Rusty-Cracker: A Multi-core Connected Components Library in Rust

Davide Rucci
davide.rucci@isti.cnr.it
ISTI-CNR
Pisa, Italy

Daniele Sampietro
d.sampietro@studenti.unipi.it
University of Pisa
Pisa, Italy

Emanuele Carlini
emanuele.carlini@isti.cnr.it
ISTI-CNR
Pisa, Italy

Matteo Mordacchini
matteo.mordacchini@iit.cnr.it
IIT-CNR
Pisa, Italy

Patrizio Dazzi
patrizio.dazzi@unipi.it
University of Pisa
Pisa, Italy

Abstract

We present Rusty-Cracker, a high-performance Rust library that implements a parallel version of the Cracker algorithm for efficiently identifying connected components in large-scale graphs. Designed to address the growing demands of graph analytics in fields such as social network analysis, bioinformatics, and infrastructure modeling, Rusty-Cracker leverages Rust's concurrency and memory safety features to ensure both speed and reliability. The adapted Cracker algorithm capitalizes on modern multi-core architectures through parallel processing, effectively minimizing synchronization overhead and optimizing workload distribution. This design significantly reduces computational time while maintaining accuracy. We present the implementation details and evaluate its performance on a real-world dataset of undirected graphs.

CCS Concepts

• **Mathematics of computing** → Graph algorithms; • **Computing methodologies** → **Shared memory algorithms**; • **Software and its engineering** → **Software libraries and repositories**.

Keywords

Graph algorithms, Parallel algorithms, Connected Components, Rust

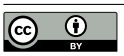
ACM Reference Format:
Davide Rucci, Daniele Sampietro, Emanuele Carlini, Matteo Mordacchini, and Patrizio Dazzi. 2025. Rusty-Cracker: A Multi-core Connected Components Library in Rust. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3731545.3744334

1 Introduction

The abstraction of graphs is highly pervasive in today's interconnected world. Consequently, graph analysis has become a fundamental tool for extracting information and unraveling complex relationships across various applications [3, 6, 22, 25]. Typical applications include social network analysis [2], community detection [10, 23], web graph mining [12, 14], and image processing [5], but network analysis is increasingly becoming important also in the optimization of resource allocation in Edge platforms [4, 10, 21]. One common problem in this context is finding the connected components in a graph, which involves identifying all subgraphs that do not share any edges. The traditional go-to algorithm for solving this problem has $O(|V| + |E|)$ linear time complexity [7], which is theoretically optimal. However, as in many other domains [17, 20], scalability concerns and the need for distributed evaluations are also arising in this context. Specifically, as graphs continue to grow in size nowadays, simple sequential solvers can struggle with such problems that require traversing the entire graph. For very large graphs, there has been a surge in approaches utilizing distributed computing [18, 19]. This holds especially in the context of connected components detection using MapReduce frameworks. In MapReduce, the "map" function distributes tasks across multiple nodes, while the "reduce" function aggregates the results. These approaches are beneficial when the graph is too large to fit into the memory of a single machine. The main challenge of these methods is structuring the computation to minimize the overhead of synchronization among the various workers.

We hypothesize that similar computation models could be effective on today's dense multi-core architectures due to their ability to achieve high levels of concurrency. To test our hypothesis, we took Cracker [15, 16], a MapReduce approach to finding connected components, and implemented a parallel version of Cracker in Rust. We tested our parallel version on a set of large-scale graphs, generally larger than a typical edge or IoT network. Overall, we found that this approach scales very well up to 16 threads, with thread overhead becoming more significant beyond that point.

The rest of this paper is organized as follows: Section 2 lays the necessary background notions and presents the algorithmic strategy of the Cracker algorithm, as well as an overview of the related work. Section 3 presents the details of our implementation, explaining the choices made for the data structures and the management of parallelism. Section 4 outlines our experimental phase, discussing the dataset used and the results obtained in terms of performance. Finally, Section 5 concludes the paper.



