Team Research Investigation Report

Topic Area Overview:

Our parallel computing research investigation centered around the topic of “pancake sorting”. This idea revolves around using one specific type of modification to sort a list of distinct elements. In particular, the modification used in pancake sorting is called a “prefix reversal”. This is an (in place) reversal of the order of some set of adjacent elements within a list, where the first element must be included. Thus, it is reversing the order of some prefix of the list. For example, we could perform a prefix reversal on the first 6 elements of a sorted list of size 10 as follows (reversed elements are highlighted in red):

\[
\begin{align*}
[1, & 2, 3, 4, 5, 6, 7, 8, 9, 10] \\
[1, & 2, 3, 4, 5, 6, 7, 8, 9, 10] \\
& \text{– perform reversal –} \\
[6, & 5, 4, 3, 2, 1, 7, 8, 9, 10] \\
[6, & 5, 4, 3, 2, 1, 7, 8, 9, 10]
\end{align*}
\]

As you can see, we inverted the prefix [1, 2, 3, 4, 5] in place.

Any given list can, in fact, be sorted using only prefix reversals. This is referred to as “pancake sorting”, after the way a chef would have to sort a stack of pancakes of distinct sizes from smallest to largest, using his spatula to flip some portion of the top of the stack (some “prefix”) upside down.

![Figure 1: The chef's spatula flip corresponding to a prefix reversal of the first three elements of a list.](image-url)
In its simplest form, pancake sorting is done by emulating a selection sort variant using prefix reversals, as follows:

1. Take as large as possible a portion of the end of the list where the elements are in their correct positions and mark it as sorted (this portion may be empty to begin with if the wrong element is at the end of the list).
   
   Example: [3, 2, 4, 1] – sorted portion: [], unsorted portion: [3, 2, 4, 1]

2. Find the largest element in the unsorted portion of the list, and perform a prefix reversal such that that element is at the front of the list (this can always be done by simply reversing the prefix that includes everything up to and including that largest unsorted element).
   
   Example: [4, 2, 3, 1] – ported portion: [], unsorted portion: [4, 3, 2, 1]

3. Now that we have the largest unsorted element at the front of the list, perform a prefix reversal of the whole unsorted portion of the list. This will flip the largest unsorted element to its proper location in the beginning of the sorted tail of the list.
   
   Example: [1, 3, 2, 4] – sorted portion: [4], unsorted portion: [1, 3, 2]

4. By repeating this for every element, the entire list will eventually become part of the sorted portion.
   
   Example:
   
   - [3, 1, 2, 4] – sorted portion: [4], unsorted portion: [3, 1, 2]
   - [2, 1, 3, 4] – sorted portion: [3, 4], unsorted portion: [2, 1]
   - [1, 2, 3, 4] – sorted portion: [1, 2, 3, 4], unsorted portion: []

However, this selection-sort-like method is rarely optimal. Take for example the list [3, 2, 4, 1] whose sorting using this method was illustrated above: it took five prefix reversals to sort using this method, when it could have been done in only three:

[3, 2, 4, 1] → [2, 3, 4, 1] → [4, 3, 2, 1] → [1, 2, 3, 4]

Thus, it is relevant to pose the question: How many prefix reversals does it take to sort a given list? As it turns out, answering this question (even for an exact, given list) is NP-Hard, as proven in [1]. Even so, there are a few other questions that can be asked about pancake sorting. Most notably: For a given list length, what is the largest possible number of prefix reversals that may be needed to sort a list of distinct elements of that length? Note that this question entails taking the maximum of many minimum values – in particular, it considers the minimum number of prefix reversals necessary to sort each permutation of a set of distinct elements of a given size/length, then takes the maximum from among them. The answer to this question for a given list length is referred to as the “pancake number” of that length, and is known to be computationally difficult to find (pancake numbers are known only for lists of size up to 19 [2][3]).

