Contents

1 Proposed Outline

- define outline - Preface - Acknowledgement - Bookmanager - Introduction - What this is doing - Why it matters - Related Research - List references - Explain what others have done - Installation - Python virtual environment - Installation of mpi4py - Installation of NumPy and Jax - Cloudmesh - Cloudmesh.StopWatch - Sequential sorting - Overview of sequential sorting algorithms - Algorithms/implementations - Sequential merge sort - Insertion sort - Bubble sort - Quick sort - Parallel sorting - Related algorithms - Bitonic sort - Multiprocessing merge sort - MPI merge sort - MPI builtin merge sort - MPI sequential merge sort - MPI adaptive merge sort - One way bitonic sort? - Sorting on GPU - Jax NumPy - Performance comparision - Visualization - Automated Jupyter notebooks for performance comparison - Conclusion

2 Preface

3 Introduction

4 Installation

5 Sequential Sorting

5.1 Recursive Merge Sort

Recursive merge sort is a standard example of a divide-and-conquer algorithm. The algorithm can be split into two parts: splitting and merging.

Merging describes the merging of two sorted lists into one sorted list. Two ways of doing so are described below.

Method 1: Iterative Method

Assign indexes to the beginning of each array. Check for the smaller of each element at the current index and add it to the end of the sorted list. Increment the index of the list whose number was found. Once one array is empty, we can simply append the other one onto the end of the sorted list.

```
python3 code to demonstrate merging two sorted lists using the iterative method

right = [1, 2, 6, 7, 15]
left = [3, 4, 5, 8, 13]

res = []
left and right indexes
i, j = 0, 0

while i < len(left) and j < len(right):
    find the smallest value at either index
    and append it to the sorted list
    if left[i] < right[j]:
        res.append(left[i])
        i += 1
    else:
        res.append(right[j])
        j += 1
```

```
at least one array is empty
append both onto the end of the sorted list
res = res + left[i:] + right[j:]
```

Method 2: Using *sorted()*

By using the built-in Python function *sorted()*, we can merge the two lists in one line.

```
right = [1, 2, 6, 7, 15]
left = [3, 4, 5, 8, 13]

concatenate and sort
res = sorted(left + right)
```

Once we can merge two sorted arrays, the rest of mergesort follows as such:

Given an array of length n,

1. If n > 1, divide the array into two halves, or subarrays; 2. Use mergesort to sort each of the two subarrays; 3. Merge the two sorted subarrays into one sorted array.

```
def sequential_mergesort(array):
    n = len(arr)
    if n > 1:
        left = array[:n / 2]
        right = array[n / 2:]

        sequential_mergesort(left)
        sequential_mergesort(right)

        merge(left, right)
```

A variant on this approach is to stop splitting the array once the size of the subarrays gets small enough. Once the subarrays get small enough, it may become more efficient to use other methods of sorting them, like the built-in Python *sorted()* function instead of recursing all the way down to size 1.

```
def sequential_mergesort(array):
    n = len(arr)
    if n < SMALLEST_ARRAY_SIZE:
        array = sorted(array)
        return

    left = array[:n / 2]
    right = array[n / 2:]

    sequential_mergesort(left)
    sequential_mergesort(right)

    merge(left, right)
```

The average time complexity of classic merge sort is O(n logn), which is the same as quick sort and heap sort. Additionally, the best and worst case time complexity of merge sort is also O(n log n), which is the also same as quick sort and heap sort. As a result, classical merge sort is generally unaffected by factors in the initial array.

However, classical merge sort uses O(n) space, since additional memory is required when merging. Quicksort also has this space complexity, while heap sort takes O(1) space, since it is an in-place method with no other memory requirements.

6 Parallel Sorting

Parallel programming describes breaking down a task into smaller subtasks that can be run simultaneously. Since merge sort is a classic and well-known example of the divide-and-conquer approach, we use merge sort as a test method to explore parallelization methods that may be generalizable to other divide-and-conquer methods.

6.1 Related Research

The theory of merge sort parallelization has been studied in the past. Cole (1998) presents a parallel implementation of the merge sort algorithm with O(log n) time complexity on a CREW PRAM, a shared memory abstract machine which neglects synchronization and communication, but provides any number of processors. Furthermore, Jeon and Kim (2002) explore a load-balanced merge sort that evenly distributes data to all processors in each stage. They achieve a speedup of 9.6 compared to a sequential merge sort on a Cray T3E with their algorithm.

