be used as an input to a "downward pass" (described in more detail below) which can result in the construction of an updated secret-shared union tuple list. This updated secret-shared union tuple list can effectively comprise a secret-shared union tuple list on which one iteration of a combined Scatter-Gather step has been performed. In this way, using aggregation trees in this manner may have an equivalent result to performing a combined Scatter-Gather step using other techniques.

[0249] One advantage of the use of aggregation trees is they enable the multi-party computation network to better parallelize SGA clique detection, as each aggregation or

propagation operation is performed on pairs of inputs (e.g., secret-shared tuples or cells). This parallel processing may be faster (e.g., have low time complexity) than serial processing, and as such, the use of aggregation trees may improve the speed and efficiency of methods according to embodiments. For the exemplary secret-shared union tuple list 1202 of FIG. 12, six processors in a pool of processors could each be assigned a sequential pair of secret-shared tuples and could generate the resulting six output cells 1204 in parallel. Afterwards, three processors from the pool of processors could each be assigned a sequential pair of output cells 1204, to produce three output cells 1206 in parallel, and so on until the final output cell 1210 is produced. As a consequence of this parallelism, the time-complexity of an upward pass is log N, where N is the number of secret-shared union tuples in the secret-shared union tuple list 1202.

2. Upward Propagation Rules Table

[0250] For reference, the following table details some of the rules that describe how a processor can process two inputs (either tuples or cells) during the upward pass phase. The upward pass table uses the "W, G, Y" notation described above. In the table below w can refer to a vertex tuple or "white" tuple, g can refer to an original edge tuple or "gray" tuple, and y can refer to a duplicate edge tuple, or "yellow" tuple. These rules are listed by reference ID, and generally describe the result and storage result (i.e., the output) corresponding to the state of the inputs and the data stored in the inputs persistent and ephemeral storage. Generally speaking, when inputs are organized in sequence, the "left input" refers to the input that is located earlier in the sequence (e.g., input n) and the "right input" refers to the input that is located later in the sequence (e.g., input $n + 1$).

REF ID	Left Input	Right Input	Result	Storage
102	g_1	g_2	$g_1 \oplus g_2$	
104	g	w	$g \oplus w$	
106	g_1	$w \parallel g_2$	$(g_1 \oplus w) \parallel g_2$	
108	g	$w \parallel y$	$g \oplus w$	$(; y)$
110	w	g	$w \parallel g$	
112	w_1	w_2	$w_1 \parallel w_2$	
114	w_1	$w_2 \parallel g$	$w_1 \parallel g$	$(; w_2)$
116	w	$y \parallel g$	$w \parallel g$	$(; y)$
118	w_1	$w_2 \parallel y$	$w_1 \parallel w_2$	$(; y)$
120	w	y	w	$(; y)$
122	y	g	$y \parallel g$	
124	y	w	$y \parallel w$	
126	y	$w \parallel g$	$y \parallel g$	$(; w)$

128	y_1	$w \parallel y_2$	$y_1 \parallel w$	$(; y_2)$
130	y_1	y_2	y_1	
132	$w \parallel g_1$	g_2	$w \parallel (g_1 \oplus g_2)$	
134	$w_1 \parallel g$	w_2	$w_1 \parallel (g \oplus w_2)$	
136	$w_1 \parallel g_1$	$w_2 \parallel g_2$	$w_1 \parallel g_2$	$(; g_1 \oplus w_2)$
138	$w_1 \parallel g_1$	$w_2 \parallel y_1$	$w_1 \parallel (g_1 \oplus w_2)$	$(; y_1)$
140	$w_1 \parallel g_1$	$w_2 \parallel w_3$	$w_1 \parallel w_3$	$(; g_1 \oplus w_2)$
142	$w \parallel y$	g	$w \parallel g$	$(y;)$
144	$w_1 \parallel y$	w_2	$w_1 \parallel w_2$	$(y;)$
146	$w_1 \parallel y$	$w_2 \parallel g$	$w_1 \parallel g_2$	$(y; w_1)$
148	$w_1 \parallel y_1$	$w_2 \parallel y_2$	$w_1 \parallel w_2$	$(y_1; y_2)$
150	$w \parallel y_1$	y_2	w	$(y_1; y_2)$
152	$y \parallel g_1$	g_2	$y \parallel (g_1 \oplus g_2)$	
154	$y \parallel g$	w	$y \parallel (g \oplus w)$	
156	$y \parallel g_1$	$w \parallel g_2$	$y \parallel g_2$	$(; g_1 \oplus w)$
158	$y_1 \parallel g_1$	$w \parallel y_2$	$y \parallel (g_1 \oplus w)$	$(; y)$
160	$y \parallel w$	g	$y \parallel (w \oplus g)$	
162	$y \parallel w_1$	w_2	$y \parallel w_2$	$(w_1;)$
164	$y \parallel w_1$	$w_2 \parallel g_2$	$y \parallel g_2$	$(w_1; w_2)$
166	$y_1 \parallel w_1$	$w_2 \parallel y_2$	$y_1 \parallel w_2$	$(w_1; y_2)$
168	$w_1 \parallel w_2$	g	$w_1 \parallel g$	$(w_2;)$
170	$w_1 \parallel w_2$	w_3	$w_1 \parallel w_3$	$(w_2;)$
172	$w_1 \parallel w_2$	$w_3 \parallel g_2$	$w_1 \parallel g_2$	$(w_2; w_3)$
174	$w_1 \parallel w_2$	$w_3 \parallel y$	$w_1 \parallel w_3$	$(w_2; y)$
178	$w_1 \parallel w_2$	y	$w_1 \parallel w_2$	$(;)$

3. Summary of Downward Pass

[0251] FIG. 13 shows a visualization of a downward pass performed on a "root cell" 1302 that can be generated by during an upward pass, such as the exemplary upward pass visualized in FIG. 12. In contrast to the upward pass, in which two inputs (tuples or cells) are aggregated to produce a single output (a cell), in the downward pass, a single cell input can be processed to produce two outputs, which can either comprise tuples or cells. As depicted in FIG. 13, root cell 1302 can be processed to produce output cells 1304 and 1306. Output cells 1304 can be processed to produce output cells 1308 and 1310. Output cell 1308 can be processed to produce output cells 1312 and 1314, Output cell 1310 can be processed to produce output cells 1316 and 1318, and output cell 1306 can be processed to produce output cells 1320 and 1322. The six output cells 1312-1322 can each be processed to collectively produce an updated plurality of secret-shared union tuples 1324. This updated plurality of secret-shared union tuples can effectively comprise a plurality of secret-shared union tuples on which an iteration of a combined Scatter-Gather step was performed.

[0252] The manner in which inputs are used to produce outputs can be consistent with a set of downward pass propagation rules, presented in the downward pass propagation rules table below. By following these rules, the multi-party computation network can implement an SGA clique detection method using aggregation trees, upward passes, and downward passes. As one example, output cell 1312 can be processed to produce an updated vertex tuple and an updated duplicate edge tuple (i.e., the first and second tuples in the updated plurality of secret-shared union tuples 1324). This is consistent with the fourth propagation rule in the downward pass propagation rules table (i.e., the rule at REF ID 206) presented below.

[0253] As described above with reference to the upward pass, an advantage of the use of aggregation trees is that they enable the multi-party computation network to better parallelize clique detection, as cell processing operations can be divided among a pool of processors, enabling the downward pass to be performed by multiple processors in parallel. This parallel processing may be faster (e.g., have lower time complexity) than serial processing, and as such, the use of aggregation trees may improve the speed and efficiency of methods according to embodiments. For the root cell 1302 of FIG. 13, in a first "processing round," a single processor can process this root cell 1302 to produce output cells 1304 and 1306. In a second round, two processors can process output cells 1304 and 1306 to produce output cells 1308, 1310, 1320, and 1322. In a third round, four processors can process output cells 1308, 1310, 1320, and 1322 to produce output cells 1312-1318 and four updated secret-shared union tuples (e.g., the bottom four updated secret-shared union tuples in the updated secret-shared union tuple list 1324). In a fourth round, four processors can process output cells 1312-1318 to produce eight updated secret-shared union tuples (e.g., the top eight updated secret-shared union tuples in the updated secret-shared union tuple list 1324), thereby completing the downward pass. As a consequence of this parallelism, the time-complexity of a downward pass is log N, where N is the number of updated secret-shared union tuples in the updated secret-shared union tuple list 1324. As such, the total time complexity of performing both an upward pass and a downward pass is 2 log N.

4. Downward Propagation Rules Table

[0254] For reference, the following table details some of the rules that describe how a processor can process two inputs (either tuples or cells) during the downward pass phase. Like the upward pass table, the downward pass table can use the "W, G, Y" notation described above. These rules are listed by reference ID, and generally describe the result and

storage result (i.e., the outputs) corresponding to the state of the inputs and the data stored in the inputs persistent and ephemeral storage. Generally speaking, when inputs are organized in sequence, the "left input" refers to the input that is located earlier in the sequence (e.g., input n) and the "right input" refers to the input that is located later in the sequence (e.g., input $n + 1$).

REF ID	Parent Input	Storage	Left Output	Right Output
200	g_1			
202	w	$(y_1; y_2)$	w	w
204	w_1	$(;)$	w	
206	w_1	$(; y)$	w	y
208	w_1	$(; w_2)$	w_1	w_2
210	$w_1 \parallel w_2$	$(w_3; \star)$	$w_1 \parallel w_3$	w_2
212	$w_1 \parallel w_2$	$(\star; w_3)$	w_1	$w_3 \parallel w_2$
214	$w_1 \parallel w_2$	$\star$	w_1	w_2
216	$w \parallel g$	$(;)$	w	
218	$w_1 \parallel g$	$(; w_2)$	w_1	w_2
220	$w \parallel g$	$(y;)$	w	
222	$w_1 \parallel g$	$(w_2;)$	$w_1 \parallel w_2$	
224	$w_1 \parallel g$	$(w_2; w_3)$	$w_1 \parallel w_2$	w_3

5. Computation Phase Implemented Using Aggregation Trees

[0255] Having summarized aggregation trees, an upward pass, and a downward pass, an implementation of the computation phase is now described with reference to the flowchart of FIG. 11. As described above, a combined Scatter-Gather step can comprise an upward pass and a downward pass, which be used to enable parallel execution of this combined scatter-gather step. FIG. 11 shows a flowchart of an exemplary method of performing a parallel private graph method according to embodiments. The parallel private graph method can comprise a clique detection method, such as a clique detection method based on breadth-first search (or depth-first search). In FIG. 11, it is assumed that the secret-shared union tuple list is already in one of the two orderings (i.e., the first ordering or the second ordering) described above.

[0256] The method can comprise two primary steps, a combined Scatter-Gather step 1102 and an Apply step 1120. The Scatter-Gather step 1102 can comprise two sub-steps: an upward pass step 1104 and a downward pass step 1112. Performing these two sub-steps in sequence can result in updating the data associated with each tuple in the secret-shared union tuple list in accordance with an iteration of both the Scatter step and Gather step.

[0257] The upward pass step 1104 generally comprises three steps 1106-1110. Initially, the multi-party computation network can define a "set of inputs" as a plurality of secret-shared union tuples in the secret-shared union tuple list. At step 1106, the set of inputs can be divided among a plurality of processors associated with the multi-party computation network. Generally, each processor can be tasked with processing its respective inputs, enabling the secret-shared union tuple list to be processed in parallel. In some cases each processor can be assigned two inputs, as this may achieve faster processing speed. However, in many practical use cases, the multi-party computation network may not have access to a large enough pool of processors. As such, each processor may be assigned more than two inputs.

[0258] At step 1108, the multi-party computation network, using the pool of processors, can process the set of inputs using a clique detection method and based on the current ordering of the secret-shared union tuple list, thereby producing a first set of outputs. The first set of outputs may comprise less outputs than the set of inputs comprises inputs, and these outputs may comprise data values referred to as cells. In some embodiments, the set of outputs may comprise roughly half as many cells as inputs (cells or tuples) in the set of inputs. After producing the set of outputs, the multi-party computation network can then define the set of inputs (for a subsequent round of upward pass processing) as the set of outputs, enabling the upward pass to be repeated until the set of inputs comprises a single input, which can comprise the root cell of the implicitly constructed binary tree.

[0259] A cell, mentioned above, generally comprises the data element used to represent an internal node of the binary tree. A cell can comprise two persistent storage elements and two ephemeral storage elements. For a given processor, its inputs and the current list ordering can influence the data stored in these persistent and ephemeral storage elements. For example, as described above with reference to FIG. 10, in the first ordering, vertex tuples may scatter to duplicate edge tuples and gather from original edge tuples. Hence, if a processor is assigned a vertex tuple and a duplicate edge tuple, it may generate a cell output consistent with a scatter operation from the vertex tuple to the duplicate edge tuple. However, in the second ordering, vertex tuples may scatter to original edge tuples and gather from duplicate edge tuples. Hence, if a processor is assigned a vertex tuple and a duplicate edge tuple, it may generate a cell output consistent with a gather operation from the duplicate edge tuple to the vertex tuple.

[0260] While processing their respective inputs during the upward pass, the processors may adhere to a set of propagation rules, defined in the upward pass propagation rules table above. The propagation rules table can indicate the corresponding cell output for a given set of inputs during the upward pass.

[0261] At step 1108, the multi-party computation network can use the plurality of processors to determine if the upward pass has been completed. As described above, the general goal of the upward pass is to construct a root cell, which can be used to reconstruct updated secret-shared union tuples. In each iteration of the upward pass, because the number of output cells is less than the number of inputs (tuples or cells), and because the set of outputs can define the set of inputs in the following iteration, eventually the set of inputs can comprise a single input (the root cell), at which point the upward pass has been completed. If the upward pass has been completed, the method can proceed to the downward pass (step 1112) otherwise the method can return to step 1106 and the upward pass can be repeated until the set of inputs comprises the root cell.

[0262] After completing the upward pass, the multi-party computation network can perform a downward pass comprising steps 1112-1118. The downward pass generally comprises the construction of a plurality of updated secret-shared union tuples using the root cell generated during the upward pass phase. These updated secret-shared union tuples can comprise tuples with data updated in a manner consistent with Scatter and Gather operations.

[0263] At step 1114, the multi-party computation network can divide a set of inputs among the plurality of processors. Initially, this set of inputs can comprise a single input, the root cell generated as a result of the upward pass.

[0264] At step 1116, the multi-party computation network, using the plurality of processors, can process the set of inputs using a clique detection method and based on the current ordering of the secret-shared union tuple list, thereby producing a second set of outputs. The set of outputs may comprise cells or tuples and may comprise more outputs than the set of inputs comprises inputs. In some embodiments, the set of outputs may comprise roughly twice as many outputs as inputs in the set of inputs. The multi-party computation network can then define the set of inputs as the set of outputs, enabling the downward pass to be repeated until the set of inputs comprises an updated plurality of union tuples in the secret-shared union tuple list.

[0265] At step 1118, the multi-party computation network can use the pool of processors to determine if the downward pass has been completed. As described above, the general goal of the downward pass is to construct the updated plurality of union tuples using the root cell generated during the upward pass. In each iteration of the downward pass, because the number of outputs is less than the number of inputs, and because the set of outputs can define the set of inputs in the following iteration, the set of inputs can grow until it is the size of the original set of inputs during the upward pass, at which point the set of inputs comprises the updated plurality of secret-shared union tuples, at which point the downward pass has been completed. If the downward pass has been completed, the method can proceed to Apply step 1120, otherwise the method can return to step 1114 and the downward pass can be repeated until the set of inputs comprises the updated plurality of union tuples.

[0266] After the upward pass and downward pass have been completed, at the Apply step 1120, the multi-party computation network can divide the secret-shared union tuple list (now comprising an updated plurality of union tuples as a result of the upward pass and downward pass) among the plurality of processors, then apply an apply function to each tuple of the updated plurality of union tuples. In some embodiments, the apply function can evaluate and update a plurality of potential clique lists and a plurality of lists of vertex tuple lists associated with the plurality of secret-shared union tuples, as described in more detail further above.

[0267] At step 1122, the multi-party computation network can determine if a terminating condition has been achieved. If the terminating condition has been achieved, the flowchart can proceed to step 1126. In this step, the multi-party computation network can detect one or more cliques in the union graph by evaluating the plurality of potential clique lists and the plurality of lists of vertex tuple lists (as described above), thereby producing a result of the parallel private clique detection method. In some embodiments, the result of the parallel private clique detection method can comprise a list of the one or more cliques corresponding to the union graph, or can comprise other data, such as data derived from the list of the one or more cliques corresponding to the union graph.

[0268] Subsequently, at step 1128, the multi-party computation network can output the result of the clique detection method to the first party computer and the second party computer. For example, the processors in the multi-party computation network can release their respective secret shares (corresponding to detected cliques) to the first party computer

and the second party computer, or alternatively to another computer system or process. Alternatively, the multi-party computation network can further process or analyze the list of cliques depending, for example, on the particular clique detection application or context. For clique detection in telecommunications network optimization, the multi-party computation network could determine an alternative, more efficient subgraph structure. If the terminating condition has not been achieved, the flowchart can proceed to step 1124, the secret-shared union tuple list can be obliviously shuffled, and the computation phase can be repeated until the terminating condition has been achieved.

[0269] In some embodiments, the terminating condition check can be integrated into the apply function applied to the secret-shared vertex tuples at the Apply step 1120. The terminating condition may depend on the particular clique detection techniques or methods being performed (e.g., depth-first vs breadth-first) being performed. For example, the terminating condition can involve checking if a predetermined number of clique detection rounds have been performed.

C. Oblivious Shuffling

[0270] At step 1124, if the terminating condition has not been achieved, the secret-shared union tuple list can be obliviously shuffled and the iterative Scatter-Gather-Apply approach (e.g., the combined Scatter-Gather step and the Apply step) can be repeated until the terminating condition has been achieved. If the secret-shared union tuple list is in the first ordering, the multi-party computation network can obliviously shuffle the secret-shared union tuple list into the second ordering using the second permutation. Otherwise, if the secret-shared union tuple list is in the second ordering, the multi-party computation network can obliviously shuffle the secret-shared union tuple list into the first ordering using the first permutation. The multi-party computation network can use any appropriate oblivious shuffling protocol, such as the oblivious shuffling protocol described by Chida et al. Afterwards, the flowchart can return to the beginning of the Scatter-Gather step 1102 and repeat until the terminating condition has been achieved.

