CSEE4824 Final Project Report

Charlotte Chen (HC3558), Jill Kang (JK4491)
Tianyun Huang (TH3116), and Yu Jia (YJ2839)

December 5, 2024

Introduction

In this project, we explored the performance of four different sorting algorithms: **Radix Sort**, **Quick Sort**, **Merge Sort**, and **Tim Sort** over 5GB 32 bit integers over two different processors. We evaluated the change of performance under different optimization levels and the cost-effectiveness of each configuration and implementation.
We choose two processors to execute the task: **Processor 1**, a general-purpose CPU and **Processor 2** is a high-performance CPU. The specific parameters of the two processors are shown in the table below.

Feature	Processor 1 (n1-standard-1)	Processor 2 (c4-standard-2)
Machine Type	n1-standard-1	c4-standard-2
CPU Platform	Intel Haswell	Intel Emerald Rapids
Threads per Core	1	2
Cores per Socket	1	2
L1 Cache	32 KiB	48 KiB
L2 Cache	256 KiB	2 MiB
L3 Cache	45 MiB	260 MiB
Monthly Estimate	$26.27	$74.11

Table 1: Comparison of Two Machine Configurations

Methodology

We randomly generate 5GB of 32-bit integers and run the naive and optimized versions of the four sorting algorithms on the two processors to compare their performance and compute the cost-effectiveness of each configuration. Four sorting algorithms are listed as follows:

Advantages of each Sorting Algorithm

- **Merge Sort**

Sorting Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Stability	Space Complexity
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Yes	$O(\log n)$
Radix Sort	$O(n \cdot k)$	$O(n \cdot k)$	$O(n \cdot k)$	Yes	$O(n + k)$
Tim Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	Yes	$O(n)$

Table 2: Comparison of Sorting Algorithms

  - **Algorithm**: Merge Sort splits the data into smaller subarrays, sorts these subarrays, and then merges them back together in sorted order, making sorting efficient.
  - **Memory Locality**: During merging, it has sequentially accesses subarrays, exhibiting good spatial locality. As recursive calls repeatedly access overlapping regions of the original array, it also has good temporal locality.
  - **Data Structure**: Merge Sort's divide-and-conquer approach fits naturally with arrays, leading to efficient memory access patterns.

- **Quick Sort**
  - **Algorithm**: Quick Sort selects a pivot, partitions the array into two subarrays, and recursively sorts these subarrays, achieving $O(n \log n)$ time complexity and $O(\log n)$ space complexity.
  - **Memory Locality**: Quick Sort exhibits good spatial locality by working on contiguous segments of the array during partitioning.
  - **Data Structure**: Quick Sort's in-place partitioning mechanism is highly efficient for arrays, minimizing additional memory usage.

- **Radix Sort**
  - **Algorithm**: Radix Sort processes the input numbers digit by digit from LSD, and uses counting sort at each step to sort the numbers by the current digit, achieving linear time complexity.
  - **Memory Locality**: Radix Sort processes data in a linear pass through the input for each digit, which results in good spatial locality.
  - **Data Structure**: Radix Sort works best on arrays where the keys have a fixed length. Its non-comparative nature makes it ideal for data that can be sorted by fixed-width keys.

- **Tim Sort**
  - **Algorithm**: Tim Sort is a hybrid sorting algorithm that combines the advantages of Merge Sort and Insertion Sort. In our implementation, we compute the optimal run size based on the input size.
  - **Memory Locality**: Tim Sort optimizes memory locality by using Insertion Sort for small runs, which operates on contiguous elements in memory.
  - **Data Structure**: Tim Sort is designed for arrays and takes advantage of pre-existing order in the data, leading to perfound performance in cache access.

Optimizations Method

1. **Parallel Optimizations with OpenMP**

   - We noticed that in each of the four sorting algorithms, we need to compare a large amount of numbers in a array with contingent indexes(in tim sort and merge sort) and multiple loops for different portion of data (radix sort). Thus, we ultized OpenMP directives to parallelize the comparison process. The effectiveness of such multi-threads instructions depends on the number of threads available on the processor. Additionally, we have to protect the shared data chunk from racing condition, which will lead to overhead and complexity.

