Proceedings of CONF-CDS 2025 Symposium: Data Visualization Methods for Evaluation
DOI: 10.54254/2755-2721/2025.PO25411

Optimization Implementation and Performance Analysis of Divide-and-Conquer Algorithm Based on Python in Big Data Sorting and Retrieval

Yixuan Zheng

Newcastle University, Newcastle, England
zhengyixuan0929@outlook.com

Abstract. This paper offers an in-depth look at divide-and-conquer algorithms, especially in big data sorting and retrieval, with a particular focus on how these techniques are implemented in Python. As data sizes grow and become more complex, having scalable algorithms that are both efficient and flexible is more important than ever. Divide-and-conquer approaches are naturally suited for this task because they break down problems into smaller parts, making it easier to run tasks in parallel or across distributed systems—something that's incredibly useful in big data projects. In our review, we explore recent Python implementations of popular sorting algorithms like merge sort and quicksort. We compare how they perform when using multiprocessing or hybrid methods. While Python makes development straightforward and flexible, it also comes with some challenges—things like the Global Interpreter Lock (GIL), recursion limits, and the overhead of managing inter-process communication can impact performance. Our findings indicate that merge sort tends to perform better than quicksort when it comes to using Python's parallel processing capabilities. Besides, tools like PySpark or Dask can help overcome certain language-specific obstacles, making large-scale data processing more manageable. Overall, this review provides practical guidance for researchers and engineers aiming to strike a good balance between algorithm design and efficient implementation in systems that handle massive amounts of data.

Keywords: Divide-and-conquer, Python, Big data, Sorting algorithms, Parallel computing, Performance optimization

1. Introduction

1.1. Background and motivation

In today's data-driven environment, the fast expansion in data volume, variety, and processing speed is revolutionizing how we handle, interpret, and use information. Many sectors — including healthcare, finance, and environmental management — are increasingly dependent on large-scale datasets, which heightens the need for scalable and efficient processing methods. As emphasized in Song [1], big data analytics plays a critical role in revealing hidden patterns and supporting

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2025.PO25411

decision-making in complex systems. To address this, innovative processing frameworks like Apache Spark have emerged, offering enhanced parallelism and in-memory computing capabilities. According to Shaikh [2], Spark delivers substantial performance gains for both batch and streaming tasks compared to traditional Hadoop setups. These technological advancements emphasize the critical importance of developing algorithms that are both theoretically sound and adaptable to today's distributed environments. In this setting, divide-and-conquer algorithms become particularly appealing—providing natural parallelism and modularity. Implemented using modern programming tools like Python, these algorithms are well-suited to meet current data processing challenges.

1.2. Research gap

While sorting and retrieval algorithms have been extensively studied in classical computer science, their performance and adaptability in large-scale, real-world data environments remain inadequately explored. Traditional algorithmic analysis often focuses on theoretical complexity under idealized conditions, yet practical scenarios involving massive, heterogeneous data introduce new constraints such as memory bottlenecks, parallel execution overhead, and scalability limitations [2,3]. As noted by Chen et al. [4], many statistical and machine learning tasks require scalable estimation techniques, but few studies offer a unified perspective on their computational frameworks. Although divide-and-conquer algorithms are generally a good fit for big problems, most of the research doesn’t include detailed tests of how they actually perform when you try to put them into real code, especially in common programming languages like Python. Yang [5] and Durrani [6] demonstrate the potential and limitations of Python-based sorting, but a comprehensive synthesis of such works is still missing. This disconnect underscores the requirements for a structured review bridging algorithm theory, language implementation, and big data execution contexts.

1.3. Language focus

Python has become one of the most widely adopted languages in data science, machine learning, and big data analytics, owing to its rich ecosystem of libraries, ease of use, and strong community support. Tools such as NumPy, pandas, and multiprocessing provide accessible interfaces for implementing and experimenting with algorithmic designs, including divide-and-conquer strategies. However, Python is often criticized for performance limitations, particularly in CPU-bound tasks such as sorting and recursion-heavy algorithms. Yang [7] demonstrates that parallel implementations of merge sort using Python's multiprocessing module can yield substantial speedups under the right configurations. In contrast, Durrani and AbdulHayan [6] reveal that Python's recursion model and interpreter overhead may hinder the performance of quicksort-like algorithms, especially when compared to Java. These findings suggest that while Python may not always match the raw speed of lower-level languages, it offers a highly accessible environment for prototyping and evaluating scalable algorithmic strategies—a key advantage in real-world big data scenarios.