Another question that can be asked with regards to pancake sorting is this: Given a randomly permuted list of N distinct elements, how many prefix reversals on average are needed to sort that list. This question, in contrast to the pancake number, is interested in the average of a set of minimums (rather that the maximum of that set of minimums, like the pancake number). In particular, it considers the minimum number of prefix reversals necessary to sort each permutation
of a set of $N$ distinct elements, then takes the average among them. It is this value that our investigation is centered around.

**Computational Problem:**

Our investigation centers around the estimation of the average number of prefix reversals necessary to sort a randomly permuted list of distinct elements of a given length. In particular, we consider the set of all possible lists of a given size that contain distinct elements, and attempt to estimate how many prefix reversals it would take to sort such a list, on average. At a high level, we do this by generating randomly permuted lists of the desired length, then computing the minimum number of prefix reversals necessary to sort each of these lists, and averaging those minimums together.

In order to perform such an operation, we need a couple of key things. First, a way to generate randomly permuted lists of a given length. We do this using a simple shuffling algorithm (the Fisher-Yates/Knuth shuffle), applied to sorted lists of the desired length. Secondly, we need a method for determining the minimum number of prefix reversals necessary to sort a given list. For this, we looked to existing research in the area of pancake sorting for ideas.

**Literature Investigation:**

As part of our investigation, we read a variety of academic literature on the topic of pancake sorting. We chose three of the papers we read to perform a more in-depth analysis on, including determining what problems they investigated, analyzing what distinct or novel contributions they made to the area of pancake sorting (and to the field of computing in general), and how their results could be applied to further our own investigation. Summaries of the results of this analysis for each paper are detailed below. Publication information and dates can be found in the citations associated with each paper, in the “References” section.

**Paper 1 - “Pancake Flipping Is Hard”:**

The first paper we investigated, titled “Pancake Flipping Is Hard” [1], was a research article from the University of Nantes by Laurent Bulteau, Guillaume Fertin, and Irena Rusu. This paper primarily investigated the computational complexity of determining the minimum number of prefix reversals necessary to sort a given list. The paper begins with a formal description of the pancake problem's “burnt” and “unburnt” variants. The “burnt” pancake variant adds a sort of polarization to each pancake, in the form of a burnt side and unburnt side on each pancake in the stack, and mandates that the final sorted stack of pancakes have all of their polarizations facing the same direction. The “unburnt” variant of the problem is the one introduced above in the “Topic Area Overview” section.

Apart from this characterization, the paper also introduces the idea of “breakpoints”, or locations in a list where there are two adjacent elements that would not be adjacent to one another if the list were sorted. An additional breakpoint exist at the end of the list if the last element is not the largest within the list. The concept of breakpoints is something that recurs throughout much of the literature on pancake sorting, and is something we make use of within our solution program, so it is worth illustrating further:

Consider the list $[6, 2, 1, 4, 5, 3]$. Breakpoints occur within this list at each of the locations marked by the commas or brackets highlighted in red. Notice that a mis-ordering of
elements does not result in a breakpoint, so long as those elements would still be adjacent in
the sorted list, such as with 2 and 1 in this example. There is also a breakpoint at the end of
this list, which will be be present any time that 6 is not the last element.

After building up these preliminaries, the core of “Pancake Flipping Is Hard” goes into a
lengthy reduction from the NP-Complete “3-SAT” problem to the problem of determining the
minimum number of prefix reversals necessary to sort a given list (which they refer to as “MIN-
SBPR”). Through this, they prove the NP-Hardness of determining this minimum value.
The paper also, through the same reduction, proves a “stronger” result: that determining whether a
given list can be sorted in a number of flips equal to its number of breakpoints is also NP-Hard.

In terms of contribution to the field, “Pancake Flipping Is Hard” offered one of the first
classifications of the computational complexity of pancake sorting problems (much of the previous
work had focused on either finding upper and lower bounds on the number of prefix reversals
necessary to sort lists of certain lengths, rather than the complexity of determining an exact value,
or on actually computing the pancake numbers for various list lengths). Additionally, the authors
introduced the new concept of “efficiently sortable” lists, which are lists that can be sorted in a
number of prefix reversals equal to their initial number of breakpoints. This concept is used to
cuaracterize the decision problem described in their “stronger” result, as mentioned above.