2. **AVX2 Manual Optimization**

   - Similar to the parallel optimization, we ultized AVX2 instructions to speed up the load and store process of contiguous data in the array. With intrinsic functions provided by the compiler, we can load 8 32-bit integers, compare them, and store the result in one instructions. This will largely reduce the time used in load and store, leading to significant improvement in performance.

3. **Algorithm level Optimization**

   - We also optimized the algorithm itself to improve the performance. In tim sort, we used binary search for the correct position of the key, reduced the complexity from $O(n)$ to $O(\log n)$. In quick sort, we used the median of three pivot selection to reduce the worst case time complexity from $O(n^2)$ to $O(n \log n)$.
   - Function and data Structure optimization is also applied. We used memory write/read instead of array access to reduce the time used in memory access. We also implemented a min-heap to merge the sorted chunks into final output. This leads to a $O(n)$ time complexity in merge step, with a $O(1)$ space complexity.

Results

Implemented on E2

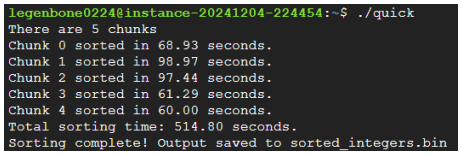
As shown in Figure 1,this is the runtime of the optimized four algorithms on Processor1.

Implemented on C4

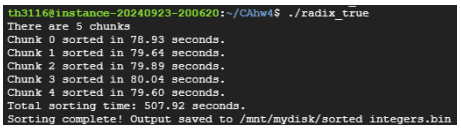
As shown in Figure 2,this is the runtime of the optimized four algorithms on Processor2.

Naive Implementation

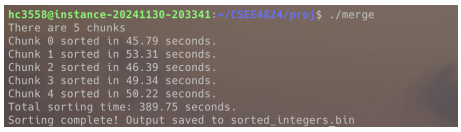
We run the naive implementation of the four algorithms on Processor1 with simple merge chunk function. The results are shown in following table.



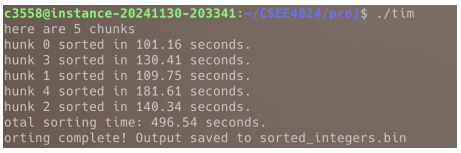
(a) Quick Sort on E2



(b) Radix Sort on E2

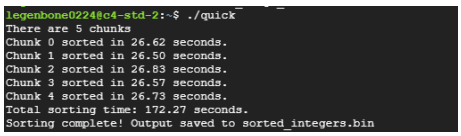


(c) Merge Sort on E2

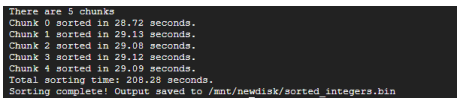


(d) Tim Sort on E2

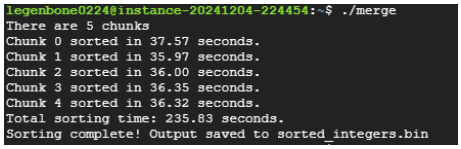
Figure 1: Execution Time of Sorting Algorithms on E2 Instance



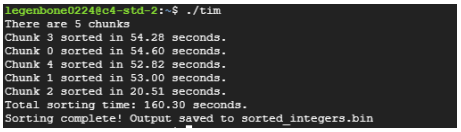
(a) Quick Sort on C4



(b) Radix Sort on C4



(c) Merge Sort on C4



(d) Tim Sort on C4

Figure 2: Execution Time of Sorting Algorithms on C4 Instance

Algorithm	Quick Sort	Radix Sort	Merge Sort	Tim Sort
Execution Time (s)	2140.6	2389.7	1960.8	2127.2
E2 Speedup	4.16	4.71	5.06	4.37
C4 Speedup	12.44	11.48	8.34	13.29

Table 3: Execution Time and Speedup

Cost-Effectiveness

From the performance results, we can observe that **Tim Sort** using **AVX-256** optimization on **Processor 2** achieves the fastest execution time. The total execution time for sorting and merging 5GB is the shortest at 160.30 seconds.

