grepR: Recursive Mutli-Threaded grep

Christopher Draper and Joshua Bittel

1. Goal

For this project we wished to implement a version of the unix command grep that is able to recursively search through directories and will make use of threading to improve search speed on multicode systems. We wanted our project to also follow symbolics links which grep -R can do as well. In order to prevent recursive looping when following a symbolic link we wanted to implement and Bloom Filter to provide better performance and memory usage over something like a list or map.For the division of work Joshua focused on implementing the boyer-moore algorithm while Christopher worked on the bloom filter and control flow of searching through directories.

2. Bloom Filter

The bloom filter is a probabilistic, hash-like data structure. It is used to query set membership status by hashing a value to create an index into a bit-vector. Multiple hashes are used to query multiple buckets within the bit vector. If one bucket is 0, then the item is definitely not in the set; however, if all queries return a 1, then the item is likely to be in the set. False positives do occur and the probability of false positives increases in relationship to both the size of the bit vector and the number of hashes used to determine set membership. This relationship can be more formally written as,

$$\sum_t \Pr(q=t)(1-t)^k \approx (1-E[q])^k = \left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^k \approx \left(1-e^{-kn/m}\right)^k$$

Figure 1: K is number of hashes; n is number of items in bitvector; m is number of bits in vector set to

grepR uses the bloom filter to keep track of visited directories so when grepR encounters a symbolic link that links to a directory which has a parent relationship to the current director, an infinite loop of recursion will not occur. When grepR proceeds inside of a new directory, two actions occur to check if that directory is already visited. First, the full directory path is hashed three times by different hash functions. The hash results are used to index into the bloom filter's bit vector. If all there is a single bucket that is set to 0, then that directory is marked as not visited, grepR spawn a thread to manage the file entries in that directory, and the directory is inserted into the bit vector. If all of the buckets are set to 1, then grepR will mark that directory as visited and will not spawn a thread to manage that directory.

3. Hash Functions

For our bloom filter we made use of three hash functions. One by Rockert Sedgwichs Algorithms in C book, one written by Justin Sobel, and one by Peter J. Weinberger. Each of these functions improved on and compiled into one webpage by Arash Partow [1]. For what hashes to use I gathered many different types of hash functions and for each one I pushed through a ton of different strings. I was looking for hashes that produced a high standard deviation and that had highly different averages from each other. I wanted a high standard deviation because having a hash that clustered its output values around the standard deviation increased the likelihood of different strings being hashed to the same value. And I wanted the hashes we used for our bloom filter to have different averages so that they be less likely to hash a string to the same value. Even though we would be using the modulus function on the outputted hash values I felt it was important to have these properties in out hash functions. Because if a hash with high standard deviation is modulated the output would be more evenly spread out across the new values than if the hash had a low standard deviation. To test the hash functions I

fed each hash the name of every directory and file in my home directory, along with some random strings of data. With the follow results of:

Hash	SD	AV	Name
1:	484464100	2349975589	RS Hash Function
2:	3010489020	300850561	JS Hash Function
3:	1477344855	427946592	PJW Hash Function
4:	1438630570	555042622	ELF Hash Function
5:	780157740	2706606255	BKDR Hash Function
6:	1117235698	474076790	DJB Hash Function
7:	942917561	2314179574	DEK Hash Function
8:	1152153626	201745817	AP Hash Function

Hash 2 was chosen for its high standard deviation, hash 1 for its large average and decent standard deviation, and hash 3 as having both a solid standard deviation and average.

4. Boyer-Moore

Boyer-Moore is an efficient string search algorithm which uses shifting rules to skip the largest possible bytes in a searchable text. Boyer-Moore uses two shifting rules, the bad character rule and the good suffix rule, to shift the desired pattern across the searchable text. One unique feature of Boyer-Moore is that the actual character comparisons between the pattern and the searchable text are performed right to left while the pattern shifts in a left to right direction. The bad character rule shifts the pattern when a mismatch is found between a character P in the pattern and a character T in the searchable text. A shift occurs which brings the next character to the left in the pattern which matches T in line with T. See figure 2 for an example. The other

shifting rule is the good suffix rule. This rule works similarly to the bad character rule but with substrings rather than single characters. Suppose there is a match on a substring between P and T. However, a character is encountered which breaks the match. The good suffix rule will locate the recurring substring which has matched between T and P to the left of P and shift P so that the substrings still align. Figure 3 depicts this rule. The motive of the good suffix rule is to preserve matches in order to increase the chance of a full pattern and text match.

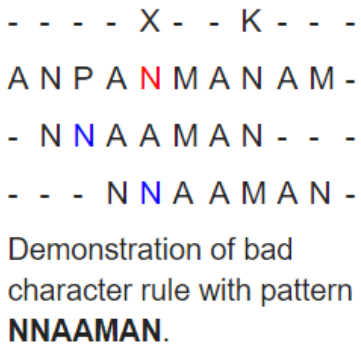

Figure 2: Bad Character Rule

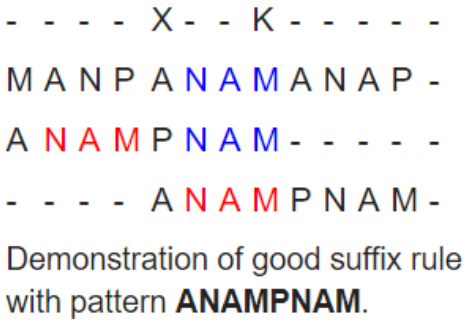

Figure 3: Good Suffix Rule

Another improvement in Boyer-moore is that it preprocesses the pattern independently of comparing the searchable text and creates two lookup tables that are used by the bad character rule and the good suffix rule. Therefore the worst case runtime of Boyer-Moore is $O(n + m)$ when the pattern is not in the text and $O(nm)$ when the pattern is present within the text where n is the time it takes to create the delta tables and m is the characters actually compared by Boyer-Moore.

