Parallelization of Rabin Karp String Matching Algorithm

CP 631 – Advanced Parallel Programming

August 3, 2022

Mohammad Bhuiyan
Email: bhui8420@mylaurier.ca

Landon Butterworth
Email: butt1110@mylaurier.ca

Abstract

Finding a pattern in a large file is a challenge. This is because of time-consuming, computation-intensive, and memory-hungry processes. The Rabin-Karp string matching algorithm is an efficient search algorithm that reduces matching time by comparing hash values. Computation time is reduced by applying a rolling hash calculation instead of comparing the pattern string with every position of the text. The time it takes to find and retrieve a pattern from a large pool of data or extremely large files by searching serially with Rabin-Karp algorithm is not practical. Especially for real-time and online applications that need to process strings as they are received. We intend to increase efficiency of the Rabin-Karp algorithm by using parallel techniques for large files. OpenMP is used to take advantage of a multicore shared memory architecture and MPI is used for a distributed memory system to build a hybrid parallel program that achieved 40 times faster search capabilities when compared to the efficiency of serial search by the Rabin-Karp algorithm. A special split method named overlap-split is developed to apply the Rabin-Karp algorithm in parallel.

Introduction

String matching algorithms are used in many fields like computer security, real-time fraud detection, signal processing, computational biology, plagiarism detection, DNA sequencing and many more. Half of the computational power is used for string matching in systems like intrusion detection [1]. The naïve String Matcher, Rabin-Karp, Knuth-Morris-Pratt (KMP) are a few well known and efficient algorithms for pattern searching in serial. Searching patterns in parallel using these algorithms poses different levels of challenges. There are various parallel routines like MPI, OMP, CUDA and languages like C, C++, Fortran, Python to opt for. MPI and CUDA based Rabin-Karp pattern matching efforts are out there [2] [3]. We built a hybrid parallel program with C bindings of MPI and OMP for pattern matching using a parallel Rabin-Karp algorithm.

Parallelization

Before parallelizing an algorithm, we should first consider whether the problem and the algorithm are good candidates' parallel execution. In the case of pattern matching, solving a subset of the problem can be a solution for a larger problem. Put more simply, finding a pattern in part of a text means that pattern exists in the entire text as well, this means that pattern matching is a good problem space to apply parallelization. The Rabin-Karp algorithm also doesn't need to be executed in any order and doesn't require any information from previous cycles, so it is also a great candidate for parallelization.

Splitting the text

One way to distribute the search work among multiple process and threads is to split the string into multiple chunks. This also reduces the risk of stack overflow caused by loading extremely large file into memory. One risk associated with splitting the text and searching is that the search pattern

might be split into different chunks handled by different processes. For example, with a pattern string of CDEF and a text of ABCDEFGH, if this was being processed by two process and the split was executed naively, we would end up with ABCD|EFGH and the pattern text of CDEF would not be found in the parallel algorithm, although it would be found in the serial algorithm. One of our criteria for success is that we produce the same results as the serial Rabin-Karp- implementation.

To overcome this issue, we can overlap the length of the pattern string between the chunks. However, this introduces another risk, if the pattern string aligns perfectly with the split point, we will find the pattern in two locations. For example, if we have a pattern string of EF and a text of ABCDEFGH split between 2 processes, the naïve split will create ABCD|EFGH and our overlap split would create ABCDEF|EFGH. As you can see our pattern string of EF now appears twice.

We can address both risks outlined above by overlapping $n-1$ length between chunks instead of n where n is the length of the pattern. e.g., we search a pattern EFG in a string of ABCDEFGHIJ; the string is split into 3 processes as ABC|DEF|GHIJ. None of the processes will find the pattern EFG. If we overlap 2 characters (3-1 = 2) at the end of each split except the last one, then the splits will be ABCDE|DEFGH|GHIJ and the pattern EFG will be found but EFG will not be duplicated. Another example of overlap-split and search can be demonstrated by splitting the string “The Wilfrid Laurier University or the University of Wilfrid Laurier” into 3 substring and search the pattern “University”. Total number of characters (including white space) in the string is 67 and number of characters in three splits without overlaps will be 22 + 22 + 23. Two instances of the pattern “University” is splitted and cannot be found (Fig-A). In Fig-B, each split is overlapped with the next split (underlined as red in below image) by “one character short of pattern length”. Please note that white space is also taken into consideration. Therefore, the search pattern will never be skipped. Duplication of the search pattern is avoided since the overlap length is not equal to the length of the pattern but one character

shorter than the length of the pattern. By applying this split-with-overlap, we can search the pattern in a large file in parallel, without skipping or duplicating the pattern.

