Query based file searching

CS F422: Parallel Computing

Harshit Garg (2018A7PS0218P)
Guntaas Singh (2018A7PS0269P)
K Vignesh (2018A7PS0183P)

1 Introduction

Here we create an algorithm for query based searching of a large file for a given set of words. The query words are separated using either of conjunction operators (AND / OR).

1.1 Assumptions

1. All queries are well-formed. Only one kind of operator (AND / OR) is used at a time.
2. The matching is case insensitive.
3. Hyphenated words in corpus are treated as two separate words.
4. Max query length is 2000 characters (including spaces) and each query term is 100 characters long.
5. Punctuation accepted is of the following characters " .,!;-"':", as specified in the WORD_SEP variable in the C program (which can be modified accordingly).

The serial algorithm for searching involves iterating through all the words of the file and through all the query terms, accordingly marking which queries were found (along with their location) and performing AND or OR operation to check if the query was found successfully.The first task of searching takes $O(mn)$ time where n is number of words in the file, and m is the number of words in the query (assuming strcmp comparisons to take $O(1)$ time. They accordingly take $O(MAX\_WORD\_LENGTH)$ time.). Therefore,

$$T_{serial} = O(mn)$$

2 Decomposition

The tasks are divided through the division of the input file into segments. Since the file size of the corpus is expected to be gigantic, it is natural to parallelize the search through dividing the corpus into smaller segments as per number of processes and each process would search the respective segment of corpus for the query words. The granularity of the algorithm is limited to this and not extended to produce a separate task for finding each query term in the corpus because the query is not expected to be too long and it would put unnecessary overhead for message passing which would slow down the algorithm. Thus we can see that recursive and data decompositions are put to use.

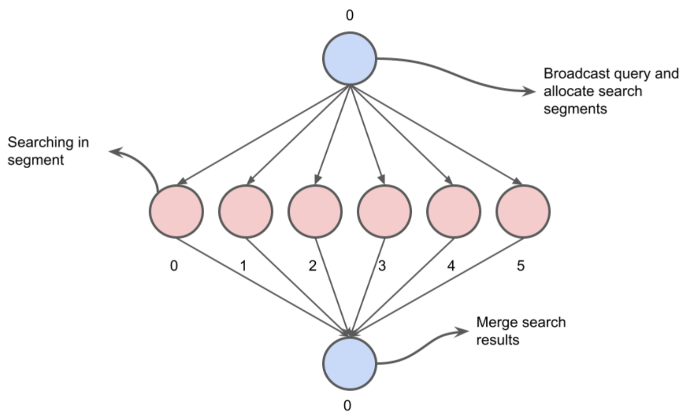

Fig. 1. Task-Dependency Graph and Task-Interaction Graph for a 6 processor machine.

The resulting task-dependency graph and task-interaction graph are as follows. There are no interactions between processes other than those necessitated by the task-dependency graph and partitioning of input data. Hence, the task-interaction graph is the same as the task-dependency graph.

$$\text{Critical Path Length} = \frac{mn}{p} + p * c \tag{1}$$

where c is the amount of work done per equality comparison

$$\text{Total Work} = mn + p * c$$

$$\begin{aligned}\text{Average Degree of Concurrency} &= \frac{\text{Total Work}}{\text{Critical Path Length}} \\ &= \frac{mn + p * c}{\frac{mn}{p} + p * c}\end{aligned} \tag{2}$$

3 Mapping

For mapping, we have to keep two basic ideas in mind:

1. Reduce idling process as much as possible
2. Reduce overhead for unnecessary message passing

Keeping these basic ideas in mind to have uniform load balancing across all the processors, we have devised some strategy. The granularity of the algorithm is also stopped at each task computing all queries on its own instead of sending it to another process.

Since the division of data is being done on the basis of number of bytes, the partitioning can be done block-wise or block-cyclic, since in either case the amount of data is definitely going to be approximately the same. (Approximate because there are minor adjustments made (not exceeding $MAX\_TERM\_LENGTH = 100$) so that each segment ends at a complete word). Thus, to have a simpler code, we have chosen block-wise partitioning of data. The file is divided into contiguous segments. The number of segments is equal to the number of processors specified in MPI.

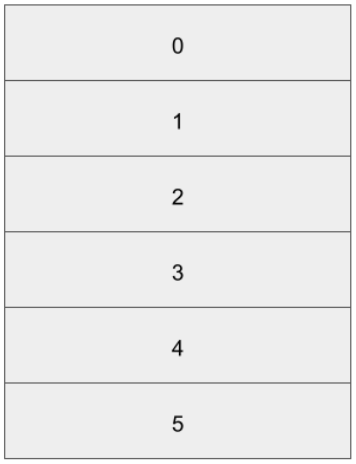

Fig. 2. The distribution is done row-wise for the segments. The figure is for a 6 processor system, where the numbers indicate the expected rank of the processes.

