CP631A - Advanced Parallel Programming

Final Project

Finding jumbled words using Parallel Programming and Quicksort

Shaurya Guliani	Sachwin Kohli	Shoieb Ur Rahman Thayal
235827170	235830680	235846310

The project aims to unjumble a vast list of jumbled words by finding them in a dictionary using parallel programming and quicksort. The basic serial algorithm consists of using quicksort to sort each word in the dictionary and the jumbled list and then find each sorted word in the jumble list by comparing it in the sorted dictionary. Whenever a match is found, the original word in the dictionary represents the unjumbled form of the original jumbled word. The results of the serial program are stored in two separate text files – one consists of all jumbled words and their unjumbled forms (a jumbled word can have multiple unjumbled forms) and the other text file consists of all jumbled words whose unjumbled form was not found. This algorithm is parallelized to make its runtime more optimal using a load balancing technique.

Both the programs - serial and parallel – use a quicksort algorithm which is defined using two functions:

Partition function:

The **partition** function is a helper function for the quicksort algorithm. It takes an array of characters, a low index **l**, and a high index **h,** as input and partitions the array based on a pivot element (the last element of the array).

Quicksort function:

The **quicksort** function is a recursive function that sorts an array of characters using the quicksort algorithm. It takes the array, a starting index **f**, and an ending index **l** as input and then uses the partition function to get the position of the element which splits the array such that all elements after that element are greater than it and all those which are before it is smaller.

Serial Program

The serial algorithm performs the following tasks:

- **Variable Declaration**: The process begins with the declaration of variables and file pointers.
- **File Operations**: This includes opening of input files “words.txt” and “jumbled.txt”, reading the count of words and jumbled words from the files, and closing of input file pointers. It also involves opening of output files “results.txt” and “WordNotFound.txt”.
- **Memory Allocation**: Memory is allocated for an array of strings (word_list) and two-dimensional arrays (jumbled_list_orignal and jumbled_list_sort). Words from “words.txt” are loaded into word_list. Jumbled words from “jumbled.txt” are read and sorted.
- **Word Matching**: The sorted jumbled words are matched with the sorted list of words and the results are written to output files.
- **Performance Measurement**: The time taken for the operation is measured.
- **Output Generation**: The total run time and status messages are printed.
- **Resource Cleanup**: The output file pointers are closed and the allocated memory is freed.

Output files:

results.txt contains all the matched pairs of jumbled words and their unscrambled counterparts while **WordNotFound.txt** consists of all jumbled words that were not found in the dictionary. Samples of both the text files are present below.

Results.txt (left) and WordNotFound.txt (right)

Parallel Program

The parallel algorithm performs the following tasks:

1. **Main Function**: The main function is where the main program logic resides. It initializes MPI, sets up variables, opens files for reading and writing, and handles parallel execution.
2. **Reading Files**: The code reads input from two files: “words.txt” and “jumbled.txt”. It extracts word counts and characters from these files.

3. **Word Jumbling and Sorting**: It reads jumbled words from “jumbled.txt”, sorts them, and stores them for later comparison. The sorting is done using the quicksort algorithm.
4. **Parallel Processing**: The program utilizes MPI for parallel processing. It divides the workload among different MPI processes. Process 0 reads the word list, divides it into blocks, and distributes these blocks to other processes for matching with jumbled words. Other processes receive blocks of words, sort them, and compare them with the sorted jumbled words to find matches. When a process is done working on a block, it sends a message to the master process (process 0) requesting more work. This is how load balancing is ensured in this program.
5. **Matching Words**: Each MPI process matches sorted jumbled words with sorted blocks of words obtained from the input file. Matches are recorded into an array in which each process records that which jumbled words were unjumbled by them and eventually reduced to a single result.
6. **Output**: The code generates output files containing matched pairs of jumbled words and their corresponding original words. It also records jumbled words that were not found in the input word list. Each process creates its own txt file (example – process1.txt) which consists of all the unjumbled forms found by that process. At the end when all the results are reduced to be received by process 0, we exactly know which jumbled forms were not found. These words are written to the WordNotFound.txt file by process 0.
7. **Resource Management and Cleanup**: Memory is allocated dynamically using malloc. Proper cleanup and deallocation of memory resources are done using free. Files are closed after use.
8. **Error Handling**: Basic error handling is implemented for file operations and MPI initialization.

