PARALLEL COMPUTING ASSIGNMENT 2

Sanjay Srivastav 2015A7PS0102P
Samip Jasani 2015A7PS0127P

Question-1: SEMI-ORDERED MATRIX

Algorithm-1 A: Iterative tri-partitioned Search

For this question, the first idea was to use the basic divide and conquer template of the semi-ordered matrix search i.e. by dividing the entire matrix into partitions based on the middle element of the matrix. For this implementation, the idea was to maintain a stack and dump the sub-matrices thrice as that of the current number of nodes in the stack each time. But since this was to be done in parallel, we needed to think a way of accessing the array in parallel. So we maintained two stacks , switching between them in subsequent iterations. The stack maintained had the corners to be worked on , and are processed till the key is not found. Since in each iteration the size of the stack is known beforehand (thrice as the previous size), it is easy to access the stack in parallel, and also each element in the current stack need to place it's three sub-children in 3*i ,3*i+1 and 3*i+2 locations of the stack array and hence the stack reads and stack writes could be parallelized. But this way of maintaining stacks and traversing it each time was time consuming and was taking an average query time of 300 milliseconds when run over a 10^4 x 10^4 matrix.

The pseudo code of the above goes as follows:

```
main()
        mat = readMatrix() //mat is the matrix.
        read(key) //key is the key to be searched.
        initialize(*arr[2]) // arr is a dual stack.
        size=1
        ind =0 // ind to shift between two stacks arr[0] and arr[1]
        do{
                ind = 1- ind
                #pragma omp for
                for(i= 0 to size-1)
                        size*=3
                        search(mat,arr,key,i)
        until(answer not found)

search (mat , stack, key , stksz)
        //define corner using arr in and stksz
        corner = arr[ind][stksz]
```

```
//Define middle using corner as
middle  = (corner.fromx + corner.tox) /2 , (corner.fromy + corner.toy) /2
if (corners.fromx > corners.tox  or corners.fromy > corners.toy)
        return
if (key =middle of mat)
        return answer found
if ( key > middle of mat )
        Define three quadrants q1,q3,q4 for right top, left bottom and right
bottom of the middle element respectively
        arr[1-ind][3*stksz] =  q1
        arr[1-ind][3*stksz+1] =q3
        arr[1-ind][3*stksz+2] = q4
        return
if ( key < middle of mat)
        Define three quadrants q1,q3,q2 for right top, left bottom and left top  of
the middle element respectively
        arr[1-ind][3*stksz] =  q1
        arr[1-ind][3*stksz+1] =q3
        arr[1-ind][3*stksz+2] = q2
        return
```

Algorithm-1 B: Recursive tri-partitioned Search

There were many comparisons done in this case to check if the element in the array was a genuine corner. Also to avoid the overhead of maintaining the large array, in this version the code was modified by recursively calling the three sub-problems by assigning tasks to unwind each recursion. The code implemented for this version was working parallelly but wasn't optimal as the search space wasn't reduced greatly. Also the sequential algorithm takes **O(n^1.58)** and hence wasn't optimal.

The pseudo code for this goes as follows:

```
main()
        mat = readMatrix() //mat is the matrix.
        read(key) //key is the key to be searched.
        search(mat, corners ,key)  // define corners as the four corners of matrix mat

search(mat,corners,key)
        //Define middle using corner as
        middle  = (corner.fromx + corner.tox) /2 , (corner.fromy + corner.toy) /2
        if (corners.fromx > corners.tox  or corners.fromy > corners.toy)
                return
```

```
if (key =middle of mat)
        return answer found
if ( key > middle of mat )
        Define three quadrants q1,q3,q4 for right top, left bottom and right
bottom of the middle element respectively
        #pragma omp task
                search(mat,q1,key)
        #pragma omp task
                search(mat,q3,key)
        #pragma omp task
                search(mat,q4,key)
if ( key > middle of mat )
        Define three quadrants q1,q3,q2 for right top, left bottom and left top of
the middle element respectively (define them inside the pragmas)
        #pragma omp task
                search(mat,q1,key)
        #pragma omp task
                search(mat,q3,key)
        #pragma omp task
                search(mat,q2,key)
```

The above two algorithms were clearly inefficient as the overhead of stack was there in both cases. These two codes are provided in the folder ("/SOM_0"). In all the above cases, there was an answer flag which when set, triggers every other process to halt there processing. This was done by making them to check the flag before calling any searches.