4 Algorithm and Implementation

The essence of the algorithm can be boiled down to 3 steps:

1. Divide data: Root process divides the file data and sends it to other processes
2. Search for query terms: Each process performs a linear search throughout the segment of the corpus it has received
3. Merging data: The data is received by the root process and it is decided if the query is found successfully or not

In the division step, the root process gets the file size, broadcasts the chunk size and filesize to each process. Each broadcast costs $(T_s + T_w * 4)log(p)$ time. Then each process seeks to a position of $(rank + 1) * chunksize$. This incurs an average linear cost of n. There is a constant seeking of a few bytes to ensure the whole word is included. This would not exceed MAX_TERM_LENGTH (=100). Then the pointer seeks to final and initial mark to calculate the new segment size, and sends messages to the next ranked process to adjust the initial mark. This incurs a cost of n/p and SINGLESENDCOST. Then the file data is tokenized in the form of array of strings. This costs n/p time. Then query is input in the root process and tokenized. In tokenization, it is detected if the query is of AND type or OR type. This takes m cost. The root process then broadcasts query word count and the qwords array to all the processes. This takes $(m+1)*log(p)$ time. In the searching step, all the processes (including the root) count the number of lines in each segment corpus and sends this offset to the subsequent processes through MPI_Scan (Prefix-sum type). This costs $(T_s + T_w * 4)log(p)$ time (assuming hypercube mapping). Then we search through the respective segment corpus for the query words by iterating through the file and the query arrays. This would cost us $m * n/p$. Then for the merging step, each process sends the word and line location as an array. This costs $(T_s + T_w * m)log(p)$ each. Then the root process goes through all the word Locations and line locations and finds the first occurrence of the word. This costs us $m * p$.

On the basis of these observations, the parallel run-time can be estimated as follows. (T_s and T_w are startup time and time to transfer one word respectively).

$$T_{\text{division}} = 2*(T_s + T_w * 4) * log(p) + n/p + n/p + n/p + m + (m+1) * log(p)$$

$$T_{\text{searching}} = (T_s + T_w * 4) * log(p) + m * n/p$$

$$T_{\text{merging}} = 2*(T_s + T_w * m)log(p) + m * p$$

Adding these components,

$$T_P = 2(T_s{+}4T_w4)log(p){+}3n/p{+}m{+}(m{+}1)log(p){+}(T_s{+}4T_w4)log(p){+}mn/p{+}2(T_s{+}mT_w)log(p){+}mp \quad (3)$$

5 Analysis

Theoretically,

$$\text{Speedup} = \frac{T_{series}}{T_p} = \frac{mn}{2(T_s + 4T_w)log(p) + 3n/p + m + (m+1)log(p) + (T_s + 4T_w)log(p) + mn/p + 2(T_s + mT_w)log(p) + mp}$$

$$\text{Efficiency} = \frac{T_{series}}{pT_p} = \frac{mn}{2(T_s + 4T_w)plog(p) + 3n + mp + p(m+1)log(p) + p(T_s + 4T_w4)log(p) + mn + 2p(T_s + mT_w)log(p) + mp^2}$$

$$\text{Cost} = pT_p = 2(T_s + 4T_w)plog(p) + 3n + mp + p(m+1)log(p) + p(T_s + 4T_w)log(p) + mn + 2p(T_s + mT_w)log(p) + mp^2$$

$$\text{Overhead} = pT_p - T_{series} = 2(T_s + 4T_w)plog(p) + 3n + mp + p(m+1)log(p) + p(T_s + 4T_w)log(p) + 2p(T_s + mT_w)log(p) + mp^2$$

6 Graph Analysis

To perform graphical analysis, the algorithm was implemented in C using OpenMPI and ran for the query ”Lorem AND hello” on a 4-core machine with varying file sizes.

p	n = 1000	n = 10000	n = 100000	n = 200000	n = 500000
1	0.004	0.012	0.061	0.102	0.157
2	0.006	0.014	0.027	0.040	0.187
3	0.015	0.007	0.021	0.036	0.146
4	0.007	0.007	0.013	0.025	0.151
5	0.010	0.008	0.022	0.036	0.067
6	0.012	0.029	0.025	0.033	0.063

Table 1. Execution Time in seconds for different values of p.

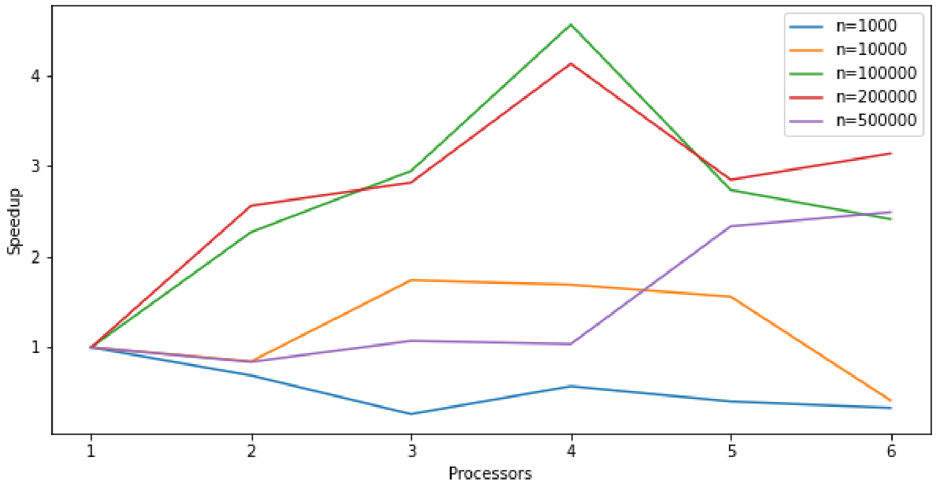

Fig. 3. Speedup plotted for different values of p.