The ideas presented in the paper by Bulteau, Fertin, and Rusu apply to our investigation in a
couple of ways. First, their reduction proves that the 'sub-problem's in our investigation, which
involve determining the minimum number of prefix reversals necessary to sort specific list
instances, are in fact NP-Hard. While this does not entirely prohibit us from solving them, it does
both limit the scope of what problem sizes we can realistically expect to solve within a reasonable
amount of time, and indicate that we may want to employ some sort of a heuristic-based approach
in order to get the most out of our implementation/algorithm.

**Paper 2 – “Pancake Flipping with Two Spatulas”**

The second paper we analyzed for our investigation was “Pancake Flipping with Two
Spatulas” [4], which explored the idea of sorting a list using prefix reversals, transpositions, and
transreversals. These new types of operations correspond to what could be done to sort a stack of
pancakes by a chef holding two spatulas. Namely, you could perform a prefix transposition by
picking up some first prefix of the stack using the first spatula, then picking up some other prefix of
the remaining stack using the second spatula, then placing the prefix picked up with the first spatula
down, and finally placing the prefix picked up with the second spatula down on the top. Effectively,
you are taking some prefix of the stack (or list) and moving it, as a single unit, down into
somewhere else in the list. For example:

Given the list: [3, 5, 4, 1, 6, 2]

We can perform a prefix transposition of the elements highlighted in red, past the elements
highlighted in blue: [3, 5, 4, 1, 6, 2]

To get: [1, 6, 3, 5, 4, 2]

Similarly, a prefix transreversal involves the same operation as a transposition, only with the first
picked up prefix flipped as its put down:

Given the same list: [3, 5, 4, 1, 6, 2]
We use the same groups of elements, performing a transreversal this time: \([3, 5, 4, 1, 6, 2]\)

To get: \([1, 6, 4, 5, 3, 2]\) (notice that the prefix highlighted in red has been flipped).

It is also worth noting that flipping both portions that we picked up will simply result in a prefix reversal:

Flipping both portions: \([3, 5, 4, 1, 6, 2]\)  Prefix reversal: \([3, 5, 4, 1, 6, 2]\)

\([6, 1, 4, 5, 3, 2]\)  \([6, 1, 4, 5, 3, 2]\)

After introducing and formalizing these concepts, “Pancake Flipping with Two Spatulas” introduces the idea of a “breakpoint graph”. Characterizing breakpoints in a similar manner as [1] (with the addition of the idea that the first position in the list is always considered a breakpoint, even in the sorted list), a breakpoint graph is an undirected graph in which the vertices correspond to the items in the list, and two types of edges, “grey” and “black” exist between certain vertices. Namely, grey edges exist between items that would be adjacent in the sorted version of a list, while black edges exist between items that are currently adjacent in the list. The authors then make (and prove) several observations about the properties of a breakpoint graph, and utilize these in the development of two approximation algorithms, one 3-approximate algorithm for sorting a list of distinct elements using only prefix reversals and transpositions, and a 2-approximation algorithm for doing so with prefix reversals, transpositions, and transreversals. Finally, “Pancake Flipping with Two Spatulas” introduces an idea of adaptive approximation ratios, which is used to demonstrate that their approximation algorithms will achieve better-than-worst-case performance on the average list.

Overall, this paper contributes several novel ideas to the area of pancake sorting. First, the idea of breakpoint graphs and their associated properties is new, and provides a distinct way to describe the state of a list. Secondly, the approximation algorithms described in the paper are original, and provide methods for practically sorting a list using the associated operations whose number of operations is at least bounded (and often better than the basic selection-sort-style approach), even if not always ideal. Lastly, “Pancake Flipping with Two Spatulas” introduces a new style of pancake problem, in which any sorted prefix of the list can be removed, and further operations carried out on the remaining unsorted portion. The authors refer to this as “forward march”.