[0271] As described above, rather than using oblivious sorting to convert the secret-shared union tuple list from the first ordering to the second ordering, the multi-party computation network can obliviously shuffle the secret-shared union tuple list using the first permutation and the second permutation determined during the setup phase. This can be more efficient, as the time complexity of oblivious shuffling is lower than the time complexity of oblivious

sorting. FIG. 14 shows an exemplary parallelized shuffling protocol that can be used in some methods according to embodiments.

[0272] Broadly, during some methods according to embodiments, the multi-party computation network can assign tuples to a collection (or "pool") of processors (e.g., processors 1402, 1404, 1406, and 1408). In FIG. 14, each processor is shown assigned a set of four tuples. The tuples are generally organized, from left to right, in an order consistent with the secret-shared union tuple list in some ordering (e.g., the first ordering). Consequently, when the secret-shared union tuple list is obliviously sorted into another ordering (e.g., the second ordering), each processor is expected, generally, to possess or otherwise be assigned a different set of tuples. As such, the oblivious shuffling process may involve processors 1402-1408 communicating and transmitting tuples to one another, so that each processor is assigned secret-shared tuples consistent with the current ordering of the secret-shared union tuple list.

[0273] At step 1410, each processor can compute the destination of all tuples. These destinations can be based on a processor ordering. For example, if processor 1402 processes the first four tuples, and processor 1404 possesses the second four tuples, if processor 1402 has a tuple (e.g., "tuple 2") which will be "shuffled" to become the 7^{th} tuple ("tuple 7"), processor 1402 can determine that the destination of tuple is processor 1404. This determination can be made while the tuples are in secret-shared form, preventing any information from leaking during the shuffling protocol. Next, at step 1412, the processors can transmit the secret share tuples to their respective destinations in batches of messages. Afterwards, at step 1414, each processor can locally reorder their respective tuples based on the shuffling order (i.e., permutation) completing the shuffling process.

VI. EVALUATION AND PERFORMANCE ANALYSIS

[0274] This section describes a variety of metrics that can be used to evaluate parallel private clique detection methods according to embodiments. These metrics can also be used to compare methods according to embodiments to convention clique detection techniques. One such metric is the "total work" metric, which broadly refers to an estimate of the total number of operations performed by computers or other systems (e.g., the multi-party computation network), when performing parallel-private clique detection methods. Total work can be evaluated based on the approximate number of operations relative to the number of inputs N (e.g., the number of vertices and edges in a union graph). "Big O" notation is

often used as an approximation or substitution for total work. An "$O(N)$" computation scales linearly with the number of inputs, such that a computer system is expected to perform an approximately constant number of operations per input. By contrast, an "$O(N^2)$" computation scales quadratically, and a computer system is expected to perform on the order of N operations per input, which may take considerably more total work to complete than an $O(N)$ computation.

[0275] Total work can be measured using a variety of means. For example, total work can be measured by evaluating the total number of operations performed on data elements (e.g., secret-shared union tuples) in a shared memory element (such as a shared memory array). As another example, if a garbled circuit is used to implement a parallel oblivious clique detection method, total work can be measured based on the size of the garbled circuit used to implement that method.

[0276] The total work involved in parallel oblivious operations (such as parallel oblivious clique detection) is often greater than similar non-parallel, insecure (i.e., non-oblivious) operations for a variety of reasons. As one example, the cost of parallelism can increase the total amount of work performed during the execution of the parallel oblivious operations. As another example, the total work may increase due to the use of oblivious processing techniques. Oblivious processing operations typically require more work than similar insecure processing operations, because extra operations are performed in order to maintain obliviousness.

[0277] Another performance metric is "parallel time" or "parallel runtime", which can be measured as the total time required to execute parallel processes assuming that a sufficient number of processors are available to "fully parallelize" such processes. If a parallel oblivious process (such as a parallel oblivious clique detection process) is implemented using a garbled circuit, the parallel runtime can be equivalent to the circuit's depth. In order to evaluate the effectiveness of methods according to embodiments, the parallel runtime of such methods can be compared against an optimal parallel insecure (non-oblivious) baseline implementation.

[0278] A third performance metric is communication cost, which can be measured as the total number of pairwise interactions between processors from among P processors participating in the parallel oblivious process. Communication costs can also be measured

using the total amount of data transferred between such processors during the execution of parallel oblivious processes.

[0279] As summarized in the table further below, performance metrics such as total work and parallel time can be used to compare methods according to embodiments against conventional SGA methods and insecure methods. In embodiments that use aggregation trees, as described above in Section E, 2 log N steps are sufficient to complete a single Scatter-Gather step. Because the size of an aggregation tree is at most $2 \times 2N$ and at each internal node a constant number of operations are perform, the total work of a single iteration is $O(N)$. As such, the parallel time associated with some clique detection methods according to embodiments is $O(\log N)$ and the underlying hidden constants are small.

[0280] Methods according to embodiments can be generalized to a case where the number of processors $P < N$. For any given P, each processor can be assigned a sub-tree of operations that the processor can evaluate serially. The subtree can be rooted at an internal aggregation tree node that is at a tree "level" where the total number of nodes at that level is less than P. For example, for a secret-shared union tuple list comprising 12 secret-shared union tuples (e.g., four vertex tuples, four edge tuples, and four duplicate edge tuples), if $P = 3$, then each processor can be assigned a distinct subtree with four tuple leaf nodes, rooted at the third level of the aggregation subtree. In such a case, each processor could process four of the twelve tuples indicated at leaf level of the aggregation tree.

[0281] As a result, with P processors, the parallel running time increases to $2\,N/P + 2\log(P)$, with total work still equal to $O(N)$. For the purpose of comparison, when using insecure clique detection methods with total graph size $M = |V| + |E|$ and P processors, the parallel time is $O(M/P + \log P)$. The total work using conventional methods under these conditions is $O(P \log P + M)$. As such, with conventional methods there is an inherent tradeoff between the parallel running time and total work. Embodiments of the present disclosure eliminate this trade-off and scale effectively with the total number of processors.

[0282] Further, in a distributed memory setting where memory is separated among the processors, conceptual "shared memory" can be implemented using inter-processor communication. One advantage of some embodiments of the present disclosure is that in a given step of parallel computation, each processor at most communicates with one other processor. This is an improvement over conventional methods, in which each processor may need to communicate with up to Log P processors. Reducing the number of communications

increases the speed and efficiency at which cliques can be detected in secret-shared union tuple lists.

[0283] The table below compares the total work and parallel time for a variety of methods, including insecure methods, conventional SGA methods, and methods according to embodiments of the present disclosure. As demonstrated by the table, the total work of methods according to embodiments of the present disclosure is a factor of log N lower than conventional SGA methods, and is equal to insecure methods, demonstrating the efficiency of methods according to embodiments.

Methods	**Total Work**	**Parallel Time**
Insecure	$O(N)$	$O(\log d)$
Conventional SGA	$O(N \log N)$	$O(\log N)$
Embodiments	$O(N)$	$O(\log N)$

VII. COMPUTER SYSTEM

[0284] Any of the computer systems mentioned herein may utilize any suitable number of subsystems. Examples of such subsystems are shown in FIG. 15 in computer system 1500. In some embodiments, a computer system includes a single computer apparatus, where the subsystems can be the components of the computer apparatus. In other embodiments, a computer system can include multiple computer apparatuses (each being a subsystem) with internal components. A computer system can include desktop and laptop computers, tablets, mobile phones and other mobile devices.

[0285] The subsystems shown in FIG. 15 are interconnected via a system bus 1512. Additional subsystems such as a printer 1508, keyboard 1518, storage device(s) 1520, monitor 1524 (e.g., a display screen, such as an LED), which is coupled to display adapter 1514, and others are shown. Peripherals and input/output (I/O) devices, which couple to I/O controller 1502, can be connected to the computer system by any number of means known in the art such as input/output (I/O) port 1516 (e.g., USB, FireWire®). For example, I/O port 1516 or external interface 1522 (e.g., Ethernet, Wi-Fi, etc.) can be used to connect computer system 1500 to a wide area network such as the Internet, a mouse input device, or a scanner. The interconnection via system bus 1512 allows the central processor 1506 to communicate with each subsystem and to control the execution of a plurality of instructions from system memory 1504 or the storage device(s) 1520 (e.g., a fixed disk, such as a hard drive, or optical disk), as well as the exchange of information between subsystems. The system memory 1504

and/or the storage device(s) 1520 may embody a computer readable medium. Another subsystem is a data collection device 1510, such as a camera, microphone, accelerometer, and the like. Any of the data mentioned herein can be output from one component to another component and can be output to the user.

[0286] A computer system can include a plurality of the same components or subsystems, e.g., connected together by external interface 1522, by an internal interface, or via removable storage devices that can be connected and removed from one component to another component. In some embodiments, computer systems, subsystem, or apparatuses can communicate over a network. In such instances, one computer can be considered a client and another computer a server, where each can be part of a same computer system. A client and a server can each include multiple systems, subsystems, or components.

[0287] Any of the computer systems mentioned herein may utilize any suitable number of subsystems. In some embodiments, a computer system includes a single computer apparatus, where the subsystems can be components of the computer apparatus. In other embodiments, a computer system can include multiple computer apparatuses, each being a subsystem, with internal components.

[0288] A computer system can include a plurality of the components or subsystems, e.g., connected together by external interface or by an internal interface. In some embodiments, computer systems, subsystems, or apparatuses can communicate over a network. In such instances, one computer can be considered a client and another computer a server, where each can be part of a same computer system. A client and a server can each include multiple systems, subsystems, or components.

[0289] It should be understood that any of the embodiments of the present invention can be implemented in the form of control logic using hardware (e.g., an application specific integrated circuit or field programmable gate array) and/or using computer software with a generally programmable processor in a modular or integrated manner. As used herein a processor includes a single-core processor, multi-core processor on a same integrated chip, or multiple processing units on a single circuit board or networked. Based on the disclosure and teachings provided herein, a person of ordinary skill in the art will know and appreciate other ways and/or methods to implement embodiments of the present invention using hardware and a combination of hardware and software.

[0290] Any of the software components or functions described in this application may be implemented as software code to be executed by a processor using any suitable computer language such as, for example, Java, C, C++, C#, Objective-C, Swift, or scripting language such as Perl or Python using, for example, conventional or object-oriented techniques. The software code may be stored as a series of instructions or commands on a computer readable medium for storage and/or transmission, suitable media include random access memory (RAM), a read only memory (ROM), a magnetic medium such as a hard-drive or a floppy disk, or an optical medium such as a compact disk (CD) or DVD (digital versatile disk), flash memory, and the like. The computer readable medium may be any combination of such storage or transmission devices.

[0291] Such programs may also be encoded and transmitted using carrier signals adapted for transmission via wired, optical, and/or wireless networks conforming to a variety of protocols, including the Internet. As such, a computer readable medium according to an embodiment of the present invention may be created using a data signal encoded with such programs. Computer readable media encoded with the program code may be packaged with a compatible device or provided separately from other devices (e.g., via Internet download). Any such computer readable medium may reside on or within a single computer product (e.g., a hard drive, a CD, or an entire computer system), and may be present on or within different computer products within a system or network. A computer system may include a monitor, printer or other suitable display for providing any of the results mentioned herein to a user.

[0292] Any of the methods described herein may be totally or partially performed with a computer system including one or more processors, which can be configured to perform the steps. Thus, embodiments can involve computer systems configured to perform the steps of any of the methods described herein, potentially with different components performing a respective steps or a respective group of steps. Although presented as numbered steps, steps of methods herein can be performed at a same time or in a different order. Additionally, portions of these steps may be used with portions of other steps from other methods. Also, all or portions of a step may be optional. Additionally, and of the steps of any of the methods can be performed with modules, circuits, or other means for performing these steps.

[0293] The specific details of particular embodiments may be combined in any suitable manner without departing from the spirit and scope of embodiments of the invention.

However, other embodiments of the invention may involve specific embodiments relating to each individual aspect, or specific combinations of these individual aspects. The above description of exemplary embodiments of the invention has been presented for the purpose of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form described, and many modifications and variations are possible in light of the teaching above. The embodiments were chosen and described in order to best explain the principles of the invention and its practical applications to thereby enable others skilled in the art to best utilize the invention in various embodiments and with various modifications as are suited to the particular use contemplated.

[0294] The above description is illustrative and is not restrictive. Many variations of the invention will become apparent to those skilled in the art upon review of the disclosure. The scope of the invention should, therefore, be determined not with reference to the above description, but instead should be determined with reference to the pending claims along with their full scope or equivalents.

[0295] One or more features from any embodiment may be combined with one or more features of any other embodiment without departing from the scope of the invention.

[0296] A recitation of "a", "an" or "the" is intended to mean "one or more" unless specifically indicated to the contrary. The use of "or" is intended to mean an "inclusive or," and not an "exclusive or" unless specifically indicated to the contrary.

[0297] All patents, patent applications, publications and description mentioned herein are incorporated by reference in their entirety for all purposes. None is admitted to be prior art.

VIII. REFERENCES

[1] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "Graphsc: Parallel secure computation made easy," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 377-394.

[2] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, N. Kiribuchi, and B. Pinkas, "An efficient secure three-party sorting protocol with an honest majority." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 695, 2019.

[3] R. C. Rocha and B. D. Thatte, "Distributed cycle detection in large-scale sparse graphs," *Proceedings of Simpósio Brasileiro de Pesquisa Operacional (SBPO'15)*, pp. 1-11, 2015.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135-146.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *10th {USENIX} Symposium on Operating Systems Design and Implementation* ({OSDI} 12), 2012, pp. 17-30.

[6] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, 2012

[7] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framwork for fast privacy-preserving computations," in *European Symposium on Research in Computer Security*. Springer, 2008, pp. 192-206.

[8] P. Rindal and P. Schoppmann, "Vole-psi: Fast oprf and circuit-psi from vector-ole," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 901-930.

[9] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30-37, 2009.

[10] J. Bennet, S. Lanning *et al.*, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007. New York, NY, USA., 2007, p. 35.

[11] P. Rindal, "The ABY3 Framework for Machine Learning and Database Operations." github.com/ladnir/aby3.

[12] P. Rindal, "libOTe: an efficient, portable, and easy to use Oblivious transfer library," github.com/osu-crypt/libOTe.

[13] "CryptoTools," https://github.com/ladnir/cryptoTools.

[14] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority." ACM CCS 2016.

WHAT IS CLAIMED IS:

1. A method of performing privacy-preserving detection of one or more cliques in directional electronic communications, the method comprising performing, by a multi-party computation network:

receiving a secret-shared union tuple list from a first party computer and a second party computer, wherein the secret-shared union tuple list was generated using a first tuple list corresponding to the first party computer and a second tuple list corresponding to the second party computer, wherein the secret-shared union tuple list comprises a plurality of secret-shared union tuples corresponding to a representation of a union graph, and wherein the secret-shared union tuple list comprises a plurality of secret-shared vertex tuples representing a plurality of vertices in the union graph and a plurality of secret-shared edge tuples representing a plurality of edges in the union graph;

detecting one or more cliques in the secret-shared union tuple list by performing a multi-party computation on the secret-shared union tuple list, the one or more cliques comprising one or more complete subgraphs of the union graph, wherein each complete subgraph of the one or more complete subgraphs comprises a plurality of subgraph vertex tuples corresponding to a plurality of subgraph vertices in the union graph and a plurality of subgraph edge tuples corresponding to a plurality of subgraph edges in the union graph, such that each subgraph vertex of the plurality of subgraph vertices is connected to each other subgraph vertex of the plurality of subgraph vertices via the plurality of subgraph edges; and

providing, to the first party computer, the second party computer, or an additional computer system, an output corresponding to the one or more cliques in response to detecting the one or more cliques.

2. The method of claim 1, wherein detecting the one or more cliques in the secret-shared union tuple list by performing the multi-party computation on the secret-shared union tuple list comprises:

performing a private multi-party Scatter-Gather-Apply implementation of a depth-first or breadth-first search method until a terminating condition has been achieved.

3. The method of claim 2, wherein:

the multi-party computation network stores, in association with each secret-shared vertex tuple of the plurality of secret-shared vertex tuples in the secret-shared union

tuple list, a vertex tuple potential clique list and a first list of vertex tuple lists, the multi-party computation network thereby storing a plurality of vertex tuple potential clique lists and a plurality of first lists of vertex tuples lists;

the multi-party computation network stores, in association with each secret-shared edge tuple of the plurality of secret-shared edge tuples in the secret-shared union tuple list, an edge tuple potential clique list and a second list of vertex tuple lists, the multi-party computation network thereby storing a plurality of edge tuple potential clique lists and a plurality of second lists of vertex tuple lists; and

the plurality of vertex tuple potential clique lists, the plurality of first lists of vertex tuple lists, the plurality of edge tuple potential clique lists, and the plurality of second lists of vertex tuple lists are used during the private multi-party Scatter-Gather-Apply implementation of the depth-first or breadth-first search method to detect the one or more cliques in the secret-shared union tuple list.

4. The method of claim 3, wherein the private multi-party Scatter-Gather-Apply implementation of the depth-first or breadth-first search method comprises:

performing a Scatter step by scattering, for each secret-shared vertex tuple of the plurality of secret-shared vertex tuples, the vertex tuple potential clique list and the first list of vertex tuple lists to a plurality of secret-shared outgoing edge tuples corresponding to the secret-shared vertex tuple, thereby updating the plurality of edge tuple potential clique lists and the plurality of second lists of vertex tuple lists;

performing a Gather step by updating the plurality of vertex tuple potential clique lists and the plurality of first lists of vertex tuple lists based on the plurality of edge tuple potential clique lists and the plurality of second lists of vertex tuple lists;

performing an Apply step by performing:

for each secret-shared vertex tuple of the plurality of secret-shared vertex tuples:

for each vertex tuple potential clique in the vertex tuple potential clique list:

updating the vertex tuple potential clique based on an identity of the secret-shared vertex tuple,

updating a corresponding first list of vertex tuples based on one or more secret-shared outgoing neighbor vertex tuples of the secret-shared vertex tuple, and

evaluating the corresponding first list of vertex tuples, and if the corresponding first list of vertex tuples contains no secret-shared vertex tuples, removing the vertex tuple potential clique and the corresponding first list of vertex tuples from the vertex tuple potential clique list and the first list of vertex tuple lists respectively; and

upon achieving the terminating condition, evaluating, for each vertex tuple, (1) each vertex tuple potential clique and/or (2) each first list of vertex tuple lists in the vertex tuple potential clique list and the first list of vertex tuple lists, thereby detecting the one or more cliques.

