GRAPES: A Software for Parallel Searching on Biological Graphs Targeting Multi-Core Architectures

Rosalba Giugno[1]*, Vincenzo Bonnici[2], Nicola Bombieri[2], Alfredo Pulvirenti[1], Alfredo Ferro[1], Dennis Shasha[3]

1 Department Clinical and Molecular Biomedicine, University of Catania, Catania, Italy, **2** Department Computer Science, University of Verona, Verona, Italy, **3** Courant Institute of Mathematical Sciences, New York University, New York, New York, United States of America

Abstract

Biological applications, from genomics to ecology, deal with graphs that represents the structure of interactions. Analyzing such data requires searching for subgraphs in collections of graphs. This task is computationally expensive. Even though multicore architectures, from commodity computers to more advanced symmetric multiprocessing (SMP), offer scalable computing power, currently published software implementations for indexing and graph matching are fundamentally sequential. As a consequence, such software implementations (i) do not fully exploit available parallel computing power and (ii) they do not scale with respect to the size of graphs in the database. We present GRAPES, software for parallel searching on databases of large biological graphs. GRAPES implements a parallel version of well-established graph searching algorithms, and introduces new strategies which naturally lead to a faster parallel searching system especially for large graphs. GRAPES decomposes graphs into subcomponents that can be efficiently searched in parallel. We show the performance of GRAPES on representative biological datasets containing antiviral chemical compounds, DNA, RNA, proteins, protein contact maps and protein interactions networks.

Citation: Giugno R, Bonnici V, Bombieri N, Pulvirenti A, Ferro A, et al. (2013) GRAPES: A Software for Parallel Searching on Biological Graphs Targeting Multi-Core Architectures. PLoS ONE 8(10): e76911. doi:10.1371/journal.pone.0076911

Editor: Lennart Martens, UGent/VIB, Belgium

Received June 3, 2013; **Accepted** August 26, 2013; **Published** October 22, 2013

Funding: DS's work was partly supported by U.S. National Science Foundation grants 0922738, 0929338, 1158273 and National Institutes of Health grants GM32877-21/22 and 2R01GM032877-25A1. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: The authors have declared that no competing interests exist.

* E-mail: giugno@dmi.unict.it

These authors contributed equally to this work.

Introduction

Biological sequences will always play an important role in biology, because they provide the representation of a fundamental level of biological variability and constitute "evolution's milestones" [1]. However, technological advances have led to the inference and validation of structured interaction networks involving genes, drugs, proteins and even species. An important task in cheminformatics, pharmacogenomics and bioinformatics is to deal with such structured network data. A core job behind complex analysis is to find all the occurrences of given substructures in large collections of data. This is required for example in (i) network querying [2–5] to find structural motifs and to establish their functional relevance or their conservation among species, (ii) in drug analysis to find novel bioactive chemical compounds [6,7], and (iii) in understanding protein dynamics to identify and querying structural classification of protein complexes [8].

The networks consist of vertices as basic elements (i.e., atoms, genes, and so on) and edges describe their relationships. All cited applications build on the basic problem of searching a database of graphs for a particular subgraph.

Formally, graph database searching is defined as follows. Let $D = \{G_1, \ldots, G_m\}$ be a database of connected graphs. A graph G_i is a triple (V_i, E_i, L_i). $V_i = \{v_j\}, j = 1, \ldots, n$ is the set of vertices in G_i. $E_i = \{(v_j, v_k)\}$, $j, i = 1, \ldots, n$, is the set of edges connecting vertices in V_i. We consider edges to be undirected. The degree of a vertex v_j is the number of edges connected to it. Each vertex may have a label, representing information from the application domain.

Let U be the set of all possible labels. Let $M : V \rightarrow U$ be the function that maps vertices to labels. Let $L_i = \{M(v_i)$, for all $v_i \in V_i\} \subseteq U$ be the set of labels of G_i. For each graph in the database, each vertex has a unique identifier, but different vertices may have the same label. Figure 1 shows an example of a database of graphs and a query. In this case $U = \{A, B, C\}$ coincides with L_1, L_2, L_3 and L_4. Examples of mapping may be $M(v_0) = C$ in G_1 and $M(v_0) = B$ in G_2.

Two graphs $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$ are *isomorphic* if and only if there exists a bijective function $I : V_1 \rightarrow V_2$ mapping each vertex of G_1 to a vertex of G_2 such that $(u,v) \in E_1$ if and only if $(I(u), I(v)) \in E_2$ and vice versa. We must respect also the *compatibility* of the labels of each mapped items, such that $\forall v \in V_1\ M(v) = M(I(v))$.

A *subgraph isomorphism* (hereafter, also called subgraph matching or matching) of $Q = (V_q, E_q, L_q)$ in $G(V, E, L)$ is an injective function $I : V_q \rightarrow V$ such that $(u,v) \in E_q$ if and only if $(I(u), I(v)) \in E$ and $M(u) = M(I(u))$ and $M(v) = M(I(v))$. Note that, there may be an edge $(u', v') \in E$ without any corresponding edge in E_q. Given a set of graphs $D = \{G_1, G_2, .., G_m\}$ and a graph Q, called query, the problem consists of identifying the graphs in D containing Q as a subgraph together with all the locations, called occurrences, of Q in those graphs. This problem is called *graph sub-isomorphism* and the complexity of all existing exact approaches is exponential. In Figure 1, colored vertices and thick edges highlight the subgraph isomorphisms of Q in the set of graphs.

Much research has been done to try to reduce the search space by filtering away those graphs that do not contain the query. This is achieved by indexing the graphs in D in order to reduce the required number of subgraph isomorphism tests. Because graphs are queried much more often than they change, indexes are constructed once by extracting structural features of graphs in a preprocessing phase. Features are then stored in a global index. Later, given a query graph, the query features are computed and matched against those stored in the index [9]. Graphs having the features of the query are *candidate* to contain the query. The set of candidates is then examined by a subgraph isomorphism algorithm and all the resulting matches are reported. The time spent searching on these graphs is exponential in the graph size. Heuristic (sub)graph-to-graph matching techniques [10,11] are used to solve this exponential step. Other tools based on the identification of discriminant characteristics of graphs [12,13] are used.

Almost all solutions build the index on subgraphs (i.e., paths [14–16], trees [17,18], graphs [19]) of size not larger than ten vertices to save on time and space.

The memory footprint and the time spent for building the index may be prohibitive even when applied to small subgraphs encouraging the use of compression heuristics. SING [15] stores paths of vertex identifiers in its index as well as the label of the first node in each path. GraphGrepSX [16] stores the paths in an efficient data structure called trie which compacts common parts of features.

Other systems use data mining techniques, which have been applied to store only non-redundant frequent subgraphs (i.e., a subgraph is redundant when it filters as much as its subgraphs) [18–20]. GraphFind [21] uses paths instead of subgraphs but reduces the number of indexed paths by using low-support data mining techniques such as Min-Hashing [22]. In spite of these heuristic techniques, index construction remains an expensive step.

The topology of the features affects both construction time and query effectiveness. TreePi [17] pioneered the use of trees as features. The authors describe a linear-time algorithm for computing the canonical labeling of a tree. They experimentally show that tree features capture topological structures well. Therefore, using trees may result in a good compromise between construction efficiency of query filtering effectiveness. As shown by the authors, a unique center can be defined for a tree. Consequently the distance (shortest path) between pairs of features in a graph can be computed. TreePi uses an additional pruning rule based on distances between features to improve the quality of the match. Tree+δ [23] uses as features both trees and a restricted class of small graphs to improve the filtering performance. A most recent contribution in the literature which uses trees is CT-index [24]. In particular it hashes the indexed features which are a combination of trees and cycles. Trees and cycles are represented by their canonical form and mapped into the fingerprint by the hashing function. The loss of information caused by the use of hash-key fingerprints seems to be justifiable by the compact size of the indexing as long as the amount of false positive does not increase significantly due to collisions. The matching algorithm is driven by a fixed ordering of the query vertices. The vertex sequence is built looking for connectivity and using a priority based on label frequencies respect to the whole graphs database. Unfortunately, as for graphs, enumerating all trees of bounded size still produces a large number of features.

