Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)

SUMMARY

We implemented a parallelized solver of the Duotrigordle game, which consists of solving 32 Wordle games at the same time using one guess per turn (same guesses get applied to each board). We implemented several different versions of a parallel solver using OpenMP, and we also attempted a solver that made use of OpenMPI (but ran into versioning issues, among other things). Our best OpenMP version achieves a ~3.5x speedup on the Gates machines (running with 8 threads), while our experience with OpenMPI shall be discussed below.

BACKGROUND

The algorithm we parallelized was written from scratch, but Wordle-solving algorithms tend to follow similar overarching strategies. For a single Wordle, with the goal being to find the correct word in the fewest guesses, algorithms rely on a 5-letter-word dictionary (from which the goal word and potential guesses are selected) and often use a word or letter-ranking system to determine the best next guess. Then, they collect feedback on the guess in relation to the goal word, which gives the solver information about the letters from the guessed word (inclusion in the goal word and the correctness of their positions in the word). The feedback can then be used to reduce the dataset (word dictionary) by deleting words that don't contain required letters, or have letters in the wrong positions. This process is then continued, making better and better guesses (more informed), until the goal word is obtained.

The complexity of the algorithm is often centered on the selection of the next best guess from the word dictionary, particularly at the beginning when the dataset is still large. The letter/word ranking is also often computationally expensive (part of the next guess computation). Incidentally, these are the parts of the algorithm that benefit the most from parallelization, as both require sifting through large quantities of data (possible words and letters) and carrying out operations on that data to arrive at a solution.

However, this algorithm isn't quite amenable to SIMD execution as the program iterates through a variable dataset of words, and the program uses and responds to these words in different ways – which is not what SIMD execution is best suited to, similar execution on different pieces of data.

APPROACH

Our algorithm was written without an existing codebase. We begin by creating a dictionary containing all five letter words in a word dataset we had found online (containing 5,757 words). We found that this rather small dataset was limiting our capacity to realize speedup, so we actually created a way to artificially inflate (add more fake words to) the dataset by generating random 5-letter strings and adding them to the set until the desired size was met. We played around with various amplified dataset sizes, with the largest one being 500,000 words (100x the size of the original data set). Our implementation's setup routine creates 32 copies of that dataset (also referred to as a dictionary), one for each of the 32 wordles (since each wordle

Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)

board will be reducing its own list of possible next words to guess). Then, we begin the word guessing process, which is the main computational workload for the solver.

The sequential solver iterates one-by-one through each of the wordle's datasets (which all start the same on the first iteration), and considers how similar each word is to all of the other words in that set. This may seem like it would have a complexity of O(N^2), but we came up with a method to "compare" each word against the rest in a simpler way that makes the complexity O(N). Our method creates a table that tallies up the number of words that contain each letter in each of the 5 letter indices (0 through 4). Then, we loop back around and for each word, we assign a score to that word based on how many other words contain the same letters as that word in the same (worth more) and different indices. This algorithm exposes a lot of opportunities for parallelism, however, there are some intricacies that require careful attention. In particular, each board maintains its own table of tallied letters. This table could also be a global / shared table for each board. If we parallelize across the different boards, then this decision matters a lot since it directly affects the amount of communication that would need to occur between threads/processes. Sharing the letter-tallying table across threads would mean that every update (incrementing of the tally at a particular index) would require synchronization/protection in the form of a critical/atomic operation. However, if the table is not combined, then we need to have a way of eventually combining the data from each board into a singular overall guess, as opposed to 32 guesses (one for each board).

This realization led us to adapt the algorithm for our original serial implementation (which used a single global tally table) to fit a more decentralized approach with each board having its own tallying table. The key driving factor in this decision was the cost of the synchronization for all threads to be accessing different parts of the shared tally table. Our 2 main implementations (both using OpenMP) are meant to specifically show/explore this phenomenon, and the improvement that we were able to achieve by decentralizing the algorithm. Our OpenMP_2 method uses individual tables for each wordle board (each thread is assigned to one or more wordle boards), and generates a best next word guess for each wordle board solely based on that board's table (ignorant of other wordle boards). Then, each of these 32 candidate guesses are collected and evaluated against each other to determine the overall next word guess. Our main hypothesis in exploring this approach was that the approximation inherent in decentralizing the algorithm would be worth the possible improvement in speedup (as a direct result of reducing the required synchronization and communication).

After we come up with an overall best guess, the solver gathers feedback on the guess for each wordle's goal word (from the game-managing portion of the code). Each wordle does this process separately, as each has a different goal word and thus each receives different feedback.

This feedback tells the solver whether each of the letters in the guessed word was either in the correct position, in the word but at an incorrect position, or not in the word at all. Based on this feedback, the solver cuts down each wordle's dictionary of possible words (removing the words that do not match with that feedback), then the next iteration of the game is setup and a new

Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)

guess is generated in the same manner as described above. This process is repeated until each of the 32 wordles are solved (or some maximum number of guesses is reached).