String: "The Wilfrid Laurier University or the University of Wilfrid Laurier"

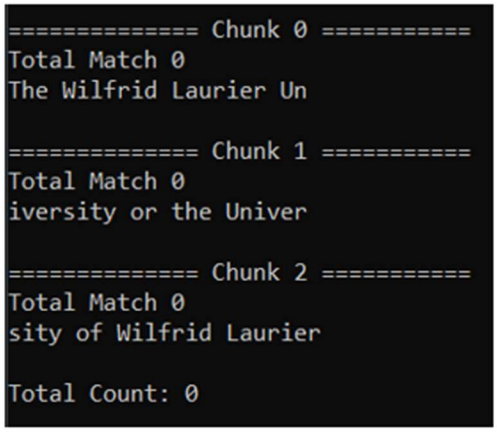


Fig-A: Split of string without overlap and failed search of a pattern "University"

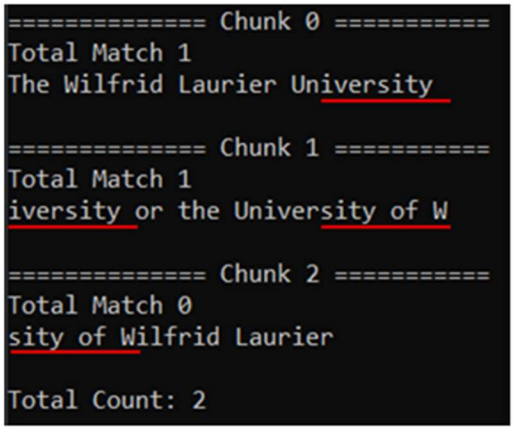


Fig-B: Split of string with overlap and successful search of a pattern "University"

Communication requirements among the processes is also reduced by overlap-split, hence improve the performance. We can see the cost of excessive communications in some previous attempts to parallelize search by Rabin-Karp [2][4].

Hybrid parallel search by Rabin-Karp algorithm

A large file or string is split into multiple processes by the overlap-split, each split of a process is further split into multiple threads by another overlap-split. The pattern string is then used by Rabin-Karp algorithm in each thread and the number of matches is added to the process level variables. The match counts of all processes are gathered, and a global counter is then updated. That global counter is the result, showing how many instances of the pattern string were found in the given text. The exact pattern location inside the large file is also recorded. For the clarity and convenience, the location of the pattern is not displayed and the code section for this is commented out.