Therefore, while the above approaches have good performance on medium-small size graphs, they are not suitable when the size and density of graphs increase. Some approaches have been proposed to deal with very large graphs [25]. In such cases, the massive data sets are decomposed and distributed onto cluster-based architectures. Nevertheless, the time overhead of the distributed computational model is high compared to the overhead on a symmetric multiprocessor (SMP) architecture.

In any parallel setting, multiple instances of the state of the art searching software can be run on the CPU cores, each one on a disjoint database graphs. These do not work when the database consists of a single large graph (as is often the case with biological networks). In addition, the parallel search may be imbalanced even on databases of many graphs, due to the differences among graph sizes and densities.

This paper presents GRAPES, a parallel algorithm for biological graph searching on multicore architectures. GRAPES is a parallelized version of the index-based sequential searching algorithms proposed in [15,16]. It introduces new algorithmic strategies tailored to parallelism. The main characteristics of GRAPES when compared to [15,16] in Table 1, are the following:

- Each single graph of the database is indexed in parallel by n threads, where n is the number of processing units. Each index construction is independent and dynamically distributed to the threads. This guarantees load balancing and minimal synchronization overhead, regardless of the number, size and density of the database graphs.
- GRAPES uses paths of bounded length as features, and *Trie* structures to represent them. Path prefix sharing in tries reduces data redundancy thus achieving a more compact representation of the index. The index also stores the number of occurrences of the features and their locations.
- GRAPES implements a filtering phase, which consists of the *trie* comparison of all query graph features against the pre-computed global *trie* of the database.
- The matching phase applies the well established graph isomorphism algorithm proposed in [10], with the exception that only suitable subgraphs within candidate graphs (i.e., that pass the filtering phase) are analyzed. This runs in parallel both within each subgraph and among subgraphs thanks to the indexes built in the first phase.

The experimental results show that GRAPES efficiently exploits the computing power of the multicore architectures by guaranteeing scalability and load balancing. The experimental results also show that GRAPES provides better performance than sequential solutions run in parallel on databases of graphs, and provides fast searching results even on databases of graphs that would be intractable by applying sequential searching algorithms.

The paper is organized as follows. Section presents GRAPES in detail. Section presents the experimental results while Section is devoted to concluding remarks.

Methods

Software Overview

GRAPES implements a parallel search algorithm for databases of large graphs targeting multicore architectures. It consists of the following three phases (see Figure 2):

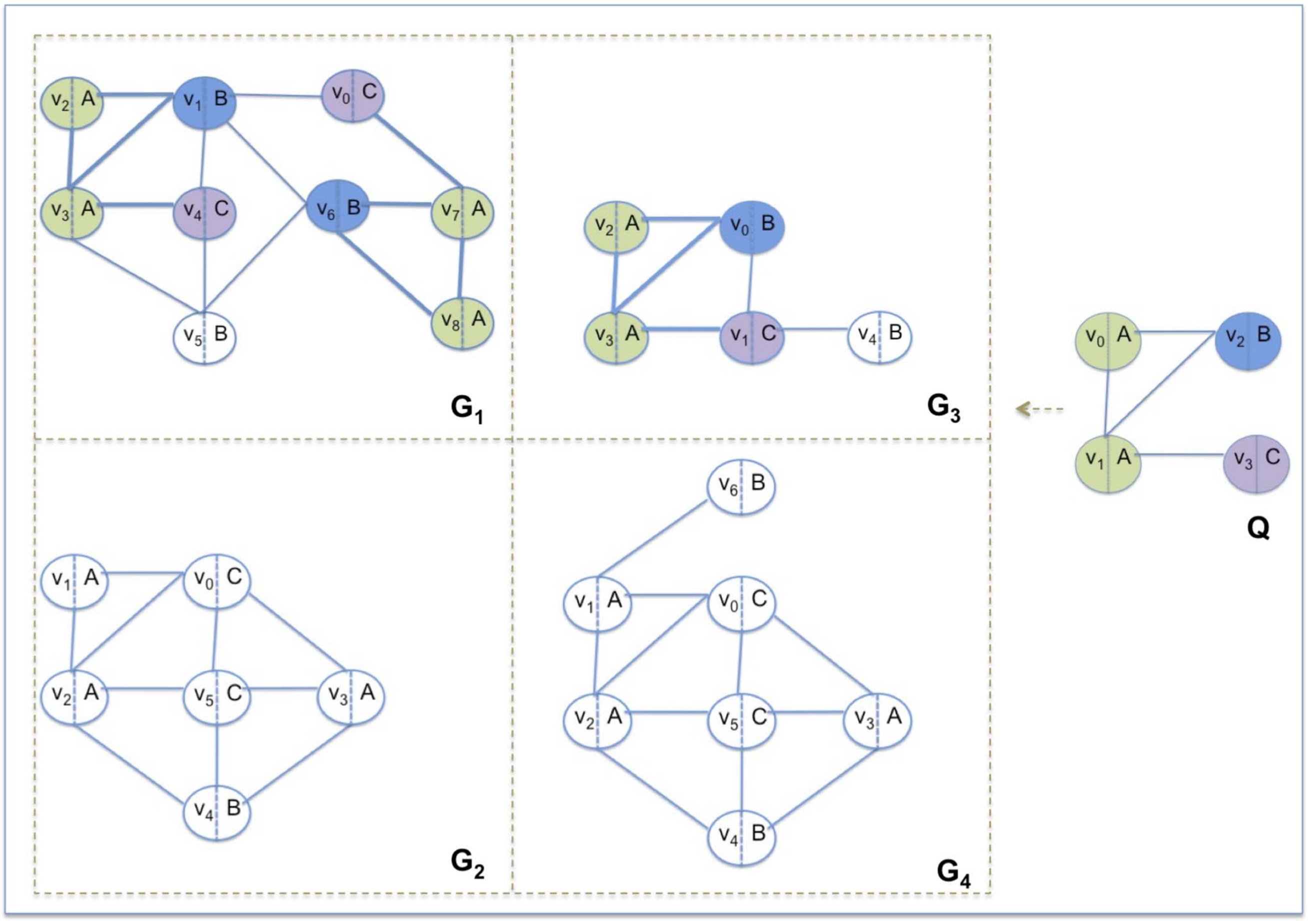

Figure 1. Graph database and query. The database is composed by four graphs G_1, G_2, G_3 and G_4. Q is the query. The occurrences of Q in the graphs are shown by colored vertices and bold edges.
doi:10.1371/journal.pone.0076911.g001

- *Building phase*. Given a database of graphs, the building phase constructs an index called a *global trie*. The global trie consists of the set of substructures (paths of length up to a fixed value called l_p), hereafter called *features*, of the database graphs, as described in Section.
- *Filtering phase*. Given a query, GRAPES builds an index of the query ($Trie_Q$) and compares that index to the features of the graphs. It discards graphs that do not contain the query features. In the remaining graphs, which are called *candidate graphs*, it filters those parts (subgraphs) that are not present in the query graph. This step may disconnect the candidate graphs by decomposing them into a set of connected components, hereafter called *candidate connected components*, as described in Section.
- *Matching phase*. A subgraph matching algorithm (i.e., [10]) is run to find all the occurrences of the query in each connected component in parallel, as described in Section.

