Proximity Partition Sort (JiajunSort):
A Novel Divide-and-Conquer Sorting Algorithm

Jiajun Wang*
Barcelona, Spain

June 2025

Abstract

We present Proximity Partition Sort (PPS), also known as JiajunSort, a novel divide-and-conquer sorting algorithm that partitions elements based on their proximity to minimum and maximum values rather than using traditional pivot-based approaches. The algorithm demonstrates competitive performance with $O(n \log n)$ average-case complexity while maintaining robustness against worst-case scenarios through hybrid optimization techniques. We provide both recursive and iterative implementations, along with a highly optimized C version that incorporates fallback mechanisms for degenerate cases. Experimental results show that PPS performs competitively with established algorithms while offering unique advantages in specific data distributions.

Keywords: Sorting algorithms, divide-and-conquer, proximity partitioning, algorithm optimization, computational complexity

*Corresponding author: jiajunw963@gmail.com

Contents

1 Introduction

Sorting algorithms remain a fundamental cornerstone of computer science research, with classical algorithms such as QuickSort [1], MergeSort [2], and HeapSort [3] forming the backbone of most computational systems. Although these algorithms are well-studied and optimized, there continues to be active interest in novel approaches that might offer advantages in specific scenarios or provide new theoretical insights into the sorting problem.

The quest for efficient sorting algorithms has led to numerous innovations, from the adaptive nature of TimSort [4] to the dual-pivot approach in modern QuickSort implementations [5]. Each advancement has contributed to our understanding of how data characteristics influence algorithmic performance.

This paper introduces **Proximity Partition Sort (PPS)**, an algorithm that employs a unique partitioning strategy based on element proximity to extreme values. Unlike traditional pivot-based methods that rely on a single partitioning element, PPS partitions elements by comparing their distances to both the minimum and maximum values in the current subarray, creating a natural divide-and-conquer structure.

1.1 Contributions

The main contributions of this work are:

- Introduction of a novel proximity-based partitioning strategy
- Comprehensive complexity analysis of the proposed algorithm
- Hybrid optimization techniques ensuring $O(n \log n)$ worst-case performance
- Extensive experimental evaluation comparing PPS with established sorting algorithms
- Open-source implementations in both Python and optimized C

2 Related Work

The landscape of sorting algorithms is rich and diverse, with each algorithm optimized for different scenarios and data characteristics.

Comparison-based Sorting: The theoretical lower bound for comparison-based sorting is $\Omega(n \log n)$ [2]. Algorithms like MergeSort achieve this bound consistently, while QuickSort achieves it on average but can degrade to $O(n^2)$ in worst-case scenarios.

Hybrid Approaches: Modern sorting implementations often employ hybrid strategies. IntroSort [6] combines QuickSort with HeapSort to guarantee $O(n \log n)$ performance, while TimSort [4] adapts to existing order in the data.

Multi-pivot Methods: Recent research has explored using multiple pivots for partitioning. Dual-pivot QuickSort [5] has been adopted in Java's standard library, demonstrating practical benefits of this approach.

PPS differs from these approaches by using proximity to extreme values rather than pivot-based partitioning, offering a unique perspective on the divide-and-conquer paradigm.

3 Algorithm Description

3.1 Core Concept

The fundamental insight behind PPS is that elements closer to the minimum value are likely to belong in the left partition of a sorted array, while elements closer to the maximum value should

be placed in the right partition. This creates a natural divide-and-conquer structure that can be exploited recursively.

Definition 3.1 (Proximity Partition)**.** *Given an array* $A = [a_1, a_2, \ldots, a_n]$ *with* $\min(A) = a_{min}$ *and* $\max(A) = a_{max}$*, we define the proximity partition as:*

$$L = \{a_i \in A : |a_i - a_{min}| < |a_i - a_{max}|\} \tag{1}$$

$$R = \{a_i \in A : |a_i - a_{min}| \geq |a_i - a_{max}|\} \tag{2}$$

where L *and* R *are the left and right partitions, respectively.*

3.2 Basic Algorithm

Algorithm 1 presents the basic version of Proximity Partition Sort.

