# Airplane flight routing in parallel

15-618, Fall 2024 – Final Project

**Team members:** Jesse Liu (jzliu@andrew.cmu.edu), Oscar Han (enxuh@andrew.cmu.edu)
**GitHub URL:** https://github.com/jliu64/15618-F24-Final
**Web Page URL:** https://jliu64.github.io/15618-F24-Final/

## Summary

Our project consists of three main components: first, we implement a sequential aircraft flight routing algorithm, to serve as a benchmark. Next, we implement two parallelized versions of said flight routing algorithm—one in OpenMP and another in MPI— in an attempt to optimize its speed. Finally, we evaluate the performances of each version of our algorithm. Based on the speed of our implementations as measured on the PSC machines, we found that OpenMP is better suited for this type of problem than MPI.

## Background

In the modern air transport industry, aircraft and crew schedule planning involves four main steps: schedule design, fleet assignment, aircraft routing, and crew scheduling. Our project focuses on the third step, aircraft routing. The objective of our implementations is to compute a feasible “flight string” representing a routing (i.e. a sequence of flights) for each aircraft in a given fleet of fixed size. More specifically, given an input consisting of a set of flight legs (which are predetermined by this stage in the scheduling process) and the starting number of airplanes at each airport involved in our input, our objective is to determine how to route the available aircraft such that each flight leg is covered without violating the laws of time and space. If it isn’t possible to cover all the flight legs with the given aircraft quantity and/or positions, then we must report that the schedule is infeasible.

For the purposes of our project, we only consider the aforementioned time and positional constraints; however, in reality, maintenance constraints must also be considered. Each airplane requires periodic maintenance, and not all airports are actually equipped to perform such maintenance. For our project, we essentially assumed that all airports are able to perform maintenance. Additionally, the routing problem is typically solved for an input period (or “horizon”) of some predetermined number of days; in the actual industry, it’s generally either a week or a month. The initial positions of each aircraft in the fleet are usually determined by the end positions of the previous routing period, but for our purposes, we specified their starting positions in our input.

The sequential version of our algorithm is partially based on the algorithm described in Axel Parmentier's paper, "Aircraft routing: complexity and algorithms" (1). Our input consists of a file containing the flight legs and starting fleet positions. Specifically, the first line consists of two integers giving the number of flight legs, *n*, and the number of airports with starting airplanes, *a*, in that order. The next *n* lines each contain individual flight legs in the following format: departure airport IATA code, departure day, departure hour, arrival airport IATA code, arrival day, arrival hour. IATA codes are always three characters in length, day is an integer, and hour is an integer between 0 and 23, inclusive. Following these *n* flight legs, the last *a* lines each contain an airport IATA code, followed by an integer representing the number of airplanes at that airport when the horizon begins. The fields on each line are all separated by whitespace. Our output consists of a set of flight strings, each of which consists of a concatenated sequence of flights that could feasibly be taken by a single plane. Each flight is represented by its departure airport IATA code, departure day, and departure hour. Each separate flight string is outputted on a separate line.

The key data structure in our algorithm is an acyclic directed graph that we construct from the aforementioned input. Each node in this graph represents the combination of an airport and a discretized timestep computed from a day and hour. The edges in the graph represent potential connections between flights; that is to say, each node is connected to the nodes representing airport-timestep combinations that could be reached from the current node by taking a flight (i.e. a flight connection), as well as to the node representing the same airport as the current node, but at the next relevant timestep (i.e. a ground connection). Thus, each node can only have edges going to nodes at later timesteps. Notably, in order to save memory and computation time, only nodes with departing flights are constructed. Also, if any impossible flights (e.g. flights that go back in time) are detected at this stage, then we can immediately report the infeasibility of the input and exit.

After initializing the graph, our key operation is a form of depth-first search; by following the edges and concatenating strings based on the departure airports and timesteps (for flight connections, as we don't consider ground connections to be flights), we can obtain feasible flight strings, assuming that we take measures to avoid duplicates, and that it was possible to obtain feasible flight strings given the input in the first place.