This work is licensed under a Creative Commons Attribution 4.0 International License.
HPDC '25, Notre Dame, IN, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3744334

2 Background

2.1 Problem definition

Let $G = (V, E)$ be an undirected graph where V is a set of vertices, and E is the set of edges. A connected component of G (the *supergraph*) is a maximal subgraph $S = (V_S, E_S)$ such that for any pair of vertices $u, v \in V_S$ there is an undirected path in S connecting them, and S is not connected to additional vertices in the supergraph.

2.2 The Cracker Algorithm

CRACKER is a distributed approach for finding connected components designed for the MapReduce paradigm. The strategy of CRACKER involves iteratively building a spanning tree for each connected component in the supergraph. Once a node is added to the tree, it is removed from the supergraph, making the supergraph smaller with each iteration. This reduction decreases the computation time and the amount of information exchanged. The removal of nodes from G necessitates rewiring the remaining nodes to maintain the semantics of the connected components. In the original paper by Lulli et al. [16], this operation is called "Seed Identification."

The purpose of seed identification is to identify vertices, called seeds, that represent a connected component. Vertices are associated with unique identifiers, which are positive natural numbers. The vertex with the lowest identifier is the seed of the connected component. During computation, the candidate seeds are those with the lowest identifier in a given neighborhood. At any iteration, a generic vertex v may discover another vertex q in its neighborhood with a lower identifier value. Consequently, v connects to q. If v is not chosen by any neighbor, it is excluded from the computation and becomes the child of q in the spanning tree. The first output of seed identification is a directed graph, called H, in which all nodes point to candidate seeds. This graph is then used to create a further undirected supergraph, G^t, where all candidate seeds of the same connected component are part of the same component. Ultimately, all vertices are removed from the connected components, with the seed being the last to be removed.

Edge Pruning Optimization. Edge pruning is an optimization of CRACKER designed to reduce the number of edges created during the seed identification phase. This optimization is detailed in an enhanced version of CRACKER by the same authors [15]. To minimize the number of messages, edge pruning eliminates those used in the original version to notify about a candidate seed with the lowest identifier in a neighborhood, as this information is redundant. Essentially, if a vertex is a candidate seed, it does not add any edges in the intermediate directed graph H^t. H^t is then required to create the subsequent supergraph. According to the experimental evaluation in the CRACKER paper, edge pruning results in a 6% reduction in completion time [15].

2.3 Related work

To the best of our knowledge, Rusty-Cracker is the first Rust library for parallel connected component computation on graphs. Rust combines a strong concurrency model, a minimal runtime, and C++-level performance with memory safety, making it ideal for tasks like decentralized, large-scale graph analysis. Thus, we provide here a brief overview of the state of the art for the same task, using other programming languages.

Parconnect [9] is a library, written in C++ with MPI (Message Passing Interface), designed for graph connectivity queries, including connected components computation. Initially designed for De Bruijn graphs, it is also able to handle generic undirected graphs.

FastSV [27] is a parallel algorithm built using the GraphBLAS [8] framework, to compute the connected components of a graph using linear algebra calculations. The GraphBLAS framework itself, implemented in C, has been forked to a parallelized version using OpenMP. Additionally, there exists an official library by GraphBLAS that implements common graph algorithms, including one to compute the connected components, called LAGraph [26].

Another very recent C++ parallel algorithm for this task is DSELP, by Khan and Boddu [13], which again exploits parallelism and message exchanges through OpenMP and MPI. The core of the algorithm is similar to that of Cracker, based on the exchange of seeds between vertices in order to reach a common consensus on the connected components (see Section 2.2).

NetworkX [11] is a well-known graph processing library written in Python, containing many different algorithms, among which we find one to compute connected components. However, this library does not offer parallelism unless different versions of Python are used.