Next, the cost-effectiveness of each configuration is evaluated as follows:

$$\text{Cost per second} = \frac{\text{Price per hour}}{3600 \times \text{Total execution time (seconds)}}$$

For **AVX-256 on Processor 2**, with a price per hour of \$0.10:

$$\text{Cost per second} = \frac{0.1}{3600} \times 160.30 = 4.4 \times 10^{-3}$$

For **Auto-Optimized on Processor 1**, with a price per hour of \$0.04:

$$\text{Cost per second} = \frac{0.04}{3600} \times 389.75 = 4.33 \times 10^{-3}$$

From these calculations, we can see that Tim Sort on **AVX-256 on Processor 2** is the fastest, and it also offers the best cost-performance ratio.

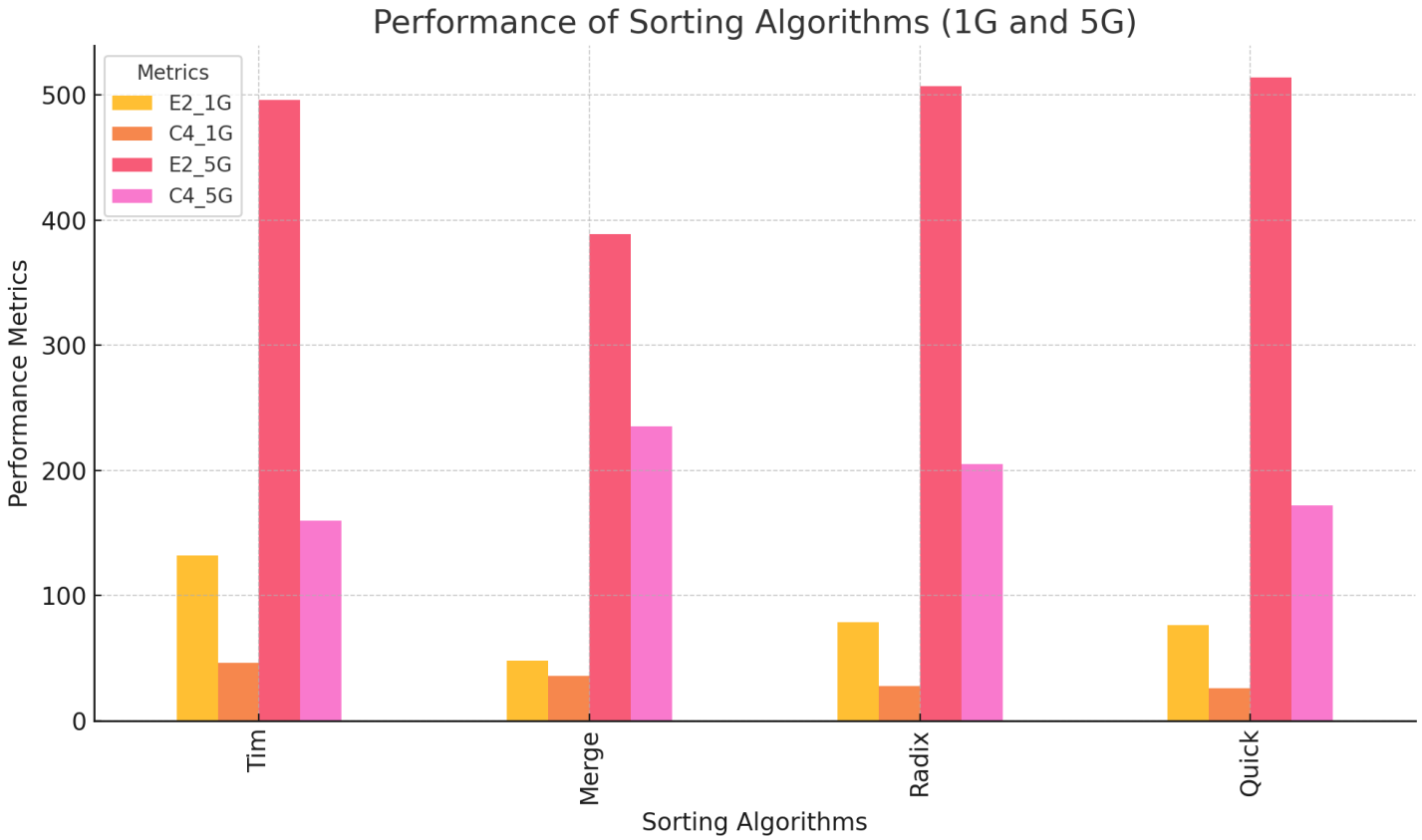


Figure 3: sort performance

Conclusion

- **Different algorithms are suited for different data scales:** For example, quick sort is the fastest for 1G-sized data but becomes the slowest for 5G-sized data. This is because its partitioning and recursive nature make it time-consuming for large-scale data. In contrast, the multiple passes and additional memory overhead of Radix sort make it more suitable for smaller-scale data.
- **Multi-threaded processors improve performance:** Due to C4 utilizing two threads, its processing time is at least halved compared to E2, demonstrating the benefit of parallelism.
- **Programs with multiple sequential array traversals benefit more from SIMD operations:** Both quick sort and radix sort show significant performance improvements when SIMD operations are added. This is likely because these algorithms require multiple array traversals and thus gain more from optimized memory access patterns.

A Appendix

Work Log

Date	Time	Task	Collaborator(s)
2024-11-20	17:00 - 19:00	Discuss the topic in a meeting.	All
2024-11-21	21:00 - 20:00	Set up the environment.	T.Y. Huang, Y. Jia
2024-11-23	15:00 - 17:00	Finish the Radix Sort Baseline.	C. Chen, J. Kang
2024-11-24	18:00 - 22:00	Finish the Quik Sort Baseline.	Y. Jia
2024-11-26	11:00 - 20:00	Working on the optimization	C. Chen, T.Y. Huang
2024-11-27	15:00 - 18:00	Complete the optimization.	T.Y. Huang
2024-11-28	17:00 - 20:00	Modify the code in the merge chunk section.	C. Chen
2024-12-1	17:00 - 20:00	Complete the report.	All