The most computationally expensive part of the algorithm is easily the DFS; however, with large inputs, the graph initialization can also take some time. In order to determine the feasibility of our input, each node tracks the number of available airplanes and the number of required departures at its airport and timestep; this data must be computed from the connected nodes at earlier timesteps. Furthermore, when computing the flight strings, the selection of later flights is based on the earlier flights. Thus, each node is dependent on the connected nodes "earlier" in the graph. However, parallelism exists in the nodes that originate from different starting airports (i.e. the source nodes at early timesteps that only have outgoing connections), and generally between non-connected nodes (that is to say, different airports that aren't connected by flights at the current timestep). That being said, the graph is not a set of completely independent trees, so care must be taken to avoid duplicates.

## Data

### Dataset

For our project, we utilized the **US Flight Data 2008** dataset sourced from Kaggle (2). This dataset is a rich and comprehensive resource containing detailed information on 7,009,728 flights conducted in 2008 across various US airports. The dataset provides extensive flight details, including departure and arrival times, airports, delays, carriers, and distances. This wealth of data makes it an ideal choice for projects involving flight scheduling, routing, and performance analysis. Its size (approximately 700 MB) ensures a realistic workload for benchmarking sequential and parallel implementations while capturing diverse scenarios across different times of the year.

The key features we used include **departure and arrival times** as well as **origin and destination airports**, which are essential for constructing a time-expanded graph to simulate flight schedules. The dataset's precision in representing times down to the minute ensures the accuracy of our computations, such as calculating overlaps and delays. Additionally, the dataset includes useful metadata like day of the week, carrier information, and delay causes, allowing for further extensions or analysis beyond routing.

A particularly advantageous aspect of this dataset is the distribution of flights across numerous airports, which helps mitigate load imbalance issues in our parallel implementation. While major hubs like Atlanta (ATL) and Chicago O'Hare (ORD) handle a larger share of flights (5.9% and 5.0%, respectively), the distribution remains sufficiently even to avoid extreme hot spots. This ensures more balanced thread workloads in the OpenMP version, minimizing bottlenecks. Moreover, the dataset's even distribution reduces the likelihood of very short flight chains, which could adversely impact routing performance or skew results.

In summary, this dataset fulfills all the requirements for our project: it is large enough to stress-test both sequential and parallel implementations, it includes sufficient precision and variety for constructing time-expanded flight graphs, and it avoids significant issues of uneven load distribution that could complicate parallelization.

### Data Processing

To utilize the US Flight Data 2008 dataset for our project, we implemented a parsing strategy to transform the raw data into a concise and relevant format suitable for flight routing simulations.

The original dataset contains detailed information for over 7 million flights, including departure and arrival times, airport codes, delays, and additional metadata. Since our project focuses on building a time-expanded graph of flights, we filtered the data to retain only essential columns: **flight date (FL_DATE)**, **departure time (DepTime)**, **arrival time (ArrTime)**, **origin airport (Origin)**, and **destination airport (Dest)**.

We further processed the data to ensure compatibility with our simulation requirements. Missing or incomplete records (e.g., canceled flights or null values in critical columns) were removed. The raw departure and arrival times, stored as integers in the HHMM format, were converted to hourly precision for simplicity. Additionally, we constructed a new FL_DATE field by combining the original year, month, and day columns into a proper datetime format. This allowed for precise filtering and simulation over arbitrary time horizons (e.g., 2 days, 7 days).

Finally, the parsed data was exported to plain text files in a format designed for our simulation. Each flight is represented as a record containing the **origin airport, departure day and hour, destination airport, and arrival day and hour**. A separate section of the file lists the starting airplane allocations for each airport, calculated based on proportional departure statistics and the estimated number of airplanes needed for the simulation horizon. This format ensures compatibility with our sequential and parallel implementations, while being compact enough to streamline processing.