We primarily targeted the parallel machines of the GHC cluster. Initially, we thought we might want to run our parallel solver on many more cores by accessing the PSC Bridges-2 machine, but after profiling our first sequential implementation, we realized that even the bare-bones sequential solver was already pretty fast (solving the game in a reasonable number of word guesses while completing in about 0.1 seconds). Thus, we hypothesized that testing on a machine with more than 8 cores would likely lead to extremely diminishing returns in speedup because of the relatively large overhead of setting up and launching many threads (the solver was not on the right side of amdahl's law). Thus, we focused on implementing our solver targeted at the 8 core GHC machines, which would hopefully allow us to realize a decent speedup while not pushing the limits of thread launching overhead. We did eventually come up with the idea of artificially inflating the dataset to better hide the effect of overhead costs, but this was relatively late in our project timeline, so we did not have time to test and record data on the PSC machines using this workload scaling exploration. However, it is somewhat reasonable that we avoided exploiting this workload scaling too much, given that it does somewhat diverge from the original purpose of the project (which is centered around a real game which uses only a set of real words).

With our sights set on parallelizing the solver across the 8 cores of a GHC machine, we essentially have the number of wordle boards (32) divided evenly across the number of requested threads, so that each thread will likely work on several boards. We had several different initial ideas for how we wanted to parallelize the solver, so we began by working with OpenMP because it is straightforward and easily adaptable to changes such as which data structures are private versus shared, and whether or not a particular code block needs to be protected as critical or not. Our ideas converged on the 2 main different approaches that have been discussed in this section (OpenMP_1 which uses a single shared letter-tallying table, and OpenMP_2 which uses separate local tables for each wordle board). We did also experiment with parallelizing over the number of words for each wordle board (iterating through the wordle boards but then dividing up the work of considering each word amongst threads), but this turned out to be quite tricky since the size of each wordles dataset changes from iteration to iteration, and also we had been using a set to contain the possible next guess words which was difficult to partition amongst different threads.

Finally, as a curiosity experiment, we spent a decent amount of time trying to implement a message passing model of our parallel solver using OpenMPI. This idea aligned with what we had explored already with the OpenMP_2 version, where each wordle board was essentially worked on separately, with communication required at the end to come to an agreement on the next word to guess. Of course, this method seems to be something that would be well-suited for a message passing model (especially because it relies on very little shared data between different processors/cores). Unfortunately, our attempt to use OpenMPI was unable to be completed as a result of some compatibility issues that we encountered when trying to port some of our previous code over (we had been using several global sets and vectors to maintain

Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)

the state of the game, but this lead to some memory errors with OpenMPI which were unable to be solved easily with the version of OpenMPI available on the GHC machines). Nevertheless, the exercise was interesting, and while it may not have been the most practical of implementations to divide up a 0.1 second workload across multiple nodes via message passing, it would have been more applicable if the dataset was bigger (as in the case of our pseudo-inflated datasets).

RESULTS

We measured results based on the speedup of parallelized code compared to our baseline code, all of which were run on GHC machines. Our baseline was single-threaded CPU code, which was identical to our optimized parallel implementation for a single CPU. We held trials for thread counts ranging from 1-16, and took note of the average speedup as well as the average number of guesses required over 10 trials of the code running. One trial is defined as one game of Duotrigordle (so 32 parallel Wordles are generated and solved at once) and its execution time is recorded.

We also held trials for different workload sizes, as we varied our word dictionary size from around 5,000 words to as much as 500,000 words. This was important to include because the time taken to execute, as well as the recorded speedups, vary depending on the workload size – as this directly affects the number and length of iterations that the solver must execute to reduce the dataset size enough to converge to the correct goal words. It also allowed us to test and debunk our initial hypothesis that the speedup was being limited by the relatively high overhead cost of launching the threads. We believed that increasing the workload size would help to amortize this overhead, but it actually had somewhat of the opposite trend.

We noticed that the general speedup for our OpenMP_2 implementation with respect to the sequential version actually gradually decreases as the workload size (dataset) increases. We discovered through profiling that this was the result of an increase in the variation of thread completion time in parallelized loops (meaning that other threads are forced to spin until lagging threads finish). It seems that with a larger dataset (with each wordle starting out with that dataset as the list of possible guesses), there is greater opportunity for disparity in the size of the thread's lists of possible next guesses. This means that the workload after the first iteration essentially becomes imbalanced.

It's worth noting that our final profiling confirms our initial investigations into the division of time our solvers spend on the different parts of the algorithm. We found that the parts of the solver that worked on the feedback provided from a recent guess to reduce the remaining list of possible words to guess was taking a peak of around 0.005 seconds (<10% of total time) and 0.12 seconds (~8% of total time) for the 5,757 word dataset and 500,000 word dataset respectively. This informed us that the true bottleneck was the next guess selection process, which notably takes the longest on the first iteration and progressively takes less and less time as the number of guesses increases.