5. Project Design

grepR uses pthreads to improve performance on systems with multiple cores. Every time a directory or file is encountered it is given its own thread for it to be processed. We give files

their own threads to deal with the case where a large file is encountered at the beginning of the search and prevents grepR from making full use of threads til the file has been fully searched. The basic flow of grepR is depicted below in Figure 4.

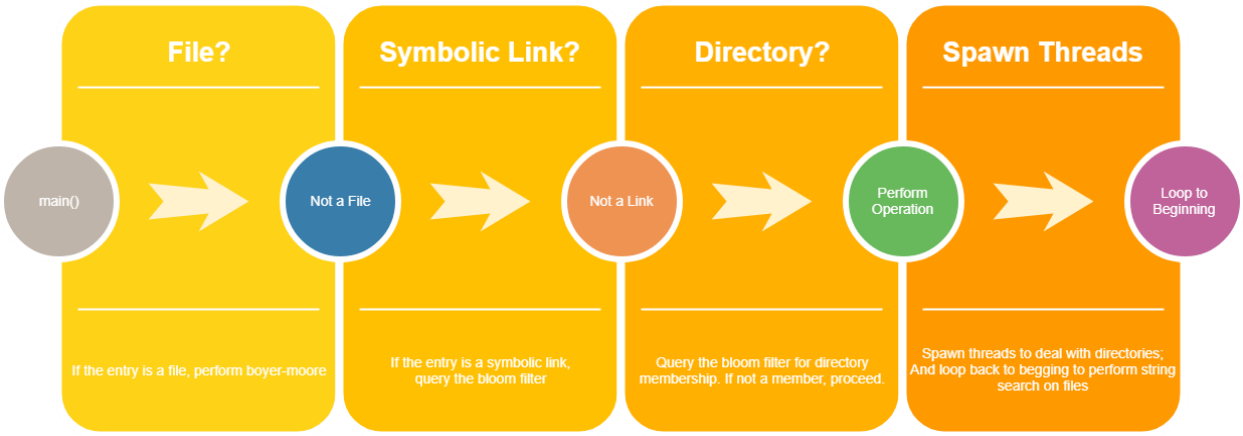

Figure 4: General overview of the code-flow within grepR

6. Evaluation

For evaluation we tested on linprog and timed grep -R and grepR using the time function on linux and gave them a number of different strings to search for. Our code ran roughly twice as fast as grep -R. With some results listed bellow:

String	grep -R	grepR
hash	0.17	0.09
cout	0.14	0.08
int	0.15	0.09
double	0.16	0.07
return	0.16	0.07
bloom	0.15	0.09
size	0.17	0.09
String	0.14	0.09
buffer	0.13	0.08
endl	0.15	0.09

argv	0.14	0.09
userFile	0.15	0.07

String	grep -R	grepR
messanger_server user_info_file	0.18	0.13
Failed to open user_info_file	0.18	0.11

We can see that our version of grepR was more impacted by longer strings than grep -R how ever grepR still ran faster than grep -R.

One reason for grep -R still being competitive with grepR is that grep has made massive amounts of improvement to the boyer moore function especially for shorter strings. Taking into consideration things like page size, improving the delta table used in boyer moore, and other small improvements to the algorithm [2]. One note of consideration is when testing on Joshua's laptop grep -R and grepR often ran at similar timings. With grep -R often being just barely beaten or matching with the performance of grepR. This makes sense as the speed of grepR is determined by the number of cores on a system. On lower core systems the overhead for grepR and the improvements grep has made to boyer-moore will mean grep is the move viable choice for searching a directory tree for information.

8. Future Improvements

There is a number of improvements that can be made to grepR. First there is implementing more of the improvements to boyer moore that grep has. Adding in these improvements should speed up our code by quite a bit for smaller strings and when searching smaller directories. We want to add optional flags for the bloom filter. First a flag for changing the size of the bloom filter. Letting the user increase the size of the bloom filter will help make it more usable for larger directory trees. Another flag that changes code behavior so it no longer

follows symbolic links. This will mean we no longer need to worry about recursive loops and so we can turn off the bloom filter functionality for a speed up. Lastly there is some improvements to our code that can be done when dealing with larger file systems.

7. Conclusions

In conclusion our implementation of grepR is a viable way of searching a file system for strings. grepR outperforms native grep -r when ran on systems with many cores by roughly a factor of 2. This performance could increased if similar modifications to Boyer-Moore were implemented in accordance to how native grep is implemented. The bloom filter works well even under heavy loads and with some of the improvements mentioned before could be modified to support systems of any size.

8. Code

Our code has been hosted on github: https://github.com/zim123abc/grep/tree/master**8.**

References

[1] Partow, Arash. “General Purpose Hash Function Algorithms - By Arash Partow.” *Partow.net*, www.partow.net/programming/hashfunctions/.

[2] Haertel, Mike. “Why GNU Grep Is Fast.” *Why GNU Grep Is Fast*, 21 Aug. 2010, lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html.