## Approach

### OpenMP

#### Settings

Our implementation leverages C++17 for the OpenMP parallelization due to compatibility constraints on the PSC machines, where the GCC version does not fully support C++20. OpenMP provides straightforward multithreaded execution by offering constructs for managing threads and workload distribution.

This adjustment does not affect our implementation significantly, as it does not rely on key features introduced in C++20, such as concepts or ranges. By using C++17, we ensured compatibility without compromising the functionality or performance of the program.

The program was tested on two systems. The first was a GHC machine equipped with an Intel Core i7-9700 CPU, featuring 8 physical cores, 8 threads, a clock speed of 3.00 GHz, and 15 GB of RAM. This setup was used to evaluate performance for moderate thread counts, typically using THREAD_COUNTS of 2, 4, 8, and 16. The OpenMP environment on the GHC machine complies with the 201511 standard, ensuring compatibility with modern parallel programming constructs.

The second system was a PSC (Pittsburgh Supercomputing Center) machine with an AMD EPYC 7742 64-Core Processor, offering 128 physical cores across two sockets (64 cores per socket) and 251 GB of RAM. This high-performance system was utilized for testing larger thread counts, up to THREAD_COUNTS of 128, and provided insights into the program's scalability under extreme parallelism. The PSC machine also used an OpenMP environment adhering to the 201511 standard, ensuring consistent support for modern parallelism.

## Mapping to Parallel Machines

The problem is inherently parallelizable because the computation of flight strings for each airport is independent. Each airport's flight strings are calculated based on its adjacency list in the graph, which allows us to map these computations directly to individual threads.

We use OpenMP #pragma omp parallel for to distribute the computation of flight strings across threads. The starting airports are stored in a std::map<std::string, Airport>, and each airport is assigned to a thread for independent computation. Within each thread, the recursive computation of flight strings is handled using the function compute_flight_string, which ensures correctness by tracking visited nodes to prevent endless recursion.

The thread results are stored in a std::vector<std::list<std::string>> to avoid shared data structures, minimizing synchronization overhead. After all threads complete their computations, the results are merged into a single std::list<std::list<std::string>> to represent the flight strings for all airports.

## Changes to the Original Serial Algorithm

The original serial algorithm was inefficient because it computed flight strings sequentially. This meant that each airport's computation had to complete before the next could begin, despite there being no interdependencies between airports. Additionally, the use of a global container for flight strings introduced significant synchronization overhead due to locking.

To address these issues, we made the following changes:

1. Thread-Local Storage: Each thread writes its results to a thread-local container (std::list<std::string>). This eliminates contention for a global data structure and avoids the need for locking.
2. Dynamic Scheduling: OpenMP's dynamic scheduling was used to address workload imbalances caused by highly connected airports (e.g., ATL, ORD), which required significantly more computation.

3. Visited Tracking: To prevent endless recursion in cyclic graphs, a std::unordered_set<Airport *> was introduced to track visited nodes.

### Optimization Iterations

We went through several iterations of optimization to achieve the final implementation:

1. Initial Sequential Baseline: The program initially computed flight strings sequentially, and synchronization overhead was not a concern. However, performance was poor due to the inability to leverage multiple CPU cores.
2. First Parallel Implementation (Global Locking): The first parallel version used OpenMP but relied on a global container for storing flight strings. This required a critical section (#pragma omp critical) for updates, which caused excessive synchronization overhead, negating the benefits of parallelism.
3. Thread-Local Results: We replaced the global container with thread-local storage (std::vector<std::list<std::string>>) for each thread. This significantly reduced synchronization overhead and improved performance, achieving near-linear speedup for up to 8 threads.
4. Avoiding Unnecessary Copies: An attempt was made to copy the entire start_airports map for each thread to reduce memory access contention. However, the additional memory usage and overhead of copying negated any performance benefits.