Building Phase: Extracts the Features and Constructs a Trie Index

This phase builds a trie index of the whole database. The index is built in parallel by the n threads run on the CPU cores. For each graph, GRAPES groups the vertices in lists, all the vertices in the same list have the same label (see Figure 2). The lists are dynamically distributed to the threads. Each thread instantiates a partial index ($trie_i$), which stores all feature, i.e. all paths of labels, information starting from the vertices of the assigned list. Each partial index is built independently and asynchronously. A thread that collects all the information of the assigned list proceeds with a further list related to the same graph or even to another graph. This guarantees load balancing on the CPU cores regardless of the number, size, and density of the database graphs.

For each vertex v_i, the thread stores all label paths starting from v_i, $(M(v_i),M(v_j),...,M(v_k))$ containing up to l_p vertices. Recalling that different vertices may have the same label, the thread records the number of time each feature (which is a topological pattern, here a sequence, of labels) appears in the graph and the identification (v_i) of the first vertex of the paths. For example, the feature (C,C,A) of length two in graph G_2 of Figure 1 occurs 5 times (see Figure 3 to locate the feature indexed in the trie). The paths mapped to it are $(v_0,v_5,v_3),(v_0,v_5,v_2),(v_5,v_0,v_2),(v_5,v_0,v_1),(v_5,v_0,v_3)$. The identifications of the first vertices of such paths are v_0 and v_5.

After it completes its collection phase, each thread stores all label path features in the partial trie. Each node of the trie tree of height k is a path of labels $(M(v_i),M(v_j),...,M(v_k))$, of length k. The height of the trie is l_p. All the descendants of a node share a common prefix of the feature associated with that node. Each node in the trie links to (i) the list of graphs G_i containing the feature associated with that node, (ii) the number of times the

Table 1. GRAPES vs GraphGrepSX and SING.

	GRAPES	GraphGrepSX	SING	Description
TRIE indexing	x	x		Compact indexing. Efficiency on indexing and filtering
	x	x		
Path indexing	x	x	x	Lighter than subgraphs or trees
Store path node	x		x	More filtering power
Parallel Indexing and Matching	x			Necessary for biological data size and density
Connected components	x			Load balancing onto CPU cores

List of the new GRAPES components together with the components of GRAPES taken from GraphGrepSX [16] and SING [15].
doi:10.1371/journal.pone.0076911.t001

feature occurs in each G_i (that is the number of paths $(v_i,v_j,...,v_k)$ mapped to $(M(v_i),M(v_j),...,M(v_k))$ in each G_i), and (iii) the starting vertices v_i of each such path.

Then, all the tries partially built by each thread are merged into the final global trie (this is done sequentially since it is very short and fast). For example, Figure 3 depicts the global trie of a database composed of a single graph. Let's G_2 of Figure 1 be such a graph. The features are paths of length two ($l_p = 2$). Note that the root is virtual (does not correspond to any vertex). All paths in the trie (starting from nodes at level 1 to the leaves) are features in G_2. All nodes link to a list of information as described before (the figure shows the result for only some nodes). Figure 4 depicts the list of information for the third node in path (B,A,A) for all graphs in Figure 1.

Filtering Phase: Reduce the Search Space

The filtering phase consists of two steps: (i) filter away those graphs that cannot have a match with the query graph (i.e., candidate generation) and (ii) filter away the portions of potentially matching graphs that have no chance of matching the query graph (i.e., candidate decomposition into connected components).

1. Given a query Q, GRAPES extracts all features from Q. Note that features in Q are paths up to length l_p, the same length value used to construct the index for the graphs of the databases. All features are stored in a trie ($Trie_Q$).
2. GRAPES matches the trie of the query Q against the *global trie*. The algorithm discards those graphs that either have a feature with an occurrence number less than the occurrence number of the query or do not contain some features of the query graph. For example in Figure 1, G_2 is discarded because the feature (B,A,A) appears once in G_2 and twice in the query.
3. For each vertex u of the query graph Q and a potentially matching vertex v_j in graph G_i, any feature starting from u should also start from v_j. Otherwise v_j cannot be a match. GRAPES achieves this by looking at the tries of the starting vertices v_i of the occurrences of features in the graphs. Thus, for the paths in the trie of Q matching paths in the trie of D, and for all of those starting from the same vertex u in Q, all occurrences of the matching features in those graphs G_i must also start from the same vertex v_j. Otherwise, G_i cannot match Q starting at v_j. Figure 1 depicts such a case, in which G_4 contains (B,A,A) twice as does the query. However, in the query the paths start from a single node, i.e. v_2, whereas, in G_4, the two paths start from two vertices, i.e. v_4 and v_6.
4. GRAPES retrieves from the trie the identities of the starting vertices of the paths in the data graphs that match the query features. The net effect is that GRAPES extracts all maximally connected components in the graphs involving only possible vertices. This is done in parallel by dynamically assigning each candidate graph to threads for guaranteeing load balancing. For example, when searching for Q in the database of Figure 1, GRAPES reduces the search space by returning only the connected components depicted in Figure 5.

This approach is particularly helpful when the graphs in the database are very large. Because the filtering step decomposes the graphs with respect to the query, large graphs become a set of smaller components, greatly reducing the load on the exponential matching phase as well as enabling that phase to be parallelized. Parallel operation (for index creation, index searching, and heuristic subgraph isomorphism) and the decomposition of graphs into small connected components are the main reasons GRAPES is so fast.

Matching Phase: Finds All Occurrences of the Query in the Graphs in the Database

GRAPES runs multiple instances (i.e., one for each thread) of the combinatorial subgraph isomorphism algorithm *VF2* [10] on the connected components resulting from the filtering phase.

VF2 is a combinatorial search algorithm which induces a search tree by branching states. It uses a set of topological feasibility rules and a semantic feasibility rule, based on label comparison, to prune the search space. At each state if any rule fails, the algorithm backtracks to the previous step of the match computation.

The size and the density of the generated connected components may be different. As a consequence, distributing each connected component as a whole for matching to each thread may cause load imbalance. On the other hand, many instances of the matching algorithm on the same connected component likely lead to redundant visits in the same search space.

GRAPES applies a heuristic to select (i) the number of searching instances to be run in parallel on a single connected component at a time, and (ii) the starting nodes for the searching instances of each connected component. The heuristic relies on the lists of *matchable vertices* that have been extracted for each vertex of the query during the filtering phase (see step 3 of Section). There is one list of matchable vertices per connected component. Longer lists are processed first, because one instance of the matching algorithm can be run in parallel starting from each node of the list.

Availability and Requirements

Project name: GRAPES

Project home page: http://ferrolab.dmi.unict.it/grapes.html, https://code.google.com/p/grapessmp