1.4. Purpose of this review

Given these considerations, this paper aims to provide a comprehensive review of divide-and-conquer algorithms applied to big data sorting and retrieval, with a particular emphasis on Python-based implementations. The review focuses on three interrelated dimensions: (1) the evolution and parallelization of divide-and-conquer algorithms in big data contexts; (2) the implementation practices, challenges, and optimizations of such algorithms using Python; and (3) the comparative

DOI: 10.54254/2755-2721/2025.PO25411

performance and adaptability of sorting and retrieval techniques under large-scale data conditions. This study doesn't introduce a brand new algorithm. Instead, it reviews what others have done, pointing out common patterns, pros and cons, and areas where more research is needed. It’s meant to give a clear starting point for future work.

2. Divide-and-conquer algorithms for big data processing

2.1. Theoretical foundations of divide-and-conquer algorithms

Divide-and-conquer is a common way to tackle tricky problems. It works by splitting the big problem into smaller parts, solving each one on its own, and then putting everything back together. This approach is well-known for its elegance and effectiveness in a wide range of computational problems, including sorting, matrix multiplication, and geometric computation.

A canonical example is merge sort, which recursively divides an array into halves, sorts each half, and then merges the sorted halves. This yields a time complexity of O(n log n), with a well-structured recursive process that lends itself naturally to parallel execution.

Bentley [7] was the first to clearly describe how the divide-and-conquer approach works in multiple dimensions. This method is more than good on paper; it’s also very practical for tackling complex, high-dimensional geometric problems. Horowitz and Zorat [8] later analysed how divide-and-conquer maps onto parallel architectures, showing that its modularity aligns well with distributed or multicore computation models.

The key strength of divide-and-conquer lies in its inherent decomposition structure, which enables:

- Independent subproblem execution (parallelism),
- Localized memory access (cache efficiency),
- Simplified reasoning about correctness and complexity.

These features make divide-and-conquer an attractive candidate for optimization in large-scale data processing, especially in systems where data can be naturally partitioned or streamed.

2.2. Applicability of divide-and-conquer in big data processing

Divide-and-conquer algorithms work well with big data because they're made up of smaller, manageable parts that can be processed at the same time. In large data tasks like sorting, searching, or calculating statistics, the data is often split across multiple computers or processors. This approach is so effective because each small piece can be handled independently, reducing the need for constant communication between parts.

For instance, when you're sorting a huge dataset—like a terabyte of information—you can adapt algorithms like merge sort to work in parallel. You can do this either on one machine with multiple cores or across several machines using tools like Apache Spark. This way, each part is sorted separately, which helps keep data close to where it’s processed and cuts down on unnecessary data transfer, leading to faster overall performance [2,4].

As Zikri [3] and Chen et al. [4] have pointed out, many divide-and-conquer strategies perform well on a large scale. They often beat traditional, single-block algorithms in speed. Plus, because they're recursive—meaning they regularly break problems down into smaller pieces—they fit nicely with frameworks like MapReduce, where the “divide” phase corresponds to the map step, and the “combine” phase mirrors reduce operations.

DOI: 10.54254/2755-2721/2025.PO25411

Another advantage is that divide-and-conquer methods are not tied to any one specific platform. You can write them in high-level languages like Python, then scale them up using multiprocessing or run them on distributed systems like Spark, even for handling petabytes of data. This flexibility makes them a top choice for optimizing algorithms in real-world big data setups.

3. Python-based implementation practices

3.1. Overview of Python's role in big data

Python has become one of the top choices for data science, analytics, and machine learning. People love it because it's easy to read, there's a huge range of libraries like NumPy, pandas, and scikit-learn, and it has a very active community. In big data projects, Python often sits at a high level, making it simple for researchers and engineers to try out ideas quickly, test new approaches, and implement algorithms without getting stuck in complex details.

While Python is not the fastest language out there—especially when compared to compiled languages like C++ or Java—it makes up for that with its flexibility and ability to work smoothly with modern computing frameworks. For example, it integrates well with tools like Apache Spark through PySpark, and libraries like multiprocessing, joblib, and Dask make parallel processing easier.

These features have encouraged numerous implementations of divide-and-conquer algorithms in Python, particularly for sorting and retrieval tasks. Researchers have used Python not only for algorithm prototyping, but also to benchmark performance and test parallel strategies on multi-core systems or distributed clusters.

3.2. Parallel implementation of mergesort

In her 2022 study, Yang [5] looked at different ways to speed up merge sort in Python. She mainly tried using the multiprocessing module and MPI with mpi4py to make sorting big arrays faster. The study showed that even though Python's Global Interpreter Lock (GIL) can be a bit of an obstacle, it can still boost performance by running tasks in parallel. Using multiprocessing to split jobs across multiple CPU cores makes things run much faster.

