Purdue University, West Lafayette

Fall 2022

ECE56300 – Programming Parallel Machines

Default Large Course Project

Submission Date- 12/14/2022

Submitted by
Nahian Ibn Hasan
PUID: 0032764564
Email: hasan34@purdue.edu

Implementation Strategy of the Map-Reducer Problem

OMP Implementation:

Overview

Steps:

1. Assume the thread availability are as follows-
    a) Total available threads = N
    b) Total number of Reducer threads = R
    c) Total number of mapper threads = total number of files = M
    d) M+R <= N.
    e) The reducer threads are spawned first and they keep waiting until they receive data from mapper threads. That means there are (N-R) threads available for the mapper threads. If N-R < M, some of the mapper threads wait for any thread availability. If N-R>=M, all the mapper threads start working immediately on separate files.
2. The mapper threads read line by line from the text files. A custom code has been implemented which reads through each sentence and separate out the words. A 8 bit hash code is generated for each word. A reducer ID is generated from the hash code. There are R reducer threads, hence, there are R unique reducer IDs (0,1,2,…,R-1).
3. Based on the reducer ID the word is placed on global shared queues. There are R separate queues.
4. The reducer threads gets word from the queues and then compares with the word with those that the thread has already found so far and increases the count accordingly. If there is no match, it then creates a new entry for that newly encountered word and starts counting.
5. After the mapper have finished reading the lines, they send signals to all the mapper threads. The mapper threads get that signal and once the ques are empty they start writing the data to separate files.

Hash Code Generation

Steps:

1. Assume, we want to generate a hash code for the word “Project”. There are 7 characters in this word. The corresponding 8 bit ascii codes are as follows-

Character	ASCII Code (8 bit)
P	01010000
r	01110010
o	01101111
j	01101010
e	01100101
c	01100011
t	01110100

2. The 8 bit ascii codes are XOR’ed with each other sequentially. For example, the XOR’ed result is as follows-

Character	ASCII Code (8 bit)	XOR result with previous character
P	01010000	01010000
r	01110010	00100010
o	01101111	01001101
j	01101010	100110
e	01100101	01000011
c	01100011	00100000
t	01110100	01010100

3. So, the final XOR'ed version of the word "Project" is 01010100.
4. The decimal equivalent of this number is 84. Therefore, the hash code for the word "Project" is 84. Since, we have considered 8 bit of hash code, there are 128 different hash code generation possible for every word in our project.

Reducer ID Generation

The reducer id is generated by the modulus operation mod(hash_code,R). For example, if R=8, mod(84,8) = 4. That means the word belongs to reducer 5, whose ID is 4. It is evident from the description that there are R different reducer ID possible for all the words. No word will have different reducer ID or hash code. All the occurrences of a word will generate the same hash code and will be sent to the same reducer. Therefore, al the files that are being written to by the reducers will include unique words.

Algorithm for OMP implementation

Step 1: Initialize global Array of Structure Queues for each reducer. The size of the queue is R*T*W. R=number of reducer threads. T=size of the queue,W=maximum length of each entry in the queue (i.e. word length). Initially, the array is completely empty, hence, the QUEUE_STATUS flag is 0 for every R queue.

Step 2: Spawn the R threads which are reducer queues. They keep running until QUEUE_STATUS flag is 1. They initialize local arrays to store unique word and their respective counts, each of length X.

Step 3: Spawn the mapper threads. The M mapper threads work on a single file.

Step 4: Each mapper thread generate a hash code and consequently a reducer ID for each word. The mapper then puts the word in the corresponding reducer queue. Since, all the threads are trying to put words to the queue, atomicity is maintained and proper locks are setup.

Step 5: Once the queue is full, the QUEUE_STATUS is set to 1 for that reducer ID. This is done by a arbitrary mapper thread. All the threads encounter this and they keep waiting. On the other hand, the reducer threads also sees the QUEUE_STATUS flag, then they collect words from the queue and keep counting. They store the words and their counts in their local arrays. If the local array lengths X is filled up, the reducer threads reallocates some more memory to the arrays with an increment size of Y. However, once the reducer has collected and counted the words in the queue, it resets the queues for the mapper and sets the QUEUE_STATUS flag back to 0.

Step 6: The mapper flags again encounters this QUEUE_STATUS flag is 0. Hence, they start putting words into the respective queue again. The process repeats from step 4.