Algorithm 2: Searching on Smaller sub-Matrices

In order to decrease the search space and make the algorithm to work on smaller search space, the design was modified. In this (provided in "./SOM/SOM_1"), our idea was to use an O(n) algorithm and parallelize it. In order to do so, our first idea was to split the matrix into multiple smaller sub-matrices and make a check in each portion for the top-left and bottom-right corners with the key, confirm whether key may occur in that region and then proceed. For each smaller matrix satisfying this property, an O(n) search was done in parallel and since most of these smaller matrices are eliminated, this was highly parallel and results were 1000 times better than the above two methods.

The pseudo code for the same is as follows:

```
main()
        mat = readMatrix() //mat is the matrix.
        read(key) //key is the key to be searched.
```

```
width = power of 2 which is a multiple of size of matrix
#pragma omp parallel for
for(i= 0 to width-1)
        for (j=0 to width-1)
                if (key in range between mat's left top and right bottom corners)
                        search(mat, corners ,key) //corners are for this portion
```

search(mat,corners,key)

```
        i= corners.fromx
         j = corners.toy;
        while  (i <= corners.tox and j >= corners.fromy)
                if (mat[i][j]==key)
                        return key found
                if (mat[i][j] > key)
                        j--
                else
                        i++
        return
```

The width considered, optimally should be in such a way that its neither too small nor too large. This is a necessity because had the width been too small, number of comparisons increases and had it been too large it will hamper parallelism (also too large partitions may be very insignificant as the O(n) algorithm should run on it which might take time). In order to do this, our approach was to divide the matrix size by 2 till its not divisible and assume that to be the base case and proceed which might not always be optimal but will be good assumption to proceed.

Algorithm-3: bi-partitioned Search

Even though the above method was good, we tried another approach where we were dividing the matrix into two parts based on an element in the middle row, instead of dividing into 3 sub parts or doing an entire matrix traversal for comparing corner elements of smaller sub-matrices. The motivation behind doing this was the fact that, in the first two methods we were only comparing the key with one of the element. But if we compare it with an entire row and eliminate the possibilities of a major chunk of the matrix, it might improve performance. That is, if the key can be found to lie between two elements in a row, then the likeliness of that element occuring in the matrix is reduced to search only the two portions of the entire matrix, in the bottom left of it and top right of it. This is because the matrix is semi sorted. If the search end up to be at extremes then we will have to search only one half of the bigger matrix, top half or bottom half. For the base case though, a search similar to the third method (algorithm-2) was done.

This algorithm was also giving performance similar to the third approach and is far better than the first two approaches taken. The results observed gave a significant rise in the performance compared to first two but were slightly off when compared with third. This algorithm takes O(log n) for a binary search in a row of the matrix and divides it into almost half size as that of the previous one in each iteration. The total time complexity of the algorithm is determined by the relation :
T(n) = 2T(n/2) + O(log n). So the time complexity in the worst case is O(n).

The pseudo code of the same goes as follows:

```
main()
        mat = readMatrix() //mat is the matrix.
        read(key) //key is the key to be searched.
        search(mat, corners ,key)  // define corners as the four corners of matrix mat

search(mat,corners,key)
        Define middle by applying binary Sort on the middle row.
        middle.x,middle.y = binSearch((corner.fromx + corner.tox) /2), (corner.fromy + corner.toy) /2
        if (key =middle of mat)
                return answer found
        if (matrix size in any one direction is less than 4)
                Call search2(mat,corners,key)  //base case
        if (middle.y < corners.fromy)
                define quadrant q1 as the top half matrix.
                #pragma omp task
                search(mat,q1,key)
        else if (middle.y > corners.toy)
                define quadrant q2 as the bottom half matrix.
                #pragma omp task
                search(mat,q2,key)
        else
                define quadrant q3 and quadrant q4 as the bottom left and top right
        portions respectively
                #pragma omp task
                search(mat,q3,key)
                #pragma omp task
                search(mat,q4,key)

search2(mat,corners,key)
        i= corners.fromx
         j = corners.toy;
        while  (i <= corners.tox and j >= corners.fromy)
```