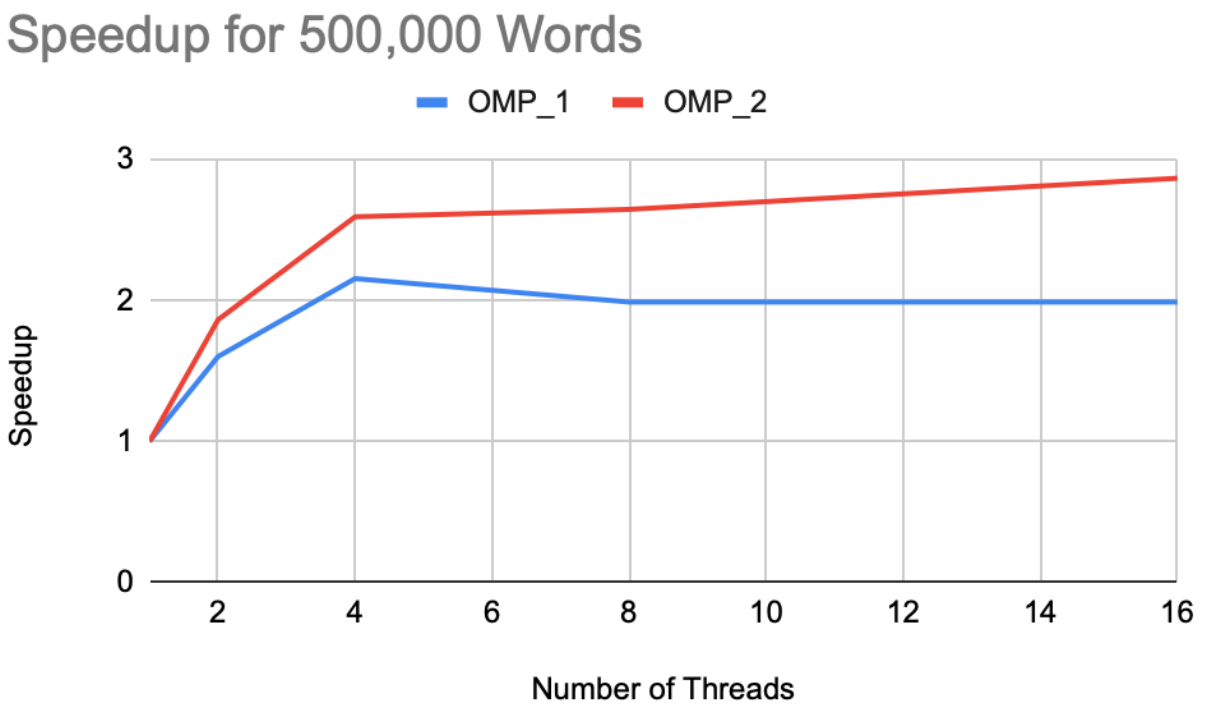
Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)

We notice that the speedup trends for our OpenMP_1 approach show that after a certain number of threads, there is no additional speedup found. This supports our hypothesis that the amount of overhead caused by synchronization in the OpenMP1 approach inhibits the speedup it can achieve with an increasing number of threads. However, the speedup for OpenMP2 consistently increases until 8 threads, which is the number of cores that the GHC machines have.

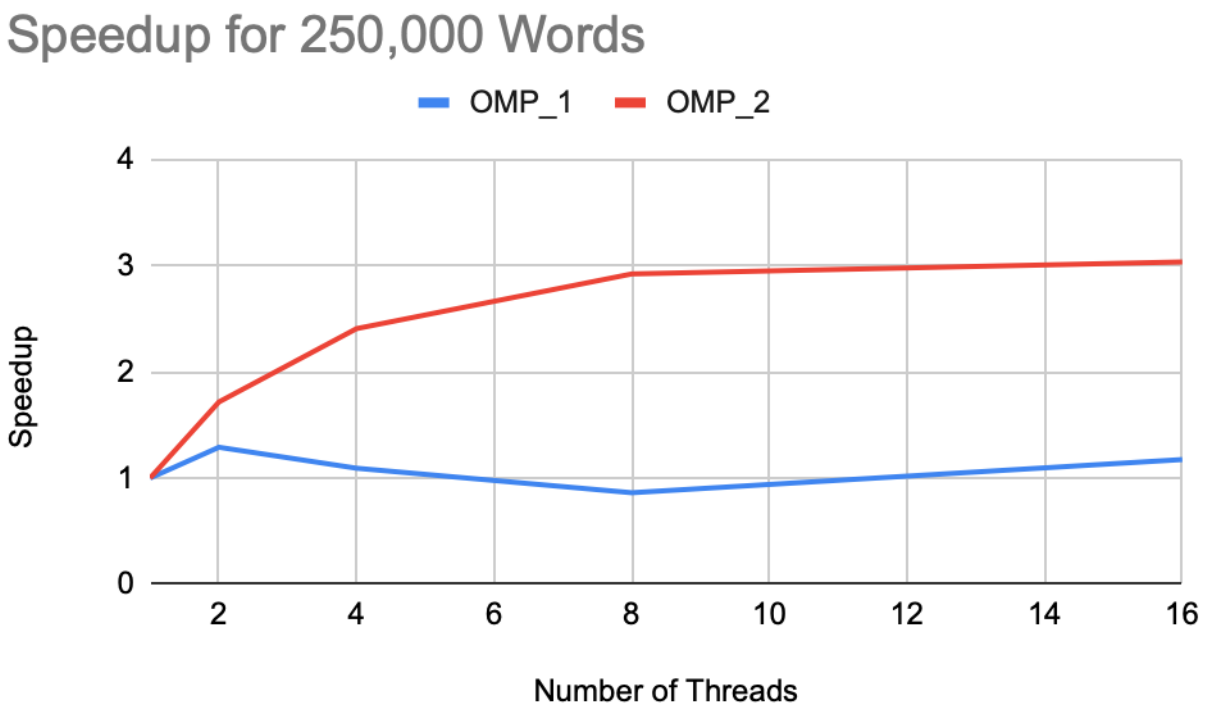
There were several factors limiting our speedup. There were certain dependencies between portions of the algorithm that prevented full parallelization, such as how it is required to make a single guess apply to all Wordles and receive feedback before being able to reduce their individual datasets. Thus, there is an element of sequential execution that is impossible to eliminate through parallelization. Also, what was described earlier as the likely explanation for why the speedup decreased as dataset size increased for OpenMP_2 is also a likely explanation for why the speedup is generally less than ideal. Since we are parallelizing across wordle boards, and each wordle board can have a drastically different current list of next possible guesses, some threads take longer to select a next guess for a particular board than others. This means that lagging threads hold up the faster threads because of the disparity in workload size in some iterations.