## MPI

### Settings

Our settings for the MPI implementation were generally identical to those of our OpenMP implementation. We tested our program on both the GHC 33 machine and the Bridges-2 machine at the PSC; in the latter case, we used the intelmpi/2021.10.0 module for MPI.

### Mapping to Parallel Machines

With MPI, we mapped each source node to a different process based on PID. In essence, each of the airports with only departing flights was assigned to a process, which handled the computation of all flight strings originating from that airport. The result was essentially a number of processes arranged in a ring, each performing DFS from a different source node, then propagating its new flight strings and passing along its received flight strings to the process on its right, before moving onto its next assigned source node. This way, each process always knows its source and destination processes, and we can avoid time-consuming broadcasts.

### Changes to the Original Serial Algorithm

The greatest change made to the sequential version was simply the addition of process synchronization via update propagation, in the aforementioned ring pattern. Since we don't know the length of a flight string or the number of flight strings to propagate at the beginning of the propagation step, the propagation is split into two stages.

In the first stage, each process has obtained its list of newly computed flight strings; each process then sends the length of its list, along with the length of the longest flight string in its list, to the process on its right. As the updates rotate around the process ring, each process tracks the greatest flight string length and list length. The purpose of this stage is simply to propagate these two values to the local memory of every process.

In the second stage, each process propagates its flight strings using the same pattern as before. The values obtained from the first stage allow the processes to determine the buffer sizes required to send and receive these strings; the exact number and lengths of the flight strings are included in the messages to aid the receiving process in parsing the data. The flight strings received from other processes are then added to the list of flight strings in the current process.

It's also worth noting that there is a significant amount of file input to be parsed and organized into various data structures in this algorithm, and all of this is done by the root process before being broadcast to the other processes during the initialization phase. Thus, the initialization of our MPI implementation takes a significantly longer time than the initialization of our other implementations, although the DFS still accounts for the bulk of the computation time.