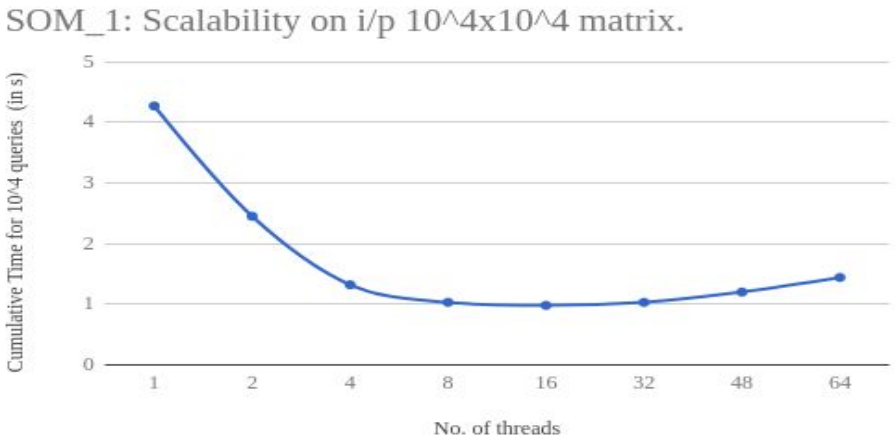
```
        if (mat[i][j]==key)
                return key found
        if (mat[i][j] > key)
                j--
        else
                i++
return
```

The above algorithm despite dividing into two partitions, was giving lesser performance compared to that of second algorithm which can be attributed to the fact that the key crossing the extreme might be larger and hence the binary search which be on same size again and hence more comparisons.

Among all the four approaches taken, the third and fourth approaches gave good results with the third method (**Algorithm-2**) being the most optimal in terms of complexity. This can be attributed to the fact that, in general, the search eliminates most cases and proceeds only for some. This was also **highly scalable** as the results varied drastically when ran on multiple cores. The reason for it is as the number of threads increased number of simultaneously working power was significant as there was scope to parallelize which was less in other cases.

The output files are attached in the directories. For example, in SOM/SOM_1(Final) the file outputall has the final output when ran on multiple cores with the number of processes varying from 1, 2, 4, 8, 16, 32, 48, 64, 96. The performance measures were checked for inputs varying from matrix of size 10^2 x 10^2 , 10^3 x 10^3 and 10^4 x 10^4 . Also each code is ran 10^3, 10^4, 10^5 times respectively for the three input matrices, with randomly generated integer being passed for each query. Thus the time represented there is the total time for all the queries together.

PERFORMANCE GRAPH:



Question 2: Document Frequency

Algorithm -1: Local Map, Global Hash table and Quick Sort. (FINAL)

In this algorithm, we first traversed the entire directory structure, and in the process of scanning the directories, each thread makes tasks for each file and proceeds to read the next directory in the current directory. When all the files are done, this will wait for the tasks to be complete and proceed. Each task here checks if the file to be read is a directory or a regular file and if it is a directory it recursively calls the same procedure. If it is a file it calls a function that reads the entire file and fill a **local map** having occurrences of each word and it also adds it to a global **Hash Table (Separate Chaining).** After the global hash table is filled, it is then passed through a quick sort function which returns a sorted list based on the count of the number of files in which that word occurs. This is the outline followed in all the algorithms implemented but the global structure and the sorting technique varies across the algorithms.

