**University of Pittsburgh**

**CS1632 – DELIVERABLE 4**

**Performance Testing**

Jamie Min: minj131

Paul Yu: pyy2

https://github.com/minj131/D4

Bill Laboon

12 Nov 2018

**Summary**

**What was the most challenging aspect of this deliverable?**

The most challenging aspect of this deliverable was figuring out an optimal way to figure out all paths from each node to each end node. This problem is an NP-hard problem as the number of paths to calculate grows exponentially with each new node added. The only (really) solution for this is to brute force. Despite this, the calculations for the given graphs were not too intensive. The second most challenging aspect was to figure out a way to generate all permutations of all possible paths for the ultra\_big graph text file. Finding permutations of strings that are length 12 takes an extremely long time as there are roughly 480 million possible strings for a string of length 12. Factor in multiple strings of this size, we needed to find a way to optimize this as the program would time out for the ultra\_big text file.

**What kind of edge cases and failure modes did you consider?**

**Using the flame graph, what methods were taking up the most CPU time?**

In the preoptimized solution, creating the dictionary set with the wordlist.txt file was taking up the most CPU time. For the ultra\_big text file, the most time was spent on finding the permutations which would eventually cause the program to time out and then after the optimization, converting the large dictionary to a hashed dictionary was taking a lot of CPU time. Additionally, generating all the possible paths used a recursive function which was evident in the large call stack in the flame graph. These points were possible avenues for optimization.

**Explain the changes that you made based on the flame graph or timing?**

Based on the timing, we found that the program would time out when trying to find the permutations of all possible paths in the ultra\_big graph. This was our first avenue of optimization. Instead of finding the permutations of all possible paths, we first read the dictionary and hashed it by mapping keys as sorted strings of incoming words (i.e., CAT would become ACT as the key) and mapping all anagrams of incoming words to that key (i.e., CAT, ACT, and TAC would map to ACT since they are all valid anagrams of each other). For each path (and each word), we simply would take the word and sort it by its character values. So, if a possible path was CAT, we would sort it and it would become ACT. Since we want to find all words possible from its permutation (i.e., a collection of its valid anagrams) we simply had to get the values for that specific key (ACT) and it would return all the valid words for that key. This allowed the ultra\_big graph text file to actually find a solution (a big step up).

This, however, caused a lot of the CPU time to be used for creating the hashed dictionary as it needed to go through the word list line by line and create the mappings. According to the flame graph, around ~60% of the total execution time was spent on this method.

Our approach to optimize this was three-fold. First, we put the work of reading in the wordlist.txt file into a list in a separate thread. Then, we split the dictionary into 4 approximately equal parts, Across 8 processes using the Parallel gem, we created 4 smaller hashed dictionaries and did the work of hashing the dictionary. After we find all of the paths, we return a sorted set of the longest paths first and as we convert each path to the word, we check if it’s a valid word and break once we find the next smallest valid word, returning the list of the valid words found. With these optimizations, we increased performance denoted by the total time execution by roughly ~14% than just using straight to hashed dictionary alone.