Pseudocode

```
int main(int argc, char* argv[]){
    // nprocs: number of processes
    // nthreads: number of threads
    // pattern: substring to find
    // text: text file/ large string
    // overlap_split: function to split string
    // rabin_karp: Rabin-Karp search function
    MPI initialization;
    proc_total; //variable to hold count of matches by a process
    root_array = overlap_split(pattern, text, nprocs);
    assign pointers of items of root_array to processes;
    pragma OMP
    thread_result; // variable to hold count of matches by a thread
    child_array = overlap_split(pattern, root_array[proc_rank], nthreads);
    assign pointers of items of child_array to threads;
    thread_result = rabin_karp(pattern, child_array[thread_Id]);
    proc_total = proc_total + thread_result;
    //end of pragma OMP
    MPI_Barrier(MPI_COMM_WORLD);
    if(rank == 0)
    {
        result[nprocs];
           gather proc_total into result array by MPI_Gather;
           total=0;
           for(i=0; i < nprocs; i++){
               total = total + result[i];
           }
        print(total);
    }
    free memory;
    close opened_file;
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Language and parallel routines

We used C and parallelized the search function by utilizing both OpenMP and MPI. We compared the results of a serial Rabin-Karp search, a parallel Rabin-Karp search using only OpenMP, a parallel Rabin-Karp search using only MPI, and lastly a hybrid Rabin-Karp search that uses both OMP and MP. Gradual implementation from serial to parallel, and then hybrid parallel helped us to understand and measure the performance of the application. All four versions of the code are attached.

Test data and test environment

One large file from Kaggle (https://www.kaggle.com/datasets/omduggineni/loghub-windows-log-data) was downloaded. The file (Windows.log 28.01 GB) is large enough to split into different sizes and generate varying loads to measure the performance of the codes. Performance analysis was done on host mcs2.wlu.ca with different combinations of file size, search pattern, number of threads and processes.

Performance analysis

The codes were tested to check the time efficiency, consistency, scaling, and optimal parameters. As compared to the serial implementation, the parallel program with OpenMP was 10 times faster, the parallel program MPI was 14 times faster, and hybrid code with MPI and OpenMP with 16 processes and/or 16 threads was 40 times faster when searching a text file 10 GB in size.

Table: Time Efficiency (Efficiency of Serial Code is 1)

Code	Process × Threads	File Size (GB)	Time (Sec)	Efficiency
Serial		10	131.42	1
OMP	12	10	15.45	8.51
MPI	12	10	12.36	10.64
OMP-MPI	12×12	10	3.37	38.97
Serial		10	131.71	1
OMP	16	10	12.76	10.32
MPI	16	10	9.53	13.82
OMP-MPI	16×16	10	3.27	40.27
Serial		15	198.31	1
OMP	12	15	23.09	8.59
MPI	12	15	18.38	10.79
OMP-MPI	12×12	15	5.14	38.57

All the code versions performed consistently with different parameters. The below figure demonstrates consistency in pattern match across the splits of a file.

Screenshot of output of Serial Code
Number of Splits 12

```
======= Chunk 0 Count 756 =======
======= Chunk 1 Count 605 =======
======= Chunk 2 Count 639 =======
======= Chunk 3 Count 632 =======
======= Chunk 4 Count 138 =======
======= Chunk 5 Count 457 =======
======= Chunk 6 Count 1044 =======
======= Chunk 7 Count 989 =======
======= Chunk 8 Count 679 =======
======= Chunk 9 Count 362 =======
======= Chunk 10 Count 285 =======
======= Chunk 11 Count 232 =======

Total Count: 6818
Total Time: 131.419 seconds
Search Pattern: passed
Text File: ./data/windows_gb10.log
File Size: 10.00 gb
```

Screenshot of output of OMP Code
Number of Threads 12

```
======= Thread 0 Count 756 =======
======= Thread 2 Count 639 =======
======= Thread 6 Count 1044 =======
======= Thread 11 Count 232 =======
======= Thread 1 Count 605 =======
======= Thread 9 Count 362 =======
======= Thread 3 Count 632 =======
======= Thread 7 Count 989 =======
======= Thread 10 Count 285 =======
======= Thread 4 Count 138 =======
======= Thread 8 Count 679 =======
======= Thread 5 Count 457 =======