5. The method of claim 4, further comprising:

prior to performing the Scatter step, the Gather step, and the Apply step:

allocating, for each secret-shared vertex tuple of the plurality of secret-shared vertex tuples, a vertex tuple memory unit, wherein the multi-party computation network uses the vertex tuple memory unit to store the vertex tuple potential clique list and the first list of vertex tuple lists corresponding to the secret-shared vertex tuple, thereby allocating a plurality of vertex tuple memory units;

prior to performing the Scatter step, the Gather step, and the Apply step:

allocating, for each secret-shared edge tuple of the plurality of secret-shared edge tuples, an edge tuple memory unit, wherein the multi-party computation network uses the edge tuple memory unit to store the edge tuple potential clique list and the second list of vertex tuple lists corresponding to the secret-shared edge tuple, thereby allocating a plurality of edge tuple memory units;

identifying an exceeded vertex tuple memory unit of the plurality of vertex tuple memory units, wherein the exceeded vertex tuple memory unit contains more data than a threshold amount of data corresponding to the exceeded vertex tuple memory unit;

randomly deleting one or more vertex tuple potential cliques and one or more corresponding first vertex tuple lists from an exceeded vertex tuple potential clique list and an exceeded first list of vertex tuple lists corresponding to the exceeded vertex tuple memory unit;

identifying an exceeded edge tuple memory unit of the plurality of edge tuple memory units, wherein the exceeded edge tuple memory unit contains more data than a threshold amount of data corresponding to the exceeded edge tuple memory unit; and

randomly deleting one or more edge tuple potential cliques and one or more corresponding second vertex tuple lists from an exceeded edge tuple potential clique list and an exceeded second list of vertex tuple lists corresponding to the exceeded edge tuple memory unit.

6. The method of claim 4, wherein updating the plurality of vertex tuple potential clique lists and the plurality of first lists of vertex tuple lists based on the plurality of edge tuple potential clique lists and the plurality of second lists of vertex tuple lists comprises:

for each secret-shared vertex tuple of the plurality of secret-shared vertex tuples:

generating one or more secret-shared virtual vertex tuples corresponding to the secret-shared vertex tuple and further corresponding to one or more secret-shared incoming edge tuples, wherein each secret-shared virtual vertex tuple of the one or more secret-shared virtual vertex tuples is associated with a virtual vertex tuple potential clique list and a virtual list of vertex tuple lists, and wherein the virtual vertex tuple potential clique list and the virtual list of vertex tuple lists are equivalent to a corresponding incoming edge tuple potential clique list and a corresponding second list of vertex tuple lists respectively;

for each secret-shared vertex tuple, including the one or more secret-shared virtual vertex tuples in the secret-shared union tuple list and the plurality of secret-shared vertex tuples, thereby including a plurality of secret-shared virtual vertex tuples in the secret-shared union tuple list and the plurality of secret-shared vertex tuples;

removing one or more secret-shared edge tuples from the secret-shared union tuple list; and

generating and including one or more new secret-shared edge tuples in the secret-shared union tuple list based on the plurality of secret-shared vertex tuples and the plurality of secret-shared virtual vertex tuples.

7. The method of claim 1, wherein the secret-shared union tuple list is generated by the first party computer and the second party computer using a private set union protocol.

8. The method of claim 7, wherein the private set union protocol comprises a private set union garbled circuit protocol, wherein the private set union garbled

circuit protocol comprises a modified private set intersection garbled circuit protocol, wherein the modified private set intersection garbled circuit protocol is configured to produce a plurality of secret-shared disjoint tuples based on the first tuple list and the second tuple list, then combine the plurality of secret-shared disjoint tuples with either the first tuple list or the second tuple list, thereby generating the secret-shared union tuple list.

9. The method of claim 1, wherein the multi-party computation network comprises a three-party honest majority semi-honest multi-party computation network comprising a first computer, a second computer, and a third computer, and wherein the step of detecting the one or more cliques in the secret-shared union tuple list is performed by the first computer, the second computer, and the third computer using a three-party honest majority semi-honest multi-party implementation of a clique detection method.

10. The method of claim 9, wherein the first party computer and the second party computer are members of the multi-party computation network, such that the first party computer is the first computer and the second party computer is the second computer.

11. The method of claim 1, further comprising, prior to detecting one or more cliques in the secret-shared union tuple list by performing the multi-party computation:

generating an induced secret-shared union tuple list corresponding to an induced subgraph of the union graph, wherein the induced secret-shared union tuple list is initially equivalent to the secret-shared union tuple list and the induced subgraph is initially equivalent to the union graph;

performing an oblivious iterative process on the induced secret-shared union tuple list until the induced secret-shared union tuple list comprises zero secret-shared vertex tuples, the oblivious iterative process comprising:

identifying and recording one or more low degree vertex tuples by evaluating a degree of each secret-shared vertex tuple in the induced secret-shared union tuple list, and

updating the induced secret-shared union tuple list by removing the one or more low degree vertex tuples and one or more associated edge tuples from the induced secret-shared union tuple list;

assigning a rank to each secret-shared vertex tuple in the secret-shared union tuple list based on an order in which that secret-shared vertex tuple was identified and recorded as a low degree vertex tuple and based on a degree of that secret-shared vertex tuple; and

for each secret-shared edge tuple in the secret-shared union tuple list:

determining a first rank of a first secret-shared vertex tuple associated with that secret-shared edge tuple,

determining a second rank of a second secret-shared vertex tuple associated with that secret-shared edge tuple,

if the first rank is greater than or equal to the second rank, modifying the secret-shared edge tuple such that it indicates that a directed edge in the union graph corresponding to the secret-shared edge tuple points from a second vertex in the union graph corresponding to the second secret-shared vertex tuple toward a first vertex in the union graph corresponding to the first secret-shared vertex tuple, and

if the second rank is greater than the first rank, modifying the secret-shared edge tuple such that it indicates that a directed edge in the union graph corresponding to the secret-shared edge tuple points from the first vertex in the union graph corresponding to the first secret-shared vertex tuple toward the second vertex in the union graph corresponding to the second secret-shared vertex tuple.

12. The method of claim 1, further comprising:

generating a plurality of secret-shared duplicate edge tuples by duplicating the plurality of secret-shared edge tuples; and

including the plurality of secret-shared duplicate edge tuples in the secret-shared union tuple list.

13. The method of claim 1, wherein the secret-shared union tuple list further comprises a plurality of secret-shared duplicate edge tuples representing the plurality of edges in the union graph, and wherein the method further comprises:

generating a first permutation corresponding to a first ordering and a second permutation corresponding to a second ordering, wherein the first permutation enables the multi-party computation network to order the secret-shared union tuple list according to the

first ordering, and wherein the second permutation enables the multi-party computation network to order the secret-shared union tuple list according to the second ordering.

14. The method of claim 13, wherein:

the first permutation corresponding to the first ordering is generated by obliviously sorting the secret-shared union tuple list into the first ordering;

the second permutation corresponding to the second ordering is generated by obliviously sorting the secret-shared union tuple list into the second ordering;

in the first ordering, each secret-shared vertex tuple of the plurality of secret-shared vertex tuples in the secret-shared union tuple list is preceded by one or more corresponding secret-shared edge tuples of the plurality of secret-shared edge tuples and followed by one or more corresponding secret-shared duplicate edge tuples of the plurality of secret-shared duplicate edge tuples; and

in the second ordering, each secret-shared vertex tuple of the plurality of secret-shared vertex tuples in the secret-shared union tuple list is preceded by one or more corresponding secret-shared duplicate edge tuples of the plurality of secret-shared duplicate edge tuples and followed by one or more corresponding secret-shared edge tuples of the plurality of secret-shared edge tuples.

15. The method of claim 13, wherein generating the first permutation corresponding to the first ordering and the second permutation corresponding to the second ordering comprises:

obliviously sorting the secret-shared union tuple list into the first ordering;

performing a secure random shuffle on the secret-shared union tuple list while the secret-shared union tuple list is in the first ordering, thereby generating a randomly shuffled secret-shared union tuple list, a shuffle permutation, and an inverse shuffle permutation;

revealing, to a plurality of computers in the multi-party computation network, a plurality of edge tuple identities corresponding to the plurality of secret-shared edge tuples and a plurality of duplicate edge tuple identities corresponding to the plurality of secret-shared duplicate edge tuples;

determining an intermediate permutation based on the plurality of edge tuple identities and the plurality of duplicate edge tuple identities, wherein applying the intermediate permutation to the randomly shuffled secret-shared union tuple list swaps a

plurality of edge tuple positions and a plurality of corresponding duplicate edge tuple positions in the randomly shuffled secret-shared union tuple list;

determining the first permutation based on the shuffle permutation, the inverse shuffle permutation, and the intermediate permutation; and

determining the second permutation based on the shuffle permutation, the inverse shuffle permutation, and the intermediate permutation.

16. The method of claim 13, wherein detecting the one or more cliques in the secret-shared union tuple list by performing the multi-party computation on the secret-shared union tuple list comprises an iterative process comprising:

(1) obliviously shuffling the secret-shared union tuple list into the first ordering using the first permutation;

(2) performing a combined Scatter-Gather step on the secret-shared union tuple list;

(3) performing an Apply step on the secret-shared union tuple list;

(4) obliviously shuffling the secret-shared union tuple list into the second ordering using the second permutation;

(5) performing the combined Scatter-Gather step on the secret-shared union tuple list;

(6) performing the Apply step on the secret-shared union tuple list; and

(7) repeating steps (1)-(6) until a terminating condition has been achieved, wherein the one or more cliques are determined in response to achieving the terminating condition.

17. The method of claim 16, wherein:

the combined Scatter-Gather step comprises:

defining a set of inputs comprising the plurality of secret-shared union tuples in the secret-shared union tuple list;

an upward pass comprising:

(a) dividing the set of inputs among a plurality of processors,

(b) processing the set of inputs using a clique detection method and based on a current ordering of the secret-shared union tuple list using the plurality of processors, thereby producing a first set of outputs, wherein the

first set of outputs comprises less outputs than the set of inputs comprises inputs,

(c) defining the set of inputs as the first set of outputs, and

(d) repeating the upward pass until the set of inputs comprises a single input, and

a downward pass comprising:

(f) dividing the set of inputs among the plurality of processors,

(g) processing the set of inputs using the clique detection method and based on the current ordering of the secret-shared union tuple list using the plurality of processors, thereby producing a second set of outputs, wherein the second set of outputs comprises more outputs than the set of inputs comprises inputs, and

(h) repeating the downward pass until the set of inputs comprises an updated plurality of secret-shared union tuples in the secret-shared union tuple list; and

the Apply step comprises:

(i) dividing the updated plurality of secret-shared union tuples among the plurality of processors, and

(j) applying an Apply function to each updated secret-shared union tuple of the updated plurality of secret-shared union tuples using the plurality of processors.

18. A method of detecting one or more cliques in a union graph corresponding to a secret-shared union tuple list comprising performing, by a multi-party computation network:

receiving the secret-shared union tuple list from a first party computer and a second party computer, wherein the secret-shared union tuple list was generated using a first tuple list corresponding to the first party computer, and a second tuple list corresponding to the second party computer, and wherein the secret-shared union tuple list comprises a representation of the union graph;

generating a first permutation corresponding to a first ordering and a second permutation corresponding to a second ordering, wherein the first permutation enables the multi-party computation network to order the secret-shared union tuple list according to the

first ordering, and wherein the second permutation enables the multi-party computation network to order the secret-shared union tuple list according to the second ordering;

defining a set of inputs as a plurality of secret-shared union tuples in the secret-shared union tuple list;

executing a parallel private clique detection method comprising a breadth-first or depth-first based clique detection method implemented using an iterative Scatter-Gather-Apply approach, the iterative Scatter-Gather-Apply approach comprising an upward pass, a downward pass, and an Apply step;

the upward pass comprising:

(1) dividing the set of inputs among a plurality of processors;

(2) processing the set of inputs based on a current ordering of the secret-shared union tuple list using the plurality of processors, thereby producing a set of outputs, wherein the set of outputs comprises less outputs than the set of inputs comprises inputs;

(3) defining the set of inputs as the set of outputs;

(4) repeating the upward pass until the set of inputs comprises a single input;

the downward pass comprising:

(5) dividing the set of inputs among the plurality of processors;

(6) processing the set of inputs based on the current ordering of the secret-shared union tuple list using the plurality of processors, thereby producing the set of outputs, wherein the set of outputs comprises more outputs than the set of inputs comprises inputs;

(7) defining the set of inputs as the set of outputs;

(8) repeating the downward pass until the set of inputs comprises an updated plurality of secret-shared union tuples in the secret-shared union tuple list;

the Apply step comprising:

(9) dividing the updated plurality of secret-shared union tuples among the plurality of processors;

(10) applying an apply function to each secret-shared union tuple of the updated plurality of secret-shared union tuples using the plurality of processors, wherein the apply function evaluates and updates a plurality of potential clique lists and a plurality of lists of vertex tuple lists associated with the plurality of secret-shared union tuples;

(11) determining that a terminating condition has not been achieved;

if the secret-shared union tuple list is in the first ordering, oblivious shuffling the secret-shared union tuple list into the second ordering using the second permutation, otherwise oblivious shuffling the secret-shared union tuple list into the first ordering using the first permutation; and

repeating the iterative Scatter-Gather-Apply approach until the terminating condition has been achieved; and

detecting the one or more cliques in the union graph by evaluating the plurality of potential clique lists and/or the plurality of lists of vertex tuple lists, thereby producing a result of the parallel private clique detection method, wherein the result of the parallel private clique detection method comprises a list of the one or more cliques corresponding to the union graph.

19. A computer comprising:

one or more processors; and

a non-transitory computer readable medium coupled to the one or more processors, the non-transitory computer readable medium comprising code that, when executed, cause the one or more processors to perform the method of any one of claims 1-18

20. A multi-party computation network comprising:

a plurality of processors; and

a plurality of non-transitory computer readable media coupled to the plurality of processors, the plurality of non-transitory computer readable media comprising code executable by the plurality of processors for implementing the method of any of claims 1-18.

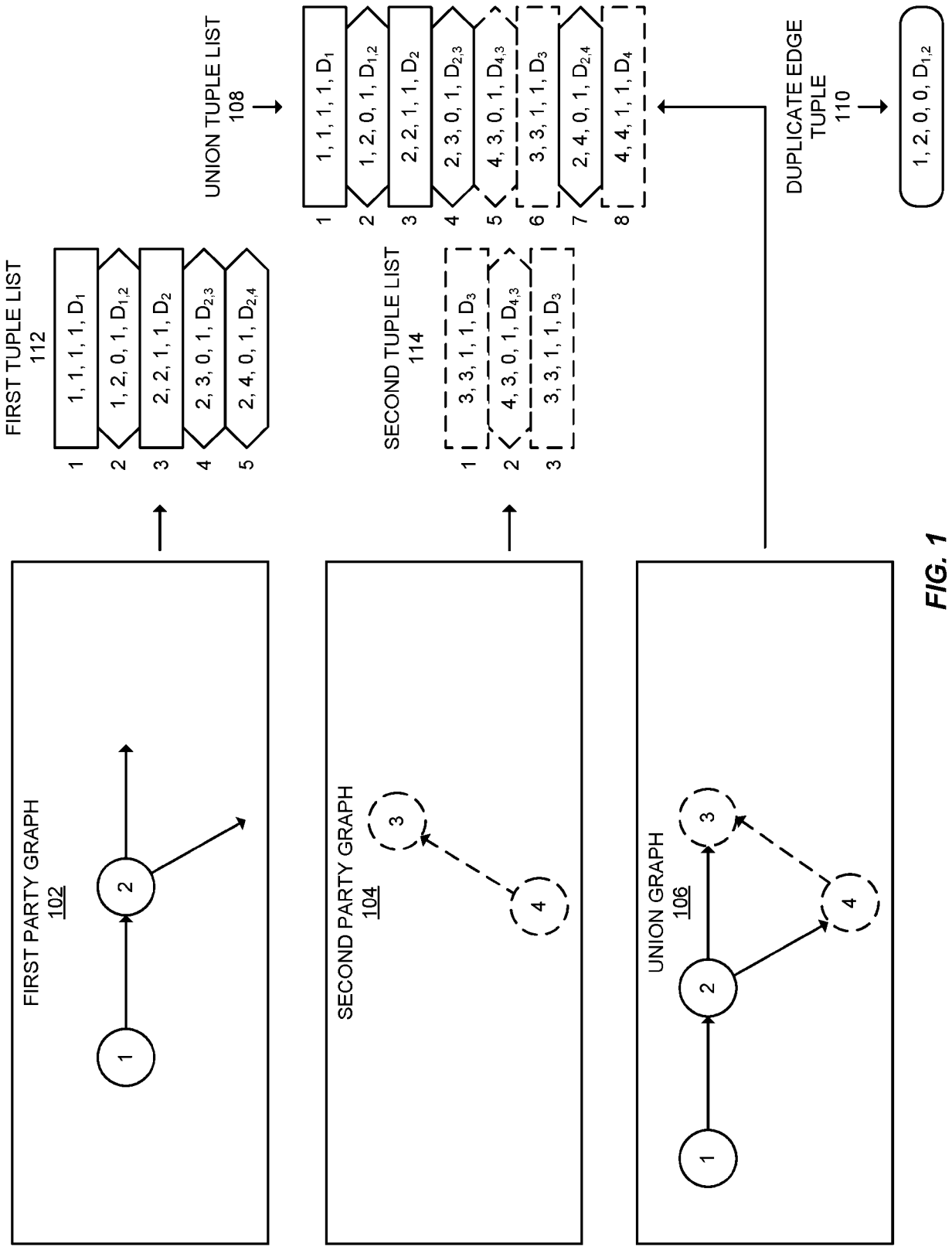
FIRST PARTY GRAPH
102
SECOND PARTY GRAPH
104
UNION GRAPH
106
FIRST TUPLE LIST
112
SECOND TUPLE LIST
114
UNION TUPLE LIST
108
DUPLICATE EDGE TUPLE
110
1, 1, 1, 1, D1
1, 2, 0, 1, D1,2
2, 2, 1, 1, D2
2, 3, 0, 1, D2,3
2, 4, 0, 1, D2,4
3, 3, 1, 1, D3
4, 3, 0, 1, D4,3
4, 4, 1, 1, D4
1, 2, 0, 0, D1,2