In order to parallelize this code, the omp tasks were used to read the directories and files across a directory. Each file is read by a single file at a time, but multiple files might be read in parallel. Thus each thread(task) maintains its own local map structure, but the global structure (hash table) might be accessed and mutated in parallel. Thus we used locks on the hash entries to prevent multiple threads accessing same chain. This global structure is only accessed if the word is not already present in the local map.

Now that the hash table is filled, we needed a way to serialize the hash table's chains to sort it. To do this we will need to know the size of the total entries in the hash table. This was done by iterating the table, and retrieving the size info already stored. To parallelize this serialization of making the hash table into a bigger array, we stored the cumulative frequency of the words at each hash table entry which thus stores two integers, a size and a cumulative frequency and a pointer to the head of the chain. This was done so that an omp parallel for can be used with each thread knowing how many entries to work on beforehand so that array accesses can be parallelized without any locking. Thus this array will be parallelized accessed and mutated. This is now sent to the quick sort function, which uses tasks to parallelize the recursions. Thus the top k entries are retrieved.

Also we used a stopwords file so that entries in a file that are generally common which might occur with a lot of frequency and may give unfruitful results are eliminated. This is done before a word is added to the local map. It is first checked where stopwords has this and then decides and proceeds. Stopwords is a hash table.

PSEUDO CODE:

main()

```
makestopwords() //makes a hash table of stopwords
Initialize locks[hash_size] //Initialize locks for hash entries
createEmptyHashTable()
#pragma omp parallel single
filetreewalk(root)
fillCumFreq() //gets cumulative freq of each element in the hash table
fillarray()
quicksort() //sorts the elements in parallel.
Return top k elements
```

filetreewalk(root)

```
if(root is empty)
        Throw error
while(file in root dir)
        #pragma omp task
        if (file is a directory)
                filetreewalk(file)
        else
                fillHashTable(file)
#pragma omp taskwait
close(root)
```

fillHashTable(file)

```
Open the file
while (words in file)
        h = hash(word)
        if (checkstopword(word))
                Contine
        if (word in localMap)
                Continue
        Add word to localMap
        omp_set_lock(h)
        insertword(word) // inserts a word to the global hash table
        omp_unset_lock(h)
```

fillarray()

```
#pragma omp parallel for
        for (i = 0 to M-1)
                temp = hash table head
                for (j = 0 to ht[i].size-1)
```

```
arr[ht[i].cf-ht[i].size+j] = temp
temp=temp.next
```

For the sake of hashing we used "**djb2**" hash function with 50000 to be the size of hash table. This algorithm performed really well when ran on the dataset /home/paracom/shared in the server with an average time of around 20 seconds for the entire process. Also the algorithm was scalable as it improve immensely on increasing the number of cores from 1 to 4 till 32, but it started to decline when the cores increased from 32 to 48 to 64 to 96. This can be attributed to the fact that when the directory has lesser files, all the threads might not be active and they might wait for other tasks to complete. This might be the reason why the performance decreased for larger number of cores. The algorithm is presented in ("DF/DF_MHQ") directory along with some outputs.

Algorithm -2: **Local Map, Global Trie and Heap Sort.**

In this implementation, in order the check the performance, a different data structure was considered. Here a trie was used as a global structure. Sorting done here is heap sort primarily because, the trie traversal can simultaneously add elements to the k-heap and since this traversal can't handle a simultaneous update to an array, this was done by using a heap which nonetheless couldn't be handled parallely as that of hash table serialization, but we can get the result immediately instead of sorting later. This method also prevents the overhead of maintaining the entire array by allowing us to maintain only k elements to be retrieved.

The file walk is similar to previous code except that this time a call is done to add to trie rather than to hash table. The trie entry insertion is done in a critical section, to avoid simultaneous update on any count. Once the trie is populated, it is then traversed and a min-heap will store a word, if its count is greater than the minimum element of the heap. In order to parallelize this, omp parallel for is used at each stage of the trie for its children, and if the count of a word is greater than the confidence variable now, we lock the entire heap and pass it to heapify(). Once this is done, confidence is reset to the heap root's count. Heapify function is a recursive function which is not done in parallel. Finally heap sort function is called which sorts the heap by iteratively making heap size 1 smaller than last iteration.