Yang [5] tested out different versions of merge sort, including a cool hybrid approach that combines multiprocessing with NumPy operations. When they ran this on a dataset of 10 million integers, the hybrid method was up to 34 times faster than doing everything sequentially. It even beat Python's built-in sorted() by about 50%. These tests were done on Indiana University's Carbonate supercomputer, using up to 24 cores.

They also investigated some of the main challenges. While multiprocessing worked well on systems where everything shares memory, its ability to scale was limited by the time it took to communicate between processes and to serialize data [5]. Using MPI-based approaches allowed for better scaling across multiple computers, but that came with more complexity in managing data and needed more advanced setup.

Overall, Yang's work shows that, if done carefully, using a divide-and-conquer approach to sorting in Python can really make effective use of multiple CPU cores [5]. This is helpful for anyone trying to find the right balance between making development easier and getting the best performance when working with big data.

DOI: 10.54254/2755-2721/2025.PO25411

3.3. Python vs Java in sorting algorithms

While many appreciate Python for its simplicity and easy-to-read code, some folks worry about how fast it runs compared to languages that are compiled, like Java. Back in 2022, researchers Durrani and AbdulHayan [6] did a thorough test comparing how quickly different sorting algorithms—such as quicksort, merge sort, insertion sort, selection sort, and bubble sort—performed when written in Python versus Java. Their tests showed some big differences in how long things took to run in the two languages. For instance, Java could sort 100,000 items in just about 0.112 seconds, while Python took over 17 seconds for the same task. This slowdown in Python is mainly because of limits on recursion stacks and the extra work the interpreter needs to do. On the flip side, merge sort performed better in Python, taking around 2.69 seconds compared to quicksort's 17.46 seconds, making it a more practical choice for Python users [6].

The researchers explained these differences by pointing out how each language executes codes. Java uses techniques like Just-In-Time (JIT) compilation and better memory management, which helps boost performance, especially for tasks like sorting. Python, on the other hand, runs through an interpreter and is affected by the Global Interpreter Lock (GIL), which can slow down CPU-heavy tasks. That said, the study also emphasized that Python's still a great option for quickly testing out ideas and algorithms, especially when combined with tools like multiprocessing to run things in parallel.

Overall, these findings show that when planning sorting methods or other performance-critical tasks, it's important to think about what the programming language is effective at and optimize based on your specific environment [6].

4. Discussion

4.1. Practical limitations of Python implementations

Python provides handy tools for implementing divide-and-conquer algorithms, but it does have some limitations, especially when working with large amounts of data or running multiple tasks at the same time. One big obstacle is the Global Interpreter Lock (GIL), which stops true multithreading in Python. This means that even CPU-heavy tasks like sorting can't take full advantage of multi-core processors just by using threads. To do this efficiently, you need to use more advanced approaches like multiprocessing or tools like joblib.

Moreover, Python imposes a limitation on recursion depth, and recursive calls are comparatively costly due to the language's interpreted nature. This affects the efficacy of algorithms such as quicksort, which depend significantly on extensive recursion. Durrani and AbdulHayan [6] indicated that Python's quicksort was markedly slower than that of Java, mostly due to this factor.

Another issue is memory management. Python dynamically allocates objects unlike statically typed languages, which causes overhead and less predictable rendering of trash collecting during high throughput sorting operations. Multiprocessing clearly shows the communication overhead since data must be serialised and sent across processes, therefore reducing performance.

Python's simplicity and adaptability help it to be constantly used despite these challenges. Understanding these constraints is essential while deciding on Python as the platform for parallel or high-performance algorithms implementation.

DOI: 10.54254/2755-2721/2025.PO25411

4.2. Summary of observations and trade-offs

The papers reviewed reveal some interesting trends about how divide-and-conquer methods are used in Python to handle large-scale data sorting and retrieval. For starters, because merge sort tends to have better recursion features and lower overhead [6], it often outperforms quicksort in Python. Also, even multiprocessing provides considerable speedups, but its benefits diminish as inter-process communication and data serialization introduce new bottlenecks [5].

These findings emphasize the many trade-offs that developers need to consider. Sometimes, they must choose between keeping things simple or pushing for better performance. For example, Python can't match the raw speed of languages like Java, even though it's easier to learn and has strong library support. Similarly, while tools like NumPy or PySpark can help you complete tasks faster, they might make it harder to fine-tune every detail or have precise control.