Table 4: Work log documenting project progress.

Folder Structure

Our project folder is structured as follows:

```
+-- code
|   +-- hw4_merge_sort.c // Merge Sort driver program
|   +-- hw4_quick_sort.c // Quick Sort driver program
|   +-- hw4_radix_sort.c // Radix Sort driver program
|   +-- hw4_tim_sort.c  // Tim Sort driver program
|   +-- chunk.c
// The sort chunk function divides a large file into multiple chunks,
// reads each chunk into memory, sorts it, and stores the sorted result in separate files.
|   +-- chunk.h
|   +-- sort.c
// Implementing four different sorting algorithms.
|   +-- sort.h
|   +-- testfile.py
// Generating a file with random integers that is 5GB in size.
|   +-- testoutput.py
// Verifying that the output file is sorted correctly.
|   +-- Makefile
// Makefile for compiling the four sorting algorithms.
+-- report
|   +-- report.tex
+-- .gitignore
```

Scripts

1. Data Generation

```
#!/bin/bash
# This script generates a file with random integers that is 5GB in size.
# Since I used a text file as output,
# the size of the file is substantially larger than 5GB.
# Thus, it takes a while to generate the file.

# Author: Charlotte Chen

import random

target_size_gb = 0.01
target_size_bytes = target_size_gb * 1024**3
num_integers = int(target_size_bytes // 4)


use_bin = True  # Set to True for binary output


output_file_1 = "random_integers_10MB.txt"
output_file_2 = "random_integers_10MB.bin"


output_file = output_file_2 if use_bin else output_file_1

print(
    f"Generating {num_integers:,} integers to create a {target_size_gb}GB file...")


if use_bin:
    print(f"Writing binary file: {output_file}")
    with open(output_file, "wb") as f:
        for _ in range(num_integers):
            random_int = random.randint(0, 2**32 - 1)
            f.write(random_int.to_bytes(4, byteorder="little"))
else:
    print(f"Writing text file: {output_file}")
    with open(output_file, "w") as f:
        for _ in range(num_integers):
```

```
            random_int = random.randint(0, 2**32 - 1)
            f.write(f"{random_int}\n")

print(f"File '{output_file}' with size {target_size_gb}GB has been created.")
```