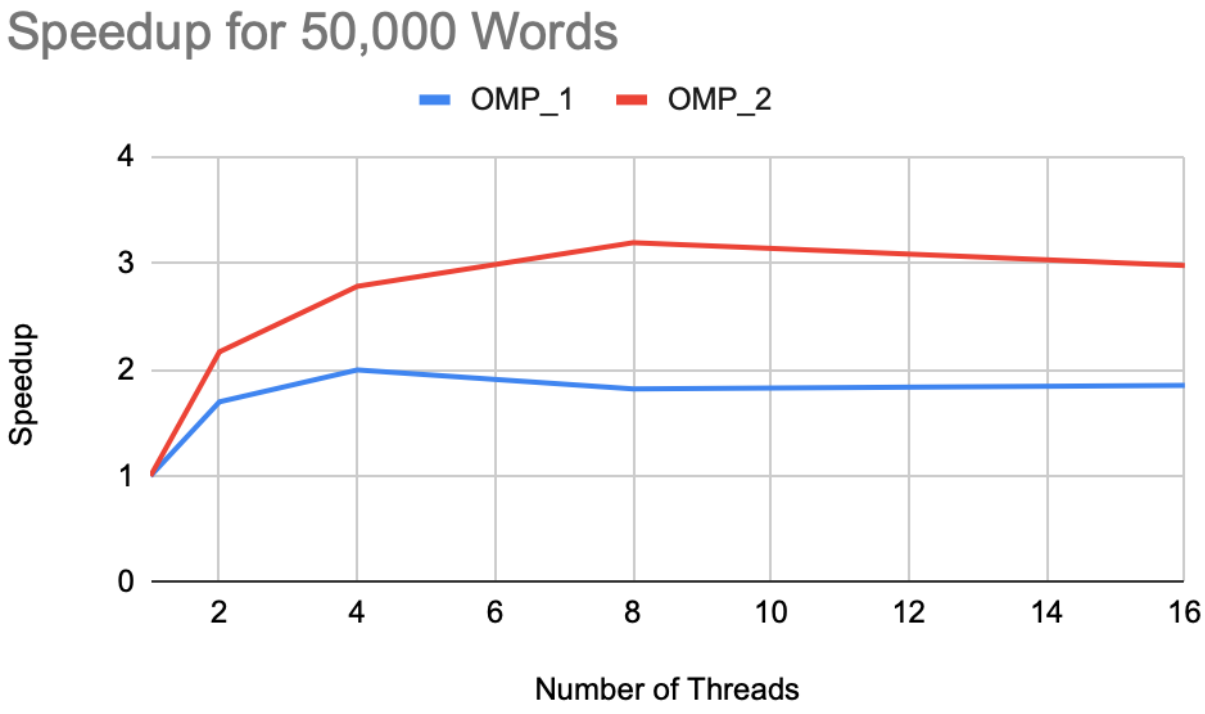
The following graphs summarize the findings discussed above:

Speedup graphs:

