Parallel Wordle Solver in C++

Charlotte Zhuang

April 2022

ABSTRACT

Wordle is a word guessing game where players guess five letter words using hints about which letters appear where in the word. The typical algorithm used to solve Wordle considers all possible words to find the guess that is most likely to narrow the set of possible words to one that is easier to solve or be the correct word. Much of the work needed in this approach is in considering each word against each possible result, which is a task that can take advantage of data parallelization. This paper will summarize building a sequential Wordle algorithm and then parallelize it for an audience with understanding of basic C++.

1. INTRODUCTION

To play Wordle the player chooses a five-letter word and makes a guess. Each letter is then revealed to be either correct, in the wrong location, or not in the word. Using this information, the player makes guess after guess until the hidden target word is guessed. Metrics to measure performance are arbitrary, but commonly used are average number of guesses or average win rate under three or four guesses. The metric being optimized matters only when considering the expected value of a possible guess. Whatever heuristic is used would hope to accurately predict either the average number of guesses left or the chance of guessing the word within the desired limit.

There are a set of 12953 guessable words, which can be considered the possible moves a player can make, but only a subset of those are possible words the game will choose. This is because most of the guessable words are uncommon words that most players do not know, such as the theorized best first guess "salet". The game considers a reduced list of 2309 words and uses them one after the other in order. This paper will consider the 2309 target words only for evaluating the performance of our solver as it can be considered hidden information not given to the player.

Wordle has a "hard mode" that restricts the guessable words to only those words that match all clues given so far, i.e., the player can only guess words that can possibly be the target word. This reduces the set of possible moves with each turn, slightly reducing the quality of possible guesses. This mode will be used for this paper as it greatly speeds up processing time by reducing the number of words that need to be considered with each successive guess. Min-max optimization can be used to ignore words that will never be used and improve performance of "normal mode" solvers, but that was not considered here.

2. SOLVER ALGORITHM

The algorithm used in the paper is one developed by 3Blue1Brown [1]. We are choosing the word that minimizes the below function $E(w)$, which is the expected number of guesses left after guessing word w. $P(w)$ is the probability that the w is the target word and $H(w)$ is the heuristic function that is used to estimate the difficulty of the game after guessing w. In other words, we consider the probability that the word is correct and the next guess will end and the probability that the next word is incorrect weighted against the estimated number of guesses to find the correct word.

$$E(w) = P(w) \cdot 1 + \left(1 - P(w)\right) \cdot H(w)$$

This equation explains why we still need to consider all 12953 guessable words even though most of them would have a probability close to zero of being the target word. Early on, we would make guesses that make the game easier to solve first, until there are few enough reasonable words left that we can try to guess the actual target word.

2.1 Word Probability

To assign probabilities to each word, we first assign a weight to each word using its relative frequency in the English language. We consider all guessable words sorted by their frequency, and apply the sigmoid function to produce a weight between 0 and 1. The parameters of the sigmoid function were chosen manually through experimentation and by considering what words seemed common and what words seemed uncommon. The goal is that all words in the target set have a weight near 1 and the rest have a weight near 0. This function can be improved using machine learning.

$$weight(i) = \sigma\left(\frac{8 \cdot 2048}{n} - 8 + i\right), \text{ for } 0 \le i < n$$

where i is the index of the word frequency

in sorted order and n is the number of words

Then finding the probability of a word w is simply dividing it by the total weight of all possible words W.

$$P(w) \frac{}{w_{weight}\{\sum_{x \in W^{x}_{weight}}}$$

2.2 Heuristic Function

The heuristic function considers the expected entropy $G(w)$ after guessing the word w . Entropy is a measure of how many possible outcomes there are with higher entropy indicating a harder game and low entropy indicating an easier game. When there is zero entropy, there is only one possible word. Entropy is measured as bits where 1 bit of entropy corresponds to 2 equally likely outcomes and k bits of entropy corresponding to 2^k equally likely outcomes. The parameters are produced from a simple line connecting 0 entropy at 1 guess to 11.5 entropy at 3.5 guesses. These parameters can be improved with a more complex function on experimental data relating entropy in a game to the number of guesses required to solve it. Actual performance of the solver was actually worse than assumed, which is a caveat the author could not resolve.

$$H(w) = \frac{11.5}{2.5} G(w) + 1$$

$G(w)$ is calculated from the entropy of each possible result $r \in R$ and multiplying it by the probability of the result occurring $P(r)$. The set R has a size of 3^5 as it is all permutations of correct, misplaced, and not in the word across all five letters. A result r is a single permutation of that.