FIG. 1

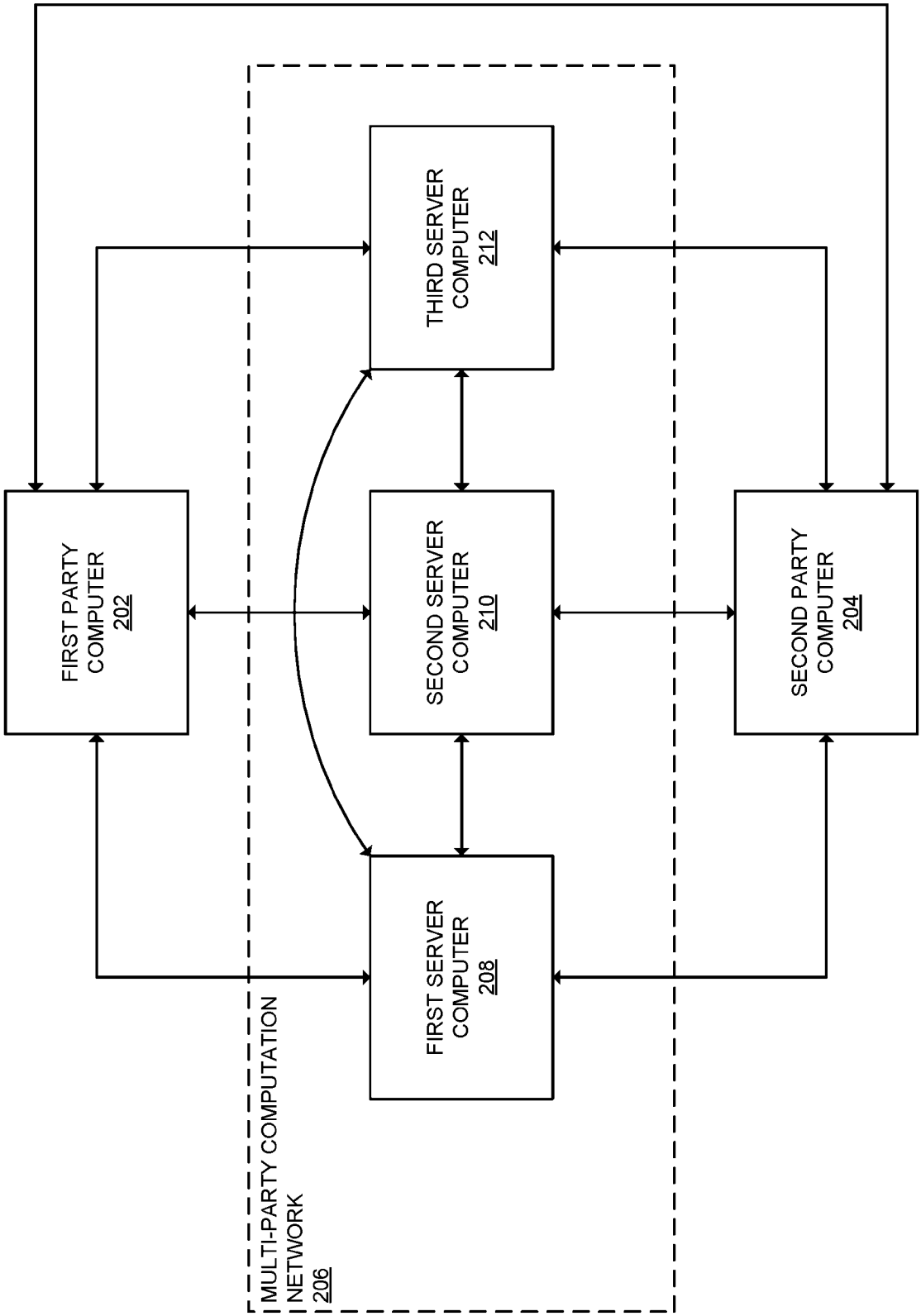
FIRST PARTY COMPUTER 202
THIRD SERVER COMPUTER 212
SECOND SERVER COMPUTER 210
FIRST SERVER COMPUTER 208
SECOND PARTY COMPUTER 204
MULTI-PARTY COMPUTATION NETWORK 206

FIG. 2

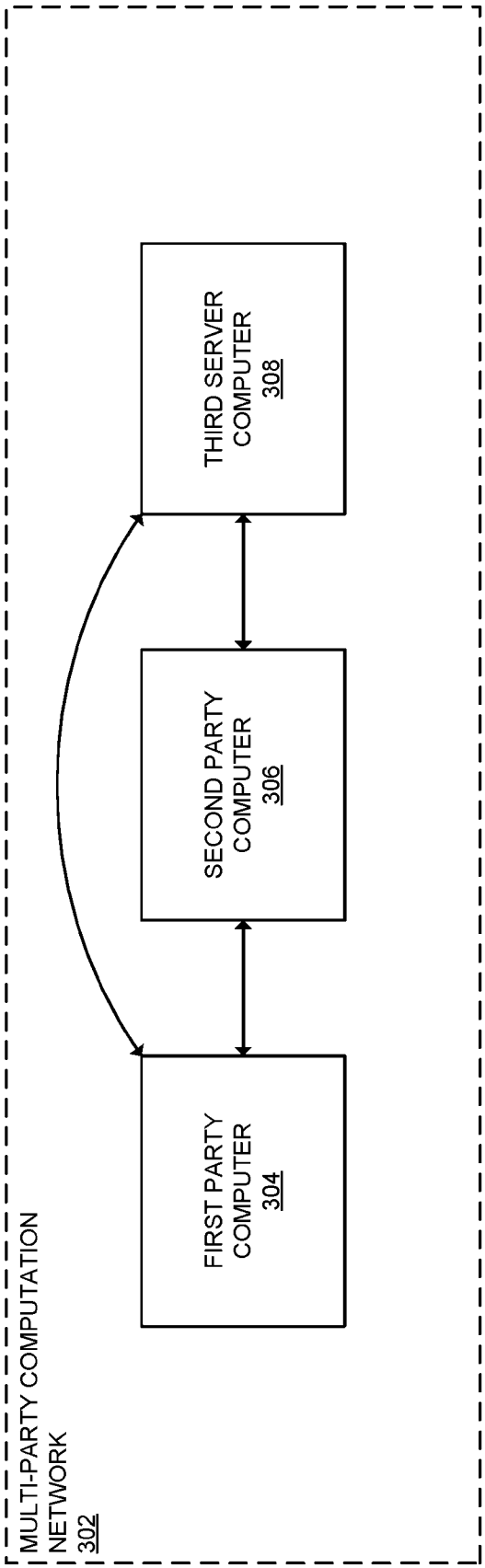
MULTI-PARTY COMPUTATION NETWORK
302
FIRST PARTY COMPUTER
304
SECOND PARTY COMPUTER
306
THIRD SERVER COMPUTER
308

FIG. 3

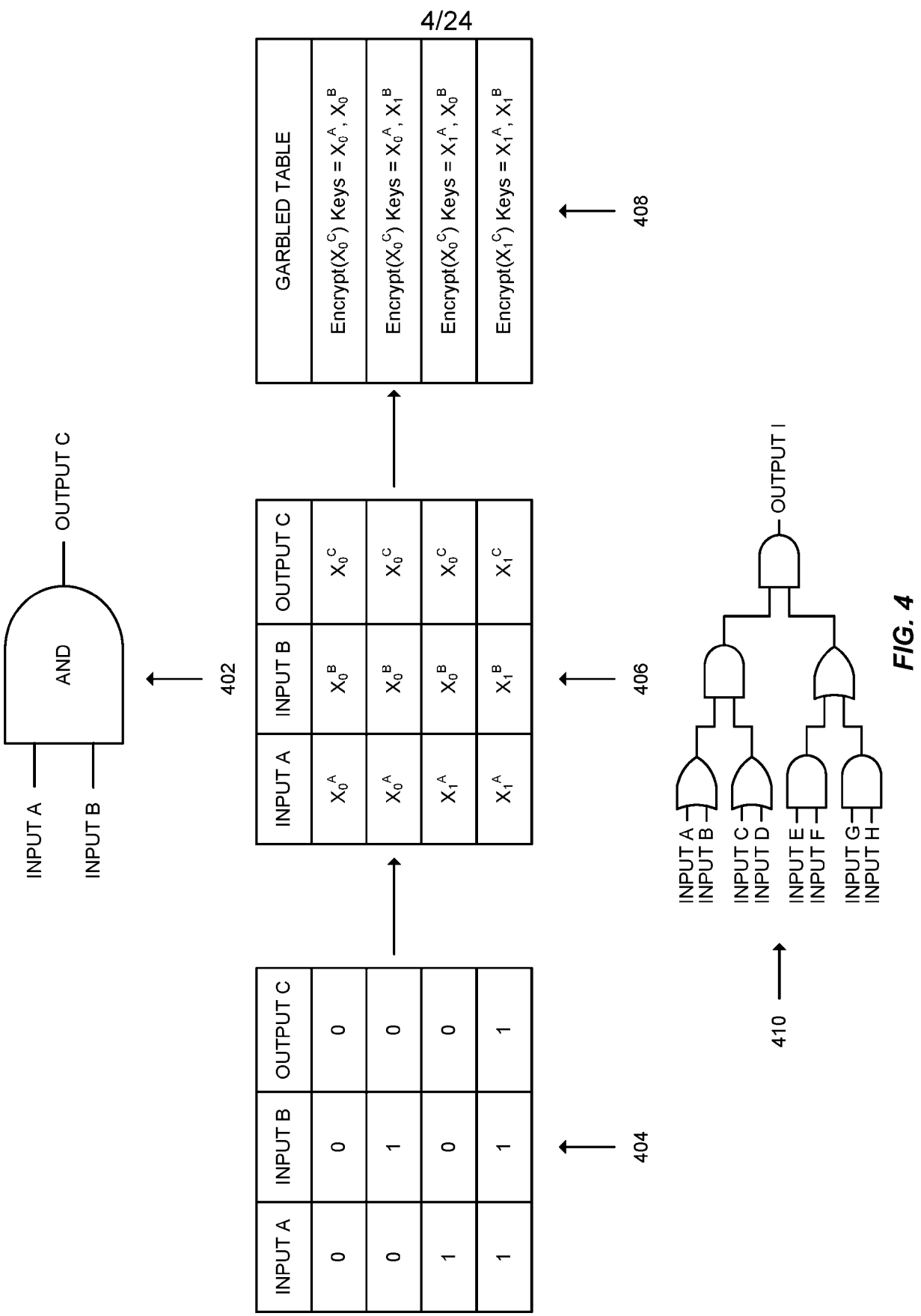
INPUT A
INPUT B
AND
OUTPUT C
402
INPUT A
INPUT B
OUTPUT C
X_0^A X_0^B X_0^C
X_0^A X_1^B X_0^C
X_1^A X_0^B X_0^C
X_1^A X_1^B X_1^C
406
GARBLED TABLE
Encrypt(X_0^C) Keys = X_0^A, X_0^B
Encrypt(X_0^C) Keys = X_0^A, X_1^B
Encrypt(X_0^C) Keys = X_1^A, X_0^B
Encrypt(X_1^C) Keys = X_1^A, X_1^B
408
INPUT A
INPUT B
0 0 0
0 1 0
1 0 0
1 1 1
404
INPUT A
INPUT B
INPUT C
INPUT D
INPUT E
INPUT F
INPUT G
INPUT H
OUTPUT I
410

FIG. 4

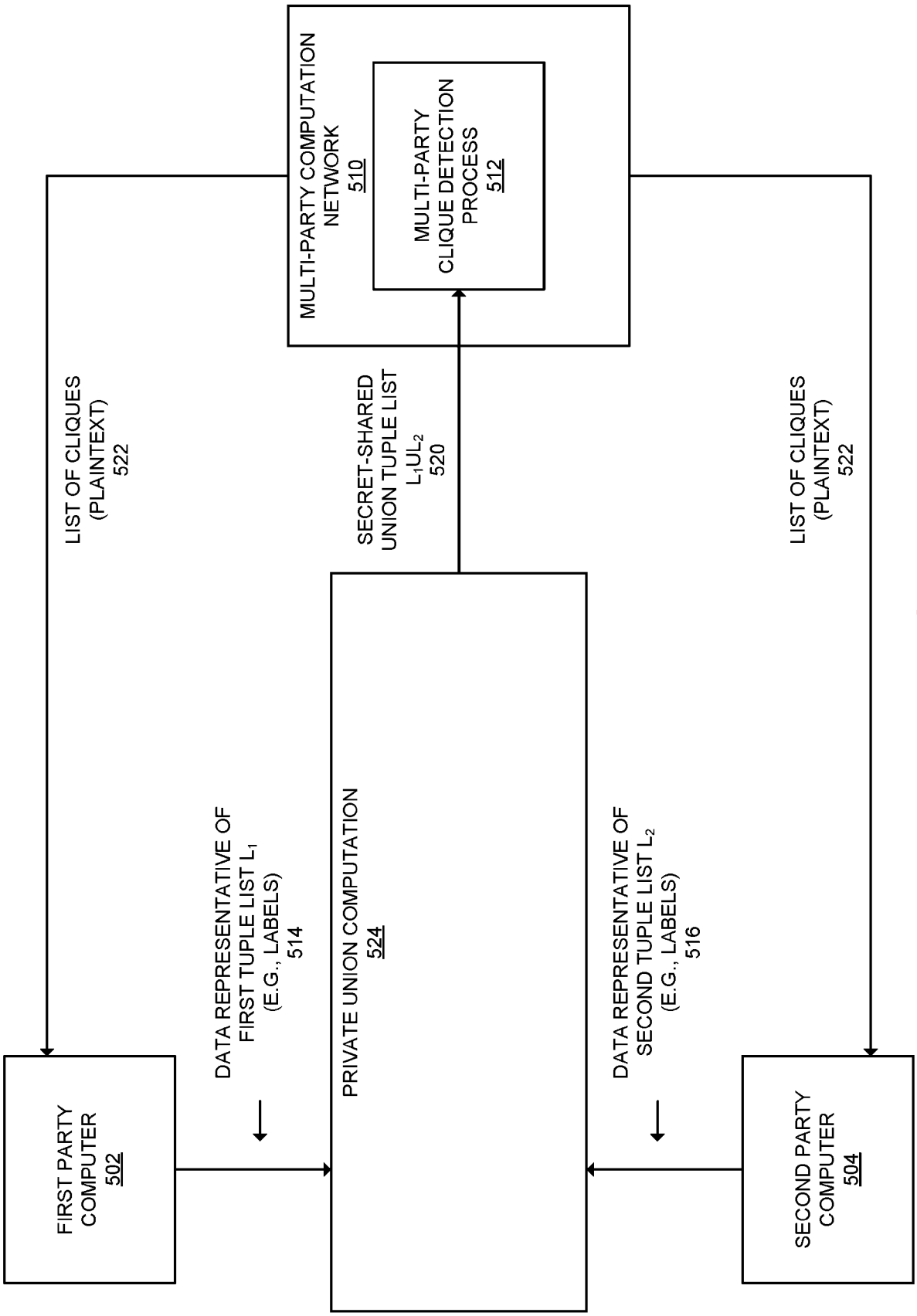
MULTI-PARTY COMPUTATION NETWORK
510
MULTI-PARTY CLIQUE DETECTION PROCESS
512
LIST OF CLIQUES (PLAINTEXT)
522
SECRET-SHARED UNION TUPLE LIST
$L_1 \cup L_2$
520
LIST OF CLIQUES (PLAINTEXT)
522
DATA REPRESENTATIVE OF FIRST TUPLE LIST L_1 (E.G., LABELS)
514
PRIVATE UNION COMPUTATION
524
DATA REPRESENTATIVE OF SECOND TUPLE LIST L_2 (E.G., LABELS)
516
FIRST PARTY COMPUTER
502
SECOND PARTY COMPUTER
504