2. Output Verification

```
#!/bin/bash
# This script verifies that the output file is sorted correctly.
# The memory usage of this script is O(1) because it only reads one element at a time.
# Please run this script with the command: python verify_sort.py <sorted_file>.
# The script supports both text and binary formats.

# Author: Charlotte Chen

use_bin = False  # Set to True if the output file is binary


def verify_sorted_text_file(filename):
    """Verifies a text-based sorted file."""
    try:
        with open(filename, "r") as file:
            prev = int(file.readline().strip())

            for line in file:
                curr = int(line.strip())

                if curr < prev:
                    print(
                        f"Error: File is not sorted. Found {prev} before {curr}.")
                    return False

                prev = curr

        print("File is sorted correctly.")
        return True
    except Exception as e:
        print(f"An error occurred while processing the text file: {e}")
        return False
```

```
def verify_sorted_binary_file(filename):
    """Verifies a binary-based sorted file."""
    try:
        with open(filename, "rb") as file:
            prev = int.from_bytes(
                file.read(4), byteorder="little", signed=True)

            while True:
                data = file.read(4)
                if not data:
                    break

                curr = int.from_bytes(data, byteorder="little", signed=True)
                if curr < prev:
                    print(
                        f"Error: File is not sorted. Found {prev} before {curr}.")
                    return False

                prev = curr

        print("File is sorted correctly.")
        return True
    except Exception as e:
        print(f"An error occurred while processing the binary file: {e}")
        return False


if __name__ == "__main__":
    import sys

    if len(sys.argv) != 3:
        print("Usage: python testoutput.py <sorted_file> <flag>(txt/binary)")
        sys.exit(1)

    filename = sys.argv[1]
    flag = sys.argv[2].lower()
    if flag == "txt":
        use_bin = False
    elif flag == "binary":
        use_bin = True
```

```
    else:
        print("Invalid flag. Please enter 'txt' or 'binary'.")
        sys.exit(1)

    if use_bin:
        print("Verifying binary file...")
        result = verify_sorted_binary_file(filename)
    else:
        print("Verifying txt file...")
        result = verify_sorted_text_file(filename)

    if result:
        print("Test passed: The file is sorted.")
        sys.exit(0)
    else:
        print("Test failed: The file is not sorted.")
        sys.exit(1)
```

3. Makefile

```
# Compiler
CC = clang

# Compiler Flags
CFLAGS = -fopenmp -O3 -march=native -mavx2 -g

# Source Files
QUICK_SRC = hw4_quick_sort.c chunk.c sort.c
TIM_SRC = hw4_tim_sort.c chunk.c sort.c
RADIX_SRC = hw4_radix_sort.c chunk.c sort.c
MERGE_SRC = hw4_merge_sort.c chunk.c sort.c

# Targets
QUICK_TARGET = quick
TIM_TARGET = tim
RADIX_TARGET = radix
MERGE_TARGET = merge

#
# Build Rules
```

```
#

all: $(QUICK_TARGET) $(TIM_TARGET) $(RADIX_TARGET) $(MERGE_TARGET)

$(QUICK_TARGET): $(QUICK_SRC)
 $(CC) $(CFLAGS) -o $@ $^

$(TIM_TARGET): $(TIM_SRC)
 $(CC) $(CFLAGS) -o $@ $^

$(RADIX_TARGET): $(RADIX_SRC)
 $(CC) $(CFLAGS) -o $@ $^

$(MERGE_TARGET): $(MERGE_SRC)
 $(CC) $(CFLAGS) -o $@ $^

#
# Clean
#

clean:
 @rm -f $(QUICK_TARGET) $(TIM_TARGET) $(RADIX_TARGET) $(MERGE_TARGET) chunk_*.bin sorted_integ
 @echo "Cleaned up build files and chunk files."
```