$$G(w) = \sum_{r \in R} P(r) \cdot \log_2\left(\frac{1}{P(r)}\right)$$

$P(r)$ is calculated as the sum of the weights of all words that match result r having guessed w divided by the total weights of all results. Note that each word produces exactly one result so the total weight of all results is equal to the total weight of all remaining words.

$$P(r) = \sum_{x \in r} x_{weight} \div \sum_{x \in W} x_{weight}$$

2.3 Deep Search

We can improve accuracy of our heuristic function by considering the actual entropy of the next guess we can make rather than estimating with a regression function. This produces a tree where each word's expected number of guesses is dependent on child words and their guesses. The computational cost in hard mode wouldn't increase that much as most of the work comes from the first guess where there are the most options, but in normal mode deep search would increase work exponentially.

Exact expectations can be calculated by recursively searching down the tree to leaf nodes, removing the need for a heuristic at all. This paper did not implement this function as the simple searching algorithm demonstrates parallel processing implementation and benefits well enough.

2.4 Algorithm Caveats

The solver achieves an average score across 10 random words of 4.1, which is not nearly as good as the state-of-the-art of 3.42 [2], but is nearly as good as human players. The performance definitely has large room for improvement, which can be improved with deep search. The C++ program also takes quite a long time to make its first guess since all words are considered. This can be easily eliminated by storing the best first guess since the game state is identical for the first guess; however, there are many situations where this is not possible and running code to find that first guess solution is non-trivial in itself.

3. SEQUENTIAL SOLVER IN C++

3.1 One-Pass Entropy Function

First a sequential solver class was written to implement the algorithm outlined before. Some changes were made to more efficiently calculate the entropy function $G(w)$ in one pass. The functions can be shown to be mathematically equivalent.

$$G(w) = \log_2(t) - \sum_{x \in W} x_{weight} \cdot \log_2(x_{weight}) \div t$$

$$\text{where } t = \sum_{x \in W} x_{weight}$$

3.2 Memory Management

Due to the large number of words, words should be stored in heap memory. Typically, this is done using the `new` keyword, followed by using the `delete` function to later free the object's memory. C++17 has the class `unique_ptr` that handles memory management of an object so that memory leaks will not happen in case of an exception or if the programmer forgets to free memory. However, if we store our objects as values in a `vector`, the `vector` class will also automatically store the object in the heap and free the object once it is no longer in scope. Only the `vector`'s header is stored in the stack, unless the `vector` is specifically created in the heap using the `new` keyword or a `unique_ptr`.

3.3 Moveable Classes

The `unique_ptr` wrapper requires its content to be move constructible and move assignable. This also makes the object compatible with the `std::move()` function, which means that it can be moved to a `vector` or between pointers more efficiently. Moving an object in C++ means giving the left-hand variable the state and all the members of the right-hand variable, and leaving the right-hand variable in an indeterminate, but valid state. This means that the right-hand value can be empty, or have incorrect data after the move, but can have new data assigned to it and continue to be used.

Movement is important as we do not wish to create copies of strings if we can simply move the same string around. This avoids the potentially costly memory allocation of thousands of short five-byte objects around memory. Additionally, the `vector` class requires that movement can be done without exception, or it will try to copy. Also be aware that defining the move constructor and move assignment operator automatically deletes the corresponding copy functions unless you define them again. This is fine as long as we always move our objects rather than copy them.

A class can be made move constructible and assignable by defining the functions in figure 1, as seen in `Solver.cpp` lines 8 to 18.

```
Word(Word &&rvalue) noexcept

Word &operator=(Word &&rvalue) noexcept
```

Figure 1. Move constructor and move assignment function headers.

3.4 Debugging

Debugging in C++ can be challenging as exceptions are not automatically thrown by the standard library for the sake of efficiency. To not sacrifice efficiency in our code, but also validate the state of our program during runtime, we can define a debug compiler directive. If we compile with the directive, in our case by adding the `-D_DEBUG` flag, we can turn on our custom defined assertion macro to check values in runtime and print them if they are not what we expect. The custom macro is necessary as the standard library assertion macro does not print anything useful. The directive and macro are defined in `Solver.hpp` and are shown in figure 2. If the `_DEBUG` directive was defined, the `ASSERT` macro will check the values passed to it with an if-statement, then print out debug information and throw an exception if the check fails. If the directive is not defined, then the macro is replaced by an empty statement that will be ignored by the compiler.

```
#ifdef _DEBUG

#define ASSERT(left, operator, right) \

{ \

  if (!((left) operator(right))) \

  { \

  std::cerr << "ASSERT FAILED: " << #left <<
#operator<< #right << " @ " << __FILE__ << "
(" << __LINE__ << "). " << #left << "="
<<(left) << "; " << #right << "=" << (right)
<< std::endl; \

  throw "Assertion error"; \

  } \

}

#else

#define ASSERT(left, operator, right) {}

#endif
```

Figure 2. The debug direction and custom assertion macro.