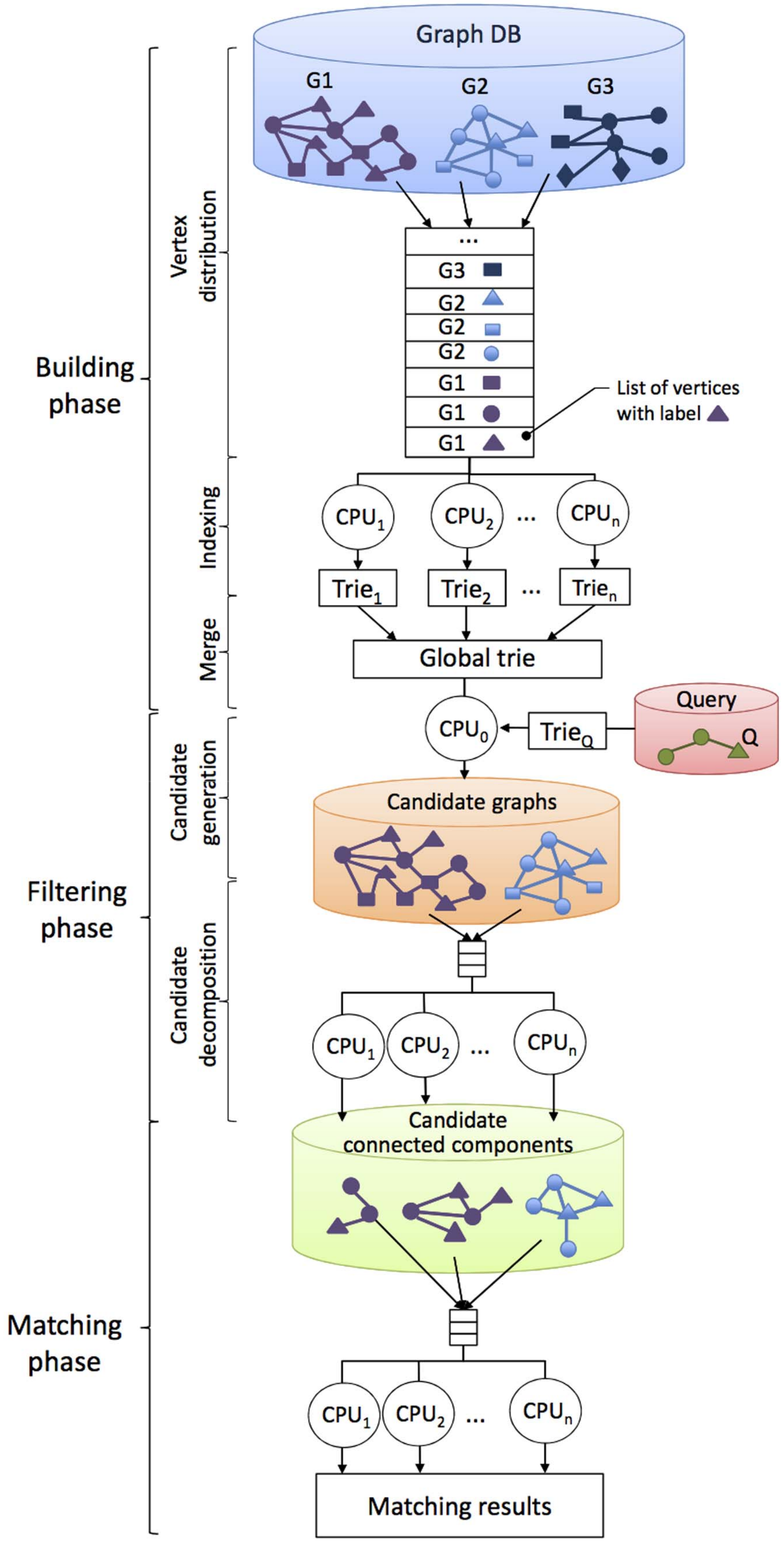

Figure 2. Schema of GRAPES.
doi:10.1371/journal.pone.0076911.g002

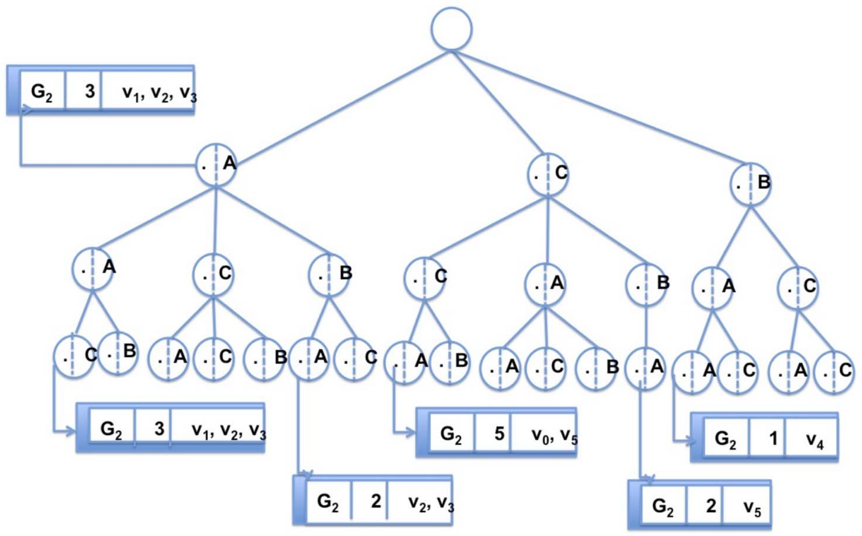

Figure 3. The trie data structure used as indexing in GRAPES. The index for the graph G_2 of Figure 1. A trie stores all features of length 2 of G_2. Each node is associated to a feature, that is the path in the trie up to this node. For each node in the trie, a list of information such as the number of times the feature appears in G_2 (equally the number of paths in G_2 mapped to it) and the identifications of the first nodes of such paths is stored.
doi:10.1371/journal.pone.0076911.g003

Operating system(s): Unix/Linux x86/x86-64

Programming language: C++

Other requirements: GRAPES runs in parallel on shared memory multi-core architectures

License: GRAPES is distributed under the MIT license.

Results and Discussion

GRAPES has been tested on four databases of biological graphs:

- *AIDS*, the standard database of the Antiviral Screen Dataset [26] of the topological structures of 40,000 molecules. Graphs in the dataset have 62 unique labels in total. The average number of vertices per graph is 44.98 with a standard deviation of 21.68. The average (resp. standard deviation) degree per vertex is 4.17 (resp. 2.28) and the average (resp. standard deviation) number of labels is 4.36 (resp. 0.86). They are small sparse graphs whose maximum size is 245.
- *PDBS*, a database of 600 graphs representing DNA, RNA and proteins having up to 16,431 vertices and 33,562 edges [27]. The dataset has been converted to graphs by using the Ball conversion library available at www.ball-project.org/; original structures can be downloaded from www.fli-leibniz.de/ImgLibPDB/pages/entry_list-all.html. As reported in [27],

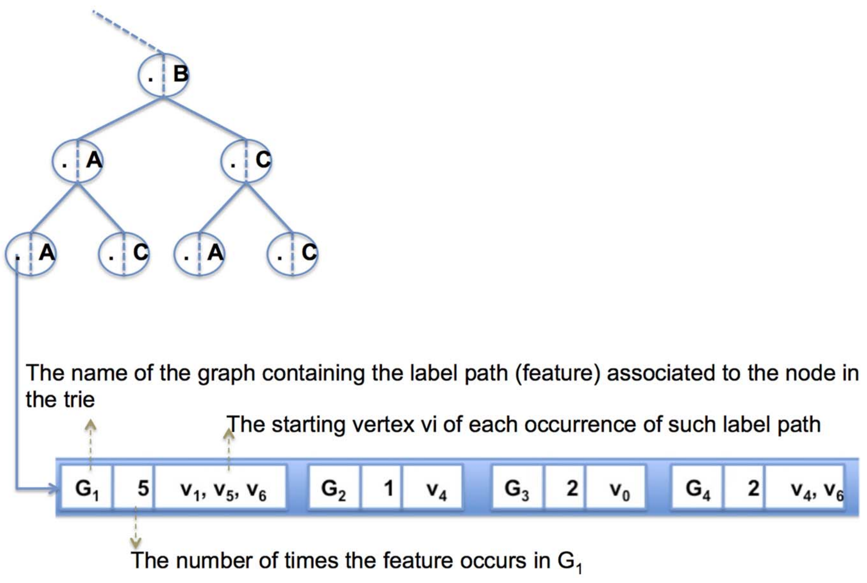
The name of the graph containing the label path (feature) associated to the node in the trie
The starting vertex vi of each occurrence of such label path
G1 | 5 | v1, v5, v6
G2 | 1 | v4
G3 | 2 | v0
G4 | 2 | v4, v6
The number of times the feature occurs in G1

Figure 4. The information stored in a node of a GRAPES index (trie) for graphs of Figure 1. Each node in the trie links to (i) the list of graphs Gi containing the feature associated with that node, (ii) the number of times the feature occurs in each Gi, and (iii) the starting vertices vi of each such path.
doi:10.1371/journal.pone.0076911.g004