Step 7: Once the mappers have read the whole file they set the respective MAPPER_STOP flag to 1.

Step 8: Each of the reducer threads checks whether all the MAPPER_STOP flags are set to 1. Once it's ensured, they start writing the unique words and their corresponding counts to separate files.

Results:
For this implementation, initially, the variables are set as follows-

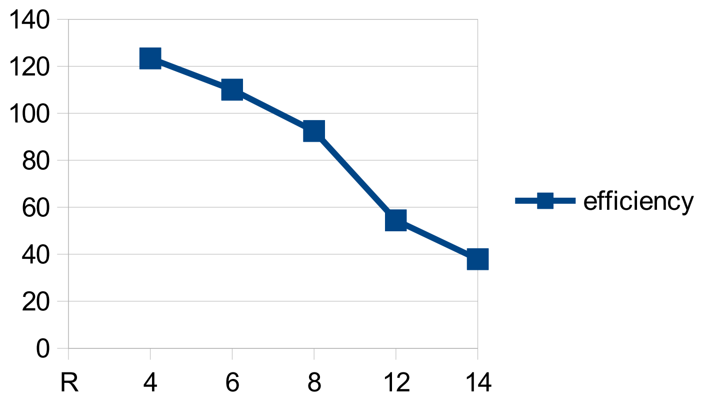
- R = number of reducer threads = 8
- T = number of total threads = 16
- W = maximum length of each word possible = 100
- X = length of reducer local arrays to store unique word and their respective counts = 1000
- Y = increment step of reducer local arrays to store unique word and their respective counts = 1000
- M = number of mapper threads = number of files = 15

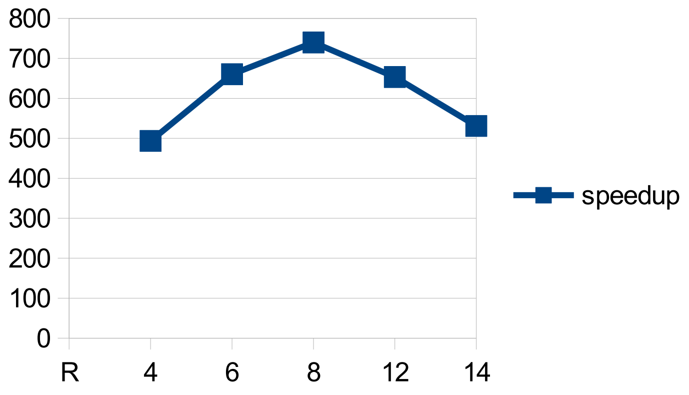
For comparison purpose, we change different parameters and the timing results are provided below.

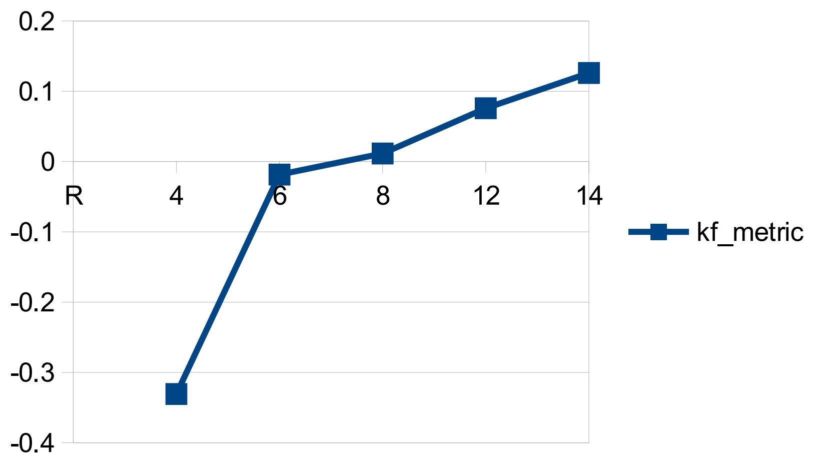
Baseline Implementation:
For comparison purpose, we have implemented a sequential code with total number of threads equal to 1. The mapper(single mapper) first reads files one by one and stores the words in separate queues. Once, mapper's job is finished, the reducer (single reducer) starts working. This implementation requires approximately **Ts = 20.467029 s**

OMP Timing w.r.t. Reducer threads (R)