4. PARALLEL SOLVER IN C++

The parallel solver was defined as a child-class of the sequential solver, allowing it to reuse the function to initialize its word list and the $E(w)$ function for the expected number of guesses to solve the game after guessing a word w . This algorithm is relatively simple to parallelize as nearly all the time is spent calculating $E(w)$ for each word, and this function is already thread-safe and reentrant as written in the sequential solver. Thread safety means that there are no race-conditions if two threads are using the function at the same time which is achieved by only reading from shared memory or by using locks. Reentrant means that the function produces the correct result if another thread is currently executing the function—this is achieved by only using locally scoped variables in the function.

4.1 Thread pool

We simply need to create a pool of worker threads that wait for the master thread to give them words to calculate $E(w)$ for. Then the results can be given back in a result stack (fig. 3).

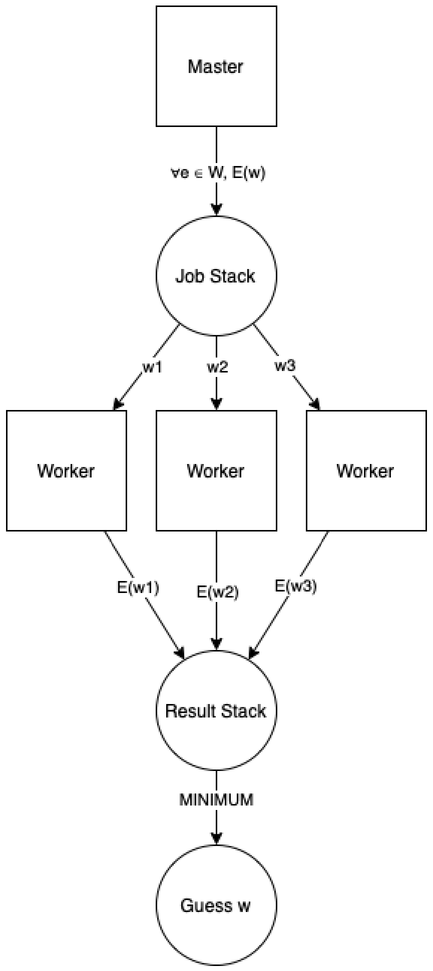

Figure 3. Master thread sends words to a job stack. A thread pool computes each job and returns the results to a result stack. The next guess is the minimum value from the result stack.

A thread pool gives the advantage of saving on thread construction and destruction costs. Threads are created upon initialization according to the number of hardware threads available and wait on a condition variable until the master signals them to start working. Then the master waits on another condition variable for when a worker completes a job. Since the master will mostly be waiting when the workers are running, they will not interfere with one another too much.

4.2 Increase Job Size to Reduce Overhead

The original thread pool described previously only manages a speedup of 3.3 on the first guess while running on 8 threads, which is lackluster. A higher speedup of 7.2 was achieved by increasing the size of each job (tab. 1). The theoretical limit of speedup with 8 threads would be a factor of 8 if we had absolutely no sequential code or resource waiting, so a factor of 7.2 on a machine with only 4 cores and 8 threads is very good indeed.

Table 1. Mean Runtimes for each Guess with 10 Repetitions

Guess Number	Serial	Parallel	Parallel Large Jobs
1	382,650 ms	117,378 ms	53,252 ms
2	381 ms	323 ms	29 ms
3	3 ms	6 ms	< 1 ms

The original parallel solver actually saw almost no speedup or very little speedup after the first guess compared to the sequential solver (tab. 1). This is due to the overhead from thread synchronization in the form of signaling and resource locking. With enough optimization, we can reduce that overhead enough that the optimized parallel solver still saw large speedup even up to the third guess, when the number of words to process was less than one hundred.

In the large job parallel solver, words were partitioned evenly across all threads, with one thread having less than the rest. Since the slowest thread determines the overall runtime of the solver, it does not matter if one thread is underloaded as long as all the others are loaded as the most heavily loaded thread has minimal load. A simple way to do this is to simply round up to the next evenly divisible number of jobs, then give the last thread less. Since we only have 8 threads, the imbalance will also never exceed that many words.