**Flame Graphs**

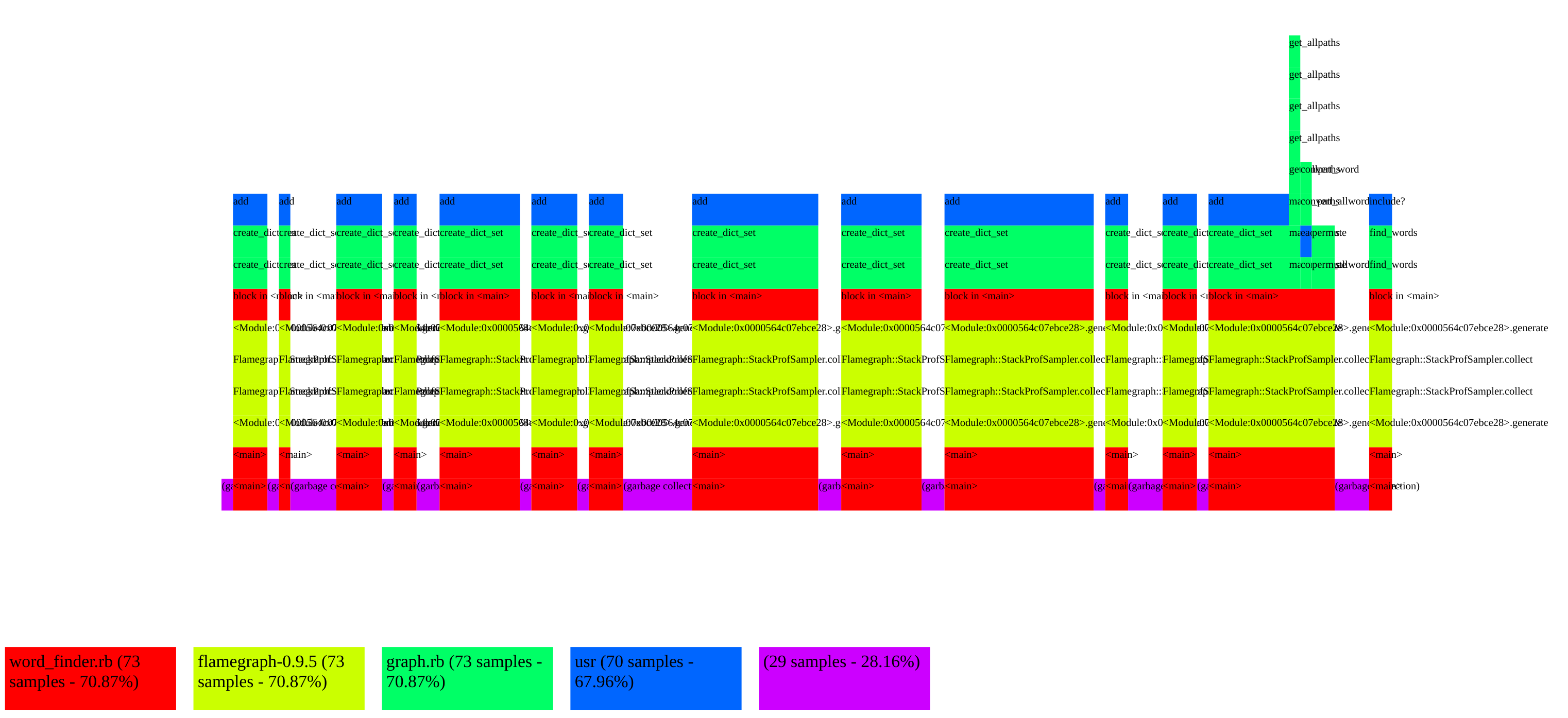
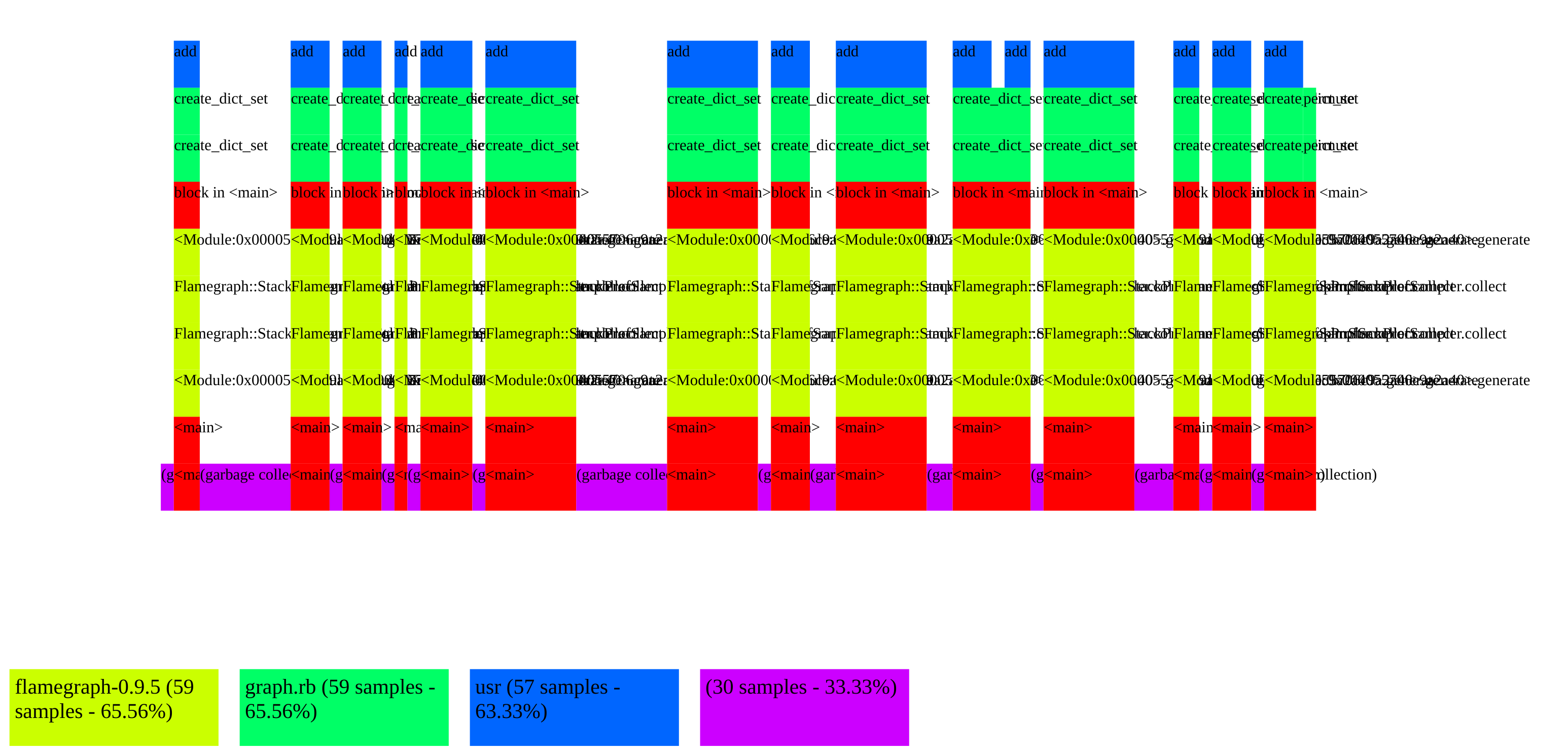
******Preoptimized:**