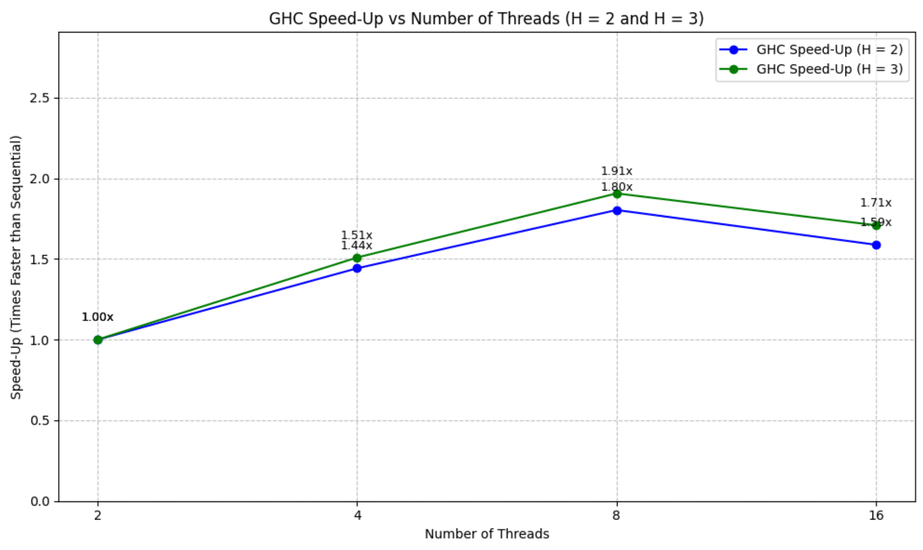
## Optimization Iterations

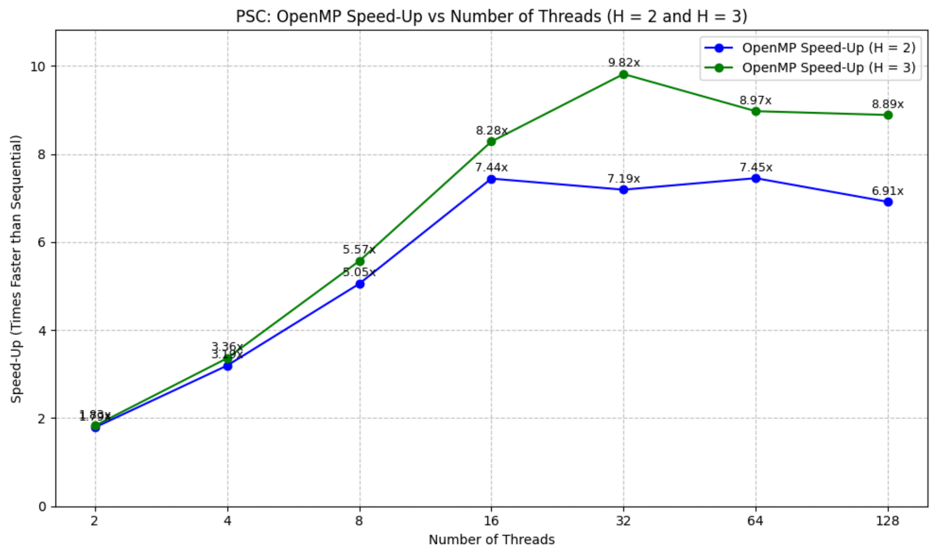
Initially, we attempted to parallelize not only the DFS stage, but also the graph construction stage, as unconnected nodes could be initialized in parallel. However, this proved to be rather pointless with MPI, as the only data we could propagate would be information on the positions and internal values of the nodes; each process would still have to initialize that node in its local memory. Given the way we implemented the sequential version, each process already had access to those pieces of information, so propagating them would be a waste of computational resources.

In addition, for the second stage of propagation, we initially attempted to find a way to send each flight string individually, in order to avoid the potentially enormous memory overhead of allocating buffers that can store all flight strings in any given process's list. However, the message sends in the ring pattern must be asynchronous to avoid blocking, and we cannot safely modify the contents of a sent buffer before we confirm that the message has been received. As a result, we couldn't repeatedly send flight strings using a single buffer, and ended up allocating the aforementioned large buffers instead. This was possible due to the data received in the first stage, but did indeed result in an enormous memory overhead, as will be seen in the results section below.

## Results

### OpenMP





We evaluated our OpenMP implementation by measuring speedup relative to the sequential baseline as the number of threads increased. Experiments were conducted on two problem sizes: H = 2 (38,405 flights, 288 airports) and H = 3 (55,544 flights, 289 airports).

On the **GHC machine**, speedup initially increases as the number of threads grows but plateaus early due to hardware limitations, including only 8 cores and limited memory capacity. For both **H = 2** and **H = 3**, the performance improvement peaks at 8 threads, after which the speedup begins to slightly decline. This behavior reflects the challenges of efficiently utilizing the available resources on GHC, where the recursive nature of the workload leads to synchronization overhead and load imbalance, particularly for highly connected airports like JFK. Additionally, the limited parallelism on GHC makes it difficult to sustain improvements beyond moderate thread counts.