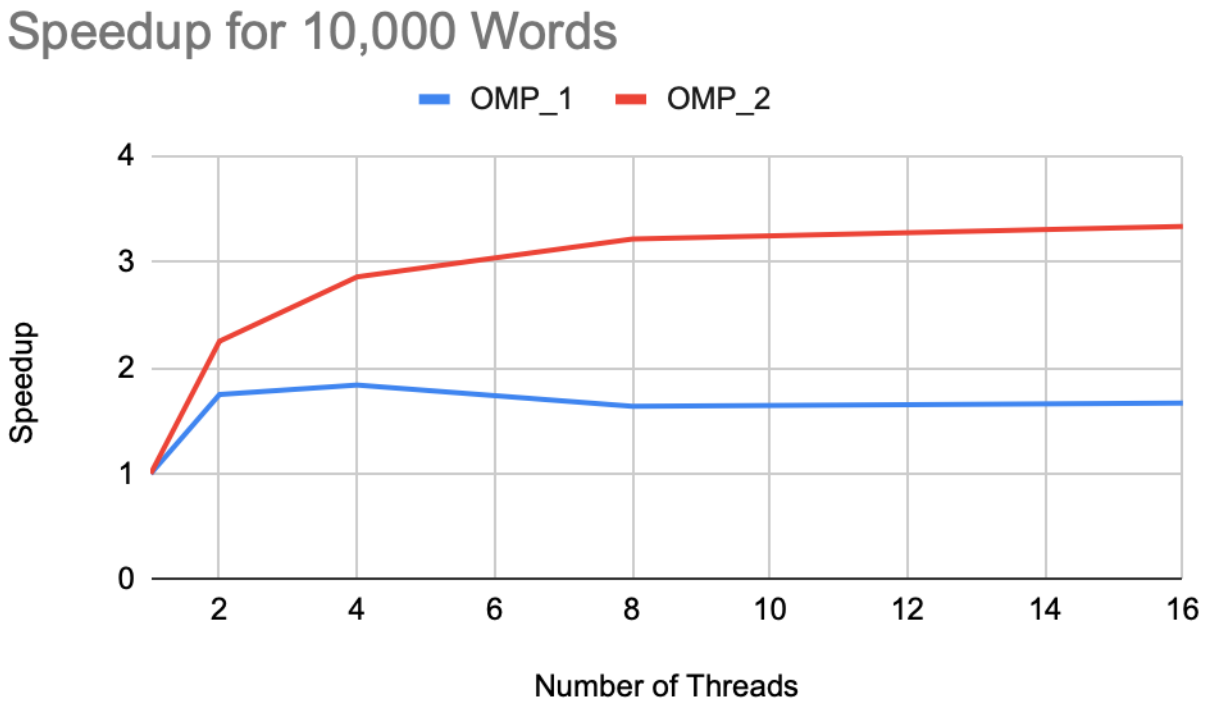


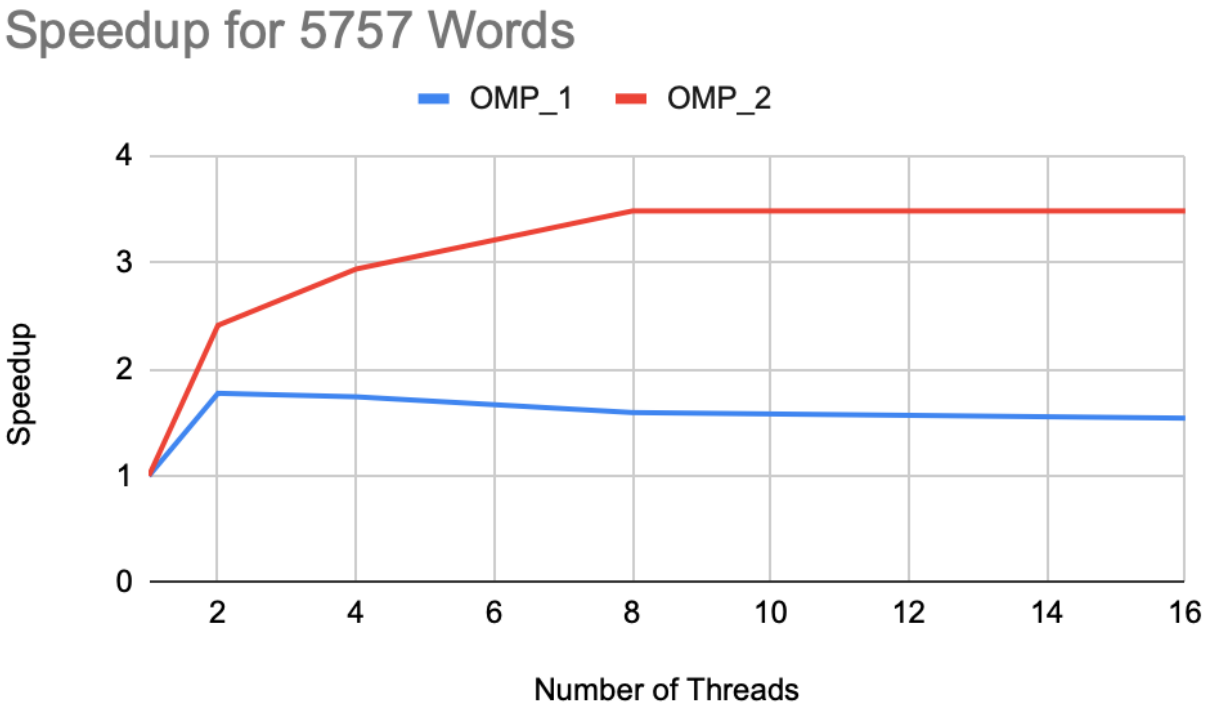
Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)





Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)