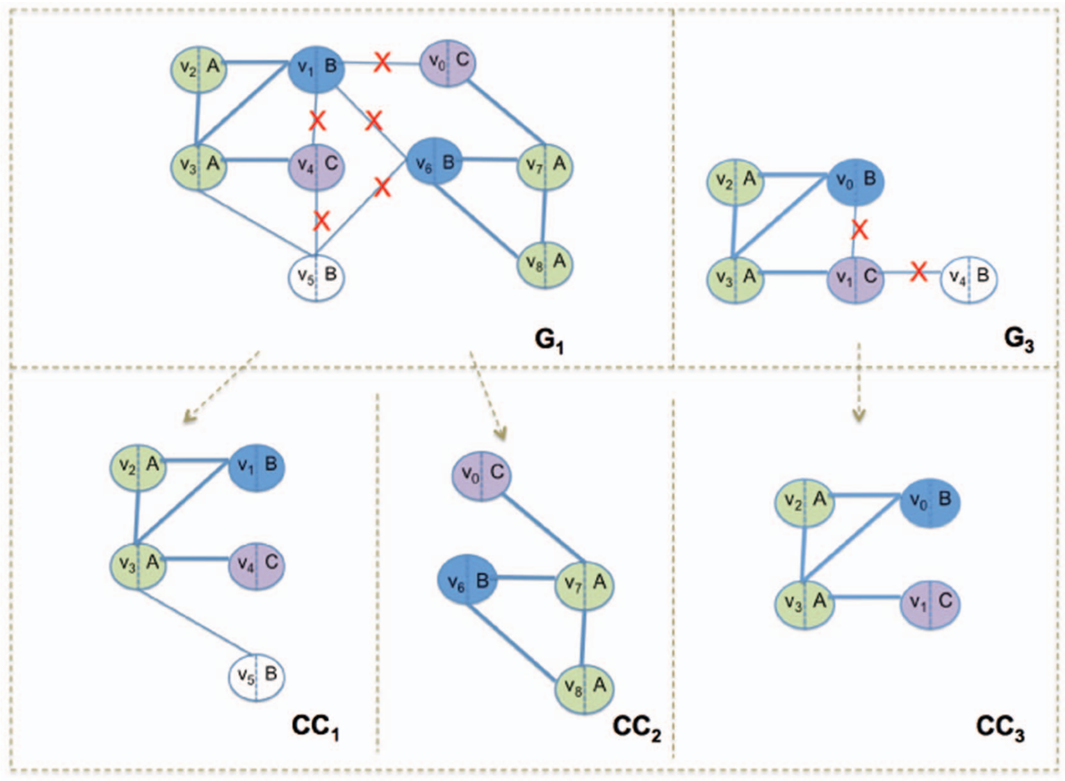

Figure 5. Filtering in GRAPES. The graphs in Figure 1 are reduced to the following components after filtering. CC_1 and CC_2 derive from G_1 and CC_3 from G_3. This reduction of search space makes GRAPES fast and suitable to parallelization. The combinatorial subgraph isomorphism algorithm (applied in the Matching phase) will be run only on these components.
doi:10.1371/journal.pone.0076911.g005

the PDBS dataset contains echoviruses and decay-accelerating factor (DAF). The dataset contains a total of 10 unique labels with an average of 6.37 and a standard deviation of 2.15 per protein. The average degree per vertex is 4.27 with a standard deviation of 2.02. The average number of vertices (resp. edges) is 2,939 (resp. 6,127) with a standard deviation of 3,214 (resp. 6,522).

- *PCM*, the protein contact maps dataset is taken from *CMView* [28]. It comes from 200 contact maps of the amino acids of the domains of the proteins having up to 883 nodes and 18,832 edges. Since they represent relationships among amino acids, the number of vertices in the contact maps is relatively small (corresponding to the length of the proteins). The average number of vertices (resp. edges) is 376 (resp. 8,679) with a standard deviation of 186.6 (resp. 3,814). The average degree per vertex is 44.78 with a standard deviation of 17.47. The average number of labels per contact map and the standard deviation are 18.86 and 3.48, respectively.
- *PPI*, a database of 20 protein interaction networks. The networks belong to 5 species: *Caenorhabditis elegans*, *Drosophila melanogaster*, *Mus musculus*, *Saccaromyces cerevisiae* and *Homo sapiens*. For each species, we generated four networks from the original ones, by selecting the edges having accurateness greater than 0.4, 0.5, 0.6, and 0.7. The networks are annotated using Gene Ontology annotations [29]. We run the Cytoscape plugin Mosaic [30], which assigns colors to the vertices of the networks depending on their most relevant GO terms. The networks have up to 10,186 nodes (average 4,942.9 and standard deviation 2,648) and 179,348 edges (average 53,333 and standard deviation 51,387). The average degree per vertex is 18.46 with a standard deviation of 42.12. The average number of labels per graph and the standard deviation are 28.45 and 9.5, respectively.

The queries for the *AIDS* database are entire compounds randomly chosen from the database of size 8, 16 and 32 edges. For the datasets *PDBS* and *PCM*, the queries have the same size but they have been generated from the original graphs in the following way. Starting from an edge of the graph, choose a random neighboring edge. In general, choose any edge that neighbors any edge of the growing graph. The process is repeated until the chosen number of edges is reached. Since queries are substructures chosen randomly from the databases, the queries reflect the average degree and label distribution of the biological data. For the *PPI* dataset, since relevant queries are often small graphs (e.g., feed forward loops), we used queries of size 4 and 8. Results are given as the average performance obtained by running from one to one hundred queries.

Experiments have been conducted on a SMP machine 6×2.8 GHz Intel Xeon with Debian 5.0×86 O.S. GRAPES has been implemented in C++ and compiled with gcc-4.4.

We compare GRAPES with available software implementing sequential graph searching algorithms based on indexing such as SING [15], GraphGrepSX (GGSX) [16], and the most recent CT-index [24]. We also report the results obtained by running *VF2* [10], which does not use indexing. We do not report comparisons with gIndex [19], GCoding [13] and CTree [12] since SING either outperforms them or gives closely comparable performance (we refer to [15] for details of such comparisons). The results (building time, matching time and building+matching time)