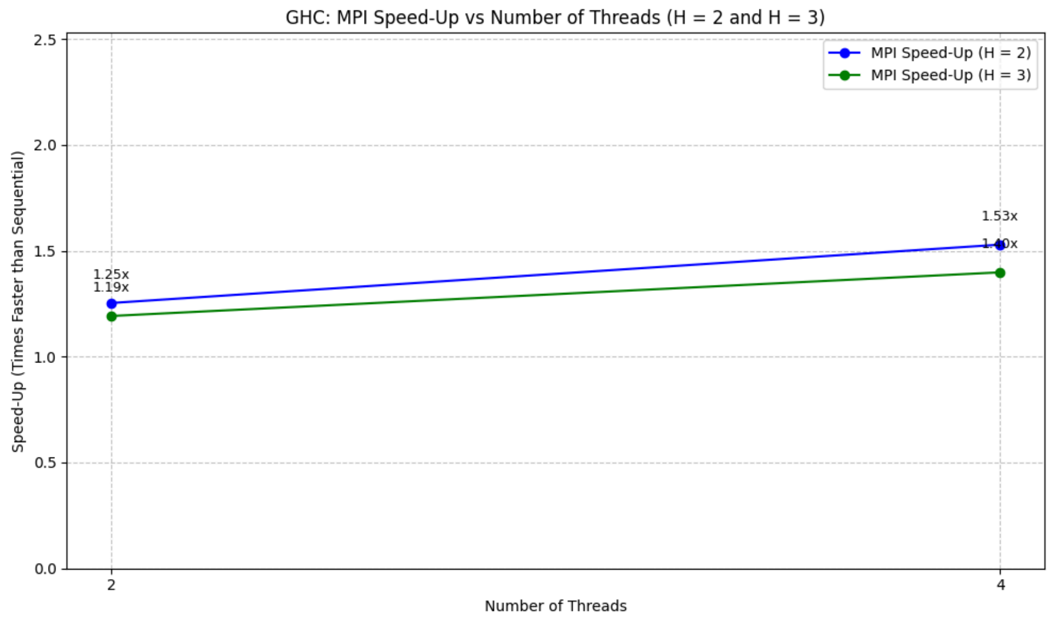
On the **PSC machine**, the speedup is significantly more impressive. Starting from 8 threads, PSC's performance surpasses GHC due to its superior hardware, which includes 128 threads and much larger memory capacity. For **H = 3**, the larger input size benefits more from the increased parallelism, allowing PSC to achieve nearly linear scaling up to 32 threads, with a peak speedup of **9.82x**. Beyond 32 threads, the performance stabilizes as the workload becomes well-distributed, although minor drops occur due to resource saturation and thread management overhead. In comparison, the smaller input size (H = 2) achieves more modest gains, with a peak speedup of **7.45x**, as the workload is not substantial enough to fully utilize all available resources.

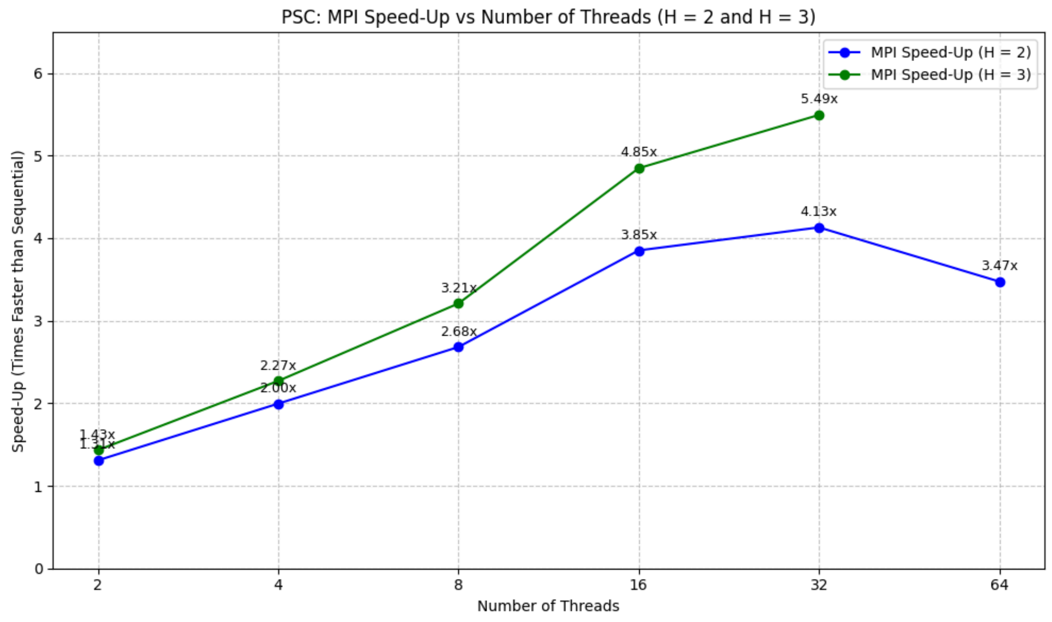
Overall, the results highlight the stark contrast between the two machines. While GHC demonstrates limited scaling due to its hardware constraints, PSC's impressive performance showcases the benefits of leveraging high thread counts and greater memory for computationally intensive tasks. The larger problem size (H = 3) on PSC provides a clear example of how increased workload can better utilize parallel resources, leading to significantly higher efficiency and speedup.