3 Implementation Details

In this Section, we outline the details of our implementation of the library, specifying the external libraries used, and the choice of data structures. We implemented Rusty-Cracker in Rust, version 1.85, the code is publicly available on GitHub[1]. This choice of the programming language is justified by the trade-off between safety and efficiency. In fact, Rust offers performances comparable to those of C++, while adopting strong measures for memory and thread safety.

Our implementation makes use of the following external *crates* (i.e. libraries):

- *rayon*: to simplify the management and allocation of threads and thread pools.
- *dashmap*: for a fast and concurrent version of the HashMap data structure,
- *getopts*: to parse command line arguments.

Parallelism Management. The Rust programming language provides a native way to work with threads. However, we found that other libraries performed better in our experimental environment [1], while also providing easier management of advanced concepts like thread pools. It is the case of the *Rayon* library, which we adopted for the handling of thread pools. Moreover, the library provides parallel iterators, automatically managing the chunking of data structures, for example, when needing to examine the neighborhood of vertices. The paradigm that we adopted for this implementation is the so-called *data parallelism*, where we apply the same operator on different sets of data. An instance of this is found in the code that computes the neighbor with the lowest identifier

[1] https://github.com/dSampietro/Rusty-CRACKER

of each vertex in the graph, where we use a parallel iterator on all nodes of the graph.

Concurrent Data Structures. Rusty-Cracker represents graphs using adjacency lists, which are internally implemented as hash maps, for fast random access. As the native *HashMap* structure of Rust is not thread-safe, we used the DashMap data structure, where each entry of the map has its lock associated. This allows to speed up the concurrent access to the map without sacrificing parallelism.

The bigger *Graph* objects are implemented via the *ConcurrentUnGraph* and *ConcurrentDiGraph* structures, respectively, for undirected and directed graphs. Both structures adopt the adjacency list paradigm using dashmaps as the underlying data structure. By doing this, we can avoid having a lock for the entire graph object, which would dramatically degrade performance, and instead we rely on many fine-grained locks that speed up the computation while maintaining thread-safety. We note that there exist other crates in the Rust registry for graph harnessing, such as *petgraph*, but they lacked the scalability factor due to their lock policies. We also emphasize that these graph structures have been carefully designed with this task in mind, hence, they may not work well in other contexts.

To summarize, our graph implementation allows for the operations listed in Table 1.

Table 1: Summary of Graph operations supported by our implementation.

Operation	Time Complexity
Node Addition	$O(1)$
Node Removal	$O(d(u))$
Node Existance Check	$O(1)$
Edge Addition	$O(1)$
Edge Removal	$O(1)$
Edge Existance Check	$O(1)$

Link-Time Optimization (LTO). The Rust compiler offers an advanced optimization framework called *Link-Time Optimization* or LTO for short. This technique is adopted by the compiler at linking time, hence its name, so it is able to work both on the source code of a program and on the libraries dynamically included in the final executable. Having access to the entire code of a project, the optimizer can apply aggressive optimizations, including:

- Dead code removal: the compiler can understand regions of code that will never be executed in the final executable and remove them.
- Function Inlining: function calls are replaced with the actual body of the called function, so that we can avoid the creation of a new frame on the call stack, and thus speed up the computation of functions called often during the execution.
- Function Un-inlining: for particularly long and complex functions, it may be better instead to not inline the code.

and more, like loop unrolling. These optimizations allow for big savings in running time at the expense of a longer compilation time.

Graph input format. We adopted the *mtx* file format for reading the input graphs. The mtx format, which stands for *Matrix Market* format, sees the graph as a matrix, and it is organized into three macro-area: *(i)* header: this contains a preamble row expressing the format of the matrix, including the data type (e.g. int, float, etc.), the symmetry of the matrix (if the matrix is symmetric, the graph is undirected), and the type of weights associated to the edges, if present; *(ii)* size: three numbers, $\langle rcnonzeros \rangle$ summarzing the size of the matrix and the number of its non-zero entries. The first two represent the size of the matrix (r rows and c columns), usually $r = c$, and the last corresponds to the number of edges present in the graph; *(iii)* data: the actual edge list of the graph, in the format of $\langle u, v, weight \rangle$, one per line. The total number of these lines is exactly *nonzeros*.

