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 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`

5 Installation

6 Single Processor Sorting

6.1 Recursive Merge Sort

Recursive merge sort is a standard example of a sequential 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.

Sort	Average Time Complexity	Best Time Complexity	Worst Time Complexity
Bubble sort	O(n^2)	O(n)	O(n^2)
Insertion sort	O(n^2)	O(n)	O(n^2)
Selection sort	O(n^2)	O(n^2)	O(n^2)
Heap sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick sort	O(nlogn)	O(nlogn)	O(nlogn)
Merge sort	O(nlogn)	O(nlogn)	O(nlogn)

6.2 Built-in Sort

For the sake of benchmarking our merge sorts, we also run and compare the times of the Python builtin sort. It uses an algorithm called Timsort, a hybrid sorting algorithm of merge sort and insertion sort. The algorithm finds "runs", or subsequences of the data that are already order. When these runs fulfill a merge criterion, they are merged.

The Python builtin sort is run by calling sorted(a), where a is the list to be sorted.

6.3 Multiprocessing Merge Sort

multiprocessing is a package that supports, on Windows and Unix, programming using multiple cores 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 spawning multiple processes instead of different 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 on one processor.

Note that even though this is a parallelized merge sort, it runs on multiple cores, not on multiple processors. This is a crucial distinction from MPI (Message Passing Interface) merge sort.

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

6.3.1 Overview

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

6.3.2 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]
```

7 Multi Processor Sorting

7.1 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)

7.1.1 Overview

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

The unsorted array is generated on rank 0. Since the unsorted array must be split into smaller subarrays, subsize is calculated. subsize is the size of each subarray. It is equal to the size of the unsorted array divided by the number of processors, since each processor must be sent a subarry. Note that *n*, the size of the unsorted array, must be evenly divisible by the total number of processors.

Once the subarrays have been distributed using the Scatter command, they are sorted on each processor using the built-in Python. This sort defaults to Timsort.

In order to merge the sorted subarrays, we can visualize the processors as being set up in a binary tree, where each parent has two children. One important note: in this situation, the left child also functions as the parent node. We can then split the tree into two sections: left children and right children. Because we are using a binary tree, we know that the number of left and right children will always be equal. Therefore, we can create a variable *split* that splits the set of processors in half.

If the rank of the processor is in the second half (between *split* and *split* 2), then it will send its sorted subarray to its "left" partner to be merged. Otherwise, if the rank of the processor is in the first half (between 0 and *split*), it will recieve a sorted subarray from its "right" partner and then merge it with the subarray that it currently contains. When a subarray is sent, it is sent to the processor with rank *rank - split*, ensuring that the processor that it is sent to has a rank between 0 and *split*. This guarantees that each subarray that is sent gets sent to a merging processor. Similarly, when a subarray is received, it is received from a processor with rank *rank + split*, ensuring that the subarray is a sorted array to be merged. This mapping guarantees a unique pairing between left and right child.

Once received, the subarrays are merged together. The merging algorithm can be defined by the user, and will be referred to as the merge algorithm. There are currently three merging algorithms that can be used. 1. sequential$_merge_fast, a"fast"mergethatsimplycombinesthetwoarraysusingthebuilt-inPython*sorted*function.Notethatsortingalgorithmscanalsobeusedtomerge.$

Then, each individual send/recieve operates as following:

1. Left child sends sorted subarray 2. Right child allocates memory for recieving subarray from left child (*local$_tmp*)3.Rightchildallocatesmemoryforarraytostoremergedresult(*local_result*)4.Rightchildreceivessubarray5.Righ$
$local_arr*topointto*local_result*$

This loop continues until the tree reaches the height that guarantees us a single sorted list. Each time, the number of nodes is halved (since two have been merged into one).

8 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 PCle 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.

Table 2: This is an example.

Figure 1: Times for different sorts

9 Performance Comparison

All performance comparison done here will be on data from the Carbonate partition.

9.1 Time

Using our results from the Carbonate partition, we analyze and graph the resulting times from selected runs.

We can analyze time in two ways. First, we analyze algorithm performance based on increasing size of arrays. Some algorithms differ within themselves (for example, two multiprocessing merge sorts might use different numbers of processes). However, we can account for this variation by aggregating all the values and showing error bands around the lines that are plotted.