FIG. 5

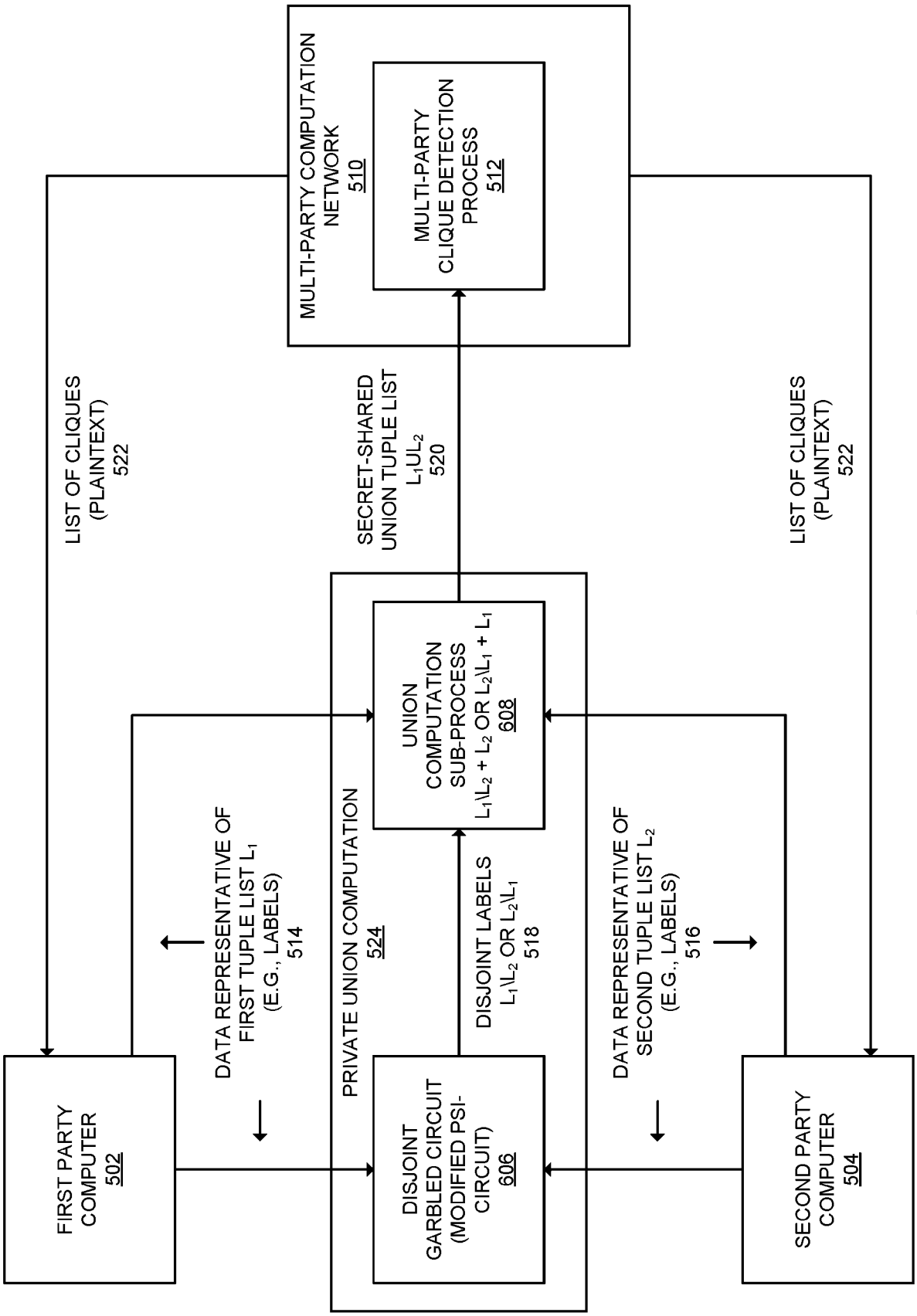
MULTI-PARTY COMPUTATION NETWORK 510
MULTI-PARTY CLIQUE DETECTION PROCESS 512
LIST OF CLIQUES (PLAINTEXT) 522
SECRET-SHARED UNION TUPLE LIST $L_1 \cup L_2$ 520
LIST OF CLIQUES (PLAINTEXT) 522
UNION COMPUTATION SUB-PROCESS $L_1 \setminus L_2 + L_2$ OR $L_2 \setminus L_1 + L_1$ 608
DATA REPRESENTATIVE OF FIRST TUPLE LIST L_1 (E.G., LABELS) 514
PRIVATE UNION COMPUTATION 524
DISJOINT LABELS $L_1 \setminus L_2$ OR $L_2 \setminus L_1$ 518
DATA REPRESENTATIVE OF SECOND TUPLE LIST L_2 (E.G., LABELS) 516
FIRST PARTY COMPUTER 502
DISJOINT GARBLED CIRCUIT (MODIFIED PSI-CIRCUIT) 606
SECOND PARTY COMPUTER 504

FIG. 6

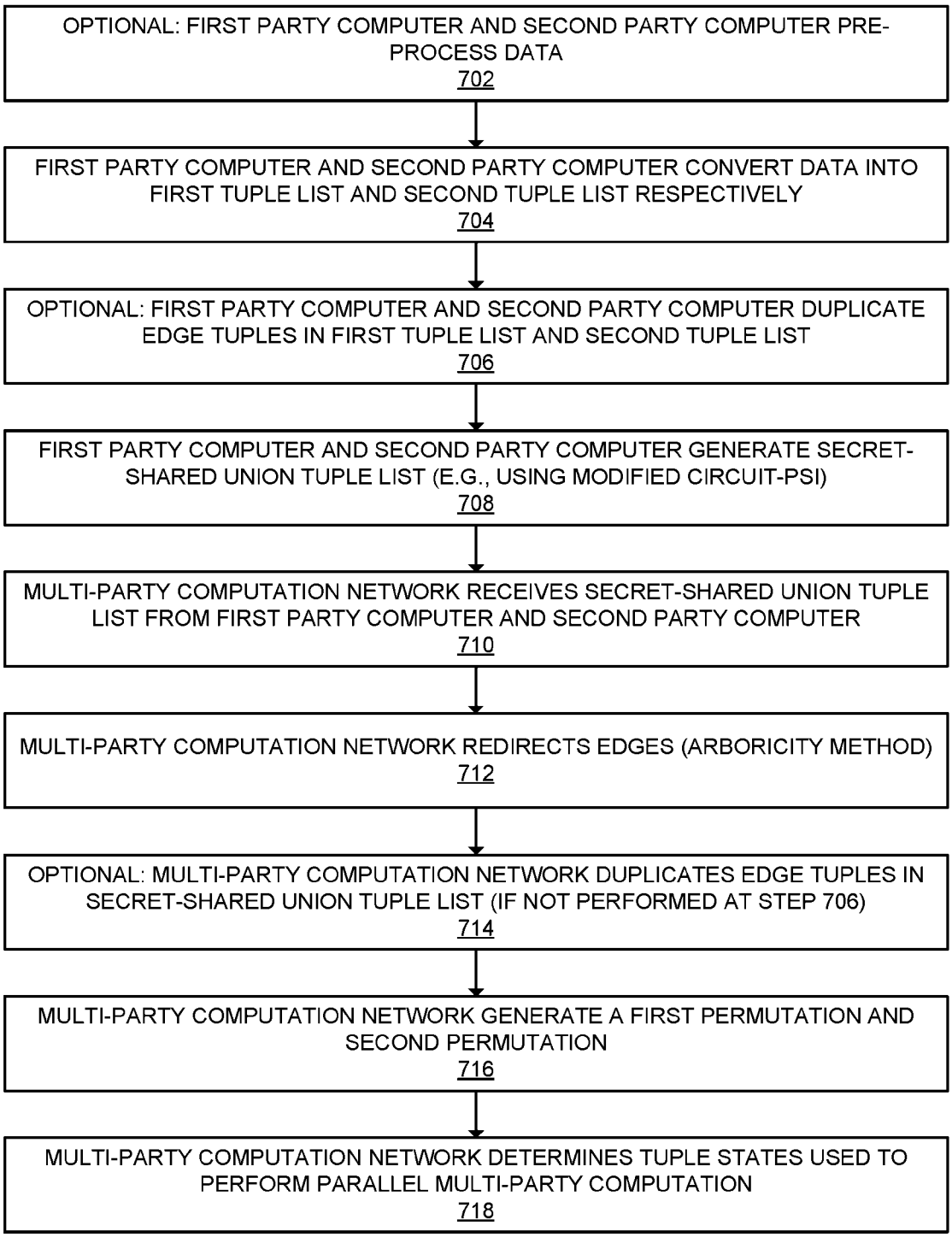
OPTIONAL: FIRST PARTY COMPUTER AND SECOND PARTY COMPUTER PRE-PROCESS DATA
702
FIRST PARTY COMPUTER AND SECOND PARTY COMPUTER CONVERT DATA INTO FIRST TUPLE LIST AND SECOND TUPLE LIST RESPECTIVELY
704
OPTIONAL: FIRST PARTY COMPUTER AND SECOND PARTY COMPUTER DUPLICATE EDGE TUPLES IN FIRST TUPLE LIST AND SECOND TUPLE LIST
706
FIRST PARTY COMPUTER AND SECOND PARTY COMPUTER GENERATE SECRET-SHARED UNION TUPLE LIST (E.G., USING MODIFIED CIRCUIT-PSI)
708
MULTI-PARTY COMPUTATION NETWORK RECEIVES SECRET-SHARED UNION TUPLE LIST FROM FIRST PARTY COMPUTER AND SECOND PARTY COMPUTER
710
MULTI-PARTY COMPUTATION NETWORK REDIRECTS EDGES (ARBORICITY METHOD)
712
OPTIONAL: MULTI-PARTY COMPUTATION NETWORK DUPLICATES EDGE TUPLES IN SECRET-SHARED UNION TUPLE LIST (IF NOT PERFORMED AT STEP 706)
714
MULTI-PARTY COMPUTATION NETWORK GENERATE A FIRST PERMUTATION AND SECOND PERMUTATION
716
MULTI-PARTY COMPUTATION NETWORK DETERMINES TUPLE STATES USED TO PERFORM PARALLEL MULTI-PARTY COMPUTATION
718

FIG. 7

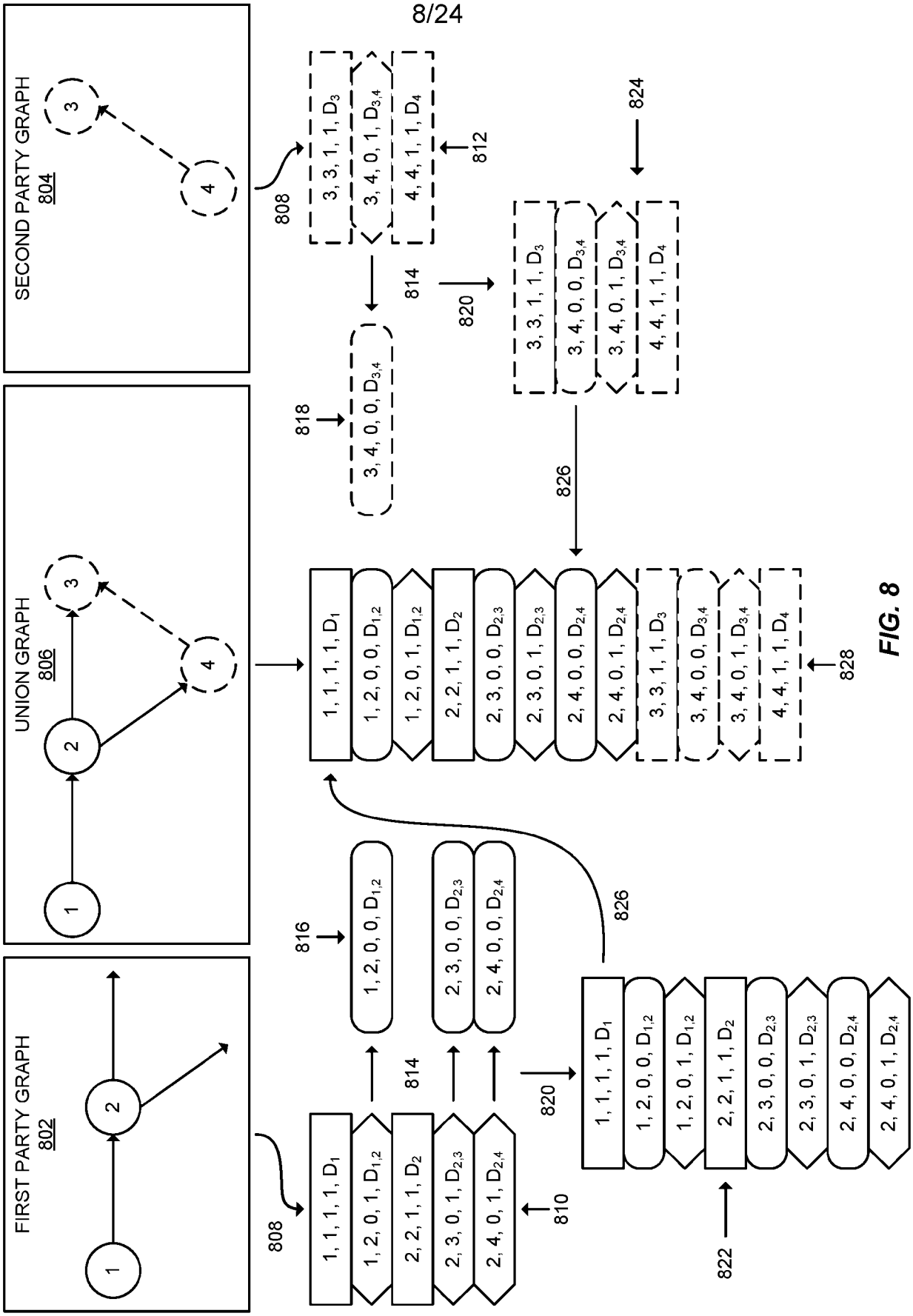
FIRST PARTY GRAPH
802
SECOND PARTY GRAPH
804
UNION GRAPH
806
808
810
812
814
816
818
820
822
824
826
828
1, 1, 1, 1, D1
1, 2, 0, 1, D1,2
2, 2, 1, 1, D2
2, 3, 0, 1, D2,3
2, 4, 0, 1, D2,4
3, 3, 1, 1, D3
3, 4, 0, 1, D3,4
4, 4, 1, 1, D4
1, 2, 0, 0, D1,2
2, 3, 0, 0, D2,3
2, 4, 0, 0, D2,4
3, 4, 0, 0, D3,4

FIG. 8

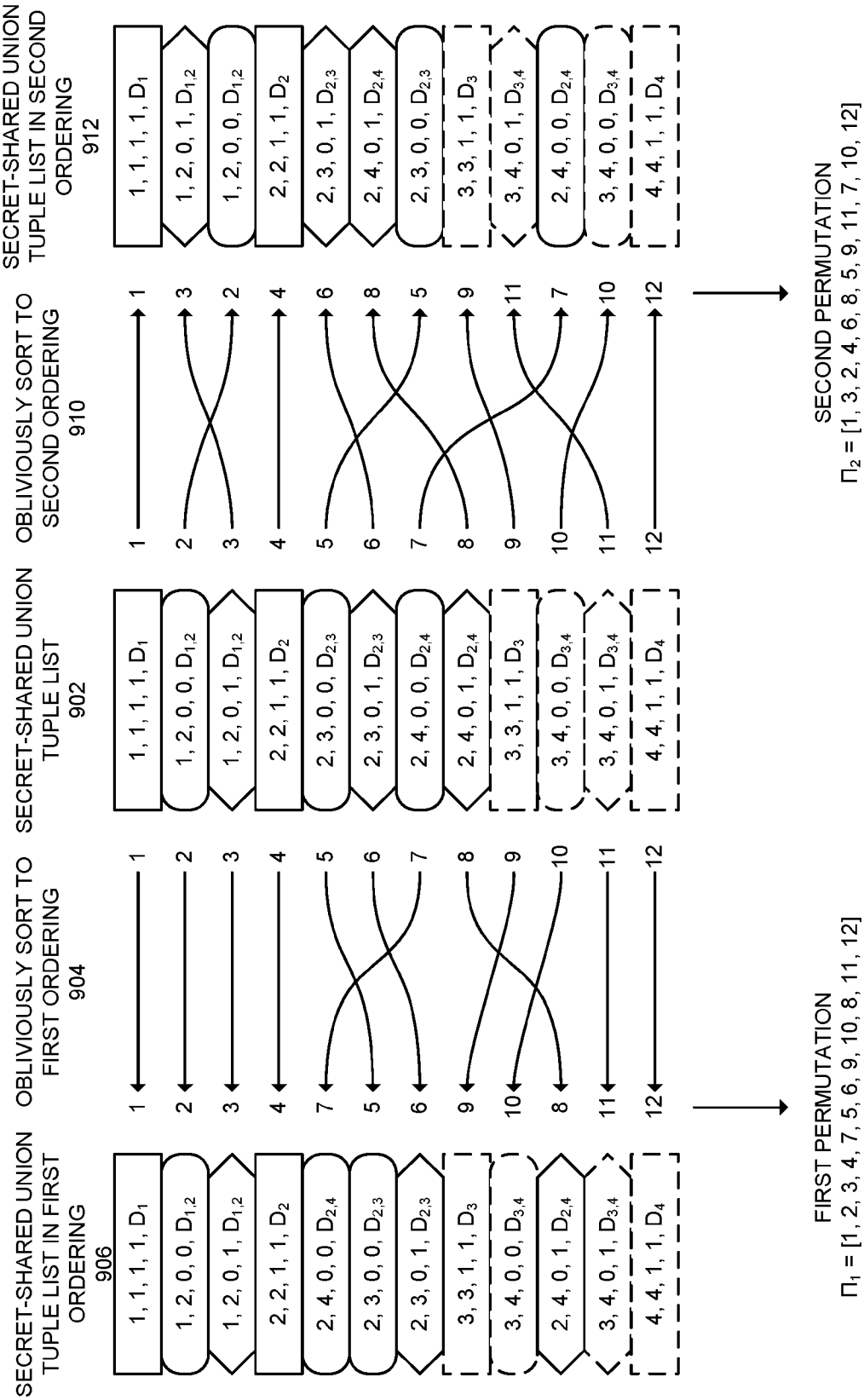
SECRET-SHARED UNION TUPLE LIST IN FIRST ORDERING 906
OBLIVIOUSLY SORT TO FIRST ORDERING 904
SECRET-SHARED UNION TUPLE LIST 902
OBLIVIOUSLY SORT TO SECOND ORDERING 910
SECRET-SHARED UNION TUPLE LIST IN SECOND ORDERING 912
FIRST PERMUTATION
Π1 = [1, 2, 3, 4, 7, 5, 6, 9, 10, 8, 11, 12]
908
SECOND PERMUTATION
Π2 = [1, 3, 2, 4, 6, 8, 5, 9, 11, 7, 10, 12]
914