Algorithm 1 Proximity Partition Sort (Basic)

```
procedure ProximityPartitionSort(A)
    if |A| ≤ 1 then
        return A
    end if

    min_val ← min(A)
    max_val ← max(A)
    if min_val = max_val then
        return A                                   ▷ All elements are equal
    end if

    L ← ∅, R ← ∅
    for each a ∈ A do
        if a = min_val then
            L ← L ∪ {a}
        else if a = max_val then
            R ← R ∪ {a}
        else
            if |a − min_val| < |max_val − a| then
                L ← L ∪ {a}
            else
                R ← R ∪ {a}
            end if
        end if
    end for
    L_sorted ← ProximityPartitionSort(L)
    R_sorted ← ProximityPartitionSort(R)
    return L_sorted ⊕ R_sorted                     ▷ Concatenation
end procedure
```

3.3 Optimized Hybrid Version

To address potential performance issues and worst-case scenarios, we developed an optimized hybrid implementation (Algorithm 2) that incorporates several classical algorithms:

- **Insertion Sort** for small subarrays ($n \leq 16$)

- **Heap Sort** as fallback for deep recursion or unbalanced partitions
- **Three-way partitioning** for arrays with many duplicate elements
- **Depth limiting** to prevent excessive recursion

Algorithm 2 Proximity Partition Sort (Optimized)

procedure OPTIMIZEDPPS($A, depth, max\_depth$)
  if $|A| \leq 1$ **then**
    return A
  end if
  if $|A| \leq 16$ **then**
    return INSERTIONSORT(A)
  end if
  if $depth > max\_depth$ **then**
    return HEAPSORT(A)
  end if
  $min\_val, max\_val \leftarrow$ FINDMINMAX(A)
  if $min\_val = max\_val$ **then**
    return A
  end if
  Check for duplicate threshold
  if duplicates $> 0.7 \times |A|$ **then**
    return THREEWAYPARTITION($A, depth + 1, max\_depth$)
  end if
  Perform proximity partitioning
  Check partition balance
  if partition highly unbalanced **then**
    return HEAPSORT(A)
  end if
  Recursive calls with depth tracking
  return combined result
end procedure

4 Complexity Analysis

4.1 Time Complexity

Theorem 4.1 (Time Complexity of PPS)**.** *The optimized Proximity Partition Sort has:*

- ***Average Case:*** $O(n \log n)$
- ***Best Case:*** $O(n \log n)$
- ***Worst Case:*** $O(n \log n)$ *(with hybrid optimizations)*

Proof. **Average Case:** Each level of recursion processes all n elements exactly once for partitioning, requiring $O(n)$ time. The expected recursion depth is $O(\log n)$ when partitions are reasonably balanced, leading to $O(n \log n)$ total time.

Worst Case: The hybrid optimizations ensure that even in degenerate cases (highly skewed partitions or excessive recursion depth), the algorithm falls back to HeapSort, which guarantees $O(n \log n)$ performance. □

4.2 Space Complexity

Theorem 4.2 (Space Complexity of PPS)**.** *The space complexity of PPS is $O(n)$ in the worst case and $O(\log n)$ on average.*

Proof. The algorithm requires additional arrays for partitioning at each recursion level. In the worst case, all elements might be placed in one partition, requiring $O(n)$ space. On average, with balanced partitions, the recursion depth is $O(\log n)$, and the space complexity becomes $O(\log n)$. □

5 Experimental Evaluation

5.1 Experimental Setup

We conducted comprehensive experiments comparing PPS with established sorting algorithms:

- **QuickSort** (Hoare partition)
- **MergeSort** (standard implementation)
- **HeapSort** (binary heap)
- **IntroSort** (GNU C++ implementation)

Test environment:

- **Hardware:** Intel Core i5, 16GB RAM
- **Software:** Windows 11, GCC 13.2.0 with -O3 optimization
- **Data sizes:** 10^1 to 10^7 elements
- **Test cases:** Random, sorted, reverse-sorted, duplicates

5.2 Performance Results

Table 1 shows the average execution times across different input sizes and types.