Although our investigation requires calculating optimal values for the number of prefix reversals needed to sort a given list, we were still able to apply some of the concepts in the paper by prioritizing operations on the list that removed breakpoints, as the authors did in their approximation algorithms.

**Paper 3 – “Landmark Heuristics for the Pancake Problem”**

The third paper we analyzed within our investigation was “Landmark Heuristics for the Pancake Problem” [5], a shorter paper that covered the use of a heuristic based methods for finding the minimum number of prefix reversals necessary to sort a specific list. Within this paper, the authors described a value they referred to as “gaps” for a list. Gaps are effectively, by the definition given in the paper, equivalent to the “breakpoints” described in [1], with a slight difference in notation where the authors of “Landmark Heuristics for the Pancake Problem” visualize gaps at the end of the list as being between the element at the end and an immutable last element that is always
the largest in the list (they refer to this as the “plate” in the context of the pancake analogy). The paper then goes on to describe the use of these breakpoints as an admissible heuristic in an A* search for the shortest sequence of prefix reversals that will sort a given list of distinct elements.

Although the paper does not go into any detail with regards to their actual implementation, they report results that indicate the gap heuristic scaling better than previously proposed/used heuristics for the problem. Thus, their primary contribution is an empirical comparison of the gap heuristic to other existing techniques.

In terms of our investigation, we utilized the idea of an A* heuristic search that was mentioned in “Landmark Heuristics for the Pancake Problem”, and applied the same breakpoint/gap based heuristic mentioned in this and the other papers.

**Program Design and Organization:**

In this section, we describe the design, organization, and operation of a program we implemented to compute estimates of the average number of prefix reversals necessary to sort a randomly permuted list of distinct elements of a given length. We have created both sequential and parallel versions of the aforementioned program. We first describe the major components and algorithms that are common to both versions, then give an explanation of what the programs themselves do differently from one another, and where in the two programs each piece of functionality resides.

**Conceptual Design and Algorithms:**

Overall, the problem we are solving can be divided into two nested sections: an inner portion within which we generate a random list, then calculate the minimum number of prefix reversals necessary to sort it, and an outer section, within which we execute the inner section many times and compute an average over all of the results of each inner section executions. The outer portion is a fairly straightforward loop over the number of iterations requested by the user.

The inner portion, however, holds more complexity. It involves first generating a randomly permuted list of distinct elements of the user specified length (N), then determining how many prefix reversals are needed to sort that list. For the random list generation, we simply fill an integer array with elements sequentially, starting from 1. We then apply a Fisher-Yates shuffling algorithm (aka a Knuth shuffle) to the list to efficiently produce an unbiased permutation of the desired length. It is worth noting that the use of numeric elements starting sequentially from 1 does not cause a loss of generality from the case of sorting a list of N distinct elements (no two of which have equal value) using prefix reversals, as the relative position of the values in the sorted list is the only thing that matters in determining the number of prefix reversals necessary.

After generating a sorted list, we must compute the minimum number of prefix reversals necessary to sort that specific list. For this, we utilize an A* search algorithm, utilizing the number of breakpoints (or “gaps”) as an admissible heuristic value. Within the search, each 'state' is defined by one specific ordering of elements within the list. Two states are defined to be “adjacent” to one another if they can be changed into one another using only a single prefix reversal. For example, our list [3, 2, 4, 1] from earlier would be adjacent to the list [2, 3, 4, 1], as single prefix reversal of the first two elements would turn either of these lists into the other. Building off of this concept of states, we define the “starting state” as the initial, randomly permuted list, and the “goal state” as the sorted list. Thus, we have a process in which we search for the sorted list state, using a single prefix reversal to move between states. This also allows us to define a “distance” between two states
as the number of prefix reversals necessary to go from one to the other. We utilize the “distance” of each state from the starting state, in conjunction with the number of breakpoints, as a measure of cost in our heuristic search.