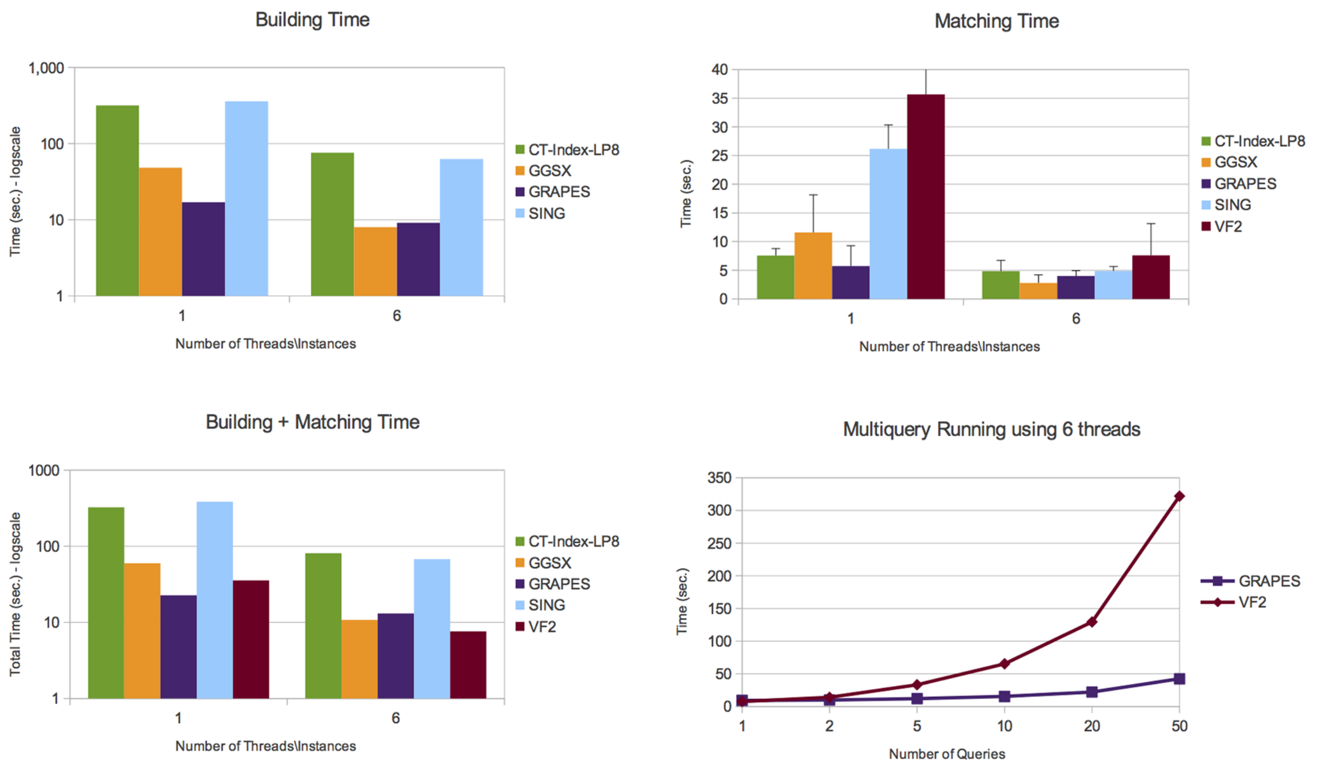

Figure 6. Experimental results on the *AIDS* database. It shows building, matching and total (building+macthing) time of the compared tools. The *AIDS* dataset contains a large number of small, simply structured graphs. Here, querying is not an expensive task. Comparisons show that GRAPES outperforms the other tools by using 1 thread. Parallelism due to the simple structure of data does not help. However, indexing induces efficiency when the index is built once and re-used for multiquery. Indeed, GRAPES results faster than non-indexing based methods such as VF2 by running two o more queries (see Multiquery running plot, run on 6 threads).
doi:10.1371/journal.pone.0076911.g006

are presented as the average time of 100 queries, run one at the time. For each query, we set the system to wait 8 hours for the searching results. When we state that a tool does not run on a dataset, we mean that the tool did not terminate (within 8 hours) in almost all tested queries. SING runs out of time on *PDBS*, *PCM* and *PPI*, while CT-index runs out of time on *PCM* and *PPI*.

Details on the number of graphs containing the queries, the number of subgraphs isomorphisms and the number of connected components are given in Table S1 in the File S1. Figures 6, 7, 8, and 9 report the obtained results on *AIDS*, *PDBS*, *PCM* and *PPI*, respectively. Each figure reports the number of the building and matching time and the performance of GRAPES and *VF2* for the runs of multiple queries (i.e., from one to one hundred queries). For a fair comparison, we also run multiple instances (six) of the state of the art algorithms on different graphs of the databases.

We report the average time and the corresponding standard deviations obtained on the tested queries. The filtering time, which is negligible for all the tested graphs, is included in the matching time. The number of candidates graphs and memory consumption of the compared tools are reported Figures S1–S4 in the File S1.

On all tested databases (Figures 6, 7, 8, 9), considering the building plus matching time, GRAPES is faster than VF2 even when only one query is run. Therefore, even though indexing is a time-consuming step, it clearly speeds up the matching time. Parallelism is critical to this success. In Figures 6 and 9, when using only one thread, the index-based tools do not outperform VF2 when a single query is run. The index-based tools outperform VF2 when we compare the performance of GRAPES and *VF2* (building plus matching time, see multiquery running graphs in Figures 6, 7, 8, 9) on one hundred queries using six threads for GRAPES and six instances of *VF2* over the databases. These results show that indexing induces efficiency when the index is constructed once and re-used for multiquery. Parallelism and our particular index strategy give substantial improvements over index-less systems such as *VF2*, with increasing benefits the more queries there are.

On the *AIDS* dataset (Figure 6), GRAPES is faster than SING and CT-index and comparable to GGSX. We run CT-Index using the size of features (i.e., 8) suggested by the CT-index authors on the same dataset (LP8 means trees of depth 6 or cycles of 8 edges).

GRAPES has a particular advantage compared to other indexing-based graph searching or subgraph isomorphism algorithms on more complex structured graph databases. For example, on the *PDBS* (Figure 7) and *PCM* (Figure 8) datasets, GRAPES is much faster than GGSX and CT-Index. We run CT-Index using also small size features (LP4), to evaluate a possible matching and building speedup. However, since CT-Index in *PCM* (Figure 8) dataset has a very high building time, GRAPES and GGSX outperformed it in the total time. For this reason, its matching time is not reported in the plot.

GRAPES is faster than GGSX also in *PPI* (Figure 9) dataset. In the *PDBS* (Figure 7), *PCM* (Figure 8), and *PPI* (Figure 9) datasets, GRAPES is much faster than VF2 even on a single query (building plus matching time).

GRAPES is faster than the compared tools in all datasets also when queries are not present in the datasets (see Figures S6–S9 in the File S1).

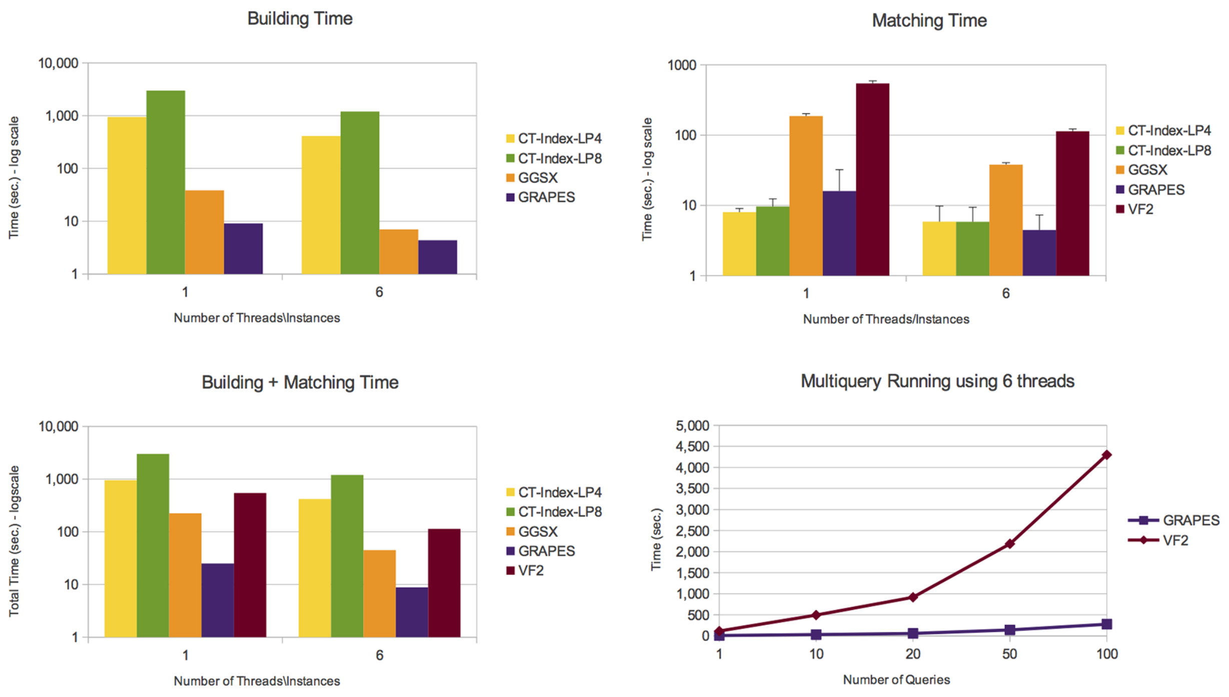

Figure 7. Experimental results on the *PDBS* database. It shows building, matching and total (building+macthing) time of the compared tools. Comparisons show that GRAPES outperforms the other tools by using 1 or 6 threads. However, parallelism helps to decrease the computation costs. We found that indexing induces efficiency when the index is built once and re-used for multiquery. Indeed, GRAPES results faster than non-indexing based methods such as VF2 by running one or more queries (see Multiquery running plot, run on 6 threads).
doi:10.1371/journal.pone.0076911.g007