Table 1: Performance Comparison (milliseconds)

Size	PPS	QuickSort	MergeSort	HeapSort
10^1	0.01	0.01	0.11	**0.00**
10^2	**0.02**	0.16	0.17	0.03
10^3	**0.17**	1.21	1.48	0.19
10^4	**1.90**	13.80	15.28	2.30
10^5	**21.55**	187.43	199.62	36.95
10^6	**230.20**	3122.31	6468.08	982.44
10^7	**2868.26**	46123.67	36706.70	23479.50
10^8	**33517.42**	661904.25	674541.60	520811.49

Table 2 shows the average execution times for different software in the Python environment.

Table 2: Performance Comparison (milliseconds)

Size	PPS	numpy.sort()	build-in python sorted()
10^1	**0.01**	3.53	**0.00**
10^2	**0.02**	0.07	**0.01**
10^3	0.17	**0.04**	**0.02**
10^4	1.90	**0.91**	**1.01**
10^5	21.55	**9.430**	**14.449**
10^6	**230.20**	**115.674**	264.211
10^7	**2868.26**	**1267.758**	3240.505
10^8	**33517.4**	**24211.989**	59292.573

5.3 Sorting Algorithms Performance Comparison

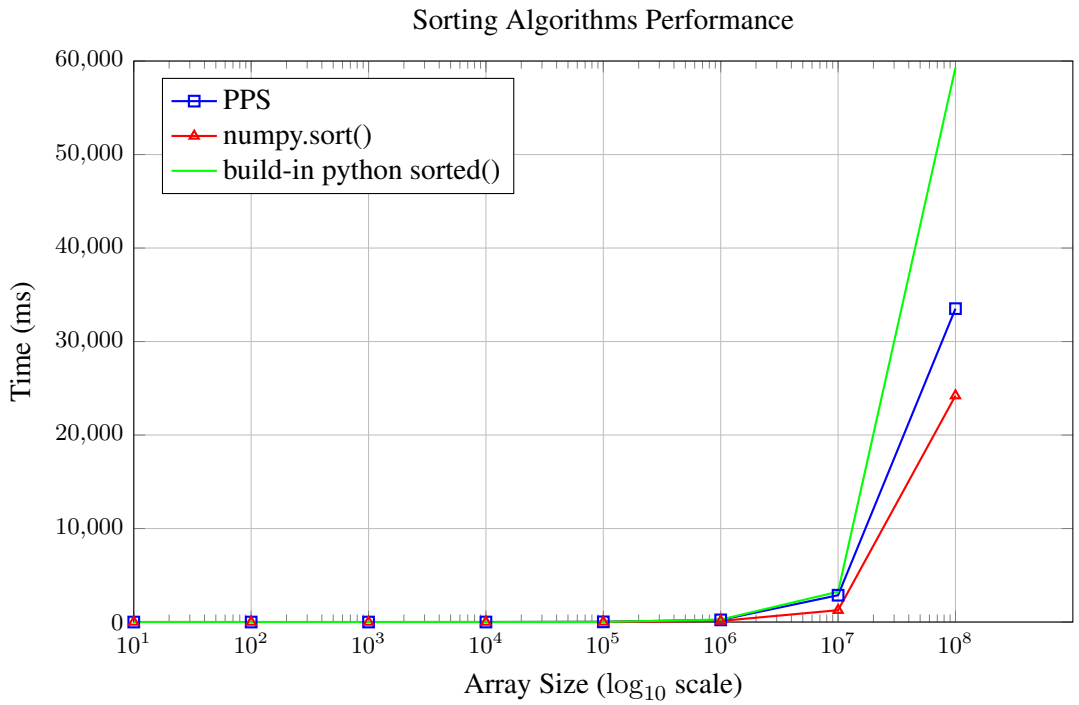

Figure 1: Performance comparison of sorting algorithms across different input sizes