4 Experimental evaluation

We conducted our experiments on a machine equipped with an AMD EPYC 7551 processor running at 2.0 GHz, featuring 64 cores, 128 MiB of shared L3 cache, and 132 GB of RAM.

Our evaluation includes both the baseline algorithm and an optimized variant incorporating edge pruning (EP), enabling us to assess the performance gains introduced by this enhancement. We collected data on execution time, iteration count, and overall scalability. Due to space constraints, we omit the plots related to the number of iterations, as they remain consistently low across all configurations and thread counts. Instead, we focus on the actual execution time of the algorithms as the number of threads in the thread pool increases, highlighting the speedup achieved through parallelism.

4.1 Dataset

We tested our implementation on a heterogeneous dataset made of real-world graphs of different sizes, ranging from thousands to millions of nodes and edges, obtained from the well-known Network Repository [24]. We give an overview of the graphs in our dataset in Table 2.

Table 2: Overview of a portion of our dataset, sorted by increasing number of edges.

Dataset	Kind	Max. Nodes	Total Edges
Facebook-Artist	Facebook Artists Pages	50515	819306
Amazon-2008	Amazon Products	548551	925872
bn-human-Jung2015_M87124670	Human Fiber Tracts	2500000	42402074
CA-Hollywood-2009	Collaboration Network	1069126	112613306
web-wikipedia_link_it	Italian Wikipedia Links	2790239	173567387

4.2 Results and Discussion

In order to do a comparison with a sequential version of the algorithms, we run the experiments with $t = 1, 2, 4, 8, 16, 32, 64$ as the number of available threads. Figures 1 through 5 show the running time (in milliseconds) of the implementation of both the base and EP implementations, for the different numbers of threads available. The results are also summarized in Table 3, showing the raw data for the running time and the speedup, calculated as the ratio between the running time of the sequential version (e.g. one thread)

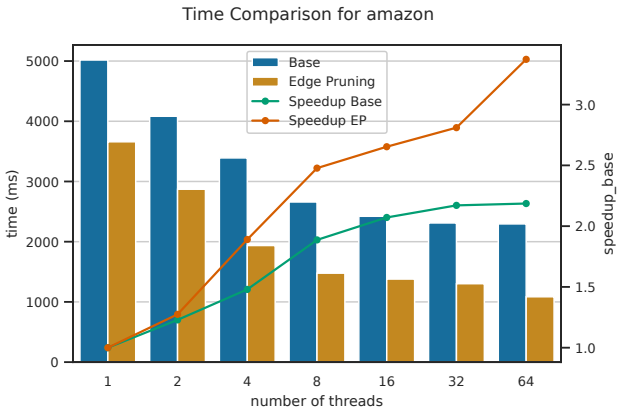


Figure 1: Running Time Comparison for the Amazon-2008 dataset.

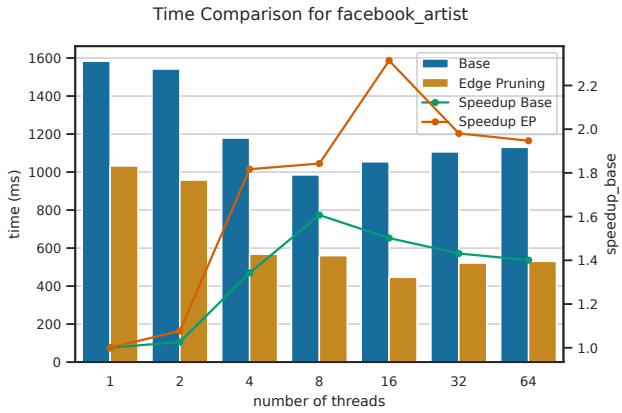


Figure 2: Running Time Comparison for the Facebook-Artists dataset.