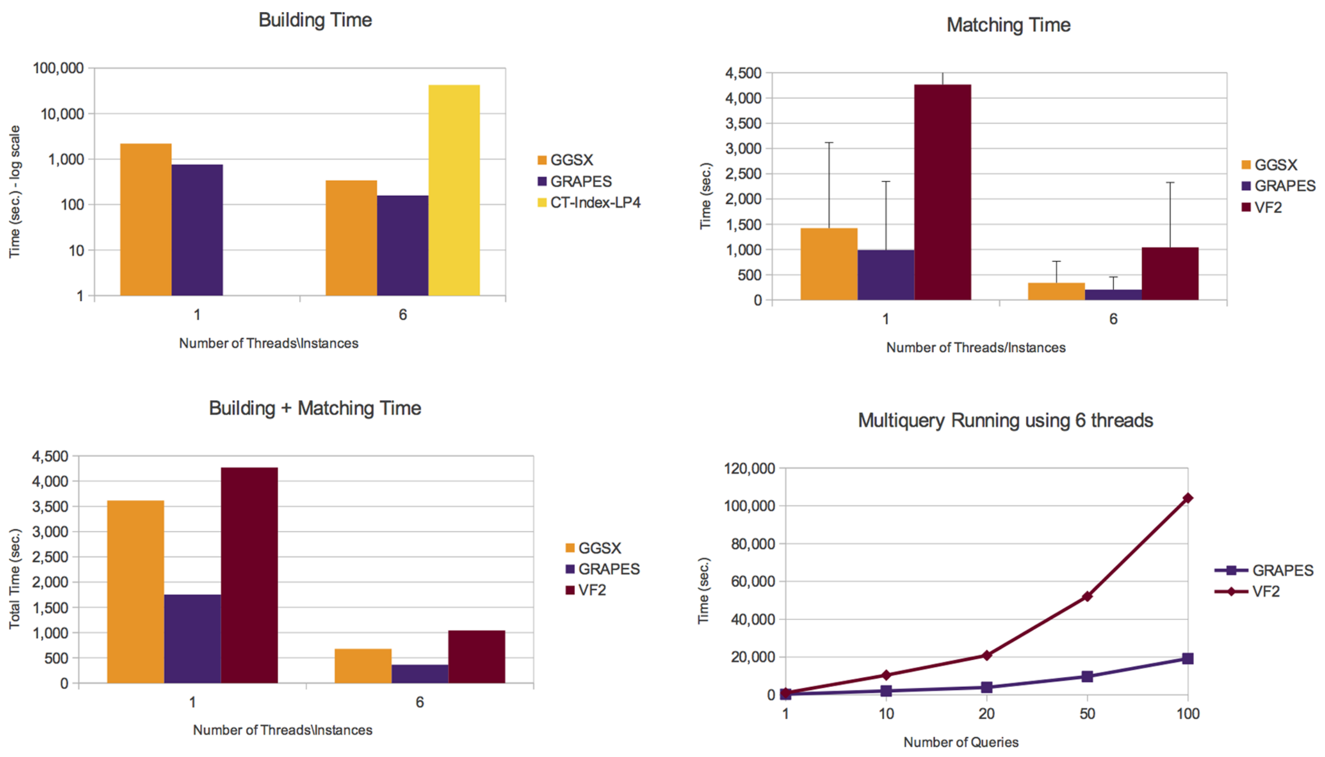

Figure 8. Experimental results on the *PCM* database. It shows building, matching and total (building+macthing) time of the compared tools. Comparisons show that GRAPES outperforms the other tools by using 1 or 6 threads. However, parallelism helps to decrease the computation costs. We found that indexing induces efficiency when the index is built once and re-used for multiquery. Indeed, GRAPES results faster than non-indexing based methods such as VF2 by running one or more queries (see Multiquery running plot, run on 6 threads).
doi:10.1371/journal.pone.0076911.g008

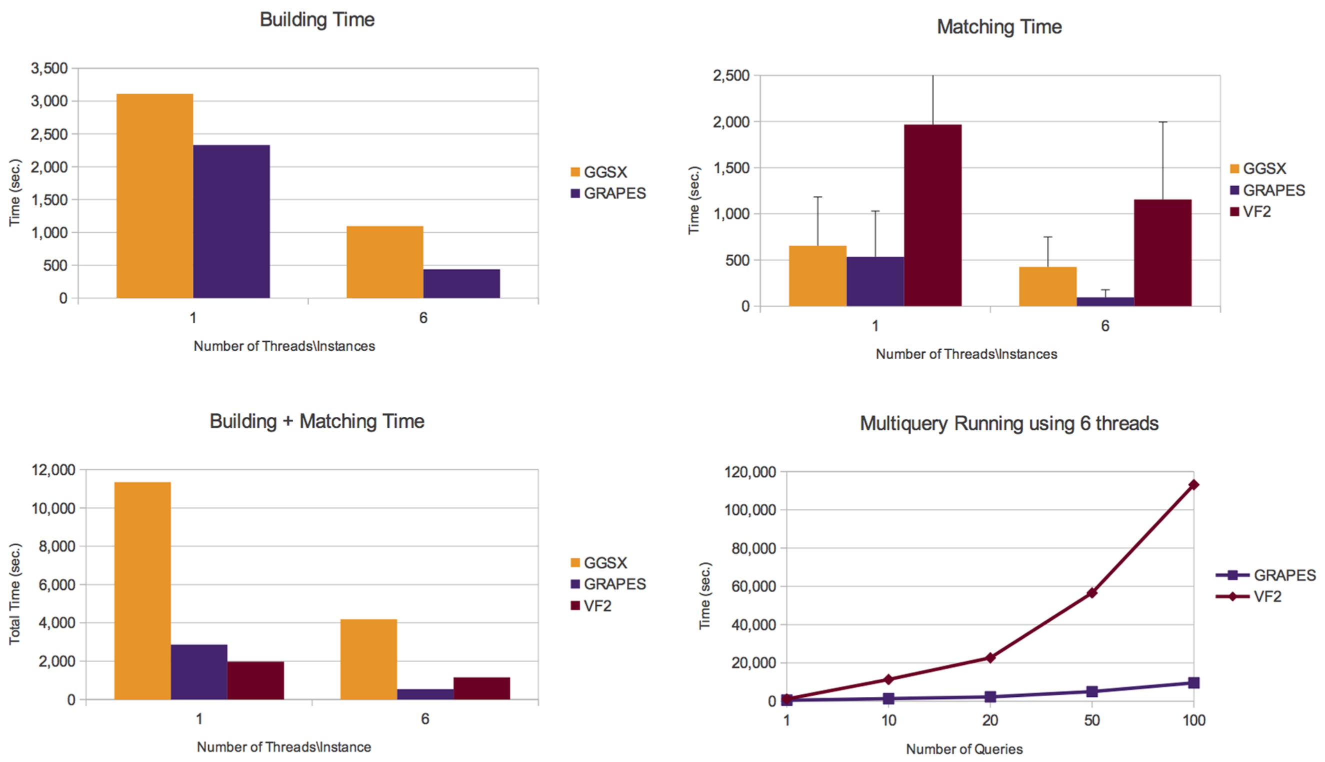

Figure 9. Experimental results on the *PPI* database. It shows building, matching and total (building+macthing) time of the compared tools. Comparisons show that GRAPES outperforms the other tools by using 1 or 6 threads. However, parallelism helps to decrease the computation costs. We found that indexing induces efficiency when the index is built once and re-used for multiquery. Indeed, GRAPES results faster than non-indexing based methods such as VF2 by running one or more queries (see Multiquery running plot, run on 6 threads).
doi:10.1371/journal.pone.0076911.g009

We conclude that the parallelism of GRAPES yields load balancing, scalability, and efficiency on all databases (see Figure S5 in the File S1). The index-based approach implemented in GRAPES is well-suited to biological data particularly when data has complex structure and queries are time consuming. Further, GRAPES performance is better than multiple instances of sequential graph-search tools of the literature.