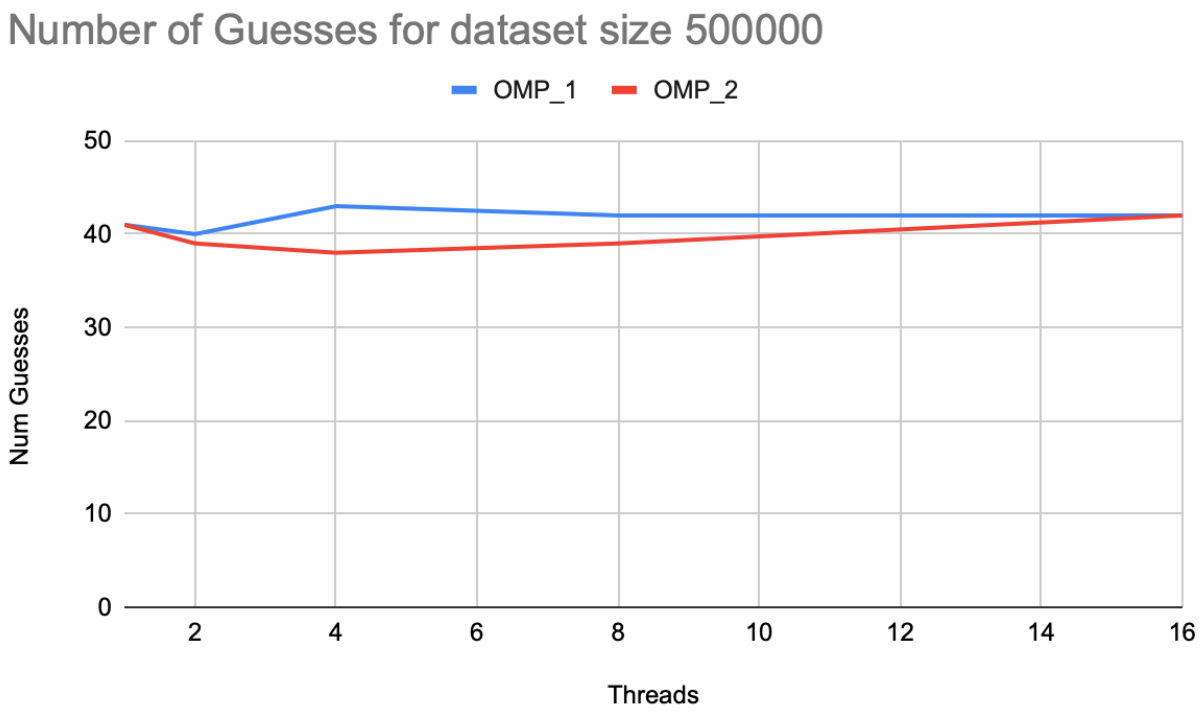


The above graphs show that the OpenMP_2 version generally outperformed the OpenMP_1 version, as expected. The OpenMP_1 version has some speedup over the sequential version,

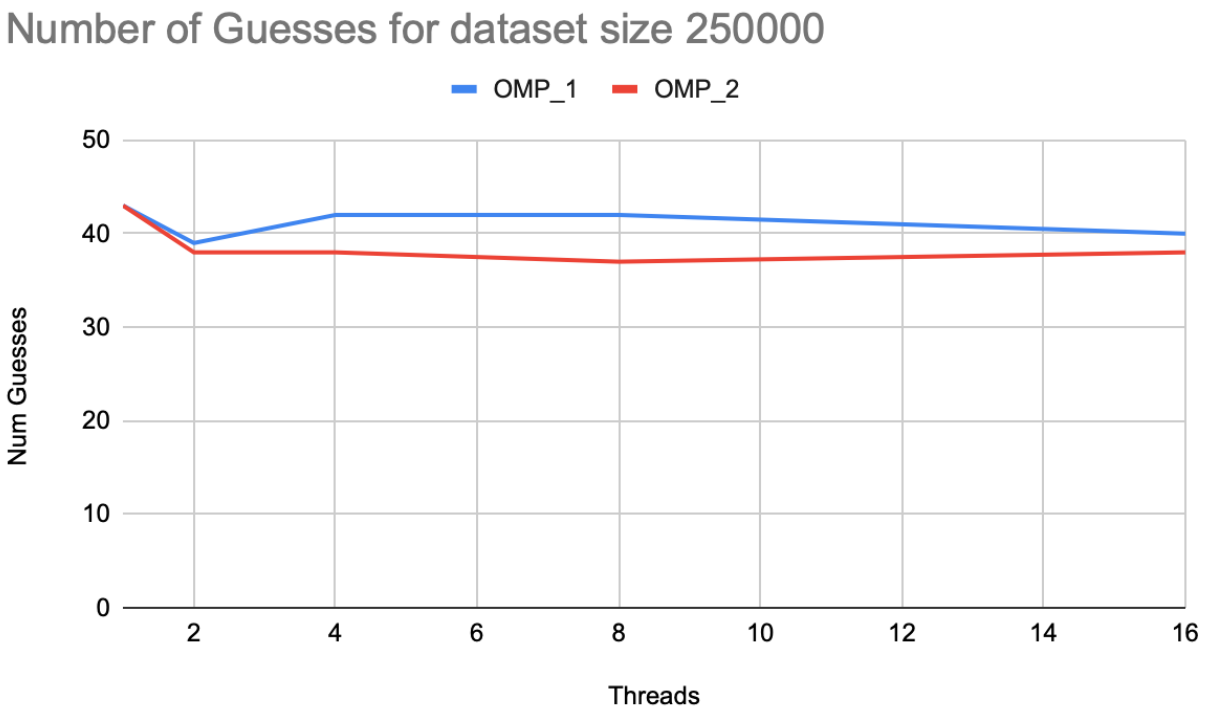
Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)