Table 3: Time Comparison for Different Graphs and Threads

Graph	Threads	Time Base (ms)	Time EP (ms)	Speedup Base	Speedup EP
amazon	1	5,017	3659	–	–
	2	4,083	2871	1.23	1.27
	4	3,391	1,936	1.48	1.89
	8	2,659	1,477	1.89	2.48
	16	2,422	1,379	2.07	2.65
	32	2,311	1,302	2.17	2.81
	64	2,295	1,085	2.19	3.37
bn-human-Jung2015_M87124670	1	80,064	55961	–	–
	2	67,575	40867	1.18	1.37
	4	52,194	20,734	1.53	2.70
	8	39,854	13,866	2.01	4.04
	16	36,216	8,772	2.21	6.38
	32	33,528	5,734	2.39	9.76
	64	32,593	5,367	2.46	10.43
ca-hollywood-2009	1	139,453	95921	–	–
	2	94,390	54942	1.48	1.75
	4	82,204	39,794	1.70	2.41
	8	65,149	28,670	2.14	3.35
	16	64,363	24,799	2.17	3.87
	32	61,531	21,413	2.27	4.48
	64	61,492	20,730	2.27	4.63
facebook_artist	1	1,583	1032	–	–
	2	1,542	958	1.03	1.08
	4	1,179	568	1.34	1.82
	8	985	560	1.61	1.84
	16	1,054	446	1.50	2.31
	32	1,106	521	1.43	1.98
	64	1,130	530	1.40	1.95
web-wikipedia_link_it	1	207,066	129597	–	–
	2	174,764	88328	1.18	1.47
	4	142,563	61,803	1.45	2.10
	8	127,685	58,809	1.62	2.20
	16	144,153	70,589	1.44	1.84
	32	144,392	74,665	1.43	1.74
	64	150,485	73,462	1.38	1.76

to the running time of the parallel versions. The speedup is also shown in the aforementioned plots.

As we expected, the results show increasingly good performance of our library tied together with the level of parallelism that we allow. While for some graphs we achieve the best performances with the maximum number of threads (64), for some graphs the best results are obtained with a smaller number of threads, such as 8 or 16. This can be explained by the overhead introduced by the large number of threads: in fact, sometimes the work needed to

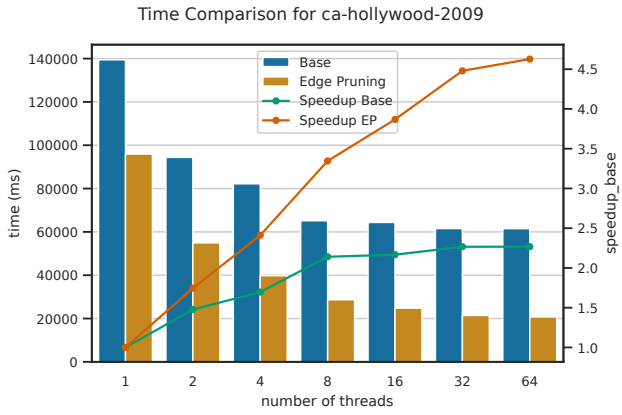


Figure 3: Running Time Comparison for the Ca-Hollywood-2009 dataset.

manage the parallel infrastructure is more prominent with respect to the core logic of the algorithms. Additionally, we note how the EP optimization of Cracker brings a valuable improvement to the running time, always performing better than the baseline implementation, thus, we can conclude that the EP version is always to be preferred.