Conclusions

GRAPES is a parallel graph searching algorithm that achieves the best performance of any algorithm on large data coming from applications such as pharmacogenomics and biology. Its advantage is the strongest for large graphs as opposed to databases of many small graphs. GRAPES is based on the following algorithmic strategies:

1. Indexing occurs in parallel on vertices, so there is parallelism even if the entire database consists of a single graph. Further, the indexing for different nodes can operate independently and asynchronously.
2. GRAPES uses the indexing system to partition large graphs into connected components, thus substantially reducing the effort when the exponential phase arises. This makes GRAPES suitable for applications on large biological graphs such as protein interaction networks.
3. To compress the resulting path-based indexes, GRAPES uses tries. In doing so, GRAPES reduces the space and time needed for the construction of large database indexes.

We have demonstrated that GRAPES improves performance compared with state of art algorithms by a factor of up 12, and offers parallel scaling with a factor varying from 1.72 to 5.34 on the biologically representative datasets containing antiviral chemical compounds, DNA, RNA, proteins, protein contact maps and protein interactions networks.

Supporting Information

File S1 Details on GRAPES performances. It shows for the compared tools memory consumption as well as the number of candidate graphs, connected components found in various setting and the performance of the tools when the queries have no matches.
(PDF)

File S2 GRAPES user manual.
(PDF)

Acknowledgments

We would like to thank the authors of VFlib and CT-Index for having kindly provided their tools for comparison purpose.

Author Contributions

Conceived and designed the experiments: RG VB NB. Performed the experiments: VB. Analyzed the data: RG VB NB AP AF DS. Wrote the paper: RG VB NB AP AF DS.

References

1. Kelley B, Sharan R, Karp R, Sittler T, Root D, et al. (2003) Conserved pathways within bacteria and yeast as revealed by global protein network alignment. PNAS 100: 11394–11399.
2. Ferro A, Giugno R, Pigola G, Pulvirenti A, Skripin D, et al. (2007) Netmatch: a cytoscape plugin for searching biological networks. Bioinformatics 23: 910–912.
3. Banks E, Nabieva E, Peterson R, Singh M (2008) Netgrep: fast network schema searches in interactomes. Genome biology 9: R138.
4. Dost B, Shlomi T, Gupta N, Ruppin E, Bafna V, et al. (2008) Qnet: A tool for querying protein interaction networks. Computational Biology 15: 913–925.
5. Lacroix V, Fernandes C, Sagot M (2006) Motif search in graphs: application to metabolic networks. IEEE/ACM Transactions on Computational Biology and Bioinformatics 3: 360–368.
6. Daylight chemical information systems, available: http://www.daylight.com/, accessed 2013 marc 10.
7. Rajapakse H (2007) Small molecule inhibitors of the xiap protein-protein interaction. Curr Top Med Chem 7: 966–71.
8. Levy D, Pereira-Leal D, Chothia C, Teichmann S (2006) 3d complex: A structural classification of protein complexes. PLOS Computational Biology 2: e155.
9. Messmer BT, Bunke H (1995) Subgraph isomorphism detection in polynominal time on preprocessed model graphs. Recent Developments in Computer Vision,Lecture Notes in Computer Science 1035: 373–382.
10. Cordella L, Foggia P, Sansone C, Vento M (2004) A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence 26: 1367–1372.
11. Ullmann JR (2011) Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. Journal of Experimental Algorithmics 15: 1.1–1.64.
12. He H, Singh AK (2006) Closure-tree: An index structure for graph queries. In: Proceedings of the 22nd International Conference on Data Engineering. ICDE'06, p. 38.
13. Zou L, Chen L, Yu J, Lu Y (2008) A novel spectral coding in a large graph database. In: Proceedings of the 11th international conference on Extending database technology: Advances in database technology. ACM, 181–192.
14. Shasha D, Wang JL, Giugno R (2002) Algorithmics and applications of tree and graph searching. Proceeding of the ACM Symposium on Principles of Database Systems (PODS) : 39–52.
15. Di Natale R, Ferro A, Giugno R, Mongiovì M, Pulvirenti A, et al. (2010) Sing: Subgraph search in non-homogeneous graphs. BMC Bioinformatics 11: 96.
16. Bonnici V, Ferro A, Giugno R, Pulvirenti A, Shasha D (2010) Enhancing graph database indexing by suffix tree structure. In: Proceedings of the 5th IAPR international conference on Pattern recognition in bioinformatics. PRIB'10, 195–203.
17. Zhang S, Hu M, Yang J (2007) Treepi: A novel graph indexing method. In: Proceedings of IEEE 23rd International Conference on Data Engineering. ICDE'07, 181–192.
18. Cheng J, Ke Y, Ng W, Lu A (2007) Fg-index: towards verification-free query processing on graph databases. In: Proceedings of the ACM SIGMOD international conference on Management of data. SIGMOD'07, 857–872.
19. Yan X, Yu P, Han J (2004) Graph indexing: a frequent structure-based approach. In: Proceedings of the ACM SIGMOD international conference on Management of data. SIGMOD'04, 335–346.
20. Williams DW, Huan J, Wang W (2007) Graph database indexing using structured graph decomposition. Data Engineering, 2007 ICDE 2007 IEEE 23rd International Conference on : 976–985.
21. Ferro A, Giugno R, Mongiovì M, Pulvirenti A, Skripin D, et al. (2008) Graphfind: enhancing graph searching by low support data mining techniques. BMC bioinformatics 9: S10.
22. Cohen E, Datar M, Fujiwara S, Gionis A, Indyk P, et al. (2001) Finding interesting associations without support pruning. IEEE Transactions on Knowledge and Data Engineering 13: 64–78.
23. Zhao P, Yu JX, Yu PS (2007) Graph indexing: tree+delta $\leq$ graph. VLDB'07: Proceedings of the 33rd international conference on Very large data bases : 938–949.
24. Kriege N, Mutzel P (2011) Ct-index: Fingerprint-based graph indexing combining cycles and trees. In: Proceeding of the IEEE International Conference on Data Engineering. ICDE'11, 1115–1126.
25. Dean J, Ghemawat S (2004) MapReduce: Simplified data processing on large clusters. In: Proc. Of Operating Systems Design and Implementation (OSDI'04). 137–150.
26. National cancer institute, available: http://www.nci.nih.gov/, accessed 2013 marc 10.
27. He Y, Lin F, Chipman P, Bator C, Baker T, et al. (2002) Structure of decay-accelerating factor bound to echovirus 7: a virus-receptor complex. Proc Natl Acad Sci USA 99: 10325–10329.
28. Vehlow C, Stehr H, Winkelmann M, Duarte JM, Petzold L, et al. (2011) Cmview: Interactive contact map visualization and analysis. Bioinformatics 27: 1573–1577.
29. Ashburner M, Ball CA, Blake JA, Botstein D, Butler H, et al. (2000) Gene ontology: tool for the unification of biology. Nature genetics 25: 25–29.
30. Zhang C, Hanspers Z, Kuchinsky A, Salomonis N, Xu D, et al. (2013) Mosaic: Making biological sense of complex networks. BIOINFORMATICS 28: 1943–4.