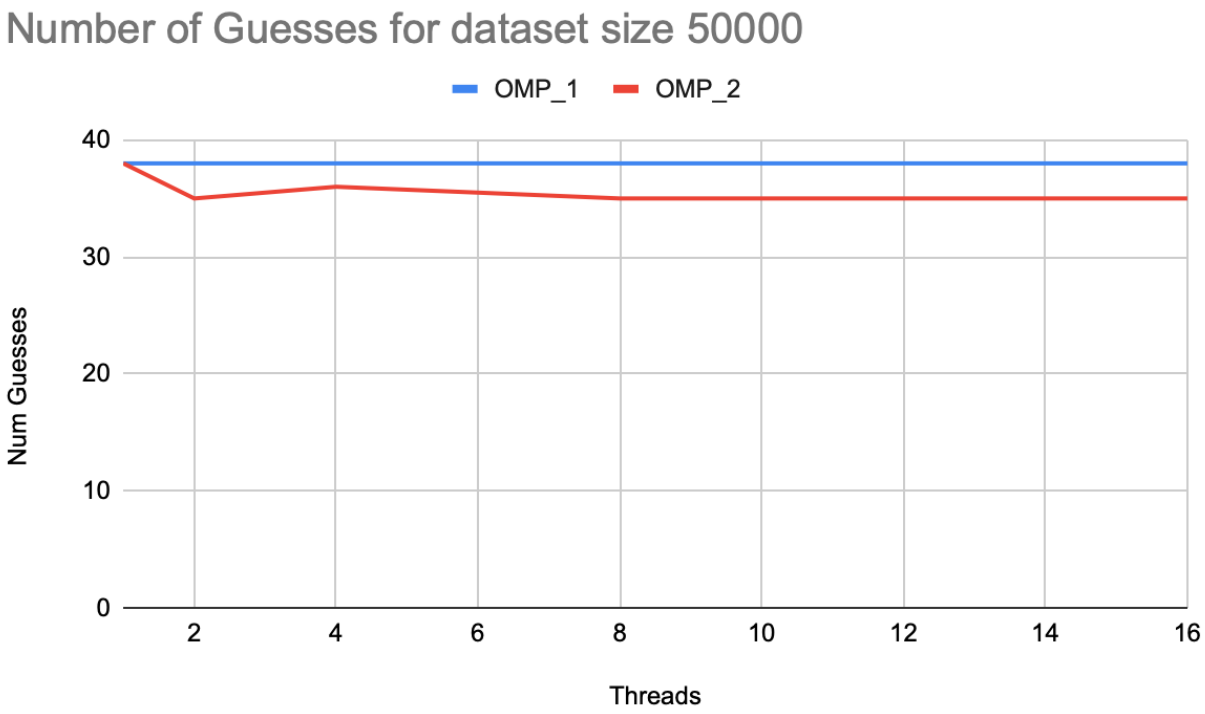
but it doesn't improve very much as the number of threads increases because of the large presence of synchronization on the shared letter-tallying table.

Graphs for number of guesses:

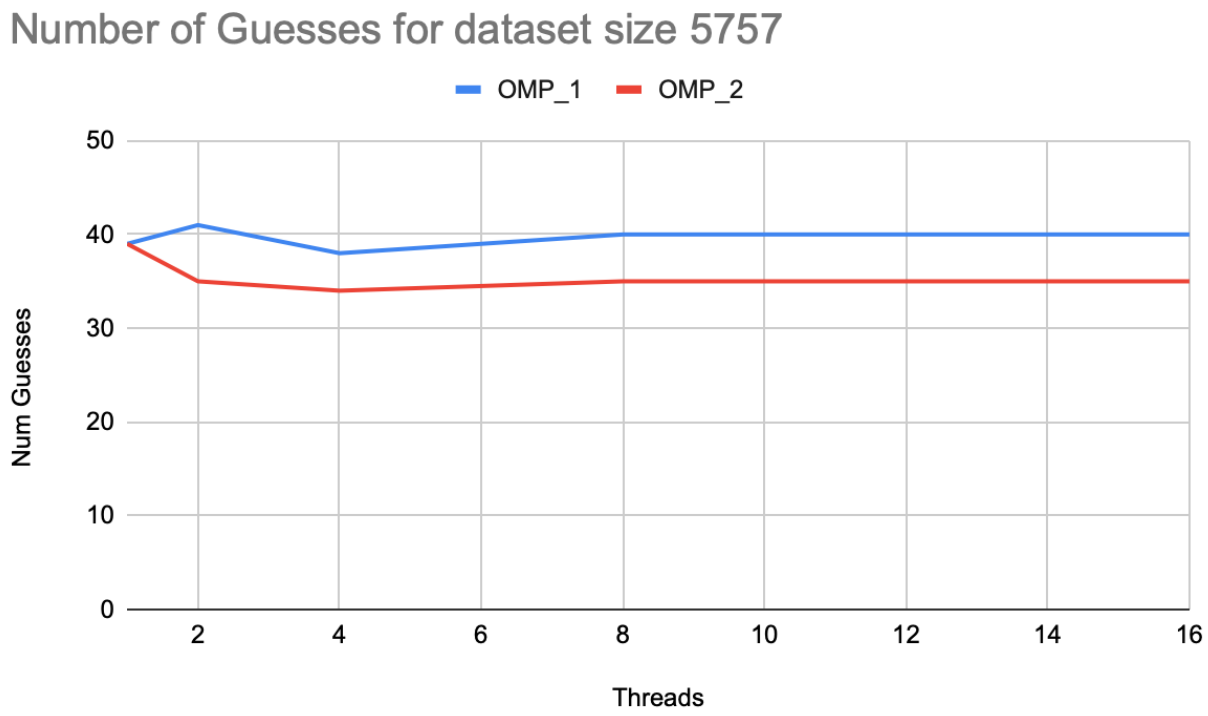


Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)





Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)

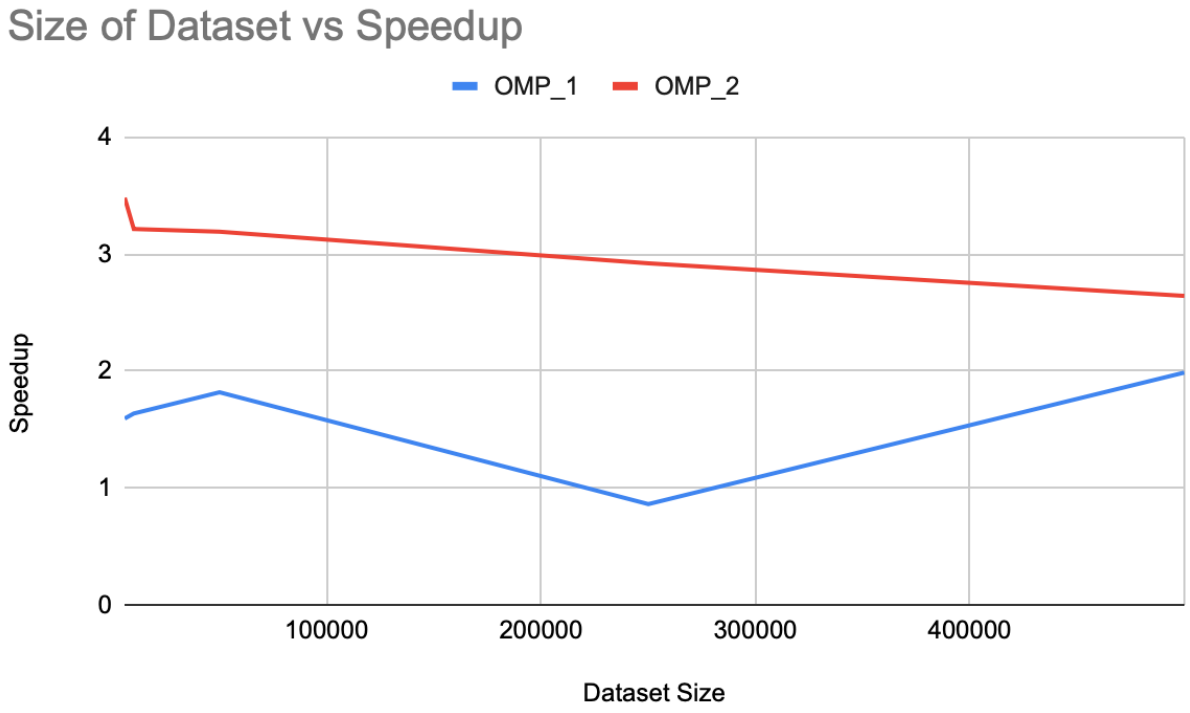


The first OpenMP approach considers the tallied data from all the boards at the same time in selecting its next guess, while the second (red) approach selects the best guess from among

Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)

candidate best guesses proposed from each individual wordle board. We notice a slight trend where OMP_2 seems to solve the duotrigordle in fewer guesses than OMP_1. This may be due to the first approach considering all boards in selecting its best guess which creates a guess that is sometimes too general (trying too much to appeal to all wordle boards all at once), and ends up paying less attention to words that are a perfect fit for a particular wordle board but not very applicable for others (but would have been an almost guaranteed correct guess for the one specific board). The second method (OMP_2) does handle this scenario a little better actually, since it gives each wordle a chance to put forth a best guess candidate from that wordle board. However, in general, both approaches still solve the duotrigordle reliably in a very similar number of guesses. Our goal was to solve the game in under 37 guesses reliably, and you can see from the above graph that this is satisfied for the realistic original dataset of 5757 words.

As discussed previously, we noticed a negative correlation between speedup and dataset size, and this is shown by the following graph:



Overall, we were able to successfully create several versions of a parallel Duotrigordle solver, and we were able to improve upon our initial ideas in a way that created reliable execution speedup on the GHC machines without sacrificing gameplay success (low number of guesses). This project was an enjoyable and interesting way to apply what we had learned in 15-418 to one of our own ideas and overall we are happy with the results we were able to achieve. Also, this project allowed us to show that we are well-equipped to not only be effective parallel programmers, but to also understand and explain why certain trends occur and why some methods are more effective than others.

Tanvi Bhargava (tanvib)
Grayson Moyer (gmoyer)

REFERENCES

- Words dataset information: https://www-cs-faculty.stanford.edu/~knuth/sgb-words.txt
- Reference for sample Wordle solvers, but actual code was unused:
  https://github.com/jason-chao/wordle-solver
  https://notfunatparties.substack.com/p/wordle-solver

Work Distribution

We spent most of this time pair programming and working on our approach over Zoom or in person together. We generally split the work as outlined in the Gantt chart included in our project Milestone report in the README in this repository. 50%-50%

LINK TO FINAL VIDEO

Please follow this link to a google photos album which contains our presentation video:
https://photos.app.goo.gl/HnKs1qeQJWyMaEMx9