Jimmy Hoerschgen
Fall 2024

CS 5802
Term Project Report

Parallel Academic Report Generation

Using Multiprocessing in Python 3.12

Project Repository: *https://github.com/jthoerschgen/CS5802-term-project*

Introduction

For the last year, I have been responsible for keeping track of academics for a student organization on campus. We have a scholarship program which places members on academic probation if their grades are below a certain threshold, where they are then placed on required study hours if necessary. This program helps keep our members focused on their academics if their grades begin to slip.

At midterms and finals, campus releases grade reports that are accessible by our organization. These reports come as CSV files. Each report lists all of the classes our members are enrolled in and their letter grade at either midterms or finals. Previously, these reports needed to be parsed and the GPA of each member was calculated by hand. Since I have been in charge of this process, I have worked to create a set of Python scripts that calculate every member's GPA instead. My interest for this project is to speed-up the GPA calculation process by utilizing the *multiprocessing* library which comes standard with Python. It is my hope that I can apply any improvements made here to my own scripts.

Configuration

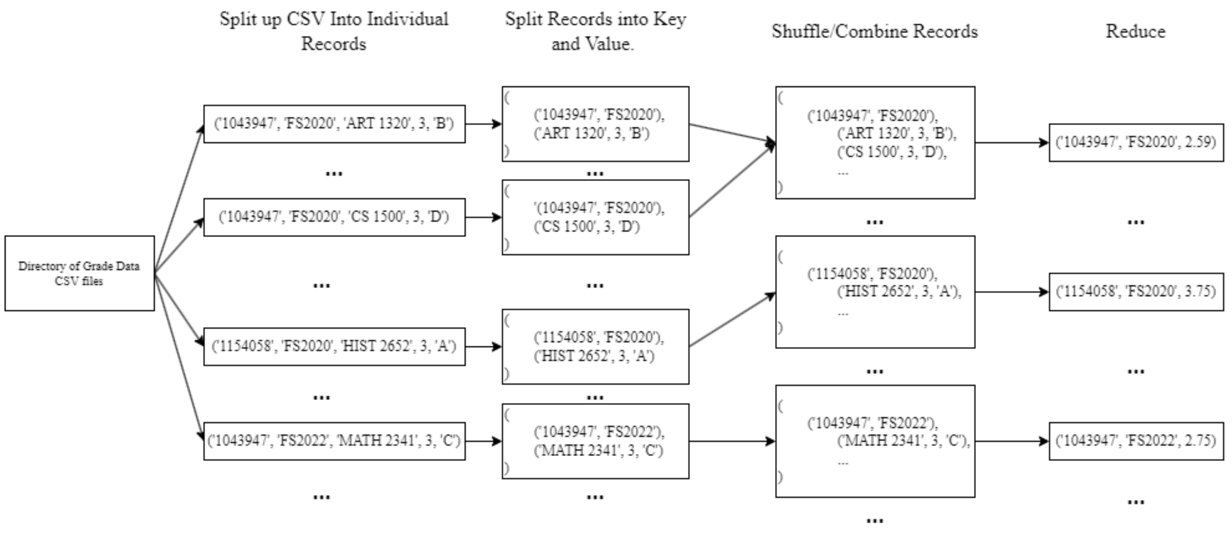
The objective of this project is to add any improvements to the GPA calculation process to my set of grade calculation scripts. Therefore, this project must be able to be run on a personal computer. For testing I used my own desktop computer, which has the following CPU:

- Intel Core i7 13700K
- 16 cores (8 performance cores and 8 efficient cores).
- 24 total threads.

All of the code for this project was written in *Python 3.12.5*.

Process

The process I chose for this project was to implement a MapReduce algorithm. The input is multiple CSV files, one per semester, which are all sorted in random order. The records from these CSV files are extracted, and contain the Student's ID, the semester the data is from, the class name, the credit hours the class is worth, and the grade the student got in the class. These records are then mapped, where the key is the combination of the Student's ID and the semester and the value is the data for the grade. This data is then shuffled and combined where all the grade data for a student and semester are all together. The combined class data is then reduced where the output is the Student's ID, semester, and GPA. Another MapReduce is done on this data to gather all of a student's GPA data for every semester. The following diagram is a demonstration of this process.