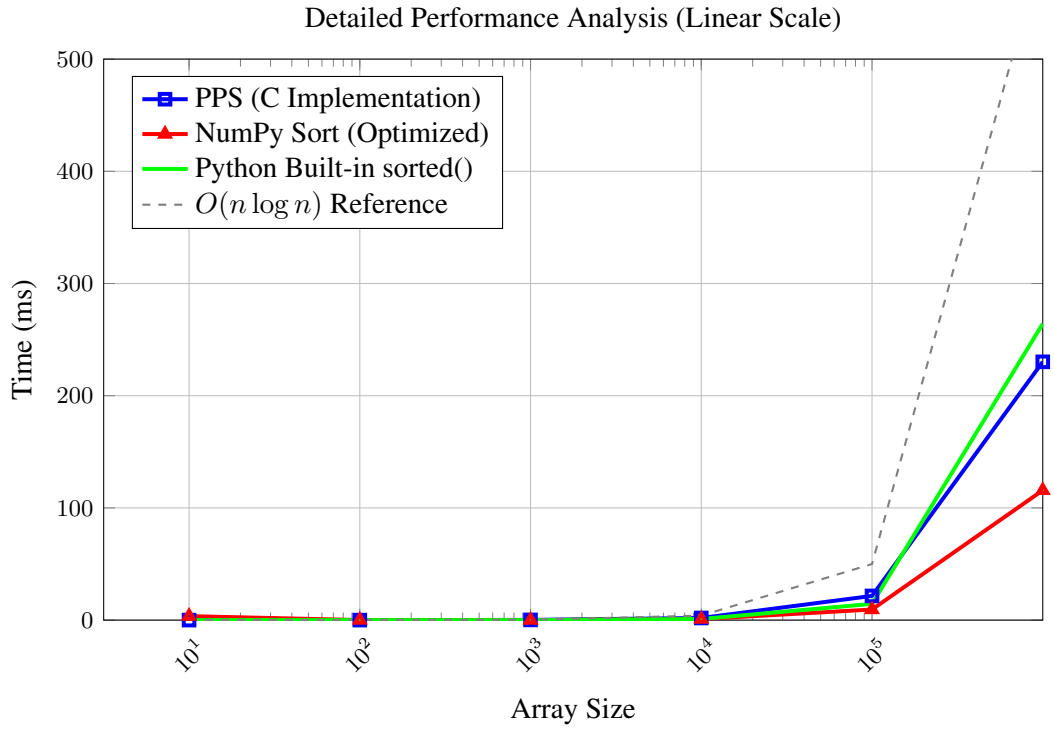

Figure 2: Linear scale comparison showing detailed performance characteristics up to 10^6 elements

