CS550 Project Progress report

Zhuolin Fu (A20501576), Rakesh Abbireddy (A20525389)

CS550, Spring 2023

Learning Dask

This project involves a heavy use of Dask library. To have a deep understanding of Dask, we did the following:

1. Understand the basics of Dask

   We started by reading the Dask documentation to understand the basics of the library. This helped us get a sense of what Dask can do, how it works, and what types of problems it can help us solve.

2. Practice with Dask examples

   After having a basic understanding of Dask, we started working through some Dask examples. Dask provides a number of example datasets and workflows, which helped us see how Dask can be used in practice.

3. Learn advanced Dask concepts

   To use Dask effectively, we need to understand some key concepts, such as Dask graphs, lazy evaluation, and task scheduling. We learned about these concepts by reading the Dask documentation and watching tutorials.

Learning merge sort

The key algorithm is the merge sort. We have studied merge sort in depth.

Merge sort involves breaking down the algorithm into "divide" step, where the array is split into smaller sub-arrays recursively, the "conquer" step, where the sub-arrays are sorted individually, and the "merge" step, where the sorted sub-arrays are combined to form a fully sorted array.

Merge sort has a time complexity of $O(n \log n)$ and is a stable sorting algorithm, but requires additional memory for the merge step.

We also implemented the basic algorithm and worked through examples to solidify understanding.

Implementing merge sort with Dask

With the basic knowledge, we built our first version of merge sort using Dask. The core idea is to break a large input array into blocks that are sufficiently small. Then we apply basic sorting algorithms (e.g. numpy's sort) on each block. Then every two adjacent blocks are combined (rechunked) into 1 block and the merge operation is performed on the block. The rechunking and merge step is done recursively until there's only 1 block left.

The core code is shown below.

```
def merge_chunks(chunks):
    if len(chunks) == 1:
        return tuple()
    return tuple(chunks[i] + chunks[i+1] for i in range(0, len(chunks)-1, 2))


def psort(x):
    chunks = x.chunks[0]
    x = x.map_blocks(np.sort)
    while len(chunks) >= 2:
        chunks = merge_chunks(chunks)
        x = x.rechunk((chunks,))
        x = x.map_blocks(merge, dtype=int)
    return x
```

We found this implementation very slow and the bottleneck is at `merge` function. The reason we found is that the `merge` function is implemented using pure Python.

Improving with Cython

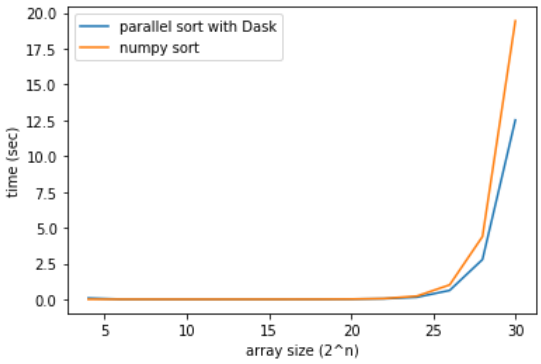
After finding the bottleneck, we managed to improve the merge operation with Cython. The code is shown below.

```
cimport cython
import numpy as np

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
def merge_c(int[::1] x):
    out = np.zeros(len(x), dtype=int)
    cdef int[::1] out_view = out
    cdef int mid = (x.shape[0] - 1) // 2 + 1
    cdef int i = 0
    cdef int j = mid
    cdef int k = 0
    while i < mid and j < x.shape[0]:
        if x[i] < x[j]:
            out_view[k] = x[i]
            i += 1
        else:
            out_view[k] = x[j]
            j += 1
        k += 1

    while i < mid:
        out_view[k] = x[i]
        k += 1
        i += 1
    while j < x.shape[0]:
        out_view[k] = x[j]
        k += 1
        j += 1
    return out
```

After using Cython implementation, the performance sees a huge improvement. We tested running time with array size ranging from 2^4 to 2^{30}. The results are shown below. We can see that when array size is small, our parallel sort is comparable to numpy sort. But as array size increases, our parallel sort gradually outperforms numpy sort.



To do

1. We found is that everything is normal when running our parallel sort in Jupyter notebook, but using it as a Python module introduces a significant overhead. This is to be fixed.
2. We found that using processes is much slower than using threads. This is to be furthur investigated.
3. We will improve the robustness of the algorithm.
4. We will improve adaptive chunking strategy based on input array and system specification.
5. We will try to imrove the runtime complexity by using more advanced optimization techniques.