Split up CSV Into Individual Records
Split Records into Key and Value.
Shuffle/Combine Records
Reduce
Directory of Grade Data CSV files
('1043947', 'FS2020', 'ART 1320', 3, 'B')
('1043947', 'FS2020', 'CS 1500', 3, 'D')
('1154058', 'FS2020', 'HIST 2652', 3, 'A')
('1043947', 'FS2022', 'MATH 2341', 3, 'C')
(('1043947', 'FS2020'), ('ART 1320', 3, 'B'))
('(1043947', 'FS2020'), ('CS 1500', 3, 'D'))
(('1154058', 'FS2020'), ('HIST 2652', 3, 'A'))
(('1043947', 'FS2022'), ('MATH 2341', 3, 'C'))
(('1043947', 'FS2020'), ('ART 1320', 3, 'B'), ('CS 1500', 3, 'D'), ...)
(('1154058', 'FS2020'), ('HIST 2652', 3, 'A'), ...)
(('1043947', 'FS2022'), ('MATH 2341', 3, 'C'), ...)
('1043947', 'FS2020', 2.59)
('1154058', 'FS2020', 3.75)
('1043947', 'FS2022', 2.75)
...

A diagram of the MapReduce Algorithm used in this project.

Software Packages

The main Python library used for parallelization in this project is *multiprocessing*. Parallelism in Python can be difficult due to Python's *global interpreter lock*, which only allows one thread to execute bytecode at a time. However, the *multiprocessing* library gets around Python's global interpreter lock by utilizing sub-processes instead of using threads.

This project uses the multiprocessing library's *Pool* object, which has an API for creating parallel programs. Below is an example of a simple program which uses *Pool* to square all of the items in a list using a *map* function.

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

An example of how to use the Python multiprocessing library.

MapReduce Implementation

The MapReduce algorithm used by this project is implemented as an object. The object has attributes for the mapping and reducing functions. It also has an attribute for how many workers should be used during the parallel MapReduce process.

```
import collections
import multiprocessing


class MapReduce:
    """
    Sources:
        https://pymotw.com/2/multiprocessing/mapreduce.html
    """

    def __init__(self, mapper, reducer, num_workers=None):
        self.mapper = mapper
        self.reducer = reducer
        self.pool = multiprocessing.Pool(processes=num_workers)

    def shuffle(self, mapped_records):
        shuffled_data = collections.defaultdict(list)
        for key, value in mapped_records:
            shuffled_data[key].append(value)
        return shuffled_data.items()

    def parallel(self, inputs, chunksize):
        map_responses = list(
            self.pool.imap_unordered(self.mapper, inputs, chunksize=chunksize)
        )
        shuffled_data = self.shuffle(map_responses)
        reduced_values = self.pool.map(self.reducer, shuffled_data)
        return reduced_values

    def sequential(self, inputs, chunksize=1):
        map_responses = []
        for item in inputs:
            map_responses.append(self.mapper(item))

        shuffled_data = self.shuffle(map_responses)
        reduced_values = []
        for item in shuffled_data:
            reduced_values.append(self.reducer(item))

        return list(reduced_values)
```

Implementation of MapReduce based on Hellmann.

The MapReduce object has a *parallel* and *sequential* method which apply the same algorithm, one with multiprocessing and the other without. This allows us to test the run times for the sequential and parallel method. The object allows for the passing in of different mapping and reducing functions as well, which is convenient in this application because we use MapReduce in two separate stages.

```
# ~~~~ Mapping Function used by Stage 1 ~~~~ #
def map_by_semester(record):
    return (
        (record[0], record[1]),              # Key (StudentID, Semester)
        (record[2], record[3], record[4]),   # Value (Class, Hours, Grade)
    )

# ~~~~ Reducing Function used by Stage 1 ~~~~ #
def calculate_gpa(record) -> tuple[str, str, float]:
    key, grade_data = record
    studentID, term = key
    total_credit_hours: float = 0.0
    total_grade_points: float = 0.0
    grade_point_map: dict[str, int] = {
        "A": 4,
        "B": 3,
        "C": 2,
        "D": 1,
        "F": 0,
    }
    for grade_datum in grade_data:
        total_credit_hours += grade_datum[1]
        total_grade_points += grade_point_map[grade_datum[2]] * grade_datum[1]
    gpa: float = (
        total_grade_points / total_credit_hours
        if total_credit_hours != 0
        else 0
    )
    return studentID, term, gpa

# ~~~~ Mapping Function used by Stage 2 ~~~~ #
def map_by_studentid(record):
    return (
        record[0],                # Key (StudentID)
        (record[1], record[2]),   # Value (Semester, GPA)
    )

# ~~~~ Reducing Function used by Stage 2 ~~~~ #
def reduce_by_studentid(record):
    studentID, grade_data = record
    result = {"StudentID": studentID, "Grades": {}}
    for semester_data in grade_data:
        result["Grades"][semester_data[0]] = semester_data[1]
    return result
```

The mapping and reducing functions for both stages of the GPA calculation process.