PSEUDO CODE:

main()

```
makestopwords()  //make a trie of stopwords
Initialize heaplock an trielocks
Initialize heap
Initialize trie root
#pragma parallel single
        filetreewalk()
traverse(trie)
heapSort()
```

filetreewalk(root)

```
if(root is empty)
        Throw error
while(file in root dir)
        #pragma task
        if (file is a directory)
                filetreewalk(file)
        else
                fillDict(file)
#pragma omp taskwait
close(root)
```

fillDict(file)

```
Open the file
while (words in file)
        if (checkstopword(word))
                Contine
        if (word in localMap)
                Continue
        Add word to localMap
        insertword(word) // inserts a word to the global trie
#pragma omp taskwait
```

insertword(word)

```
for (i =0 to  wordlength-1)
        #pragma critical
        if no children at this index, create a node
        Move down the trie
Make the end of word true
#pragma atomic
        Count++
```

traverse(trie)

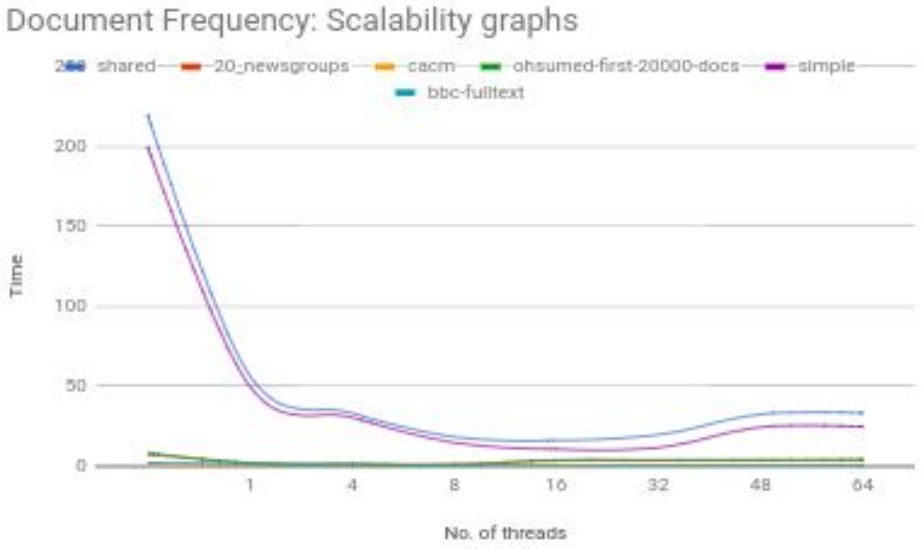
```
if (trie.count >conf)
        omp_set_lock(heap)
                Update global heap's root
                if (trie.count >  heap.arr[0].count)
                        minHeapify()
        omp_unset_lock(heap)
        Conf =new top of heap
#pragma omp for
for (i=0 to 25)
        If node exists at ith location,
                traverse(child)
```

This algorithm is also a very efficient algorithm with performance similar to that of the hash table. This is provided in ("./DF/DF_MTH") directory and output file is in the same directory.

Also in another implementation we tried implementing hash with heap instead of hash with quicksort. This wasn't completely working but was almost implemented and is provided in ("DF/DF_MHH").

Among all Map Hash QSORT was running most efficiently with the Map Trie Heap just behind it with minimal variations in time. Scalability wise both are scalable as multiple iterations of the code where different number of threads impacted the performance significantly.

PERFORMANCE GRAPH:



These measures were done for the shared directory as a whole and also the sub directories independently and a peak performance of 15.81 seconds was observed when run on 32 threads.

For both the questions the peak performance was observed when ran with 16 threads. In second question though 32 gave better result in one run but overall, 16 and 32 threads were giving peak utilization.