Total Count: 6818
Total Time: 15.450 seconds
Search Pattern: passed
Text File: ./data/windows_gb10.log
File Size: 10.00 gb
```

Screenshot of output of MPI Code
Number of Processes 12

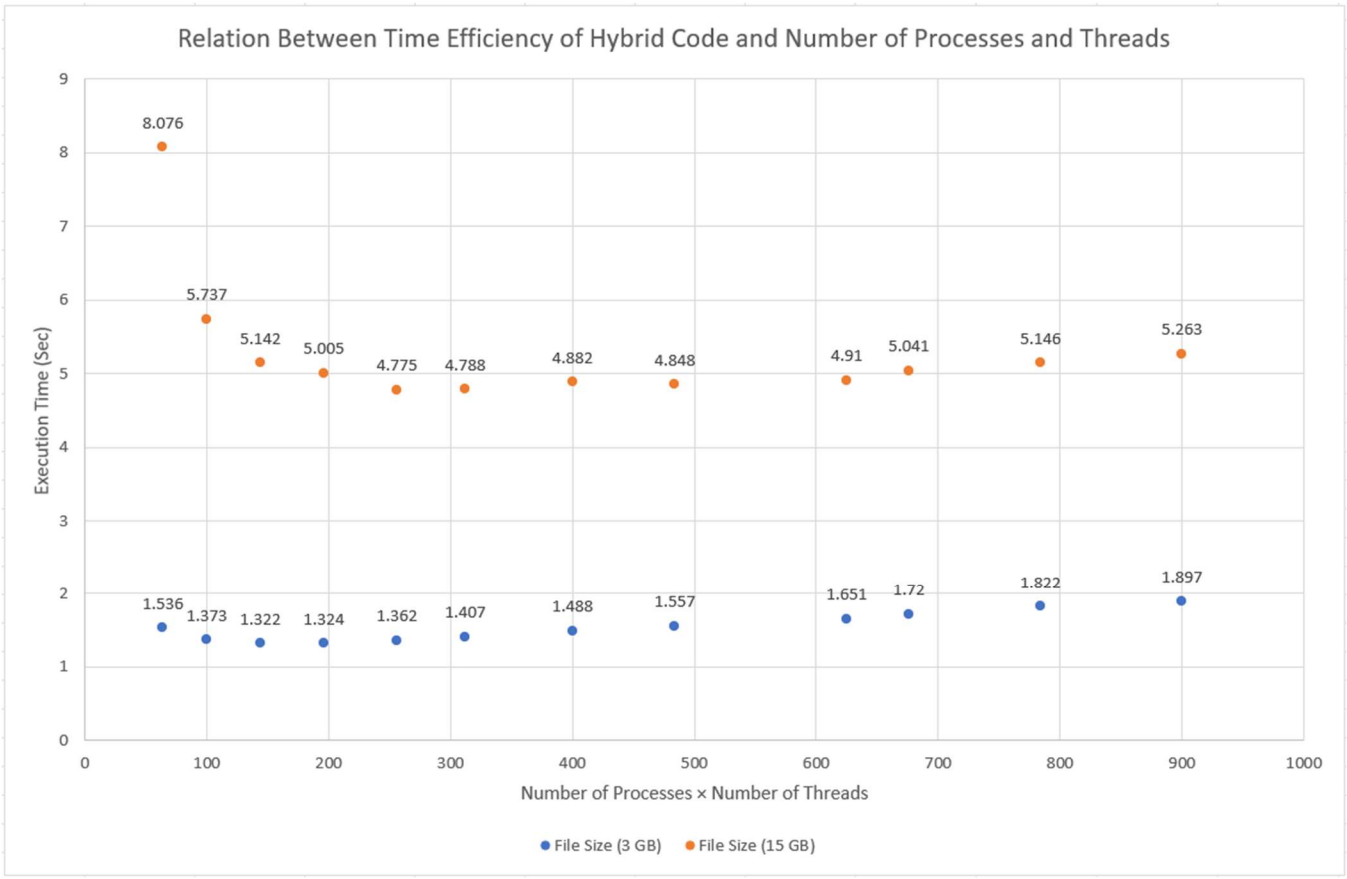
```
======= Process 1 Count 605 =======