Output files:

The output stays the same in this case with the difference that results are divided into multiple txt files (one for each process, example – process1.txt). The words which were not found are recorded into WordNotFound.txt.

Analysis and Results

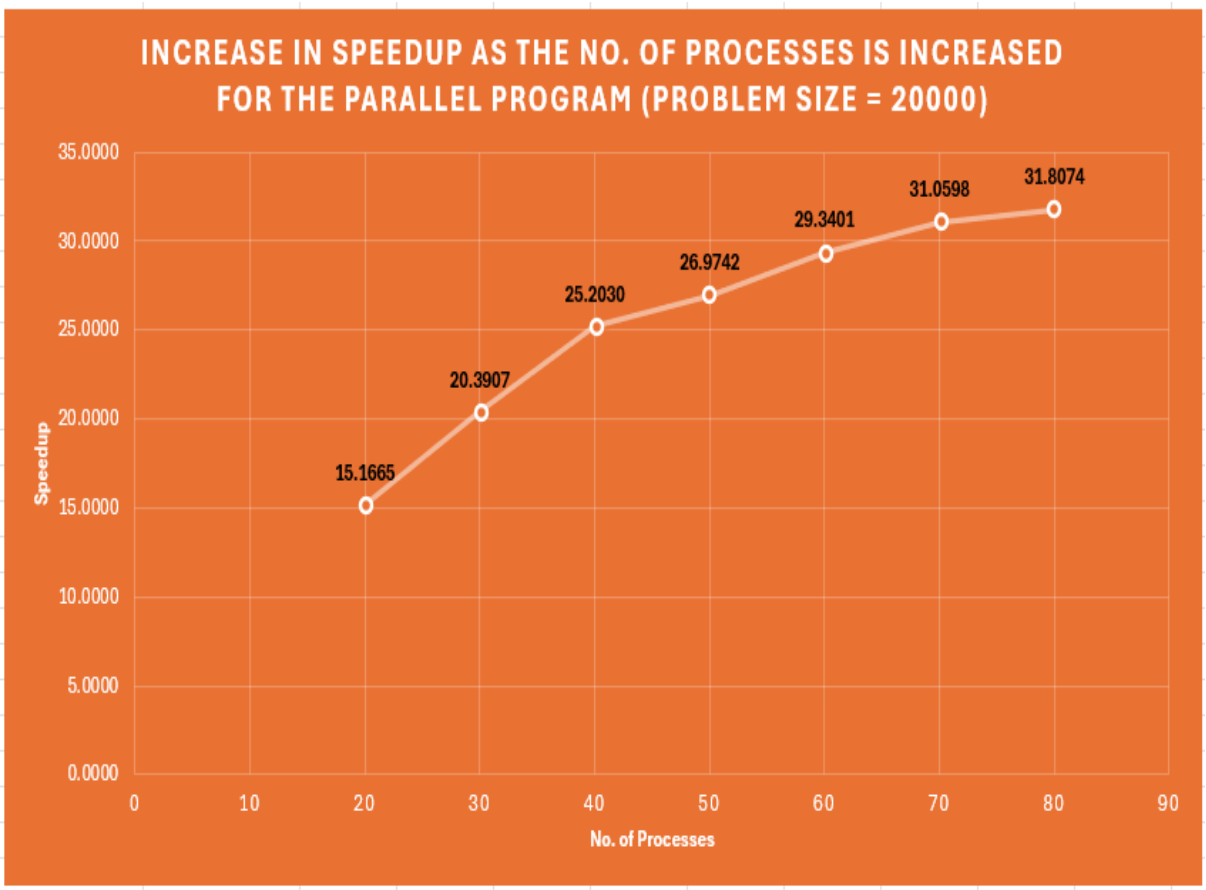
The Serial and Parallel programs were executed for multiple jumbled lists having different sizes using different number of processes for the parallel program and the following results were obtained:

Problem Size		Serial Program	Parallel Program						
	No. of Processes =>		20	30	40	50	60	70	80
10000	Time Taken (seconds)	115.9300	7.9699	5.9890	4.9741	4.8384	4.5137	4.3300	4.3435
	Speedup		14.5459	19.3572	23.3068	23.9603	25.6840	26.7738	26.6906
	Efficiency		0.7273	0.6452	0.5827	0.4792	0.4281	0.3825	0.3336
12000	Time Taken (seconds)	138.6700	9.4144	7.0841	5.8372	5.7140	5.1455	4.9214	4.9379
	Speedup		14.7296	19.5747	23.7563	24.2686	26.9497	28.1772	28.0826
	Efficiency		0.7365	0.6525	0.5939	0.4854	0.4492	0.4025	0.3510
14000	Time Taken (seconds)	162.7300	10.9187	8.1766	6.7340	6.3737	5.8266	5.6349	5.4944
	Speedup		14.9037	19.9020	24.1654	25.5316	27.9288	28.8791	29.6173
	Efficiency		0.7452	0.6634	0.6041	0.5106	0.4655	0.4126	0.3702
16000	Time Taken (seconds)	184.9800	12.3743	9.2571	7.5530	7.1350	6.5683	6.2444	6.1300
	Speedup		14.9488	19.9824	24.4910	25.9258	28.1623	29.6231	30.1760
	Efficiency		0.7474	0.6661	0.6123	0.5185	0.4694	0.4232	0.3772
18000	Time Taken (seconds)	215.2000	13.8657	10.3306	8.3865	7.9196	7.3324	6.8762	6.6522
	Speedup		15.5203	20.8314	25.6604	27.1731	29.3493	31.2965	32.3502
	Efficiency		0.7760	0.6944	0.6415	0.5435	0.4892	0.4471	0.4044
20000	Time Taken (seconds)	232.5100	15.3305	11.4028	9.2255	8.6197	7.9247	7.4859	7.3099
	Speedup		15.1665	20.3907	25.2030	26.9742	29.3401	31.0598	31.8074
	Efficiency		0.7583	0.6797	0.6301	0.5395	0.4890	0.4437	0.3976

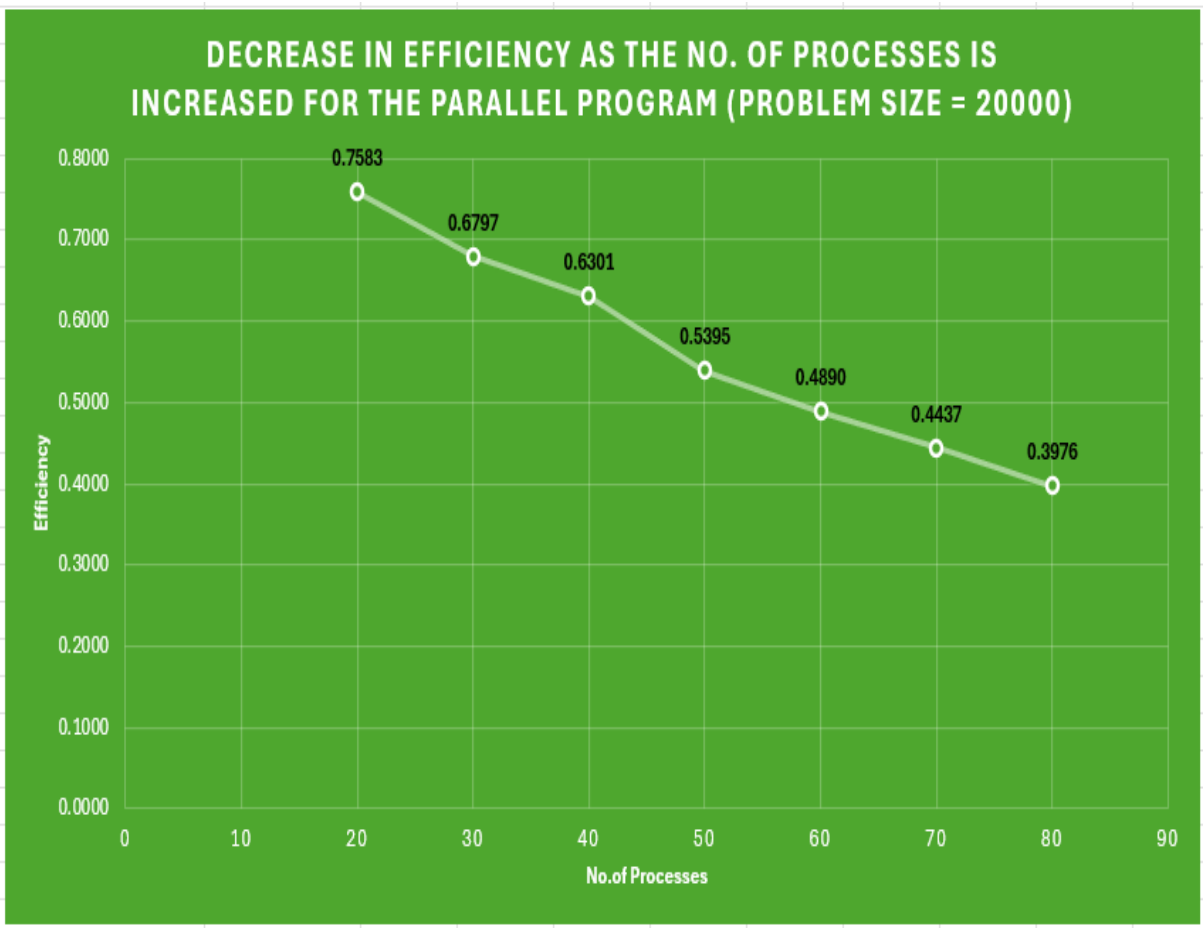
Different runtimes (in seconds) obtained for different jumbled list sizes and no. of processes.