FIG. 9

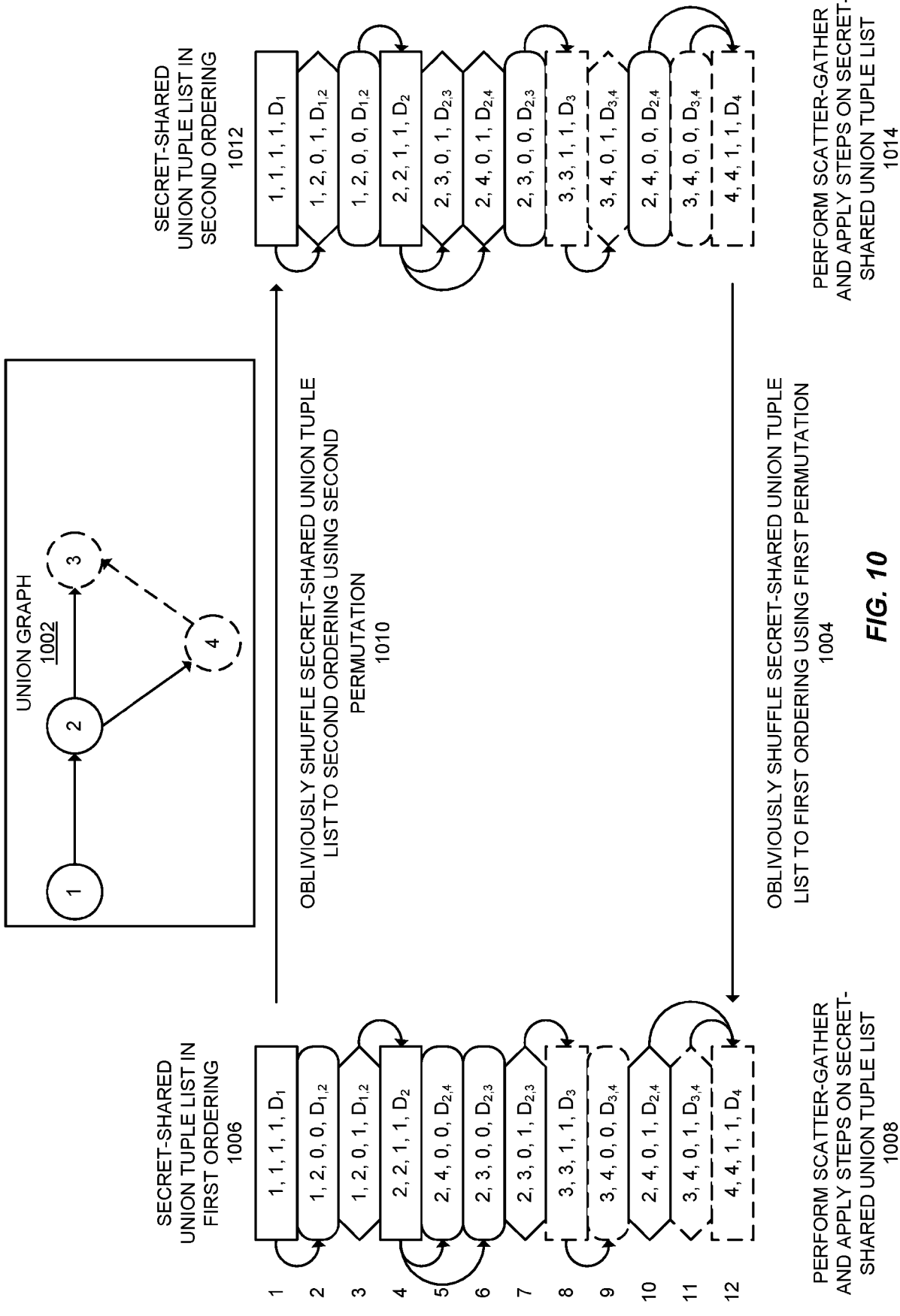
UNION GRAPH
1002
1
2
3
4
SECRET-SHARED UNION TUPLE LIST IN FIRST ORDERING
1006
1, 1, 1, 1, 1, D1
1, 2, 0, 0, D1,2
1, 2, 0, 1, D1,2
2, 2, 1, 1, D2
2, 4, 0, 0, D2,4
2, 3, 0, 0, D2,3
2, 3, 0, 1, D2,3
3, 3, 1, 1, D3
3, 4, 0, 0, D3,4
2, 4, 0, 1, D2,4
3, 4, 0, 1, D3,4
4, 4, 1, 1, D4
1 2 3 4 5 6 7 8 9 10 11 12
PERFORM SCATTER-GATHER AND APPLY STEPS ON SECRET-SHARED UNION TUPLE LIST
1008
OBLIVIOUSLY SHUFFLE SECRET-SHARED UNION TUPLE LIST TO SECOND ORDERING USING SECOND PERMUTATION
1010
SECRET-SHARED UNION TUPLE LIST IN SECOND ORDERING
1012
1, 1, 1, 1, D1
1, 2, 0, 1, D1,2
1, 2, 0, 0, D1,2
2, 2, 1, 1, D2
2, 3, 0, 1, D2,3
2, 4, 0, 1, D2,4
2, 3, 0, 0, D2,3
3, 3, 1, 1, D3
3, 4, 0, 1, D3,4
2, 4, 0, 0, D2,4
3, 4, 0, 0, D3,4
4, 4, 1, 1, D4
PERFORM SCATTER-GATHER AND APPLY STEPS ON SECRET-SHARED UNION TUPLE LIST
1014
OBLIVIOUSLY SHUFFLE SECRET-SHARED UNION TUPLE LIST TO FIRST ORDERING USING FIRST PERMUTATION
1004

FIG. 10

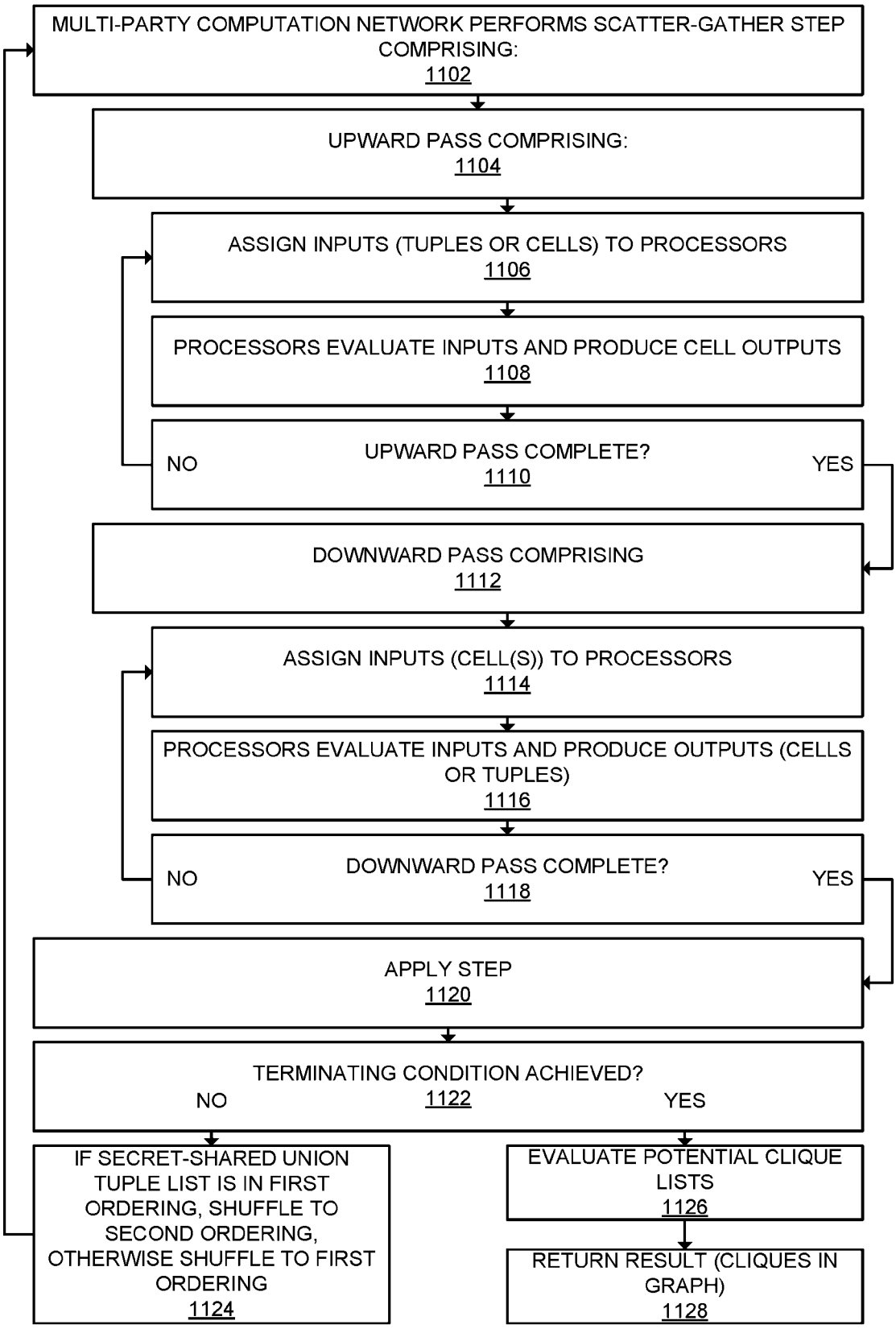
MULTI-PARTY COMPUTATION NETWORK PERFORMS SCATTER-GATHER STEP COMPRISING:
1102
UPWARD PASS COMPRISING:
1104
ASSIGN INPUTS (TUPLES OR CELLS) TO PROCESSORS
1106
PROCESSORS EVALUATE INPUTS AND PRODUCE CELL OUTPUTS
1108
NO
UPWARD PASS COMPLETE?
1110
YES
DOWNWARD PASS COMPRISING
1112
ASSIGN INPUTS (CELL(S)) TO PROCESSORS
1114
PROCESSORS EVALUATE INPUTS AND PRODUCE OUTPUTS (CELLS OR TUPLES)
1116
NO
DOWNWARD PASS COMPLETE?
1118
YES
APPLY STEP
1120
TERMINATING CONDITION ACHIEVED?
NO
1122
YES
IF SECRET-SHARED UNION TUPLE LIST IS IN FIRST ORDERING, SHUFFLE TO SECOND ORDERING, OTHERWISE SHUFFLE TO FIRST ORDERING
1124
EVALUATE POTENTIAL CLIQUE LISTS
1126
RETURN RESULT (CLIQUES IN GRAPH)
1128

FIG. 11

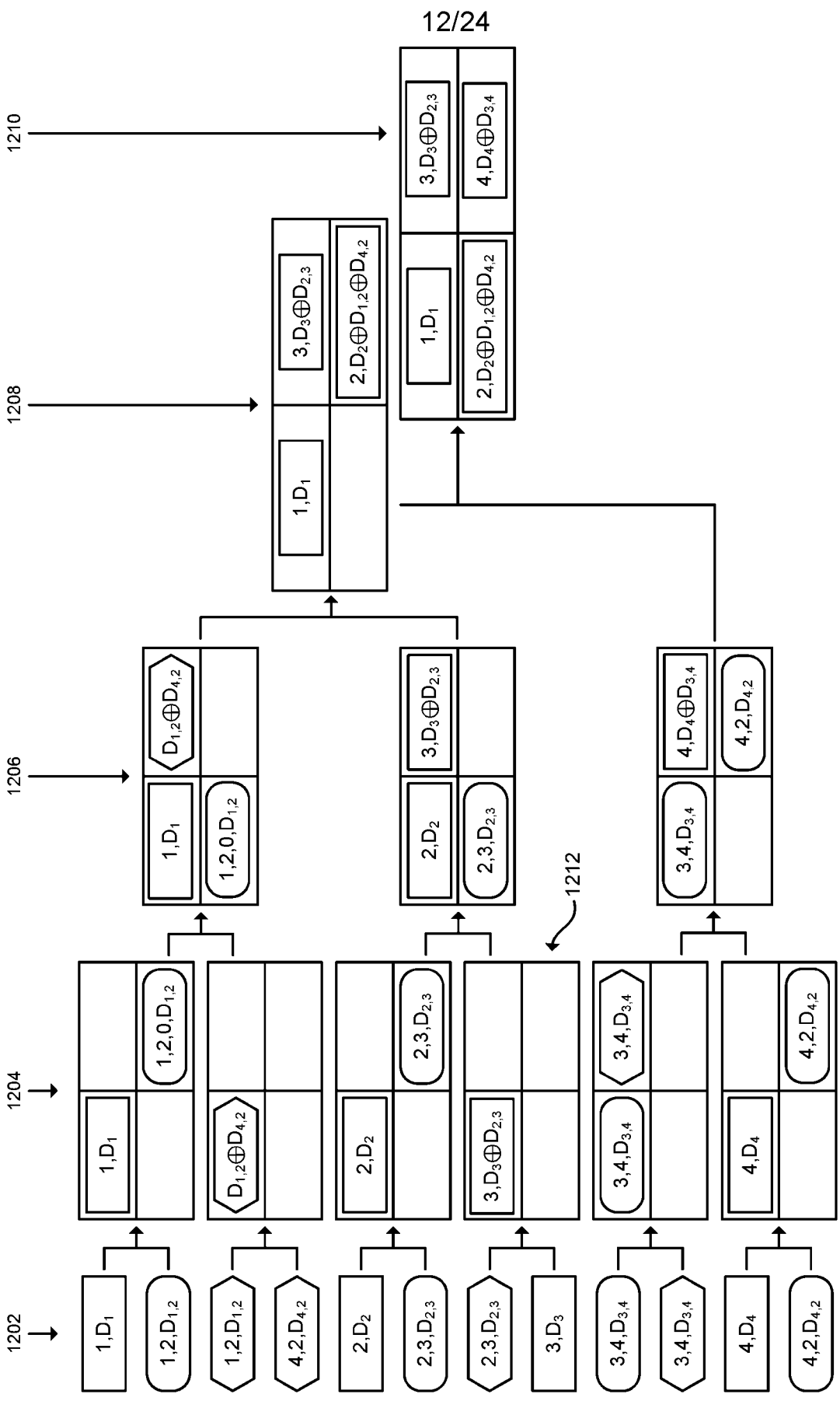
1202
1204
1206
1208
1210
1212
1,D1
1,2,D1,2
D1,2⊕D4,2
2,D2
2,3,D2,3
3,D3
3,4,D3,4
4,D4
4,2,D4,2
1,2,0,D1,2
3,D3⊕D2,3
2,D2⊕D1,2⊕D4,2
4,D4⊕D3,4

FIG. 12

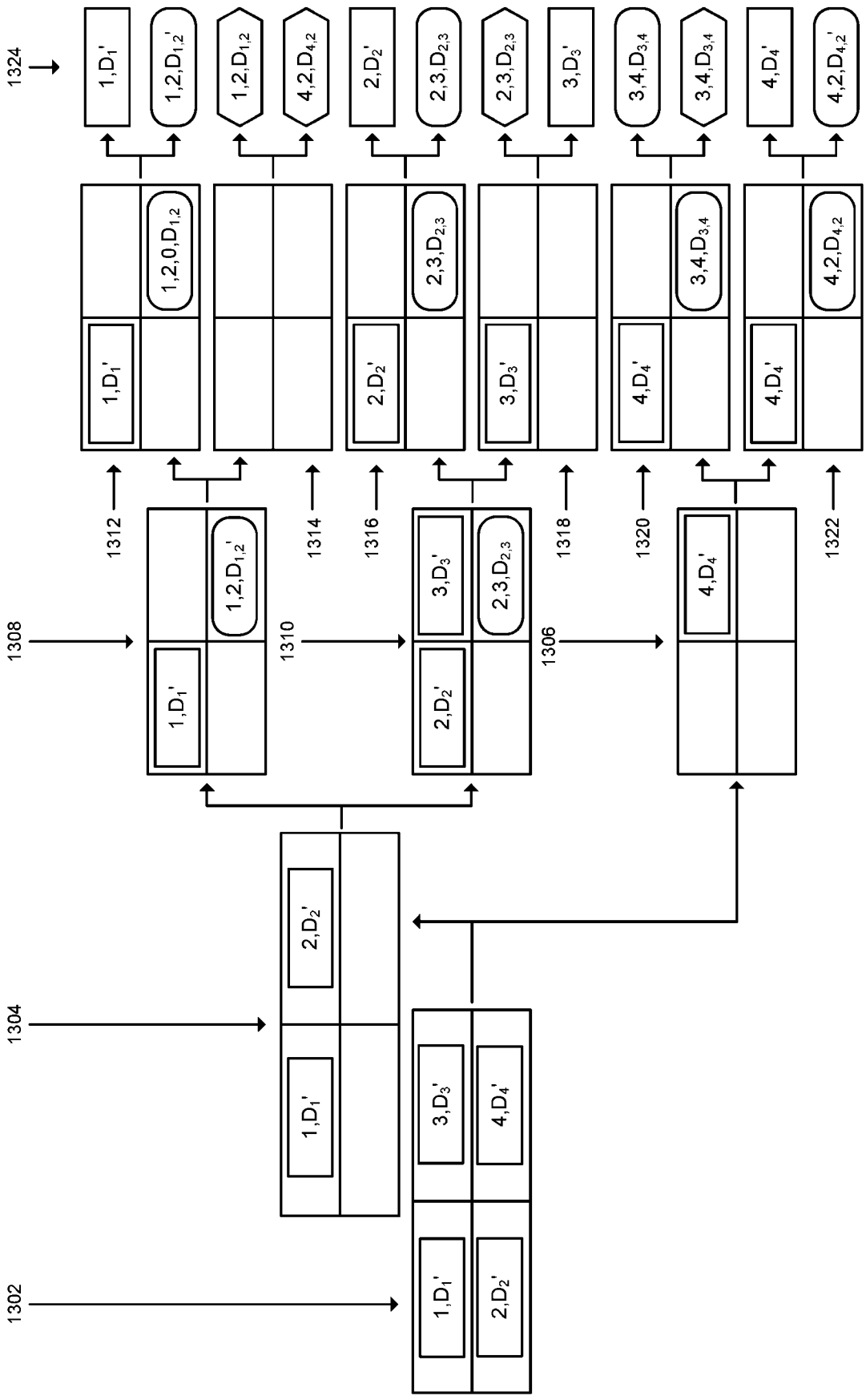
1302
1304
1306
1308
1310
1312
1314
1316
1318
1320
1322
1324
1,D1'
2,D2'
3,D3'
4,D4'
1,2,D1,2'
2,3,D2,3
3,4,D3,4
4,2,D4,2'
1,2,0,D1,2