Evaluation

For evaluating the effectiveness of my MapReduce algorithm, I compared the parallelized implementation's run times to the sequential implementation's run time. To get the average sequential time, I ran the data through the sequential implementation of the MapReduce algorithm and took the the average of the run times. Below is a snippet of the testing script that does the sequential run time evaluation.

```
NUM_TRIALS: int = 5
sequential_times = []

for i in range(NUM_TRIALS):
    random.seed(5 * i + 29)

    first_stage_map_reduce = MapReduce(
        mapper=map_by_semester, reducer=calculate_gpa
    )
    second_stage_map_reduce = MapReduce(
        mapper=map_by_studentid, reducer=reduce_by_studentid
    )
    print(f"Iteration #{i}")
    sequential_start_time = time.time()
    # ~~~~ Start Timer ~~~ #

    # ~~~~ Stage 1 ~~~ #
    sequential_gpa_data = first_stage_map_reduce.sequential(input_records)

    # ~~~~ Stage 2 ~~~ #
    sequential_result = second_stage_map_reduce.sequential(
        sequential_gpa_data
    )

    # ~~~~ End Timer ~~~ #
    sequential_end_time = time.time()
    sequential_time = sequential_end_time - sequential_start_time
    print(f"Sequential Execution Time {sequential_time:.3f} seconds.\n")

    sequential_times.append(sequential_time)
avg_sequential_run_time: float = sum(sequential_times) / len(
    sequential_times
)
```

Portion of the testing script for evaluating the average sequential run time.

The method for getting the parallel run time information is similar. The run times are stored in a *dictionary* where the key values are the number of processors used, and the values are a list of the run times for that number of processors. We can later use this data to plot the average, maximum, and minimum run times of the algorithm.

```
    NUM_TRIALS = 5
    max_num_workers: int = multiprocessing.cpu_count() * 2
    chunks_per_process: int = 4

    print(f"Max number of Workers: {max_num_workers}\n")

    parallel_times: dict[int, list[float]] = {}
    for num_workers in range(0, max_num_workers + 2, 4):
        num_workers = max(1, num_workers)
        # ~~~~ Prepare To Store Run-Times ~~~ #
        parallel_times[num_workers] = []
        for i in range(NUM_TRIALS):
            random.seed(5 * i + 29)
            print(f"({num_workers} workers), Iteration #{i}")
            # ~~~~ Create MapReduce Objects ~~~ #
            first_stage_map_reduce = MapReduce(
                mapper=map_by_semester,
                reducer=calculate_gpa,
                num_workers=num_workers,
            )
            second_stage_map_reduce = MapReduce(
                mapper=map_by_studentid,
                reducer=reduce_by_studentid,
                num_workers=num_workers,
            )
            parallel_start_time = time.time()
            # ~~~~ Start Timer ~~~ #

            # ~~~~ Stage 1 ~~~ #
            chunksize = max(
                1, len(input_records) // (num_workers * chunks_per_process)
            )
            parallel_gpa_data = first_stage_map_reduce.parallel(
                input_records, chunksize=chunksize
            )

            # ~~~~ Stage 2 ~~~ #
            chunksize = max(
                1, len(parallel_gpa_data) // (num_workers * chunks_per_process)
            )
            parallel_result = second_stage_map_reduce.sequential(
                parallel_gpa_data, chunksize=chunksize
            )

            # ~~~~ End Timer ~~~ #
            parallel_end_time = time.time()
            parallel_time = parallel_end_time - parallel_start_time
            parallel_times[num_workers].append(parallel_time)
            print(f"Parrallel Execution Time: {parallel_time:.3f} seconds.")
```