The main bottleneck lies in the **recursive computation** of flight strings, where airports with high connectivity (e.g., **JFK**) require deeper recursion and significantly longer processing times. Although OpenMP's dynamic scheduling helps distribute the workload across threads, achieving perfect load balancing remains difficult. Additionally, the overhead of managing thread-local results and merging the outputs contributes to diminishing returns at higher thread counts.

Flight string computation dominates execution time, accounting for over 99% of the runtime, making it the focus of performance improvements. While the recursive nature of the problem limits scalability, our OpenMP implementation demonstrates significant speedups, especially for larger inputs, achieving clear performance gains over the sequential baseline.

## MPI





As with our OpenMP implementation, we evaluated the performance of our MPI implementation based primarily on its speedup relative to the sequential, single-threaded implementation when run on the same machine. We generated several input files of varying sizes from the dataset mentioned in the data section above; the above two charts measure the relative speedup based on the same two input files used for evaluating OpenMP: H = 2 (38,405 flights, 288 source airports) and H = 3 (55,544 flights, 289 source airports).

Once again, there is a significant difference in the performances of our algorithm when run on the two different machines. Notably, on GHC, the smaller problem size resulted in a greater speedup, while on PSC, the larger problem size yielded much greater speedup. This is due to the greater computational resources available on PSC, but may also be in part due to the fact that the speedups on GHC are very low and close together. Logically, the results on PSC make more sense; a larger problem size presents more opportunities for parallel execution, particularly on a powerful machine like PSC, which possesses a significant amount of parallel computation resources to utilize.

That being said, there are evidently some significant problems with our MPI implementation. First and foremost, there is the issue of the enormous memory overhead that we mentioned above, which has indeed caused problems. In fact, when a large input size is combined with a large number of processes, a process will automatically be killed due to running out of memory, ending the program execution. This is why there are missing data for certain process counts in the charts shown above; GHC could not execute our MPI implementation with 8 or 16 processes on our admittedly large input files, and not even PSC could handle 64 processes on the larger of our inputs, or 128 processes on either input.

Even disregarding the out-of-memory issue, our MPI program performed worse than our OpenMP one. On PSC with 32 processes (the peak of our MPI program's speedup), we achieved a 4.13x and 5.49x speedup on the "H = 2" and "H = 3" datasets, respectively. This is noticeably inferior to our OpenMP version, which achieved 7.19x and 9.82x speedups on the same datasets. Overall, the data seem to imply that OpenMP is better suited to this sort of graph problem. This is also in line with our initial speculation and experience implementing the programs; using MPI on a DFS was rather awkward, and our end solution did not seem very optimized compared to what would be possible with a different technology/API.

## References

(1) Parmentier, Axel. "Aircraft routing: complexity and algorithms." (2013).

(2) Dongare, Vikalp. "US Flights data 2008." (2018). https://www.kaggle.com/datasets/vikalpdongre/us-flights-data-2008/data.

## List of Work

Jesse: Sequential implementation, MPI implementation.

Oscar: OpenMP implementation, dataset collection and input generation.

Both: General debugging and adjustments on all implementations, report writing.

Overall distribution: 50% – 50%