Looking at the table above, it can be concluded that the best speedup (32.3502) is attained when 70 processes are used for the second largest problem size (18000) and the best efficiency (0.7760) is obtained when least no. of processes (20 here) are used for the second largest

problem size (18000).Overall, it is quite evident that the values of speedup and efficiency are on the lower side. The reason for this is that this parallel program performs a lot of communications to ensure load balancing and hence, the optimal speedup for this program is expected to be achieved when the problem size (size of jumbled list) is very large. On a quick glance, it is evident that the speedup achieved by the parallel program tends to increase with an increase in the number of processes. However, looking at the efficiency, it is seen to be decreasing for an increase in number of processes most likely due to the massive increase in the value of communication overhead which increases with the number of processes. Hence, even though using large number of processes to attain better speedup might look like the perfect way to go, using lesser no. of processes to attain a little lesser speedup doesn't also look like a bad option. The trends experienced with the changes in the number of processes can be better seen in the following charts:

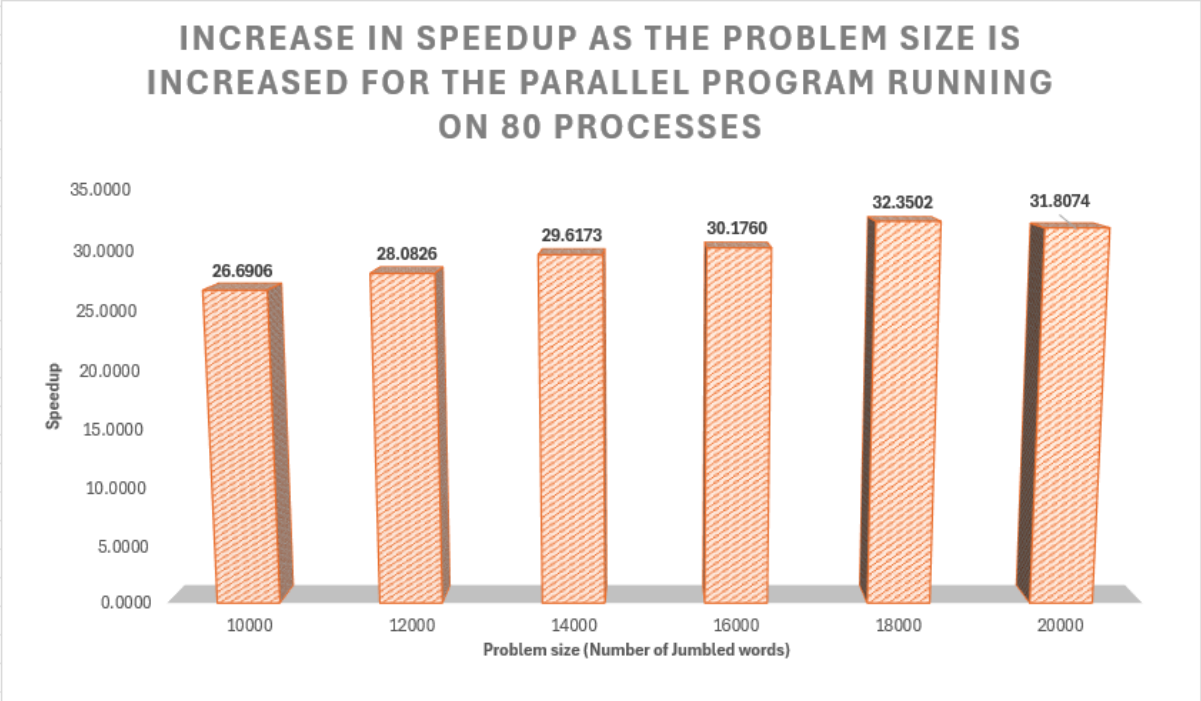


Speedup attained vs No. of processes.