Figure 2 - medium\_size\_graph.txt

Figure 1 - small\_graph.txt

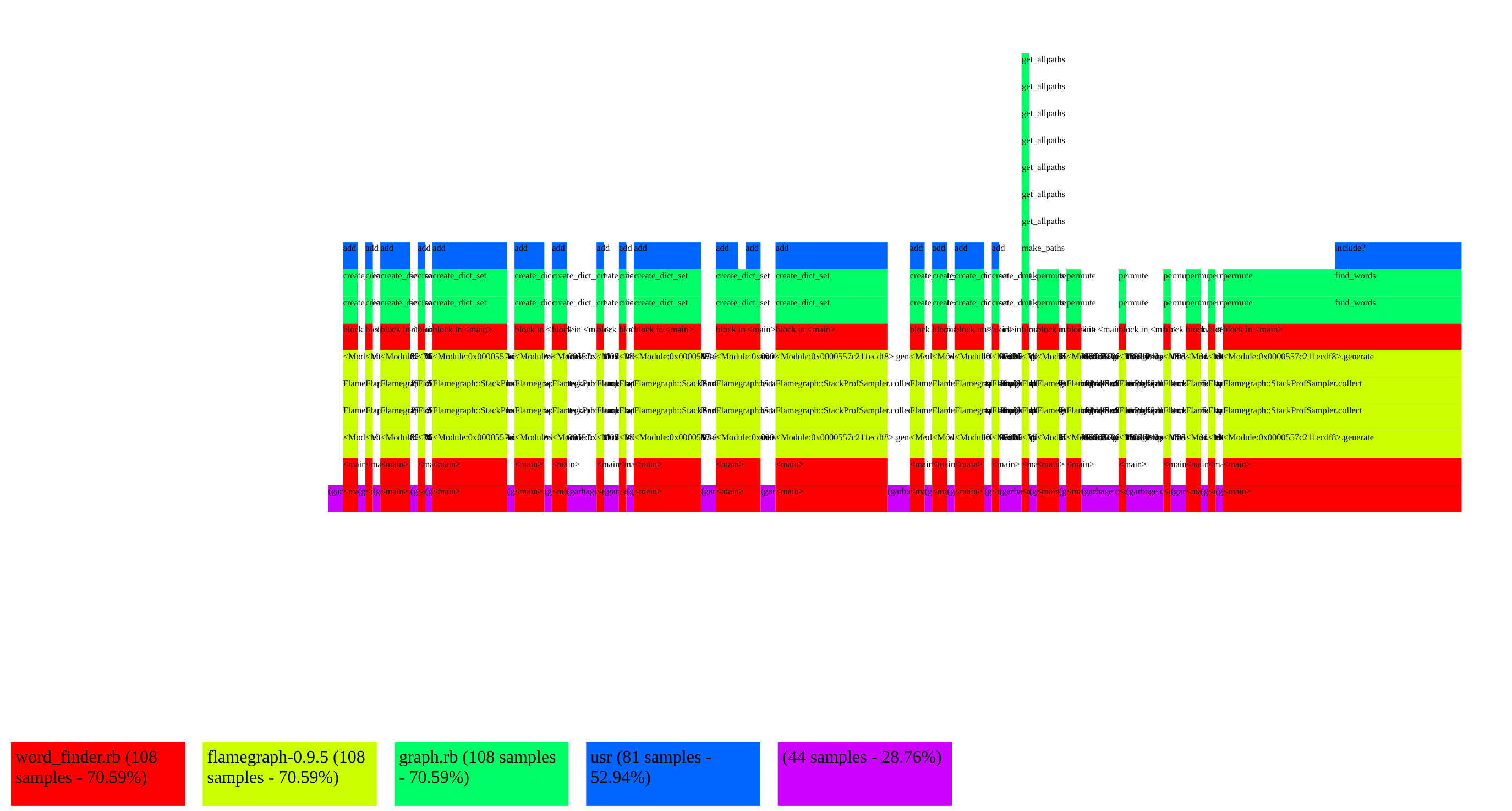


Figure 3 - big\_graph.txt

**Optimized with Hashed Dictionary (Only for ultra\_big\_graph.txt)**

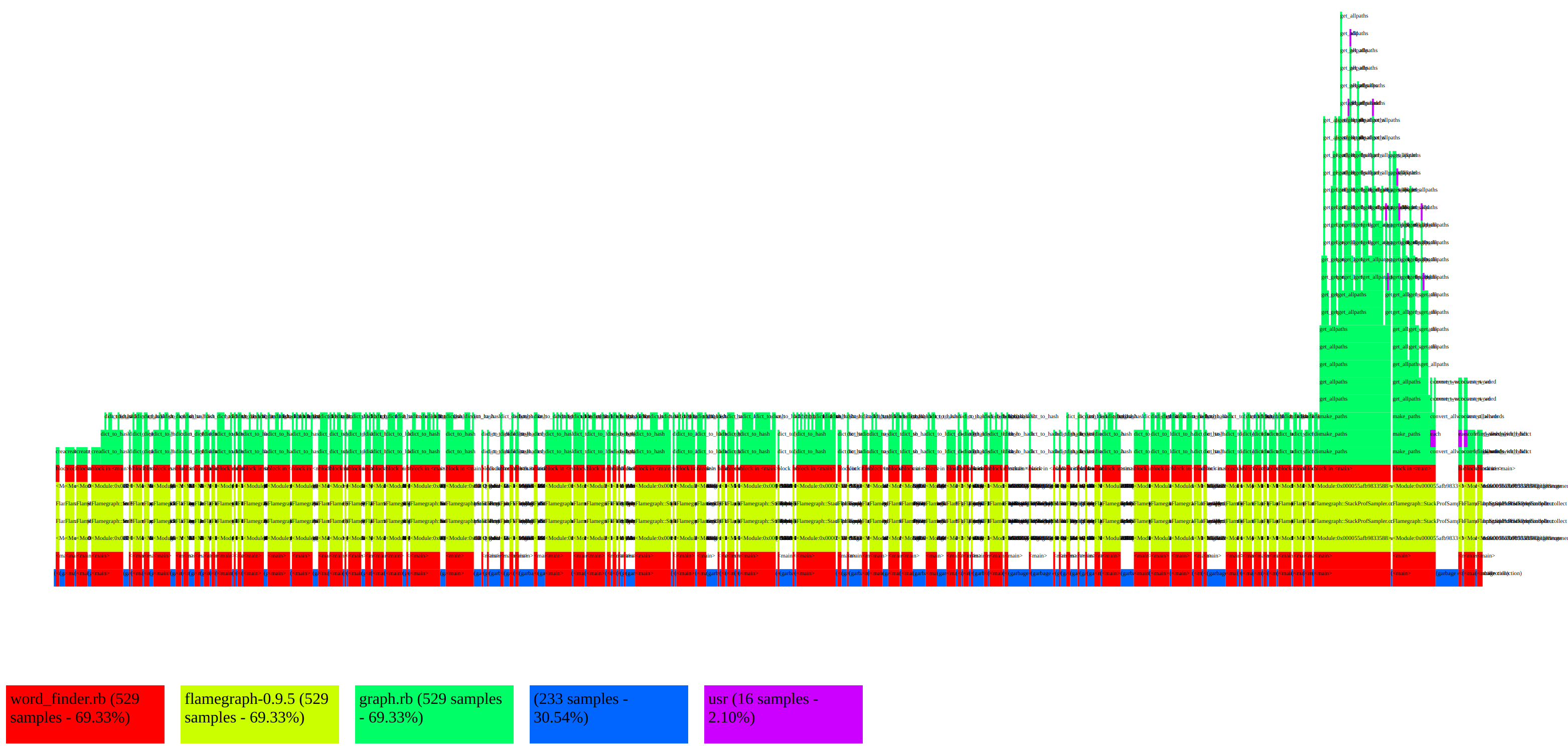


Figure 4 - ultra\_big\_graph.txt w/ Hashed Dictionary

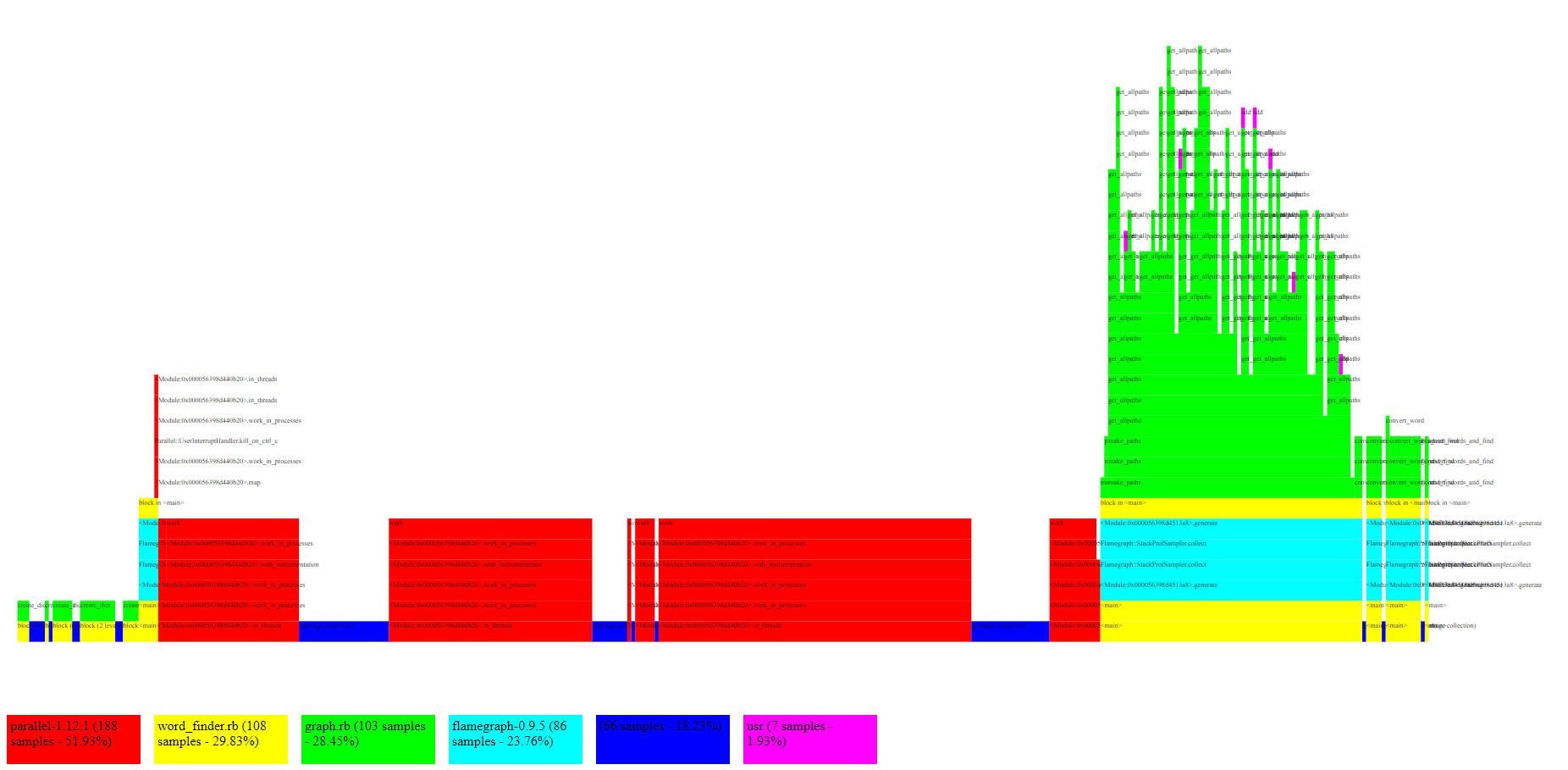
**Optimized with Threads, Processes, and Early Return in convert\_words\_and\_find**

Figure 5 - ultra\_big\_graph.txt w/ Hashed Dictionary and Threads/Processes