======= Process 8 Count 679 =======

======= Process 0 Count 756 =======

======= Process 3 Count 632 =======

======= Process 9 Count 362 =======

======= Process 5 Count 457 =======

======= Process 10 Count 285 =======

======= Process 6 Count 1044 =======

======= Process 4 Count 138 =======

======= Process 2 Count 639 =======

======= Process 11 Count 232 =======

======= Process 7 Count 989 =======

Total Count: 6818
Total Time: 12.355 seconds
Search Pattern: passed
Text File: ./data/windows_gb10.log
File Size: 10.00 gb
```

Count of Pattern Match from the screenshot of Hybrid Code
Number of Processes (P) 12, Number of threads (T) 12, File Size 10 GB

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
T0	53	33	62	125	4	0	1	81	92	119	0	12
T1	114	0	136	60	2	1	52	110	134	21	6	0
T2	65	0	28	96	0	117	122	100	116	0	42	0
T3	83	18	26	43	0	107	61	35	4	0	47	8
T4	110	65	0	14	2	0	128	18	6	0	0	62
T5	83	97	0	32	3	0	78	0	0	0	0	98
T6	25	68	12	50	72	0	97	88	22	0	0	52
T7	0	45	49	184	55	0	95	141	121	37	0	0
T8	0	105	80	27	0	16	77	65	89	107	1	0
T9	27	81	103	1	0	16	140	148	6	57	20	0
T10	97	10	26	0	0	128	67	67	1	21	109	0
T11	99	83	117	0	0	72	126	136	88	0	60	0
Total	756	605	639	632	138	457	1044	989	679	362	285	232
Grand Total	**6818**											

Fig: Consistent match of a pattern search in a file by different versions of codes

The performance of the hybrid code was measured by changing file size, search pattern size, and the number of processes and threads to find the optimal parameters and maximum scaling possibilities. It was found that the optimal performance was reached at 16×16 processes and threads regardless of file sizes up to 15 GB. Performance remained unchanged or deteriorated slowly with an increase of processes and threads beyond 16×16. This is because of increased communication time between the processes.


Relation Between Time Efficiency of Hybrid Code and Number of Processes and Threads
Execution Time (Sec)
Number of Processes × Number of Threads
File Size (3 GB)
File Size (15 GB)
8.076
5.737
5.142
5.005
4.775
4.788
4.882
4.848
4.91
5.041
5.146
5.263
1.536
1.373
1.322
1.324
1.362
1.407
1.488
1.557
1.651
1.72
1.822
1.897


Execution time of different versions of codes with different file sizes
Execution Time (Sec)
File Size (GB)
Serial Code
OMP Code 12 Threads (T)
MPI Code 12 Processes (P)
Hybrid Code 12P×12T

Conclusion

Parallel search using the Rabin-Karp algorithm is achieved by using an overlap split of the original text. Large files are split into chunks in such a way that no target pattern is split up between the chunks or duplicated due to the overlap. The chunks are then distributed among processes and threads to search the pattern by using the Rabin-Karp algorithm. The number of matches is then gathered and returned. The exact pattern location inside the large file is also recorded. The efficiency of the search function is increased by 40 times using the hybrid MPI and OpenMP code as compared to the performance of the serial implementation. The hybrid code can be scaled up to 16×16 processes and 16×16 threads to get the optimal performance.

Future Work and Other Areas Explored

We've also included a parallel implementation of the Rabin-Karp pattern matching algorithm using a GPU for parallelism (by utilizing CUDA code). The implementation works well for a block of text that can fit in the GPUs memory, but the serial portion of the code dominates. This is because the input file or text must be read on the host machine then a chunk of the text must be copied to the GPU. All threads must execute and complete their work on this chunk of text, then two reduction functions must be used to sum the counts from each block and that value must be copied back to the host machine. Only then can the next chunk of text be copied over to the GPU to start that process again

To process a 512MB file including the read time, copying from the host to device, executing, reducing, and copying back to the host it takes 2.92 seconds. Since we expect that to scale linearly with the size of the file a 10 GB file would take 58.4 seconds. While this is still much better than the serial algorithm, it performs significantly worse than the other parallel methods that we explored. For this

reason, analysis wasn't done extensively, and it wasn't included in the deeper analysis and comparison to the other parallel methods.

As mentioned above, pattern matching, and frequent pattern mining is a common problem across many domains. As a result of this, there are several interesting tools that exist for handling this problem across large data sets. Benchmarking an implementation of the Rabin-Karp algorithm against some of the built-in functions in Spark that use regular expressions would be an interesting exercise and something that could be explored in the future.

References:

1. Xu, D., Zhang, H., Fan, Y.: The gpu-based high-performance pattern matching algorithm for intrusion detection [j]. Journal of Computational Information Systems 9(10) (2013) 3791–3800.
2. Sharma, Jyotsna & Singh, Maninder. (2015). CUDA based Rabin-Karp Pattern Matching for Deep Packet Inspection on a Multicore GPU. International Journal of Computer Network and Information Security. 7. 70-77. 10.5815/ijcnis.2015.10.08.
3. Moeini, Masoumeh & Shahhoseini, Hadi. (2019). Parallel Rabin-Karp Algorithm for String Matching using GPU.
4. web: https://docplayer.net/44215812-Implementation-of-rabin-karp-string-matching-algorithm-using-mpi.html