On MPI, Randenski (2011) describes three parallel merge sorts: shared memory merge sort with OpenMP, message passing merge sort with MPI, and a hybrid merge sort that uses both OpenMP and MPI. They conclude that the shared memory merge sort runs faster than the message-passsing merge sort. The hybrid merge sort, while slower than the shared memory merge sort, is faster than message-passing merge sort. However, they also mention that these relations may not hold for very large arrays that significantly exceed RAM capacity.

`https://www.researchgate.net/publication/220091378_Parallel_Merge_Sort_with_Load_Balancing`
`http://www.inf.fu-berlin.de/lehre/SS10/SP-Par/download/parmerge1.pdf`
`https://charm.cs.illinois.edu/newPapers/09-10/paper.pdf`

6.2 Multiprocessing Merge Sort

multiprocessing is a package that supports, on Windows and Unix, programming using multiple processors on a given machine. Python's Global Interpreter Lock (GIL) only allows one thread to be run at a time under the interpreter, which means multithreading cannot be used when the Python interpreter is required. However, the *multiprocessing* package side-steps this issue by using subprocesses instead of threads. Because each process has its own interpreter with a separate GIL that carries out its given instructions, multiple processes can be run in parallel.

The docs for the *multiprocessing* package can be read [here](https://docs.python.org/3/library/multiprocessing.html).

6.2.1 Overview

We use Python to implement a multiprocessing mergesort algorithm (linked: here. The algorithm is then run and evaluated in here.

6.2.2 Algorithm

We define a merge function that supports explicit left/right arguments, as well as a two-item tuple, which works more cleanly with multiprocessing. We also define a classic merge sort that splits the array into halves, sorts both halves recursively, and then merges them back together.

Once both are defined, we can then use multiprocessing to sort the array.

First, we get the number of processes and create a pool of worker processes, one per CPU core.

```
processes = multiprocessing.cpu_count()
pool = multiprocessing.Pool(processes=processes)
```

Then, we split the intial given array into subarrays, sized equally per process, and perform a regular merge sort on each subarray. Note that all merge sorts are performed concurrently, as each subarray has been mapped to an idividual process.

```
size = int(math.ceil(float(len(data)) / processes))
data = [data[i * size:(i + 1) * size] for i in range(processes)]
data = pool.map(merge_sort, data)
```

Each subarray is now sorted. Now, we merge pairs of these subarrays together using the worker pool, until the subarrays are reduced down to a single sorted result.

```
while len(data) > 1:
    extra = data.pop() if len(data) % 2 == 1 else None
    data = [(data[i], data[i + 1]) for i in range(0, len(data), 2)]
    data = pool.map(merge, data) + ([extra] if extra else [])
```

If the number of subarrays left is odd, we pop off the last one and append it back after one iteration of the loop, since we're only interested in merging pairs of subarrays.

Entirely, the parallel merge sort looks like this:

```
def merge_sort_parallel(data):
    processes = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(processes=processes)

    size = int(math.ceil(float(len(data)) / processes))
    data = [data[i * size:(i + 1) * size] for i in range(processes)]
    data = pool.map(merge_sort, data)

    while len(data) > 1:
        extra = data.pop() if len(data) % 2 == 1 else None
        data = [(data[i], data[i + 1]) for i in range(0, len(data), 2)]
        data = pool.map(merge, data) + ([extra] if extra else [])

    return data[0]
```

6.3 MPI Merge Sort

BOGO describe MPI MPI for Python, also known as mpi4py, is an object oriented approach to message passing in Python. It closely follows the MPI-2 C++ bindings.

[Linked](https://mpi4py.readthedocs.io/en/stable/index.html)

6.3.1 Overview

This project uses Python to implement an MPI mergesort algorithm (linked The algorithm is then run and evaluated in

7 Sorting on GPU

We measure the performance of our multiprocessing and MPI mergesort algorithms on the Carbonate GPU partition. Each node is equipped with two Intel 6248 2.5 GHz 20-core CPUs, four NVIDIA Tesla V100 PCIe 32 GB GPUs, one 1.92 TB solid-state that memory constraints allowed us to experiment with arrays of up to BOGO integer elements.

Multiprocessing merge sort was executed on 1 to 24 processes by using all available processes on the Carbonate partition. Message-passing merge sort was executed on 1, 2, 4, and 8 nodes by using one core on each node for MPI processes.