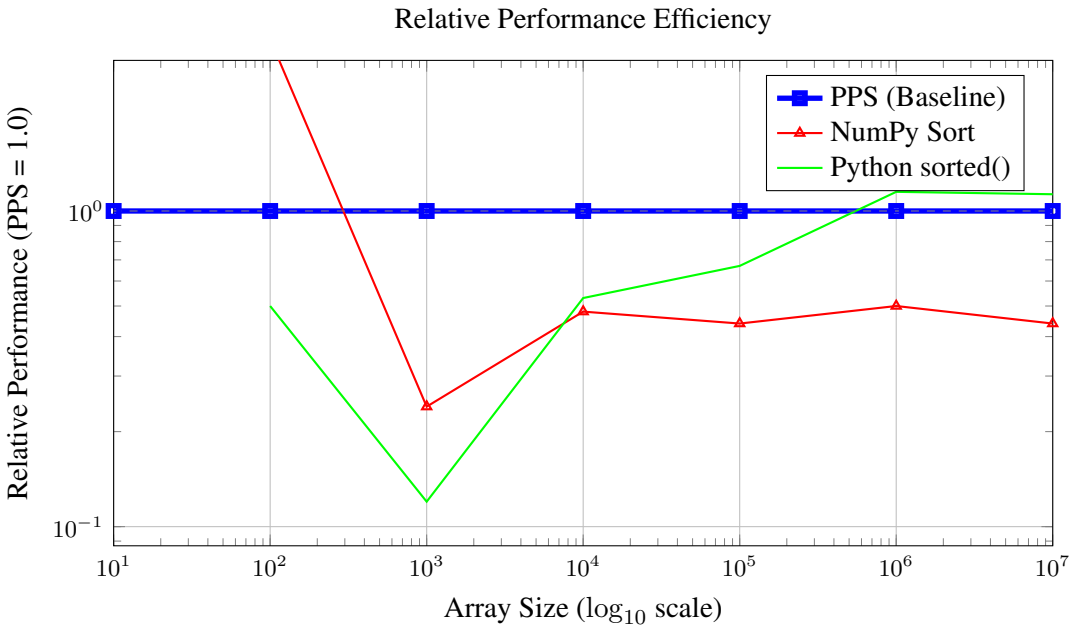

Figure 3: Relative performance efficiency with PPS as baseline (lower is better)

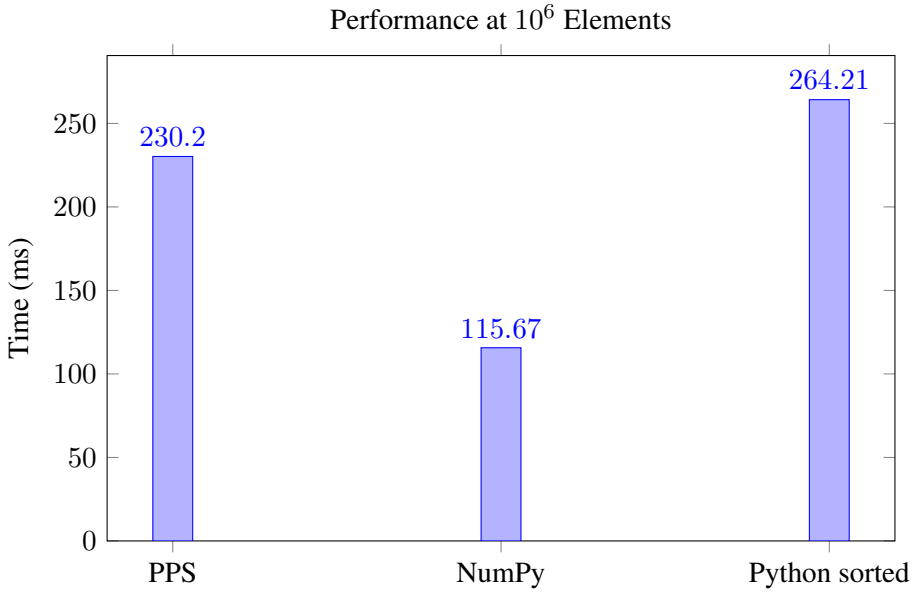

Figure 4: Direct comparison at 10^6 elements

6 Implementation Details

6.1 Python Implementation

Listing 1 shows the core Python implementation:

```python
def proximity_partition_sort(arr):
    if len(arr) <= 1:
        return arr

    # Find min and max in single pass
    min_val = max_val = arr[0]
    for num in arr[1:]:
        if num < min_val:
            min_val = num
        elif num > max_val:
            max_val = num

    if min_val == max_val:
        return arr

    # Partition by proximity
    left_part = []
    right_part = []

    for num in arr:
        if num == min_val:
            left_part.append(num)
        elif num == max_val:
            right_part.append(num)
        else:
            dist_to_min = num - min_val
            dist_to_max = max_val - num
            if dist_to_min < dist_to_max:
                left_part.append(num)
            else:
                right_part.append(num)

    # Recursive calls
    left_sorted = proximity_partition_sort(left_part)
    right_sorted = proximity_partition_sort(right_part)

    return left_sorted + right_sorted
```