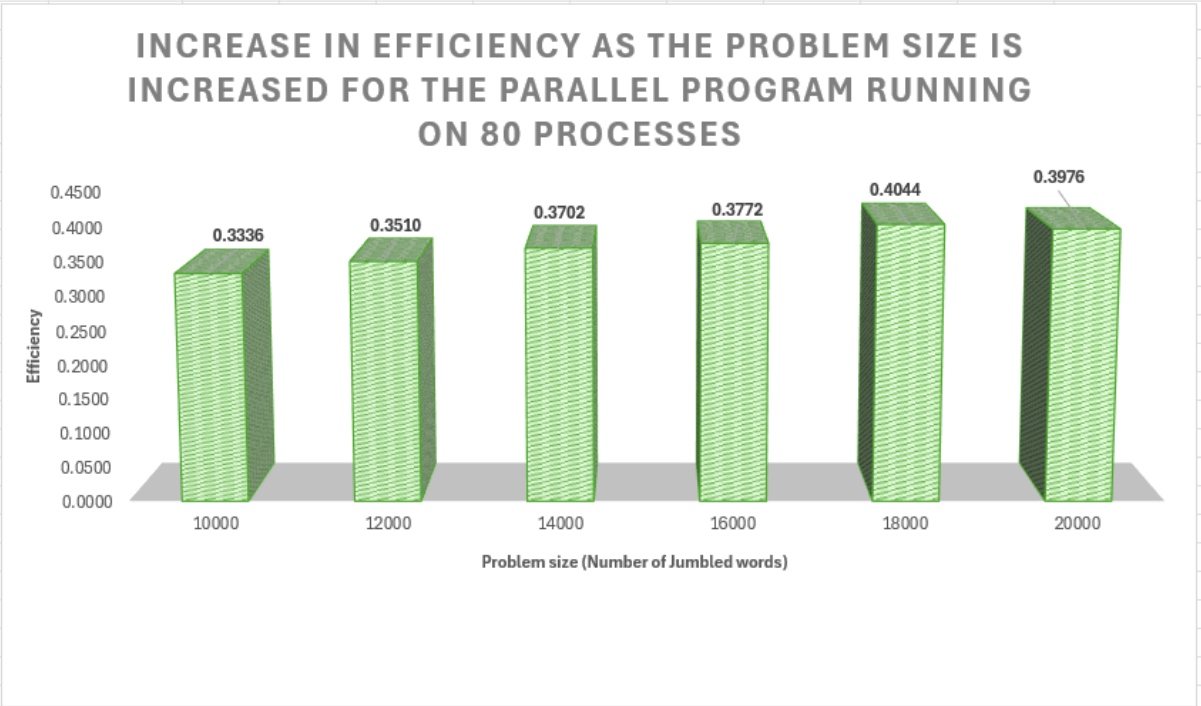


Efficiency vs No. of processes.

Looking at the two charts above, for a problem size of 20000, the program can perform with the highest speedup when using 80 no. of processes, however, also works with the least efficiency. It can be concluded that for a fixed problem size, speed up and efficiency carries an inverse relation when characterized by an increase in the no. of processes. However, the trends when characterized by an increase in the problem size shows different results. An increase in the problem size has a direct relation with both, speedup and efficiency for a parallel program executing with a constant number of processes. These trends are evident in the following charts:



Speedup attained vs Problem size.



Efficiency vs Problem Size.

Looking at the charts above the maximum speed up and efficiency is obtained when the problem size is second largest (18000) for a fixed no. of processes (80 here) and the lowest for

the smallest problem size. Understanding the trend, a constant increase in the speedup and efficiency with problem size suggest that even though the speedup and efficiency might not be optimal for smaller sizes, the same program might work extremely well for massive problem sizes.

It can be concluded that the parallel program is faster than the serial program, however, is less efficient when working with large no. of processes. Even though this is the case, the program works fastest when the no. of processes is more and hence, can work in the most optimal way if the efficiency problem is solved by increasing the problem size to massive values.