5 Conclusions and Future Work

In this paper, we introduced Rusty-Cracker, a high-performance Rust library that efficiently identifies connected components in large undirected graphs using a parallel implementation of the Cracker algorithm [16]. Through extensive testing on a diverse set of real-world datasets, we showed that Rusty-Cracker scales effectively with the number of threads, achieving significant performance improvements, particularly with the edge pruning optimization. Our results indicate that, while the library performs optimally with a high number of threads for certain datasets, the overhead associated with thread management can sometimes offset

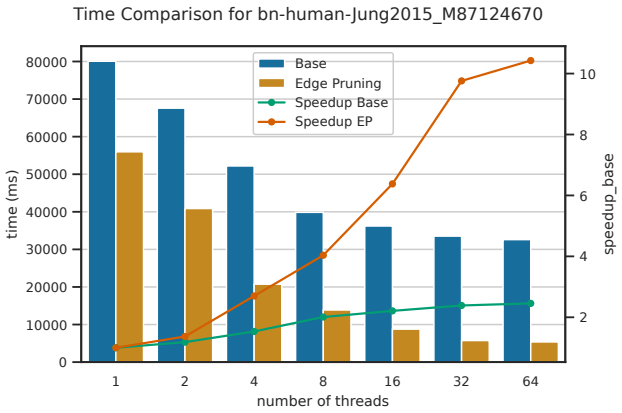


Figure 4: Running Time Comparison for the bn-human-Jung2015_M87124670 dataset.

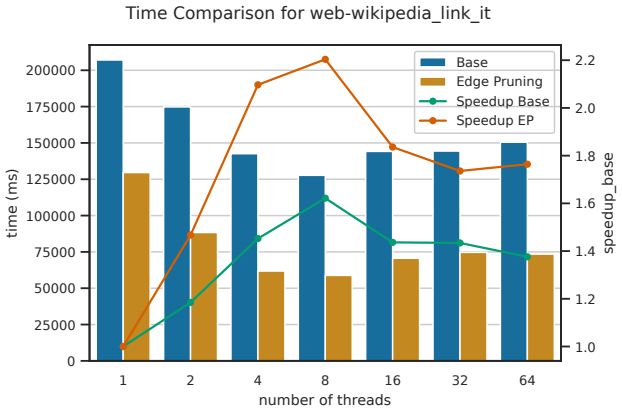


Figure 5: Running Time Comparison for the web-wikipedia-link_it dataset.

the benefits of parallelism. However, edge pruning optimization consistently improves performance, making it a preferred choice for large-scale graph analysis.

In our future work, we will focus on further optimizing the library to reduce synchronization overhead and exploring its application to other types of graphs and computational environments. Finally, we plan to investigate the integration of Rusty-Cracker with distributed computing frameworks to handle even larger datasets that exceed the memory capacity of a single machine.

References

[1] Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C. Jeffrey. 2024. When Is Parallelism Fearless and Zero-Cost with Rust?. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 27–40.

[2] Nadeem Akhtar and Mohd Vasim Ahamad. 2021. Graph tools for social network analysis. In *Research Anthology on Digital Transformation, Organizational Change, and the Impact of Remote Work*. IGI Global, 485–500.

[3] Caroline Brosse, Alessio Conte, Vincent Limouzy, Giulia Punzi, and Davide Rucci. 2024. Output-Sensitive Enumeration of Potential Maximal Cliques in Polynomial Space. In *International Workshop on Combinatorial Algorithms*. Springer, 382–395.

[4] Emanuele Carlini, Patrizio Dazzi, Antonios Makris, Matteo Mordacchini, Theodoros Theodoropoulos, and Konstantinos Tserpes. 2024. Efficient Application Image Management in the Compute Continuum: A Vertex Cover Approach Based on the Think-Like-A-Vertex Paradigm. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. IEEE, 399–403.

[5] Gene Cheung, Enrico Magli, Yuichi Tanaka, and Michael K Ng. 2018. Graph spectral image processing. *Proc. IEEE* 106, 5 (2018), 907–930.

[6] Alessio Conte, Roberto Grossi, and Davide Rucci. 2023. CAGE: Cache-Aware Graphlet Enumeration. In *International Symposium on String Processing and Information Retrieval*. Springer, 129–142.

[7] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[8] Timothy A. Davis. 2022. Algorithm 10xx: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* ((under revision) 2022).