So far, we have discuss using breakpoints or gaps as a heuristic our search, but have not covered the optimality guarantees of doing so. In particular, an A* search takes into account the cost to reach a state from the starting state, as well as a heuristic estimate of the remaining distance between that state and the goal state. It produces an optimal result only when the heuristic used for estimating this remaining cost is “admissible”, which means the heuristic never overestimates the remaining cost to the goal state. The number of breakpoints in the list can be used as such a heuristic due to a couple of properties: First, the number of breakpoints in a sorted list is 0, so we can view the process of sorting a list as the repeated removal of breakpoints until there are none remaining. Secondly, a single prefix reversal can only have one of the following 3 effects on the number of breakpoints in a list:

1. Remove a single breakpoint from the list.
2. Add a single breakpoint to the list.
3. Leave the number of breakpoints in the list unchanged.

To understand this, consider the locations at which there may be breakpoints within a list. One occurs after each element in the list. When a prefix reversal is performed on the first M elements of a list of length N, all of the potential breakpoints after elements M+1 through N remain untouched. The breakpoints after elements 1 through M-1 will have their elements in reverse order, but the presence of a breakpoint is not affected by the ordering of the elements, only whether they are adjacent in the sorted list. Thus, the absence or presence of the breakpoints after elements 1 through M-1 will be unaffected. This leaves us with a single breakpoint that might be affected by the prefix reversal: the breakpoint after item M in the list. Note that no special case is needed for the potential breakpoint position at the end of the list, as it will only be affected by a reversal of the entire list, which will not affect any other potential breakpoint.

With this property established, we can see that the number of breakpoints is a useful and admissible heuristic for the remaining number of prefix reversals necessary to sort a list with prefix reversals: fewer breakpoints mean a list is closer to the sorted list, and we can, at best, reach the sorted list by making a sequence of prefix reversals that removes one breakpoint with each move.

Thus, we have an A* algorithm that will operate (at a high level) as follows:

1. Keep a priority queue of states, ordered (from low to high) by the sum of their distance from the start state and their number of breakpoints. The queue begins containing only the start state, which is at a distance of zero from itself.

2. Repeat the following until the sorted list is pulled off of the queue (or the queue is empty):
   - Remove a state from the the front of the queue.
   - Add any possible state that is adjacent to the one we pulled off of the queue, and has not been seen before (or has been seen but with a higher cost) to the queue.

In addition to A* search, we determined experimentally that the use of iterative deepening resulted in faster computations for medium to larger list sizes. Iterative deepening is a technique in
which the depth of a search is limited and, if the depth-limited search failed to find the goal, the depth limit is increased and the search is restarted. This technique is known to increase efficiency in search cases with a large branching factor, which matches our observations from our implementation, as the branching factor in our A* search is the number of different prefix reversals that can be performed on a list, and is thus greater for larger list sizes. Our program implements iterative deepening, with the depth limit starting at the number of breakpoints in the list to be sorted, and increasing by one each time the search fails.

**Implementation Design and Organization:**

In order to support the operation of this algorithm, our program is organized into several Java classes. First, we have a 'pancake_state' class that maintains the information for a single state in the search. Namely, it stores the list of values that the state is defined by, as well as the number of breakpoints (gaps) in that list. Next, we have a 'pancake_number' class that provides the functionality for actually computing the minimum number of prefix reversals necessary to sort a given list, using the A* algorithm described above, as well as for randomly permuting a list. An additional 'pancake_state_comparator' class, that implements a comparison between two 'pancake_state' objects, was defined, to support the ordering of 'pancake_state' objects within the priority queue used by A*. Finally, the 'outer section' and general argument parsing functionality was implemented in the two main program classes – 'PancakeAvgSeq' for the sequential version of the program and 'PancakeAvgSmp' for the parallel version of the program.