FIG. 13

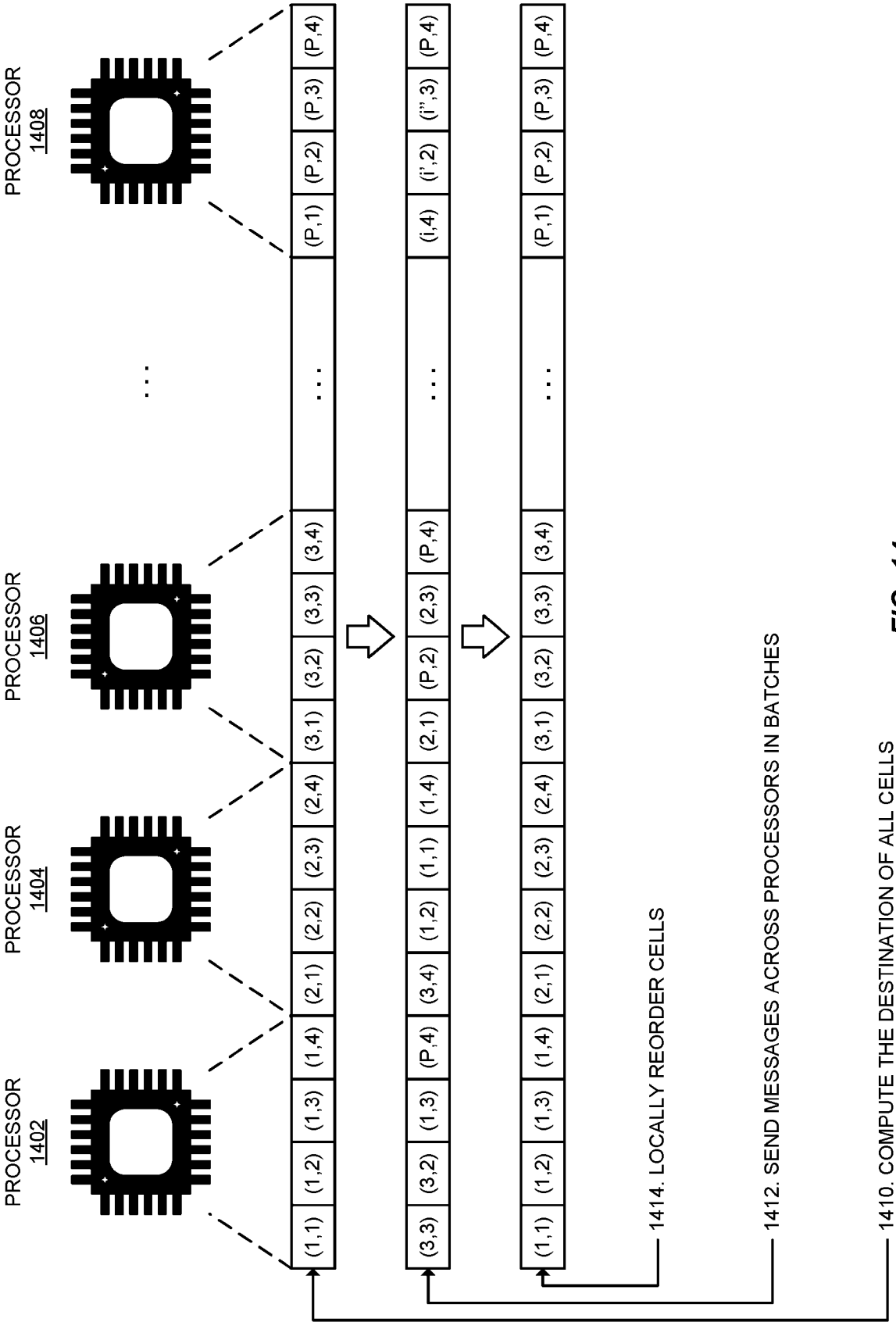
PROCESSOR 1402
PROCESSOR 1404
PROCESSOR 1406
PROCESSOR 1408
(1,1) (1,2) (1,3) (1,4) (2,1) (2,2) (2,3) (2,4) (3,1) (3,2) (3,3) (3,4) . . . (P,1) (P,2) (P,3) (P,4)
(3,3) (3,2) (1,3) (P,4) (3,4) (1,2) (1,1) (1,4) (2,1) (P,2) (2,3) (P,4) . . . (i,4) (i',2) (i",3) (P,4)
(1,1) (1,2) (1,3) (1,4) (2,1) (2,2) (2,3) (2,4) (3,1) (3,2) (3,3) (3,4) . . . (P,1) (P,2) (P,3) (P,4)
1414. LOCALLY REORDER CELLS
1412. SEND MESSAGES ACROSS PROCESSORS IN BATCHES
1410. COMPUTE THE DESTINATION OF ALL CELLS

FIG. 14

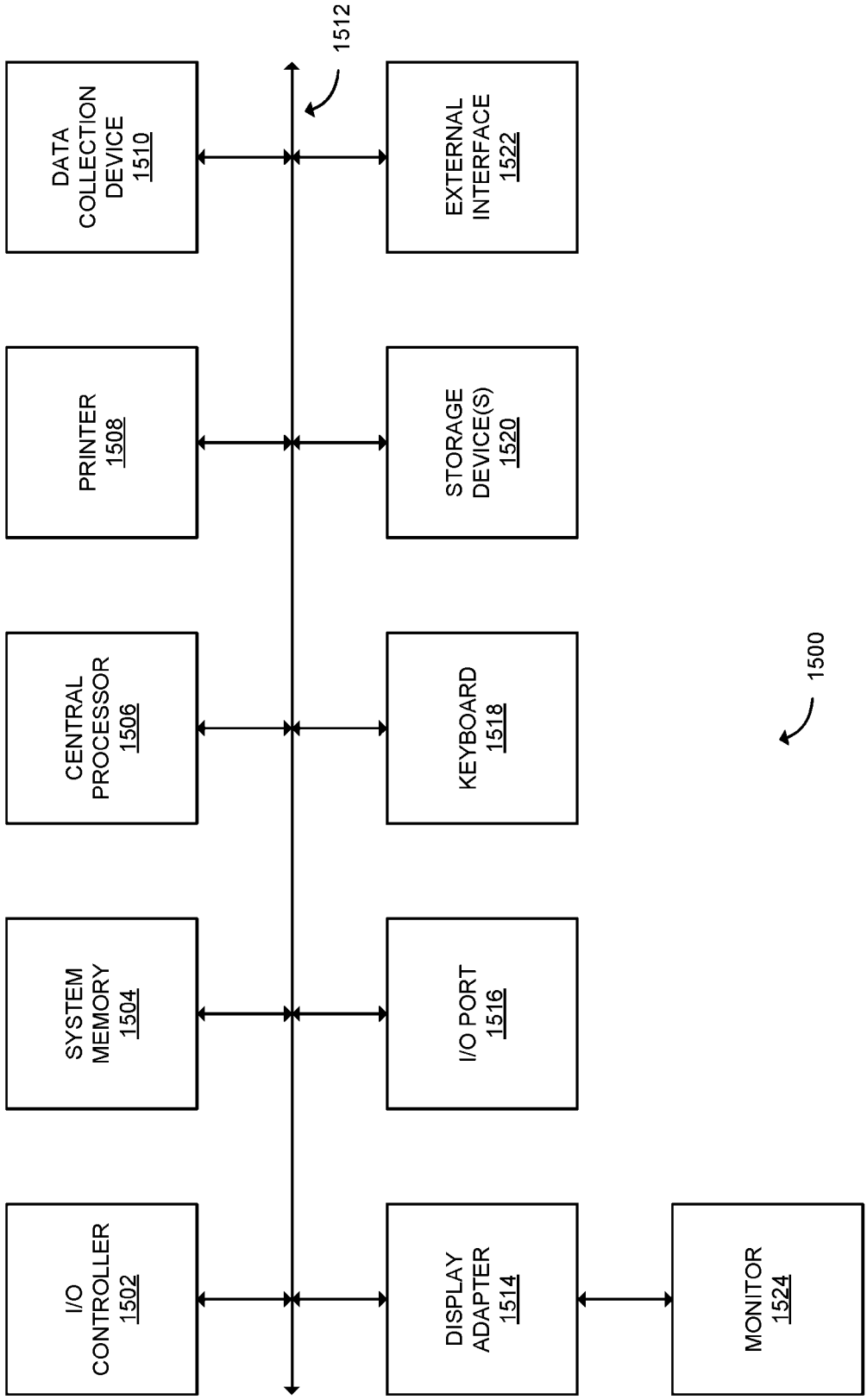
1512
DATA COLLECTION DEVICE 1510
EXTERNAL INTERFACE 1522
PRINTER 1508
STORAGE DEVICE(S) 1520
CENTRAL PROCESSOR 1506
KEYBOARD 1518
SYSTEM MEMORY 1504
I/O PORT 1516
I/O CONTROLLER 1502
DISPLAY ADAPTER 1514
MONITOR 1524
1500

FIG. 15

Method 1: k-Clique Detection-DFS

global $D \leftarrow \emptyset$;
$idx \leftarrow 1$;
for $v \in V$ **do**
  $T \leftarrow \{v\}$;
  $S \leftarrow Outgoing\_N(v)$;
  Recurse-DFS(v, T, S, idx)
end

Method 2: Recurse-DFS(v, T, S, idx)

if $idx \geq k$ **then**
  $D \leftarrow D \cup \{T\}$;
  return
end
if $S = \emptyset$ **then**
  return
end
for $v' \in S$ **do**
  $T' \leftarrow T + \{v'\}$;
  $S' \leftarrow S \cap Outgoing\_N(v')$;
  Recurse$(v', T', S', idx + 1)$;
end

FIG. 16

Method 3: k-Clique Detection-BFS

```
global D ← ∅;
for v ∈ V do
    T = {v};
    S = Outgoing_N(v);
    BFS(v, T, S);
end
```

Method 4: BFS(v, T, S)

```
(vs, Ts, Ss) ← ([v], [T], [S]);
(vs', Ts', Ss') ← (∅, ∅, ∅);
for idx ← 1 to k do
    if idx == k then
        for T ∈ Ts do
            D = D ∪ {T}
        end
        return
    end
    for (v, T, S) ∈ (vs, Ts, Ss) do
        if S ≠ ∅ then
            for v' ∈ S do
                vs' ← vs' + {v'};
                Ts' ← Ts' + {T + {v'}};
                Ss' ← Ss' + {S ∩ Outgoing_N(v')}
            end
        end
    end
    (vs, Ts, Ss) ← (vs', Ts', Ss');
    (vs', Ts', Ss') ← (∅, ∅, ∅);
end
```

1702

FIG. 17

Method 5: Scatter(v)

```
for e ∈ outgoing_edges(v) do
    e.Ts = v.Ts;
    e.Ss = e.Ss;
end
```

Method 6: Gather(v)

```
v.Ts = ∅;
v.Ss = ∅;
for e ∈ incoming_edges(v) do
    v.Ts = v.Ts ∪ e.Ts;
    v.Ss = v.Ss ∪ e.Ss;
end
```

Method 7: Apply(v)

```
for (T, S) ∈ (v.Ts, v.Ss) do
    T = T ∪ {v};
    S = S ∩ Outgoing_Neighbors(v);
    if S = ∅ then
        Remove T, S from v.Ts, v.Ss
    end
end
```

Method 8: $k-$ Clique Detection

```
for Parallelized each vertex v do
    v.Ts = {∅};
    v.Ss = {Outgoing_Neighbors(v)}
end
for i ∈ k iterations do
    for Parallelized each vertex v do
        Apply(v);
        Scatter(v);
        Gather(v);
    end
end
```

FIG. 18

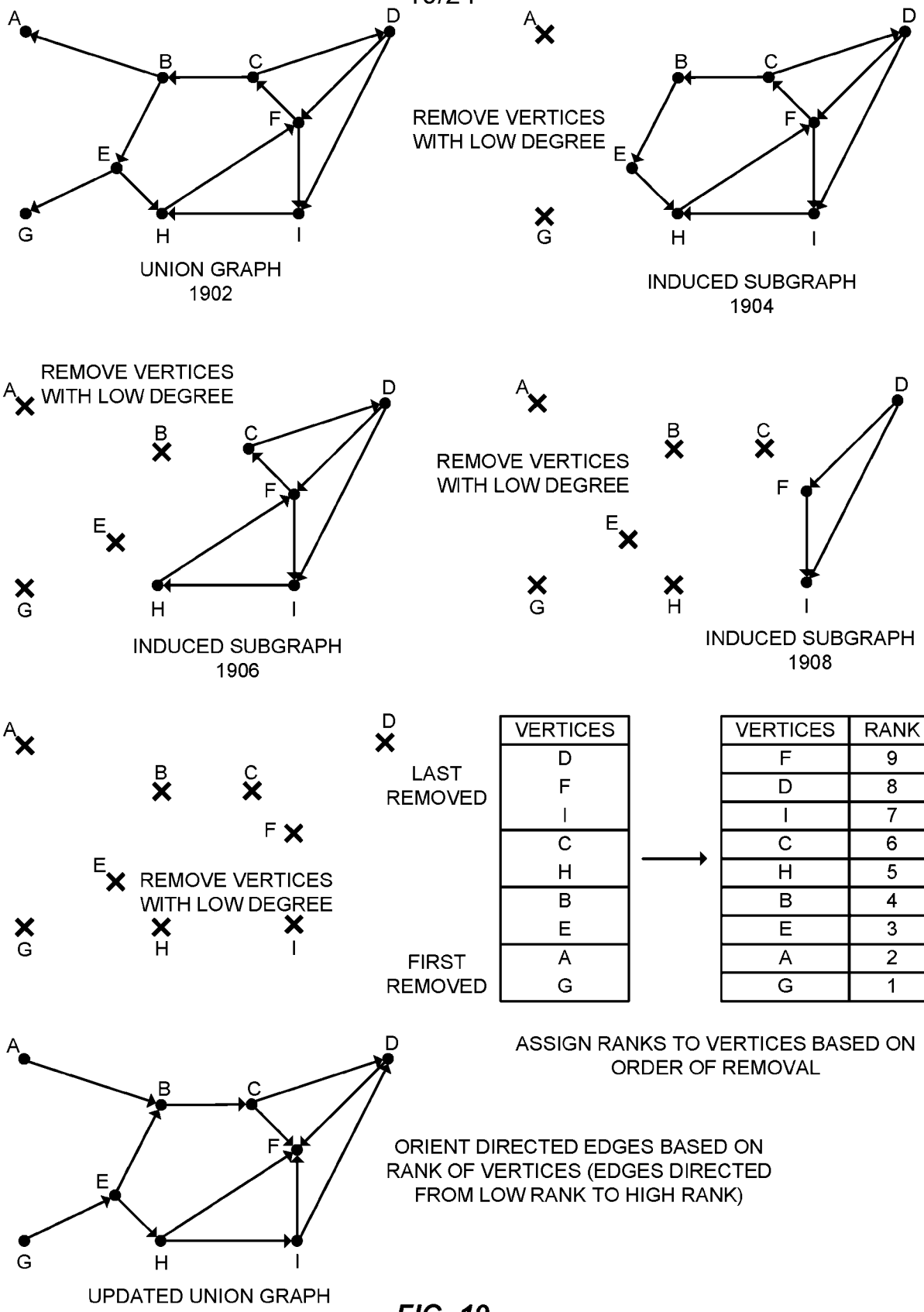
A
B
C
D
E
F
G
H
I
UNION GRAPH
1902
REMOVE VERTICES
WITH LOW DEGREE
INDUCED SUBGRAPH
1904
REMOVE VERTICES
WITH LOW DEGREE
INDUCED SUBGRAPH
1906
REMOVE VERTICES
WITH LOW DEGREE
INDUCED SUBGRAPH
1908
REMOVE VERTICES
WITH LOW DEGREE
LAST
REMOVED
FIRST
REMOVED
VERTICES
D
F
I
C
H
B
E
A
G
VERTICES
RANK
F 9
D 8
I 7
C 6
H 5
B 4
E 3
A 2
G 1
ASSIGN RANKS TO VERTICES BASED ON
ORDER OF REMOVAL
ORIENT DIRECTED EDGES BASED ON
RANK OF VERTICES (EDGES DIRECTED
FROM LOW RANK TO HIGH RANK)
UPDATED UNION GRAPH
1910

FIG. 19

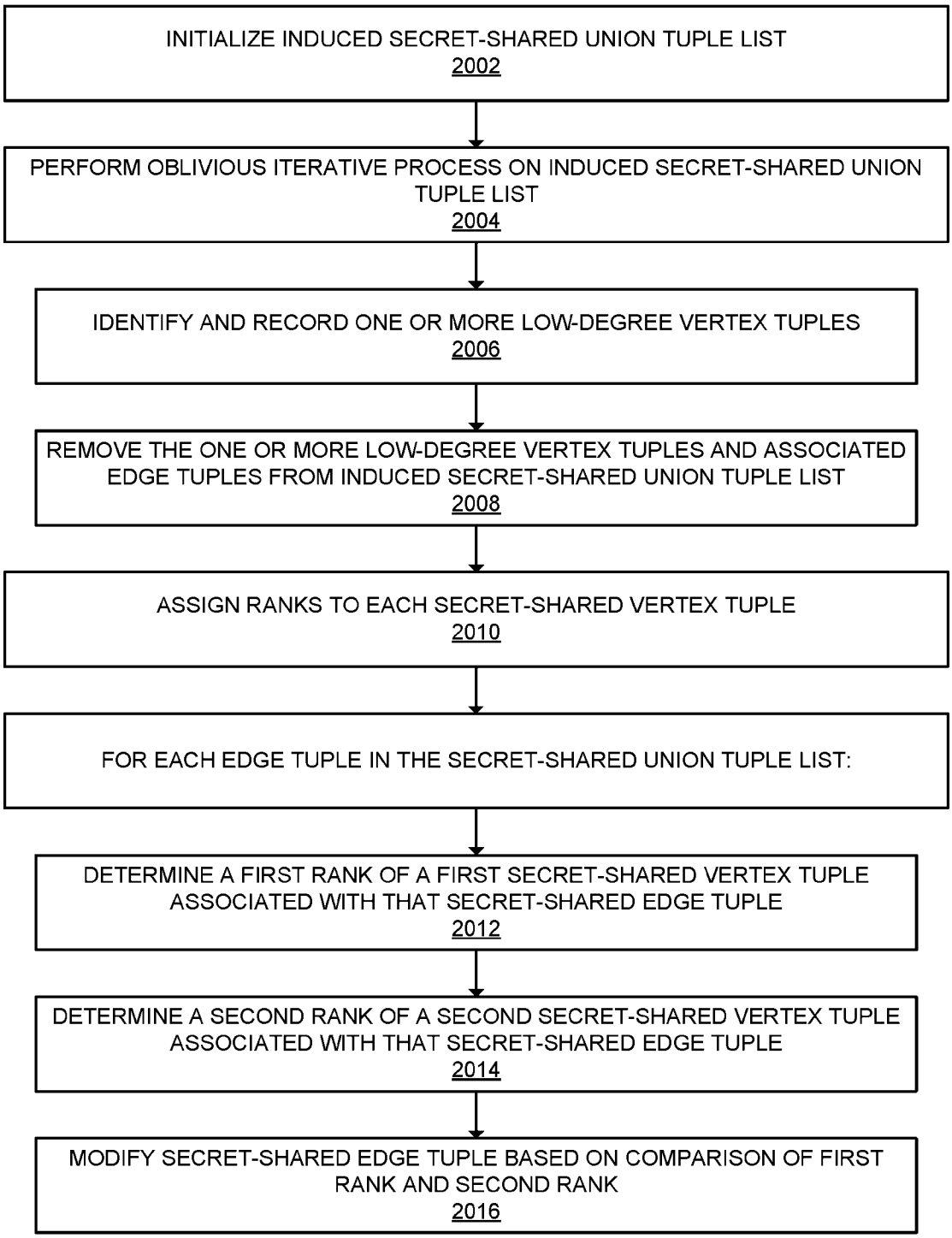
INITIALIZE INDUCED SECRET-SHARED UNION TUPLE LIST
2002
PERFORM OBLIVIOUS ITERATIVE PROCESS ON INDUCED SECRET-SHARED UNION TUPLE LIST
2004
IDENTIFY AND RECORD ONE OR MORE LOW-DEGREE VERTEX TUPLES
2006
REMOVE THE ONE OR MORE LOW-DEGREE VERTEX TUPLES AND ASSOCIATED EDGE TUPLES FROM INDUCED SECRET-SHARED UNION TUPLE LIST
2008
ASSIGN RANKS TO EACH SECRET-SHARED VERTEX TUPLE
2010
FOR EACH EDGE TUPLE IN THE SECRET-SHARED UNION TUPLE LIST:
DETERMINE A FIRST RANK OF A FIRST SECRET-SHARED VERTEX TUPLE ASSOCIATED WITH THAT SECRET-SHARED EDGE TUPLE
2012
DETERMINE A SECOND RANK OF A SECOND SECRET-SHARED VERTEX TUPLE ASSOCIATED WITH THAT SECRET-SHARED EDGE TUPLE
2014
MODIFY SECRET-SHARED EDGE TUPLE BASED ON COMPARISON OF FIRST RANK AND SECOND RANK
2016