[9] Patrick Flick, Chirag Jain, Tony Pan, and Srinivas Aluru. 2015. A parallel connectivity algorithm for de Bruijn graphs in metagenomic applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 15, 11 pages. doi:10.1145/2807591.2807619

[10] Santo Fortunato. 2010. Community detection in graphs. *Physics Reports* 486, 3 (2010), 75–174. doi:10.1016/j.physrep.2009.11.002

[11] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.

[12] U. Kang and Christos Faloutsos. 2013. Big graph mining: algorithms and discoveries. *SIGKDD Explor. Newsl.* 14, 2 (April 2013), 29–36.

[13] Maleq Khan and Sharon Boddu. 2024. Fast Distributed Memory Parallel Algorithms for Finding Connected Components in Large Graphs. In *2024 IEEE International Conference on Big Data (BigData)*. 579–588.

[14] Sangkeun Lee, Sreenivas R. Sukumar, and Seung-Hwan Lim. 2015. Graph mining meets the Semantic Web. In *2015 31st IEEE International Conference on Data Engineering Workshops*. 53–58. doi:10.1109/ICDEW.2015.7129544

[15] Alessandro Lulli, Emanuele Carlini, Patrizio Dazzi, Claudio Lucchese, and Laura Ricci. 2016. Fast connected components computation in large graphs by vertex pruning. *IEEE Trans. Parallel Distrib. Syst.* 28, 3 (2016), 760–773.

[16] Alessandro Lulli, Laura Ricci, Emanuele Carlini, Patrizio Dazzi, and Claudio Lucchese. 2015. Cracker: Crumbling large graphs into connected components. In *2015 IEEE symposium on computers and communication (ISCC)*. IEEE, 574–581.

[17] Ruben Mayer and Hans-Arno Jacobsen. 2020. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–37.

[18] Lingkai Meng, Yu Shao, Long Yuan, Longbin Lai, Peng Cheng, Xue Li, Wenyuan Yu, Wenjie Zhang, Xuemin Lin, and Jingren Zhou. 2024. A survey of distributed graph algorithms on massive graphs. *Comput. Surveys* 57, 2 (2024), 1–39.

[19] M. Mordacchini, E. Carlini, and P. Dazzi. 2025. A Decentralized Strategy for Unweighted Minimum Vertex Cover. In *Proc. of the 40th Annual ACM Symposium on Applied Computing*. ACM.

[20] Matteo Mordacchini, Marco Conti, Andrea Passarella, and Raffaele Bruno. 2020. Human-centric data dissemination in the IoP: Large-scale modeling and evaluation. *ACM Trans. Auton. Adapt. Syst.* 14, 3 (2020), 1–25.

[21] Jose Moura and David Hutchison. 2016. Review and analysis of networking challenges in cloud computing. *J. Netw. Comput. Appl.* 60 (2016), 113–129.

[22] Ciyuan Peng, Feng Xia, Mehdi Naseriparsa, and Francesco Osborne. 2023. Knowledge graphs: Opportunities and challenges. *Artificial Intelligence Review* 56, 11 (2023), 13071–13102.

[23] Francisco Aparecido Rodrigues. 2019. *Network Centrality: An Introduction*. Springer International Publishing, Cham, 177–196.

[24] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[25] Jan-Grimo Sobez and Markus Reiher. 2020. Molassembler: Molecular graph construction, modification, and conformer generation for inorganic and organic molecules. *Journal of chemical information and modeling* 60, 8 (2020), 3884–3900.

[26] Gábor Szárnyas, David A. Bader, Timothy A. Davis, James Kitchen, Timothy G. Mattson, Scott McMillan, and Erik Welch. 2021. LAGraph: Linear Algebra, Network Analysis Libraries, and the Study of Graph Algorithms. In *2021 IEEE IPDPSW*. 243–252.

[27] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. [n. d.]. *FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence*. 46–57.