**Aspects Unique to Sequential and Parallel Versions:**

For our parallelization of the process of estimating the average number of prefix reversals necessary to sort a randomly permuted list of a given size, we decided to parallelize the loop in the 'outer' portion of the program. That is, we parallelized the performance of multiple trials, rather than the computation of the number of prefix reversals necessary to sort a specific list. This decision came as a result of the fact that parallelizing the trials would result in a cleaner and simpler overall design for the program, without decreasing its usefulness (as it is necessary to perform a decent number of trials when estimating an average value, anyway).

**Sequential Version:**

The sequential version of our program, implemented in class 'PancakeAvgSeq', begins by creating a 'pancake_number' object, then executes a simple for loop for the desired number of trials, using that object to generate randomly permuted lists and compute the number of prefix reversals needed to sort them. It sums these numbers of prefix reversals, and divides that sum by the number of trials when printing the results at the end.

**Parallel Version:**

The parallel version of our program, implemented in class 'PancakeAvgSmp' utilizing the Parallel 2 Java library, executes a parallel for loop over the number of trials. Each thread executing the loop creates its own 'pancake_number' object, and utilizes that to compute the individual trials. The results are saved into a sum reduction variable, which is reduced (by the library) once the threads are finished into a final total for all of the trials. This is then ultimately divided by the total number of trials to produce the average. We determined experimentally that the 'guided' thread schedule produced the quickest and most consistent results.
Developer's Manual:

Our sequential and parallel implementations of our program each consists of several Java source files. Documentation on the purpose of each file, including docstrings for every source file, class, and method, can be found within the files themselves. Instructions on how to build our parallel and sequential programs are as follows:

Sequential Program:

Our sequential program consists of the following source files:

- PancakeAvgSeq.java
- pancake_number.java
- pancake_state.java
- pancake_state_comparator.java

Its main method is within class PancakeAvgSeq, and the program can be built with:

javac PancakeAvgSeq.java

Remember to have the Parallel Java 2 library in your classpath when building the program.

Parallel Program:

Our parallel program consists of the following source files:

- PancakeAvgSmp.java
- pancake_number.java
- pancake_state.java
- pancake_state_comparator.java

Its main method is within class PancakeAvgSmp, and the program can be built with:

javac PancakeAvgSmp.java

Remember to have the Parallel Java 2 library in your classpath when building the program.

User's Manual:

Both the sequential and parallel implementations of our program can be used by running them from the command line with the following three arguments:

1. The list length to perform testing for. This should be a valid positive integer.
2. The number of trials to perform. This should be a valid positive long value (integer).
3. A seed for the random number generator(s) used in the program. This should be a valid long value (integer).
Individually, the syntax for running the programs is as follows:

**Sequential program:**
```
java pj2 PancakeAvgSeq <list_length> <num_trials> <seed>
```

**Parallel program:**
```
java pj2 cores=<# cores> PancakeAvgSmp <list_length> <num_trials> <seed>
```

When run with valid arguments, both versions of the program will print a single line of output, in the form (font size decreased to fit on a single line):

```
Average Number of Reversals (size=<list_length>,iterations=<num_iterations>,seed=<seed>): <calculated_average_value>
```

where `<list_length>`, `<num_iterations>`, and `<seed>` are the arguments the program was run with, and `<calculated_average_value>` is the resulting estimate for the average number of prefix reversals necessary to sort a list of the specified length.

**Scaling Data:**

We measured the performance of our sequential and parallel solution programs under both string and weak scaling, running the parallel version on 1, 2, 4, and 8 cores and taking the average of five trials for each combination of problem size and number of cores. The results for each type of scaling are detailed in the charts below.