T = number of total threads	R = number of reducer threads	Required Time, Tp (s)	Speedup = Ts/Tp, (%)	Efficiency = (Ts/(Tp*R)), (%)
16	4	4.149030	493.3	123.33
	6	3.100097	660.21	110.04
	8	2.767851	739.46	92.44
	12	3.133392	653.2	54.44
	14	3.856124	530.77	37.92







Number of words found in both baseline and OMP implementation

File Name	Number of Total Words
1.txt	58,166
2.txt	26,393
3.txt	433,621
4.txt	251,836
5.txt	668,221
6.txt	296,832
7.txt	11,629
8.txt	27,481
9.txt	439,016
10.txt	148,205
11.txt	74,565
12.txt	25,948
13.txt	111,343
14.txt	159,402
15.txt	45,471

Reducer Work Load When R=8,T=16

Reducer ID	Number of Unique Words	Number of Total Words
1	32,108	262,907
2	41,541	472,877
3	32,002	221,026
4	58,798	262,782
5	36,896	488,129
6	32,866	253,624
7	18,828	157,739
8	22,563	336,414

Comments:

Result:

It seems if the percentage of reducer threads is too high, there are less number of mapper threads available, on the other hand, if the percentage of mapper threads is too low, there are less number of mappers available. In both cases, the performance is hurt. If there is a balanced number of threads available for both mappers and reducers (i.e. R=8, M=8), the optimum performance is achieved. The speedup increases w.r.t number of cores and the optimum speedup is possible when there is a balance between the number of reader and mapper threads. Efficiency decreases w.r.t. the number of cores. Also, the Karp-Flatt metric is negatives at lower number of cores because of relatively high amount of memory and cache running on multiple cores/processors, leading to less work being done handling memory page faults and cache misses.

Pros:

The reducer threads maintain their own local unique word array and counting array. The reducer threads also collect data from unique queues. Hence, the reducers should not clash in collecting words or incrementing the counting or writing the data to files. It is the mappers who should be careful in writing data to the shared queues. The queues are shared among the mappers in terms of writing. On the other hand, the same queues are private among the reducers in terms of reading. It's good thing that the reducers don't have to compete while reducing.

Cons:

The mappers need to be careful in writing data to the queues. Although each word is written to different indexes of the array, but the array index increment need to be done one by one, hence locks are mandatory. If the queue of any reducer becomes full (i.e. QUEUE_STATUS=1), all the threads keep waiting until that single queue is free. They could have continued to fill up other queues.

Future Direction:

1. In the next implementation, for each queue a backup queue is also prepared. This backup queue is private to each mapper thread only. The idea is that once the main queue is filled up, the mappers start filling-up the back-up queue. In the mean time, reducers collect data from the main queue and reduces the words and resets the main queue. When the mappers restarts mapping word, they first reload the words from the backup queue to the main queue and then starts filling up with the newly read word. In summary, when the mappers start refilling the queues, they empties the backup queues first. In this way, the mappers don't have to wait at all. However, one has to make sure that the backup queues are not overloaded. If the backup queues also overload, then the mappers should wait until both main queue or backup queues are emptied. This is a more complex code that is planned for future version of this implementation.

2. Words are classified purely based on ASCII code. Hence, "Project", "project", "PROJECT", "PROJECT ", "PROJECT.", "project," - all are considered different words. Hence, they are mapped to different reducers. Therefore, the word parsing routine could be modified for better and more meaningful reduction.

3. Also, some reducers got higher number of words to reduce. One way of reducing this bias is to increase the number of reducers or increasing the number of hash-code bits which will result in the variation of reducer ids.

MPI Implementation

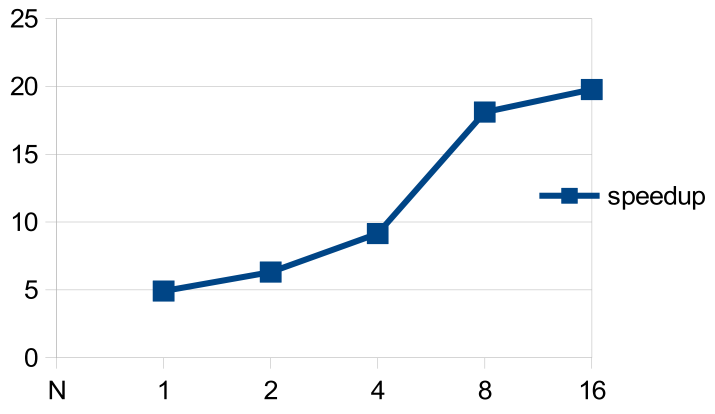
Overview:

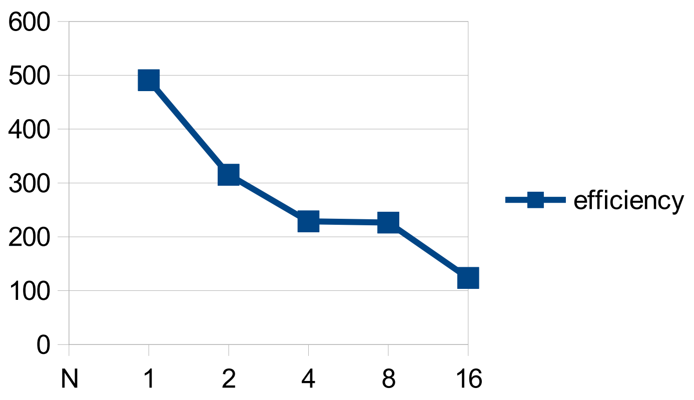
Steps:

1. Assume there are N nodes and each node can support upto T threads. Hence, there are in total N*T threads available. In the MPI implementation, each thread has a unique rank. The rank range from (0,1,2,…,N*T-1).
2. Let's say there are M files to be reduced. We want to assign a specific number of files to each thread. Hence, each node will get at least **ceil(N/M)** files to reduce. However, of course, this might cause a load imbalance issue. We will encounter it later.
3. Once the MPI system spawns the threads, the threads which have a rank of **mod(rank,T)==0,** will be the master thread for that node. These master threads distributes the files to be read to the mapper threads which are inside that node.
4. Also, the master threads tells the respective reducer threads to reduce the words with the same strategy described in OMP implementation section.
5. This MPI implementation runs the same algorithm that OMP section describes, the only difference is that each node will have its own mapper thread and reducer thread and a list of files which should be reduced.
6. **Inside each MPI node:**
    a) The mappers read lines from the files, generate hash-codes for each word, assign reducer ID, and put the word in MPI custom data lists. Each list or queue are assigned to each reducer ID. The lists are then sent to each reducer. The reducer receives a separate list from each mapper and reduces the words. In the mean time, the mappers generate another batch of word for each queue. Once the reduces have finished their work on the previous batch, it requests for a new batch which the mappers have already prepared for them. The difference between the implementation of OMP and MPI is that in the OMP, the words were put in a shared queues of which both the mappers and reducers had access to. But here, in MPI, the data are sent to the respective reducers with the MPI interface. Hence, there is an extra communication cost.

Results:

N = number of Nodes	Required Time, Tp (s)	Speedup = Ts/Tp, (%)	Efficiency = (Ts/(Tp*R)), (%)
1	4.16923	4.91	490.91
2	3.246027	6.31	315.27
4	2.237752	9.15	228.66
8	1.130372	18.11	226.34
16	1.0356221	19.77	123.52





Discussion:
As the number of processors increase, there are less amount of work to be done by each processor. Therefore, the efficiency decreases. The speedup increases w.r.t. the number of processors.
Pros:
The implementation is pretty straight-forward. There is no complicated work distribution. With minimal change to the OMP code, the MPI code can be implemented.
Cons:
The initial distribution of work is not uniform across nodes. A future implementation could be to statically define the files each node will process, but this could lead to some nodes getting many big files and other nodes getting many small files. Instead, each node should request a file from a master node which will either send a filename back to the node or an "all done" that indicates that all files have been or are being processed.

Critical Thinking:
In OMP implementation, we are strictly bounded by the number of threads available. Therefore, there is a very high probability that someone would face the shortage of threads compared to the number of mapper threads and reducer threads. There, when we increase the number of reducer threads, the speedup doesn't increase with respect to it, because, at some point we start to squeeze the number the mapper threads, hence more time are needed by the less number of mappers to read relatively higher number of files. On the other hand, in MPI, we barely face this issue, because, compared to our problem size in this project, at least 2 nodes are more than enough to accommodate a reasonable and healthy number of reducer and mapper threads. We don't have to overly squeeze any of them. Hence, the speedup is continuously increasing as we are seeing in the result.

NB:
The codes are attached with this report. Please go through the script called 'cluster_run.sh'. There someone can run specific version of the implementations, serial, OMP or MPI versions. For the specific versions, please see the attached directory for the source code.