FIG. 20

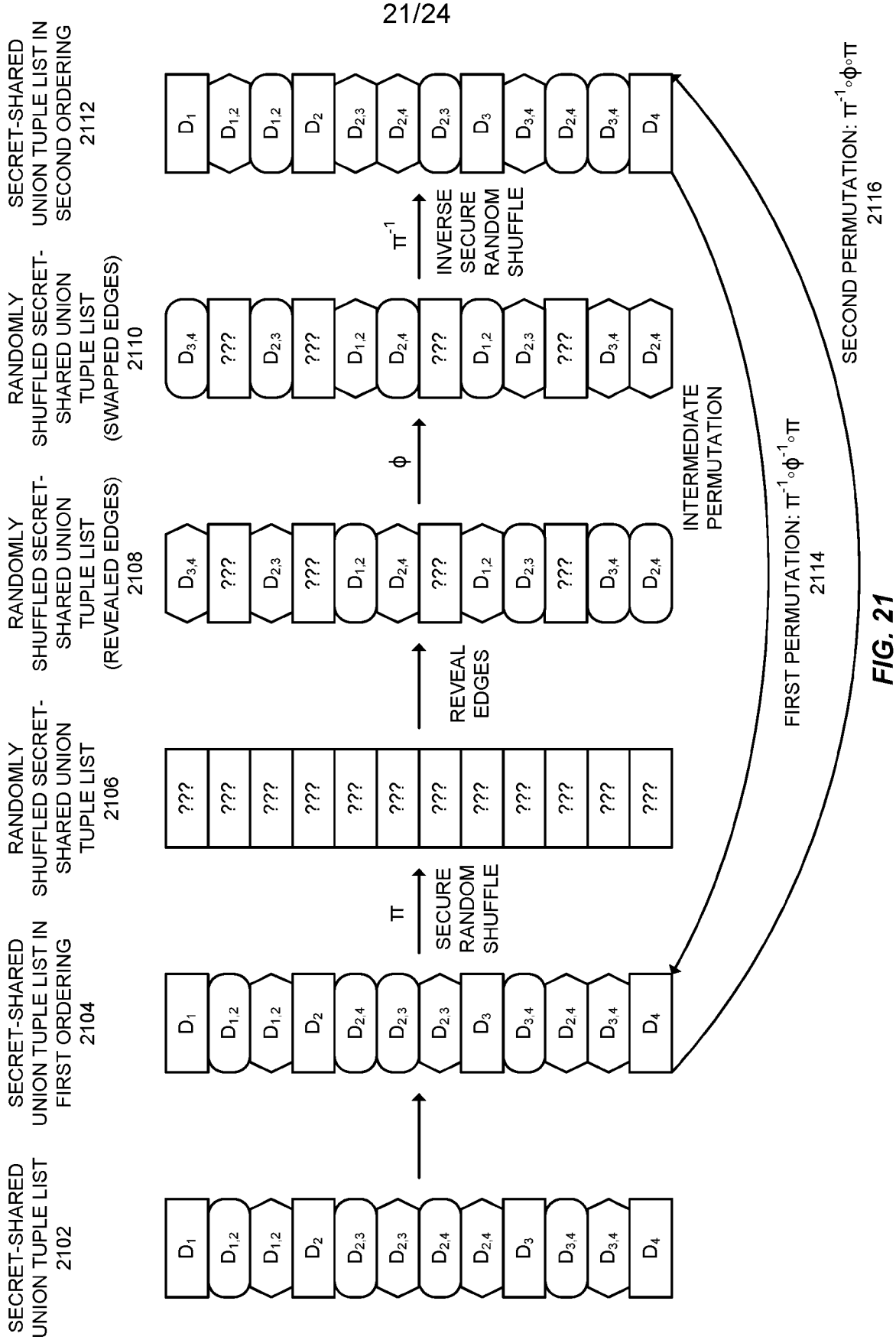
SECRET-SHARED UNION TUPLE LIST 2102
SECRET-SHARED UNION TUPLE LIST IN FIRST ORDERING 2104
RANDOMLY SHUFFLED SECRET-SHARED UNION TUPLE LIST 2106
RANDOMLY SHUFFLED SECRET-SHARED UNION TUPLE LIST (REVEALED EDGES) 2108
RANDOMLY SHUFFLED SECRET-SHARED UNION TUPLE LIST (SWAPPED EDGES) 2110
SECRET-SHARED UNION TUPLE LIST IN SECOND ORDERING 2112
π
SECURE RANDOM SHUFFLE
REVEAL EDGES
ϕ
π-1
INVERSE SECURE RANDOM SHUFFLE
INTERMEDIATE PERMUTATION
FIRST PERMUTATION: π-1∘ϕ-1∘π
2114
SECOND PERMUTATION: π-1∘ϕ∘π
2116

FIG. 21

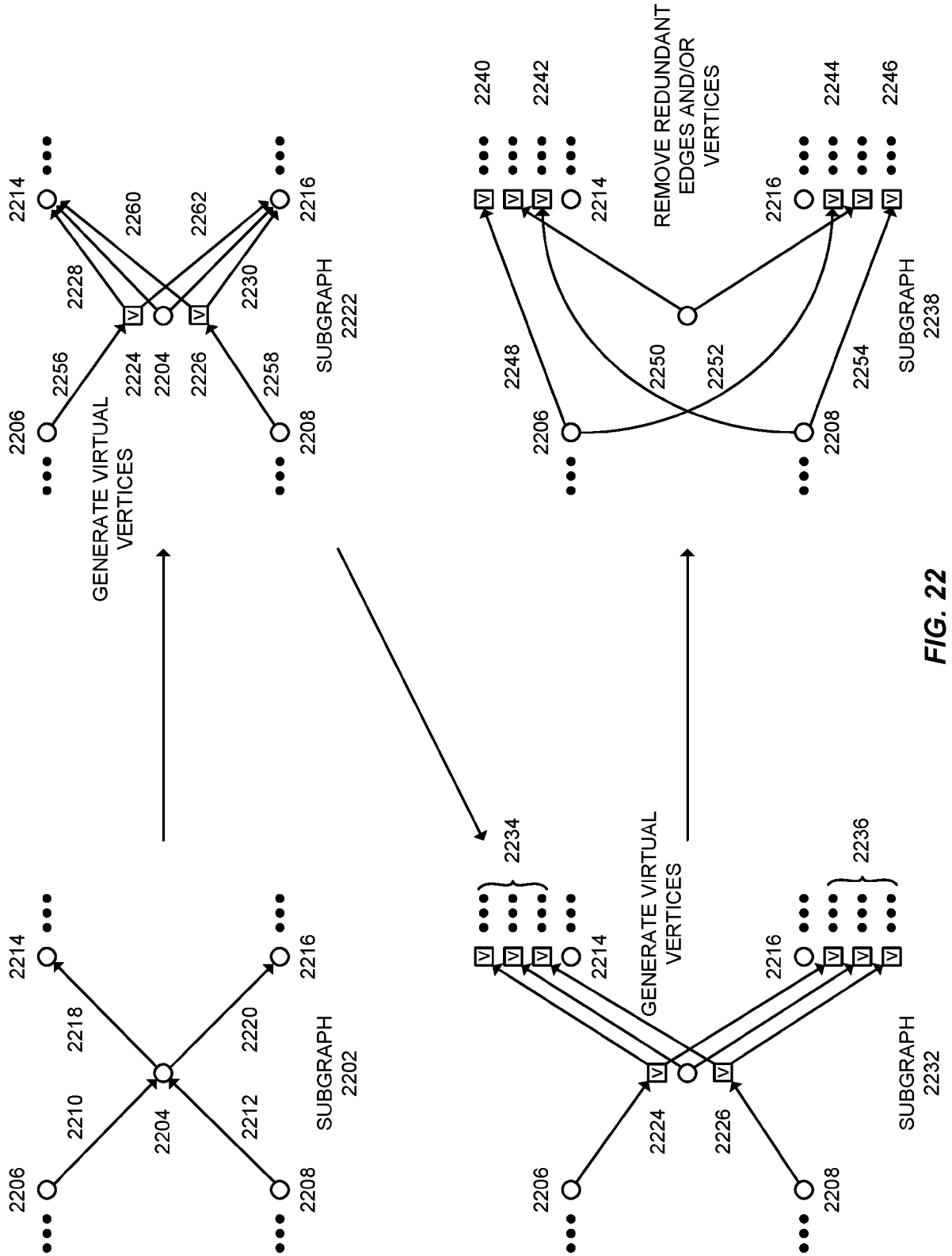
SUBGRAPH 2202
2206
2208
2210
2212
2214
2216
2218
2220
2204
GENERATE VIRTUAL VERTICES
SUBGRAPH 2222
2224
2226
2228
2230
2256
2258
2260
2262
REMOVE REDUNDANT EDGES AND/OR VERTICES
SUBGRAPH 2232
2234
2236
GENERATE VIRTUAL VERTICES
SUBGRAPH 2238
2240
2242
2244
2246
2248
2250
2252
2254

FIG. 22

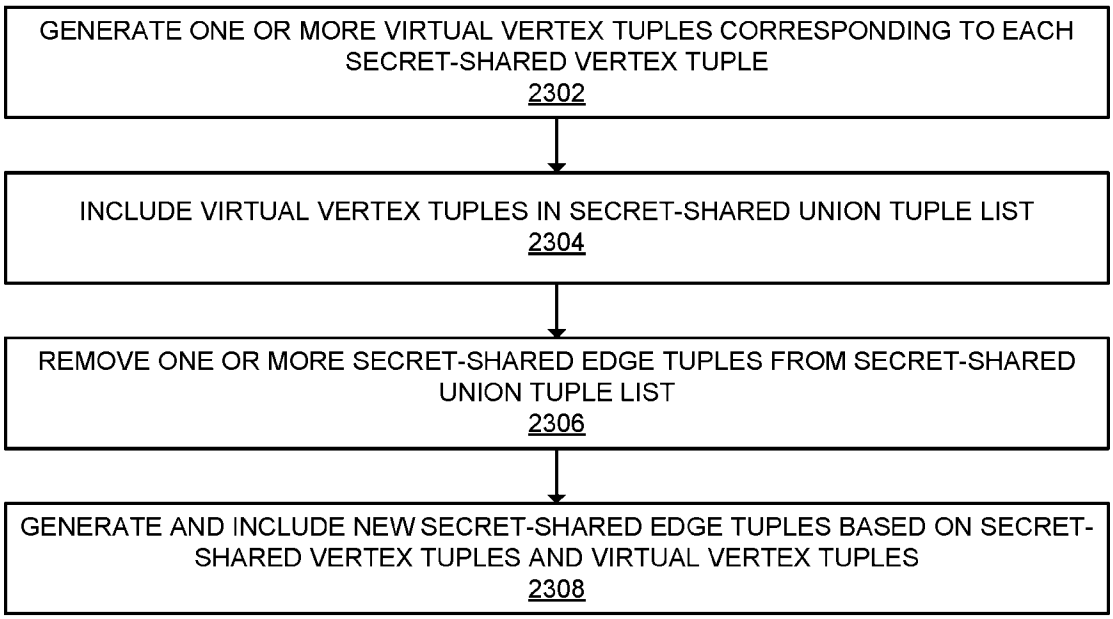
GENERATE ONE OR MORE VIRTUAL VERTEX TUPLES CORRESPONDING TO EACH SECRET-SHARED VERTEX TUPLE
2302
INCLUDE VIRTUAL VERTEX TUPLES IN SECRET-SHARED UNION TUPLE LIST
2304
REMOVE ONE OR MORE SECRET-SHARED EDGE TUPLES FROM SECRET-SHARED UNION TUPLE LIST
2306
GENERATE AND INCLUDE NEW SECRET-SHARED EDGE TUPLES BASED ON SECRET-SHARED VERTEX TUPLES AND VIRTUAL VERTEX TUPLES
2308

FIG. 23

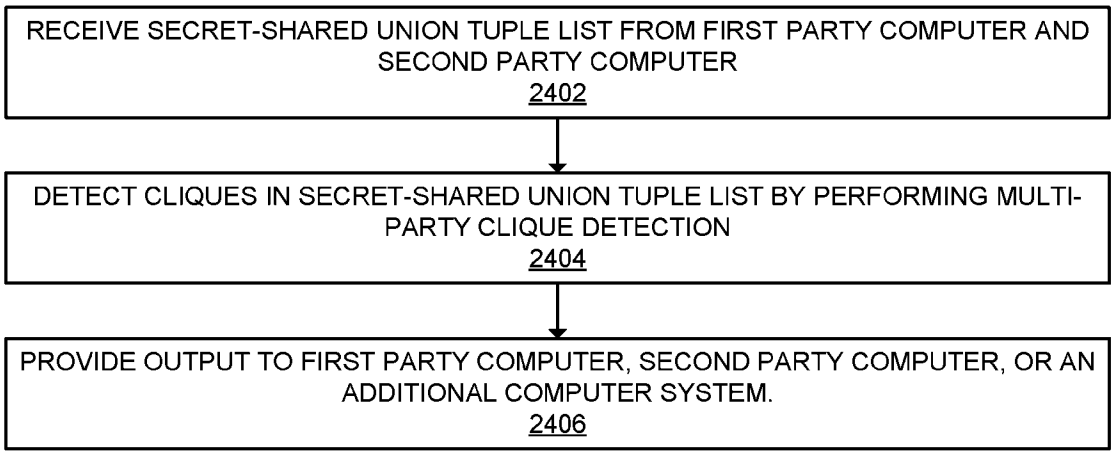
RECEIVE SECRET-SHARED UNION TUPLE LIST FROM FIRST PARTY COMPUTER AND SECOND PARTY COMPUTER
2402
DETECT CLIQUES IN SECRET-SHARED UNION TUPLE LIST BY PERFORMING MULTI-PARTY CLIQUE DETECTION
2404
PROVIDE OUTPUT TO FIRST PARTY COMPUTER, SECOND PARTY COMPUTER, OR AN ADDITIONAL COMPUTER SYSTEM.
2406

FIG. 24

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2023/075440

A. CLASSIFICATION OF SUBJECT MATTER

G06F 21/62(2013.01)i; **G06F 21/50**(2013.01)i

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F 21/62(2013.01); G06F 17/30(2006.01)

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

eKOMPASS(KIPO internal) & Keywords: privacy, preserving, clique, detection, secret-shared, union, tuple, list, party, graph, subgraph, vertex, edge, permutation, order, scatter, gather, apply

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	DANISH MEHMOOD et al., 'Privacy-preserving subgraph discovery', DBSec`12: Proceedings of the 26th Annual IFIP WG 11.3 conference on Data and Applications Security and Privacy, pages 161-176, 11 July 2012 pages 161-174; and figures 1-5	1-20
DA	KARTIK NAYAK et al., 'GraphSC: Parallel Secure Computation Made Easy', 2015 IEEE Symposium on Security and Privacy, pages 377-394, 17-21 May 2015 pages 377-393; and figures 1-6	1-20
A	BIN ZHOU et al., 'Preserving Privacy in Social Networks Against Neighborhood Attacks', 2008 IEEE 24th International Conference on Data Engineering, pages 506-515, 07-12 April 2008 pages 506-515; and figures 1-5	1-20
A	SAHAR MAZLOOM et al., 'Secure parallel computation on national scale volumes of data', Proceedings of the 29th USENIX Security Symposium, pages 2487-2504, 12-14 August 2020 pages 2487-2502; and figures 1-15	1-20

☑ Further documents are listed in the continuation of Box C. ☑ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"D" document cited by the applicant in the international application
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
27 February 2024	**27 February 2024**

Name and mailing address of the ISA/KR	Authorized officer
Korean Intellectual Property Office **189 Cheongsa-ro, Seo-gu, Daejeon** **35208, Republic of Korea**	**YANG, JEONG ROK**
Facsimile No. **+82-42-481-8578**	Telephone No. **+82-42-481-5709**

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2023/075440

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2017-0124218 A1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 04 May 2017 (2017-05-04) paragraphs [0057]-[0109]; claim 1; and figures 1-6	1-20

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/US2023/075440

Patent document cited in search report	Publication date (day/month/year)	Patent family member(s)	Publication date (day/month/year)
US 2017-0124218 A1	04 May 2017	US 10055510 B2	21 August 2018