We can also minimize the amount of memory used by passing only the indexes that a thread should work on to each thread, and allowing all threads to read from the shared word memory. The results still need to be passed as arrays if we want the master to have access to results for all words (if we wanted to have the option of choosing between the best 10 words for example), but we can also cut the overhead in this step by making the transfer happen all at once rather than one-by-one as results are calculate (fig. 4). If we use move-assignment as discussed earlier in this paper, moving the result `vector` will actually happen in constant time. Each worker receives their job, calculates all results, then copies the results to shared memory and signals the master.

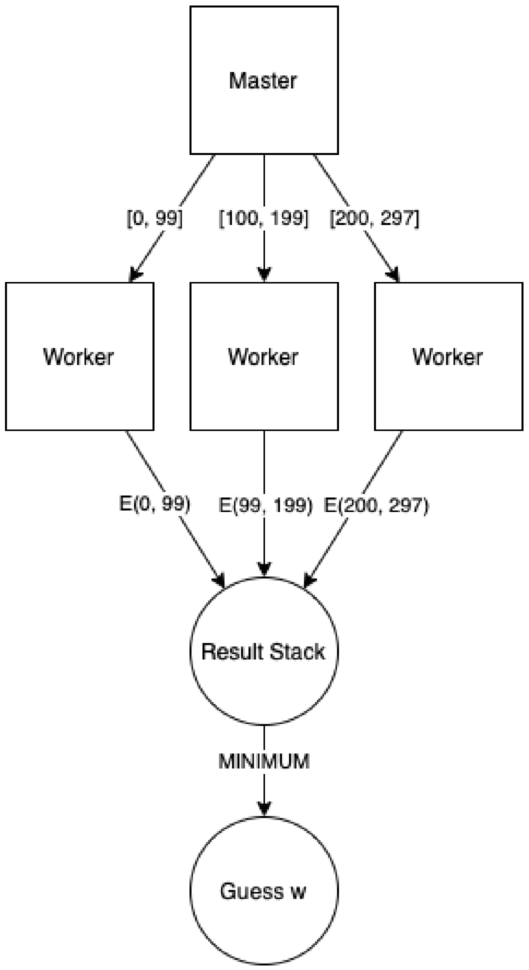

Figure 4. Large job parallelization with reduced overhead.

4.3 Thread Synchronization Details

In the parallel solver class, you will see a few lambda functions. In case the reader is not familiar with functional programming in C++, figure 5 shows a simple lambda function. The square brackets [] specify if the function can reference an external scope or any external variables (& for the entire scope, `this` for the object's scope). The parentheses () contain parameters for the lambda function, if any. If the function returns anything, that can optionally be specified with an arrow. Then curly braces {} contain the function body, much like a standard function in C++. Lambda functions are useful if static variables in the function need to be defined during runtime, or if a static function needs to be passed from a non-static context.

```
function<int(int)> my_lambda = [this](int
arg1)->int
{
  return this.member_func(arg1);
};
```

Figure 5. An example of a lambda function in C++. This lambda function can be used to execute a member function in a static context.

The parallel solver in `Solver.cpp` only uses two condition variables, `pool_cv` and `master_cv`, to signal and protect the shared resources that the thread pool and master read from respectively. In C++, condition variables require their own `mutex` to protect their `wait` function and the variables that the condition variables in turn read from. C++17 also has the `unique_lock` wrapper class, which is used with the Resource Acquisition is Initialization (RAII) programming paradigm. When we wish to access protected resources, we simply create an anonymous scope using curly-braces `{}` and initialize a `unique_lock` at the top of the scope. The lock is acquired on initialization, then released when the `unique_lock` leaves scope and is destroyed. This ensures that resources are released without having to remember to call the function releasing them (fig. 6).

```
// unprotected code
int count
{
  std::unique_lock<std::mutex>
lock(count_mutex);
  // inside critical section
  count = get_count();
}
// use count here in to minimize critical
section
```

Figure 6. Example code using RAII.

It is also important to use proper thread synchronization classes rather than boolean flags as the compiler does not know what boolean flags can change under what circumstances. A thread may never be aware that a flag's state has changed values even if no resource protection is needed. The condition variables will handle this for us, or we can use the `atomic` class if any flags are needed without a condition variable (fig. 7).

```
while (flag.load())
{
  // do nothing
}
```

Figure 7. Example of using a boolean flag in a thread. The flag must be atomic or the thread may never exit the while-loop.