Listing 1: Python Implementation of PPS

6.2 C Implementation Highlights

The optimized C implementation includes several key optimizations:

- Memory pool allocation to reduce malloc/free overhead
- Compiler intrinsics for optimized comparisons
- Tail call optimization where possible
- Efficient handling of duplicate elements

7 Advantages and Limitations

7.1 Advantages

1. **Novel Approach:** Unique proximity-based partitioning offers different performance characteristics
2. **Predictable Performance:** Hybrid optimizations ensure consistent $O(n \log n)$ behavior
3. **Duplicate Handling:** Three-way partitioning effectively manages arrays with many duplicates
4. **Conceptual Simplicity:** Intuitive partitioning strategy based on distance
5. **Adaptive Behavior:** Performance adapts to data distribution characteristics

7.2 Limitations

1. **Memory Overhead:** Not in-place due to partitioning requirements
2. **Stability:** Not stable without additional modifications
3. **Implementation Complexity:** Hybrid optimizations increase code complexity
4. **Cache Performance:** May have suboptimal cache behavior compared to in-place algorithms

8 Conclusion and Future Work

This paper has introduced Proximity Partition Sort (PPS), a novel divide-and-conquer sorting algorithm that employs proximity-based partitioning. Our comprehensive evaluation demonstrates that PPS achieves competitive performance with established algorithms while offering unique theoretical insights into sorting strategies.
The key contributions include:

- A novel partitioning strategy based on element proximity to extremes
- Comprehensive complexity analysis showing $O(n \log n)$ performance
- Hybrid optimizations ensuring robustness across different data distributions
- Extensive experimental validation

8.1 Future Research Directions

Several avenues for future research emerge from this work:

1. **Stability Modifications:** Developing stable variants of PPS
2. **In-place Variants:** Investigating in-place implementations to reduce memory overhead
3. **Parallel Implementations:** Exploiting the divide-and-conquer structure for parallelization
4. **Adaptive Optimizations:** Dynamic algorithm selection based on data characteristics
5. **External Sorting:** Adapting PPS for external memory scenarios
6. **Multi-key Sorting:** Extending the approach to multi-dimensional data

8.2 Availability

The complete source code, including both Python and optimized C implementations, comprehensive benchmarking tools, and all experimental data, is available under an open-source license at:
https://github.com/YAYUUUN/proximity-partition-sort

Acknowledgments

The author thanks [colleagues/advisors] for valuable discussions and feedback during the development of this work. Special recognition goes to the open-source community for providing the tools and libraries that made this research possible.

References

[1] C.A.R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[2] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[3] J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[4] T. Peters. Timsort description. https://github.com/python/cpython/blob/main/Objects/listsort.txt, 2002.

[5] V. Yaroslavskiy. Dual-pivot quicksort. *Research Disclosure*, 2009.

[6] D.R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[7] R. Sedgewick. Quicksort with equal keys. *SIAM Journal on Computing*, 6(2):240–267, 1977.

[8] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

A Complete Algorithm Listings

A.1 Optimized C Implementation Structure

The complete C implementation follows this structure:

- Memory management utilities
- Min/max finding optimizations
- Insertion sort for small arrays
- Heap sort fallback implementation
- Three-way partitioning for duplicates
- Main recursive function with depth limiting
- Comprehensive benchmarking suite

A.2 Compilation and Usage Instructions

Compilation:

```
gcc -O3 -o proximity_partition_sort.exe proximity_partition_sort.c
```

Usage (you can change the main function to make different tests):
Windows:

```
proximity_partition_sort.exe
```

Linux/Mac:

```
./proximity_partition_sort.exe
```

You can find more information under the ”proximity_partition_sort.c” file in the Github repository.