In the end, while divide-and-conquer approaches align well with ideas of parallel and distributed computing, how well they work in practice really depends on implementation choices, programming language features, and the platform you're working on. Picking the right algorithm for big data projects means carefully weighing these trade-offs to make the best decision.

4.3. Implications for algorithm design in big data

The outcomes of this review suggest several design problems for the implementation of divide-and-conquer techniques in applications of big data. First, programmers of algorithms must adapt their implementations to the guidelines and limitations of the targeted platform. Python can struggle somewhat with demanding duties like deep recursion or simultaneous operation of numerous processes, even if it is user-friendly and makes it easy to get things done swiftly. That emphasises the need to choose the right algorithms and implement them.

Second, split-and-conquer remains a potent and adaptable technique. However, it performs best when the effort is evenly distributed and the issue can be resolved orderly. In the actual world, where data can be mixed or arriving constantly, you often need additional techniques like energetic load balancing or combining several algorithms to handle things smoothly.

Third, decisions on language and structure should be taken knowing their trade-offs. Dask or PySpark tools enable developers to run jobs in parallel and increase the efficiency of divide-and-conquer strategies. They should also be conscious, though, of the possible loss in control and the learning curve these systems bring in.

Finally, big data algorithm design should not treat implementation and theory as separate phases. Instead, architectural decisions—including language, parallel model, and data structure assumptions—should be integrated into the early stages of algorithm design.

5. Conclusion

This paper reviewed recent advances in the implementation and optimization of divide-and-conquer algorithms for big data sorting and retrieval, with a particular emphasis on Python-based approaches. Through a systematic comparison of key studies, we observed that merge sort is consistently more suitable than quicksort for Python environments, especially when parallelized using multiprocessing. In addition, divide-and-conquer strategies align well with modern big data frameworks, offering natural decomposability and scalability.

Even though Python has some built-in limits, like the GIL, recursion depth, and communication costs, our analysis shows that, when designed carefully and with the right tools, it's still a solid

DOI: 10.54254/2755-2721/2025.PO25411

choice for developing and testing algorithms. By consolidating research findings across multiple works, this paper provides a structured reference for researchers and practitioners seeking to understand the trade-offs involved in applying divide-and-conquer techniques to large-scale data problems.

Future research could benefit from more empirical evaluations of divide-and-conquer algorithms in distributed environments, particularly using hybrid tools such as PySpark or Dask. Moreover, deeper exploration into dynamic load balancing and fault tolerance in recursive frameworks would further enhance the practicality of these algorithms in real-world big data systems.

References

[1] M.-L. Song, R. Fisher, J.-L. Wang, and L.-B. Cui, 'Environmental performance evaluation with big data: theories and methods', Ann. Oper. Res., vol. 270, no. 1, Art. no. 1, Nov. 2018, doi: 10.1007/s10479-016-2158-8.

[2] E. Shaikh, I. Mohiuddin, Y. Alufaisan, and I. Nahvi, 'Apache Spark: A Big Data Processing Engine', in 2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM), Nov. 2019, pp. 1–6. doi: 10.1109/MENACOMM46666.2019.8988541.

[3] Zikri, 'Performance Analysis of Sorting Algorithms in Big Data Environments: Efficiency, Scalability, and Practical Applications | Idea: Future Research'. Accessed: May 15, 2025. [Online]. Available: https://idea.ristek.or.id/index.php/idea/article/view/8

[4] X. Chen, J. Q. Cheng, and M. Xie, 'Divide-and-conquer methods for big data analysis', Feb. 22, 2021, arXiv: arXiv: 2102.10771. doi: 10.48550/arXiv.2102.10771.

[5] A. Yang, 'Approaches to the Parallelization of Merge Sort in Python', Nov. 26, 2022, arXiv: arXiv: 2211.16479. doi: 10.48550/arXiv.2211.16479.

[6] O. K. Durrani and S. AbdulHayan, 'Performance Measurement of Popular Sorting Algorithms Implemented using Java and Python', in 2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME), Nov. 2022, pp. 1–6. doi: 10.1109/ICECCME55909.2022.9988424.

[7] J. L. Bentley, 'Multidimensional divide-and-conquer', Commun ACM, vol. 23, no. 4, Art. no. 4, Apr. 1980, doi: 10.1145/358841.358850.

[8] Horowitz and Zorat, 'Divide-and-Conquer for Parallel Processing', IEEE Trans. Comput., vol. C–32, no. 6, Art. no. 6, Jun. 1983, doi: 10.1109/TC.1983.1676280.