Furthermore, even if different indexes of a `vector` are being modified by threads so that no race condition would appear to exist between threads, we must protect the entire `vector` if any thread is writing within it (fig 8.). Say if `thread0` writes to `vec[0]` and `thread1` writes to `vec[1]`, undefined behavior can result since vectors require mutual exclusion even when accessing separate locations.

```
// i = thread index
{
  std::unique_lock<std::mutex> lock2(solver-
>pool_mutex);
  solver->thread_ret[i] = std::move(ret);
  solver->thread_status[i] = false;
}
```

Figure 8. Lines 402 to 406 of `Solver.cpp`. Even though each thread accesses different locations in the shared vectors, mutual exclusion is necessary using a `mutex`.

In the master thread of our parallel solver, we check a counter protected by a condition variable to see if any threads still are complete, then we look for the specific thread that has results ready for us. Even though threads signal the master each time they complete, the master will only receive the signal if it was waiting at the time it was sent. The counter will let the master know if it should look for any remaining threads in the case if any signals were missed while it was busy (fig. 9).

```
for (size_t i = 0; i < threads.size(); i++)
{
  {
    std::unique_lock<std::mutex>
lock_master(master_mutex);
    master_cv.wait(lock_master, [this]()
      { return thread_done_count > 0; });
    thread_done_count--;
  }
  // process a result returned from the
finished thread
}
```

Figure 9. Lines 319 to 326 of `Solver.cpp`. The master thread checks a counter to see if any results are ready to be processed.

Since our thread pool will always wait for jobs to perform, we need a way to signal to the threads to terminate and allow the master to join with them. This was achieved using a flag protected by a condition variable. The destructor of our parallel solver class is a convenient place to join our threads and free their resources so that the caller of the parallel solver class does not need to manage our resources (fig. 10). As long as it was not declared using the `new` keyword, the parallel solver's destructor will be called once it exits scope.

```
SolverParallel::~SolverParallel()
{
  {
  std::unique_lock<std::mutex>
lock(pool_mutex);
  terminate_pool = true;
  }
  pool_cv.notify_all();
  for (auto &t : threads)
  {
    t.join();
  }
}
```

Figure 10. Lines 288 to 298 of `Solver.cpp`. The destructor will terminate and join with all its threads once the parallel solver object is destroyed.

4.4 Using a Pointer versus Reference

In C++ there is the option to store and pass objects with pointers or references. This is separate from the question of passing by value or by reference to a function (pass objects by reference unless the function needs a copy to change without affecting the caller). Also, this only applies to objects as simple types such (e.g., `double`) can very efficiently be copied by value. Generally, pointers should be used if the variable needs to point to different objects during its lifetime or if `null` is an acceptable value. References can be used if only one object will ever be referenced by the variable or if `null` is not an acceptable value. If you like, you never really need to use references so long as you properly manage your pointers (delete their objects to prevent memory leaks); however, references combined with a `unique_ptr`, `vector`, or similar class can manage the object's memory automatically, making references better stylistically and easier to use (fig. 11).

```
std::vector<Word> words; // storing references

const Word *res = nullptr; // pointer variable

double best = std::numeric_limits<double>::max();

for (size_t i = 0; i < words.size(); i++)

{

  if (expect[I] < best)

  {

    best = expect[i];

    res = &words[i];

  }

}
```

Figure 11. An example based on lines 305 to 354 in `Solver.cpp`. References are used in a `vector` for automatic memory management. A pointer is used to allow for reassignment and to avoid storing by value, which is expensive for objects. Storing by value is efficient for simple types like a `double`.

5. SUMMARY/CONCLUSION

Wordle offers simple, but extremely effective opportunities for data parallelization. The development cost to parallelize the solver code in C++ was considerable, requiring intermediate knowledge of C++'s thread synchronization libraries and compiler behavior. The number of member variables doubled and debugging proved to be non-deterministic and difficult due to reduced visibility of thread state from the parent process. In the end, performance improvements were tremendous with well-implemented parallelization, nearly reaching the theoretical speedup limit. Further research into superior Wordle algorithms would be needed to produce a more useful solver that can beat humans in a reasonable amount of time.

6. PROJECT SOURCE CODE

https://github.com/charlotte-zhuang/wordle

7. REFERENCES

[1] 3Blue1Brown. Solving Wordle using information theory. YouTube, 6 February 2022. https://youtu.be/v68zYyaEmEA.

[2] Olson, J. Optimal Wordle Solutions. 2022. https://jonathanolson.net/experiments/optimal-wordle-solutions.