p	n = 1000	n = 10000	n = 100000	n = 200000	n = 500000
1	1.000	1.000	1.000	1.000	1.000
2	0.690	0.845	2.269	2.560	0.839
3	0.265	1.739	2.943	2.815	1.071
4	0.571	1.690	4.556	4.126	1.037
5	0.404	1.558	2.733	2.846	2.334
6	0.333	0.415	2.412	3.135	2.490

Table 2. Speedup for different values of p.

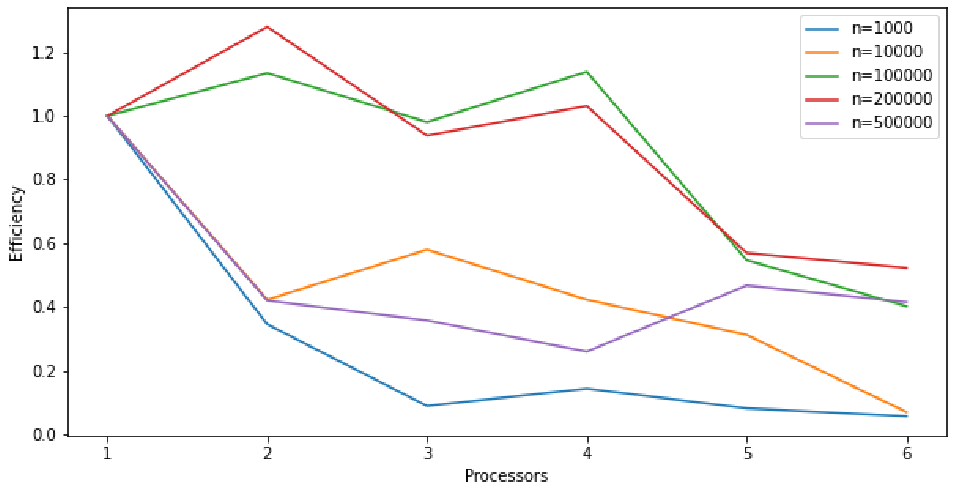

Fig. 4. Efficiency plotted for different values of p.

p	n = 1000	n = 10000	n = 100000	n = 200000	n = 500000
1	1.000	1.000	1.000	1.000	1.000
2	0.345	0.423	1.135	1.280	0.420
3	0.088	0.580	0.981	0.938	0.357
4	0.143	0.423	1.139	1.031	0.259
5	0.081	0.312	0.547	0.569	0.467
6	0.056	0.069	0.402	0.523	0.415

Table 3. Efficiency for different values of p.

p	n = 1000	n = 10000	n = 100000	n = 200000	n = 500000
1	0.004	0.012	0.061	0.102	0.157
2	0.012	0.028	0.054	0.080	0.373
3	0.045	0.021	0.063	0.109	0.439
4	0.028	0.028	0.054	0.099	0.604
5	0.050	0.038	0.112	0.179	0.336
6	0.072	0.173	0.153	0.195	0.377

Table 4. Cost for different values of p.

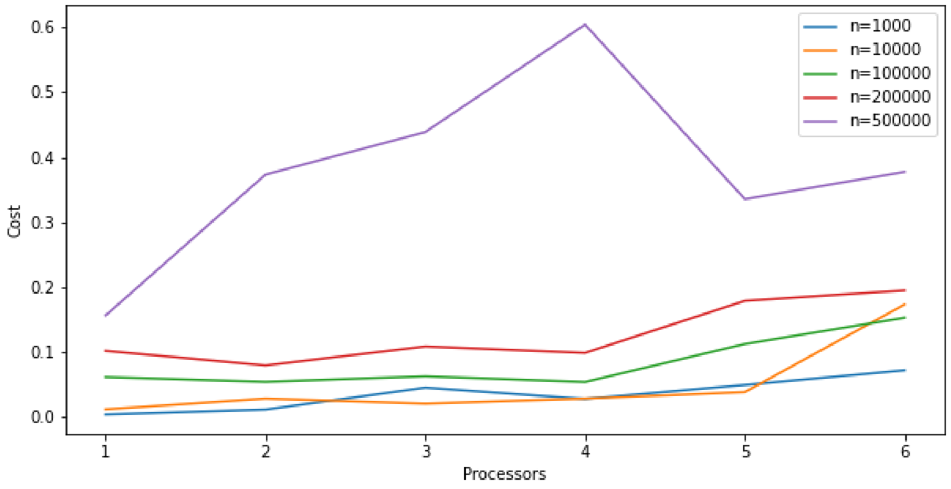

Fig. 5. Cost plotted for different values of p.