**Strong Scaling:**

![Running Time vs Cores](chart.png)
Non-Ideal Strong Scaling:

As can be seen from the charts above, our parallel program does achieve a speedup when run on the problem sizes listed in the graph legends, albeit one that is not fully ideal. We largely suspect that this non-ideal behavior is the result of memory-boundedness within our program. Specifically, within the A* search, which involves a large amount of list copying in order to enable a search where each state is defined by a list. Additionally, frequent object creation occurs due to our use of a class to represent each search state, which is what enables us to use the Java standard library implementations of the data structures needed for A*. Considering the memory intensity of our algorithm, in conjunction with the fact that our parallel program performs very few operations sequentially (only the argument parsing and the printing of final results happen outside of the parallel loop), this indicates that the intense memory usage represents a bottleneck between multiple threads running within the program. This is further evidenced by the slow-down we saw while running multiple instances of the program at the same time. While doing this, multiple runs of the program would produce noticeably longer running times when executed simultaneously than if they were executed one after another. This indicates that some sort of contention is occurring between multiple instances of our program, as well as between the threads within a single instance, further leading us to believe that memory-boundedness is the result of the non-ideal speedup we experienced while doing strong scaling.

Weak Scaling:

Running Time vs Cores
Sizeup vs Cores

Efficiency vs Cores
Non-Ideal Weak Scaling:

From the result above, we can observe that our parallel program's performance under weak scaling closely resembled those from strong scaling, with significant but non-ideal sizeups and efficiency. We believe the same memory-bounded nature is involved here, as we are using the same algorithms, and experienced the same increase in runtime when multiple separate instances of the program were run simultaneously, as compared to running the separate instances sequentially.

Future Work:

There are a couple of different ways in which our investigation could be carried further. First, we might consider a few methods for improving on the computations we are currently doing. For example, we could adapt our implementation into a cluster parallel program, to see if this results in any performance gain from not having all of the threads from competing with one another for memory. Alternatively, other heuristics besides the number of breakpoints in a list could be investigated, to see if any yield better performance.

Alternatively, we might also consider implementing programs to compute other, related metrics, such as the pancake number for a given list length, rather than the average. Another metric we could compute would be the average (or maximum) number of operations needed to sort a list with prefix transpositions and transreversals, two operations we learned about in our literature investigation that correspond to the ways a stack of pancakes could be manipulated using two spatulas. We could consider these additional operations both in combination with prefix reversals and on their own.

Insights and Lessons Learned:

There are a wide variety of things we were exposed to for the first time while researching pancake sorting and implementing our program, but a few key things stand out that we can take away from the experience. First, the amount we learned about the topic of pancake sorting itself. We had not heard of pancake sorting before we began our search for a research topic, so we had the opportunity to learn a great deal about the problem space around pancake sorting and its nuances. Researching pancake sorting also afforded us the opportunity to hone our research skills, as we had to repeatedly determine both how the concepts we read about related to those in other literature, and how we could apply them to our own investigation.

Aside from this, we also became more experienced in designing and writing parallel programs through the creation and optimization of the parallel program in our investigation, observing that programs that take an average value from various trials lend themselves well to parallelism.

Lastly, our research investigation gave us an opportunity to learn more about and implement an A* search algorithm, and provided a practical illustration of how iterative deepening can be applied to a search to increase its efficiency.

Contributions:

The work for our investigation was roughly distributed as follows:

While coming up with a topic for our investigation we each researched several topics, and
met to discuss the best few (from which we ended up deciding on pancake sorting). For creating the first three presentations, we met and worked together during our meetings to create the slides and practice giving our presentation. The fourth presentation was done entirely by Coleman (as Nik had left the course before it came time to put the presentation together). For our research investigation, we each found and read through several candidate papers, from which we chose three to formally analyze for our investigation. In terms of our programs, we collaborated on the design, discussing which ideas from our research we would apply to the program, and how certain things would be layed out. Coleman took this design and translated it into the source code for our initial implementation of our program. Further optimization work (between the third and fourth presentations) and all of the data collection for the investigation was done by Coleman. This final report was also written entirely by Coleman (as it was not started until Nik had left the course).

References:


[2] https://oeis.org/A058986