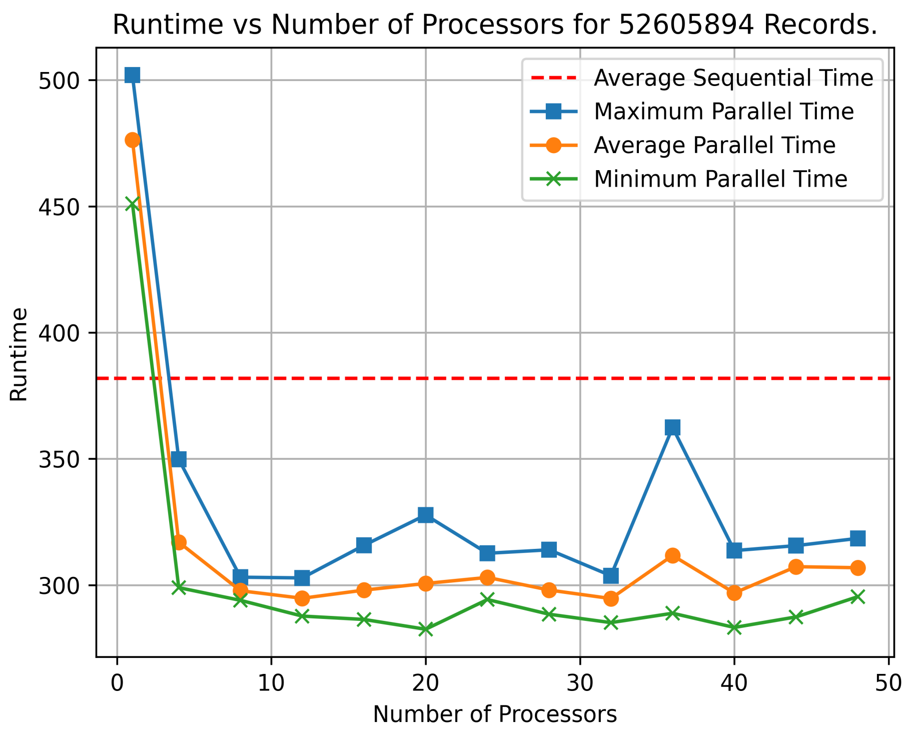
Portion of the testing script for evaluating the average parallel run times.

The chunk size is calculated based on the input and the amount of available workers. This is meant to better distribute the load among the processes. The formula used was:

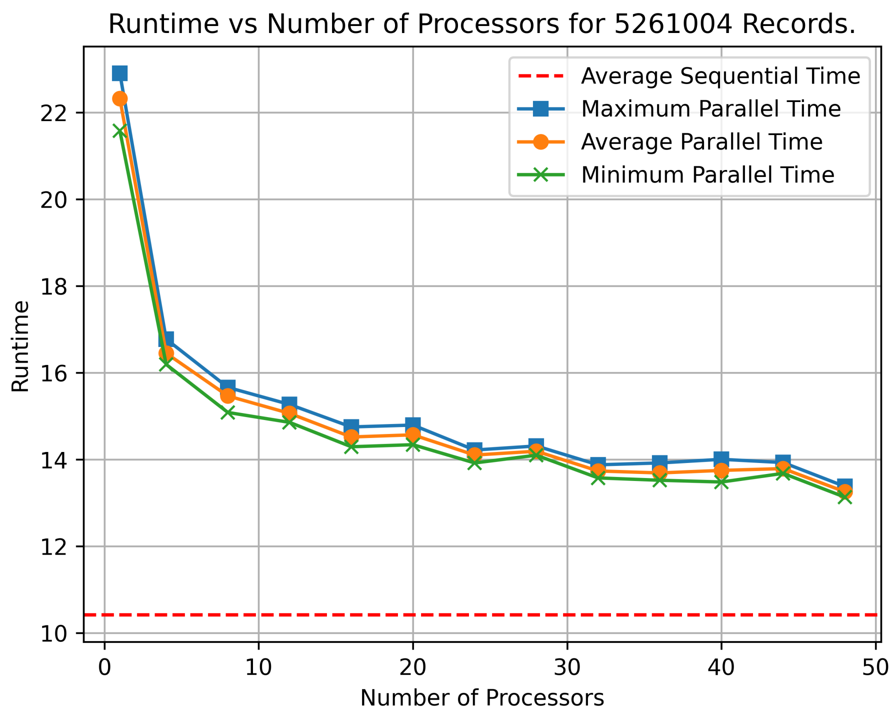
```
max(1, len(input) // (number of available workers * chunks per worker))
```

Plotting Results

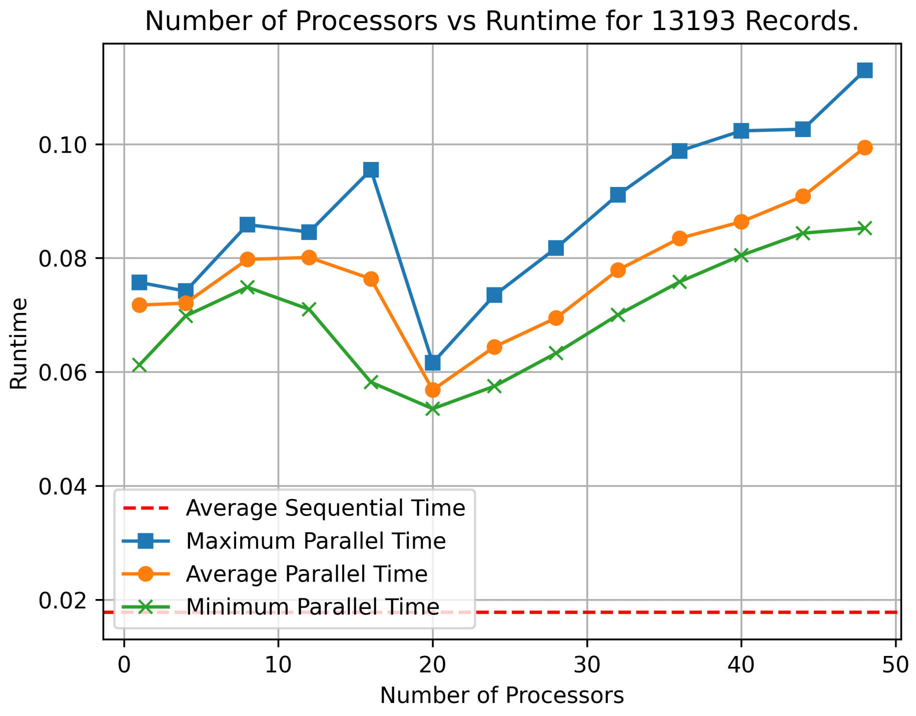
After all of the tests were ran, the results were plotted as the parallel run time versus the number of processors. The average sequential time is plotted as a dotted line in order to give a visual demonstration of what parallel run times were faster/slower than sequential. Below are the plots for a very large dataset (~50 million records), a large dataset (~5 million records), and a small dataset (~13,000 records).



Parallel Run Time (seconds) vs Number of Processors for 52,605,894 records



Parallel Run Time (seconds) vs Number of Processors for 5,261,004 records.



Parallel Run Time (seconds) vs Number of Processors for 13,193 records.

The final evaluation of the data appears to show that for the large and small datasets (~5 million records and ~13 thousand records) the parallel run time is worse than the average run time. However, for the large dataset (~50 million records), it appears that the parallel runtime is most efficient at ~12 processors and is significantly faster than sequential for all amounts of processors greater than 1-2. It appears that my MapReduce algorithm works best with very large amounts of data, but not as much so for smaller sized datasets.

Conclusion

Unfortunately it appears that the parallel MapReduce algorithm is not a reasonable solution for my current application. The amount of data that is realistically being processed from campus grade data of about 70-80 members is much closer to the small dataset rather than the very large dataset.

There are reasons that could explain why the parallel MapReduce algorithm here is slower than the sequential for smaller data sizes, which mostly have to do with overhead. The multiprocessing library *pickles*, or serializes, the data it sends back and forth during inter-process communication. The complexity of the data increases serialization overhead, slowing down the algorithm. Perhaps more research can be done for investigating how to best minimize this overhead or look deeper into alternative parallelization libraries, but due to time constraints around this project, I was unable to do this.

While developing this project, I discovered that I could not compare the *multiprocessing Pool* object's map function to the built-in python map function. I had to instead use loops for a fairer comparison. Even on very large amounts of data, the built-in python map function seemed to always significantly beat the multiprocessing one in run time speed. Upon further investigation, this discrepancy is caused by the built-in map function being implemented in C and compiled, rather than in Python (Medium). It seems that if I wanted an sequential and significant speed up for my grade calculation scripts, that it would be best to write them in a compiled language like C first, than work with the much more high-level and abstracted parallelization libraries in an interpreted language like Python.

References

Computer. (2024, October 27). *Comparing performance of map() vs for loop in Python and JavaScript*. Medium. https://medium.com/@8acking/comparing-performance-of-map-vs-for-loop-in-python-and-javascript-0fefc036d049

Hellmann, D. (2020, July 11). *Implementing MapReduce with multiprocessing*. Python Module of the Week. https://pymotw.com/2/multiprocessing/mapreduce.html

Lapets , A. (2020, August 6). *map-reduce-and-multiprocessing*. GitHub. https://github.com/python-supply/map-reduce-and-multiprocessing

The Python Software Foundation. (2024, December 14). *Multiprocessing - process-based parallelism*. Python documentation. https://docs.python.org/3/library/multiprocessing.html