CSC453: Group project

**Parallel Bar Chart Generation**

Amal Almuarik, Ghadah Tawhari, Mona Alzamil, AlAnoud AlYousef, Rzan Almagoshi, Norah AlWabel

School of Computer Science, College of Computer and Information Sciences, KSU

Riyadh, Saudi Arabia  
{~~---~~}@KSU.EDU.SA

**Abstract.**

Parallel processing has emerged as a transformative technique for enhancing computational efficiency. In today's world of complex tasks and massive amounts of data, the ability to harness multiple processors simultaneously has become increasingly important. This paper investigates the effectiveness of parallelizing a bar chart generation program using OpenMP and MPI, comparing their performance to the sequential execution. The results demonstrate that OpenMP with a higher number of threads outperforms both sequential and MPI parallelization, especially for large datasets. The limitations of MPI scalability are highlighted, emphasizing the efficiency of shared-memory parallelism.

1. Introduction

Parallel processing can be considered as one of the powerful aspects that enhances the performance of computation by leveraging the power of multi-core or even a single-core processor (i.e., with pipelining) to perform many tasks simultaneously. As a result of parallelism, complex problems can be decomposed into smaller subtasks, where each subtask is then processed concurrently. The latter leads to more efficient processing and time reduction which ultimately reduces the amount of time required to complete specific computations [1].

This project aims to parallelize a bar chart generation program using OpenMP and MPI, as well as compare and analyze the performance of these two techniques.

This document is organized as follows. Section 2 explains the objective of the program. Section 3 will showcase how the program i.e., bar chart generator is implemented sequentially as well as its performance in terms of complexity and execution time, Section 4 will display how the program is implemented in parallel using OpenMP and its performance by explaining how the work is distributed among threads, its complexity and the time taken for the program to be executed, Section 5 will show how the program was implemented using the other parallelizing technique i.e., MPI in addition to showing its performance in terms of work distribution among processes, complexity, and execution time. In Section 6 the results of the project will be discussed. Finally, conclusions are given in Section 7.

1. Problem definition

The objective of this project is to develop a program that generates a bar chart representation based on an array of numerical data. The program receives an input array consisting of numerical values and produces a visual output in the form of a bar chart of asterisks (\*), that accurately represents the numerical data. Figure 1 depicts the execution process at a high level of abstraction.

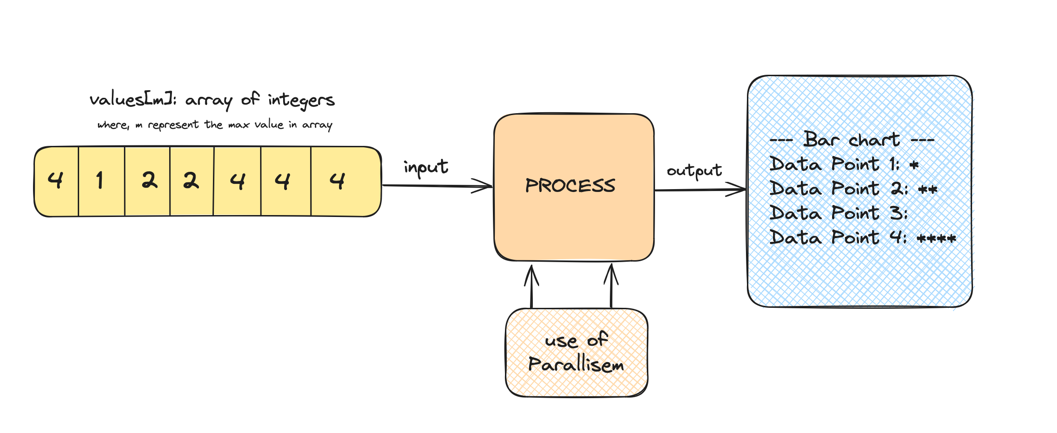


Figure 1. An overview of parallel star generation.

To ensure optimal computational efficiency, we will utilize the following parallel computing techniques. OpenMP and MPI. OpenMP is a shared memory parallelization model that will be employed to distribute the computational workload across multiple threads, furthermore, we will take advantage of MPI i.e., a distributed memory parallelization model, that is used for distributing work among processes, enabling efficient data sharing between each process via message passing [2]. By testing both OpenMP and MPI, we can develop an efficient program that effectively generates bar charts for large arrays of data, benefiting from parallelism at both the shared-memory and distributed-memory levels.

1. Sequential Algorithm
   1. Pseudocode

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm 1. Sequential Bar Chart Generation | | | | | | |
|  | **Input**: size of array, array of values | | | | | |
|  | **Output**: Bar chart | | | | | |
|  | **function** generateBarChart(**size: int , values: int**) | | | | | |
| 1 | Initialization of variable **max\_val** ß **values** [0] | | | | | |
| 2 |  | | | | | |
| 3 | for **i** ß 1 to **size** - 1 | | | | | |
| 4 |  | | if **values**[**i**] > **max\_val** | | | |
| 5 |  | | **Max\_val** ß **values**[**i**] | | | |
| 6 |  | | **end if** | | | |
| 7 | **end for** | | | | | |
|  |  | | | | | |
| 8 |  | **frequency** ß allocate an array of integers of size (**max\_val** + 1) initialized with 0 | | | | |
|  |  | | | | | |
| 9 |  | for **i** = 0 to **size** - 1 | | | | |
| 10 |  | | | frequency[**values**[**i**]] ß frequency[**values**[**i**]] + 1 | | |
| 11 |  | **end for** | | | | |
| 12 |  | print ("--- Bar chart ---") | | | | |
| 13 |  | for **i** ß 1 to **max\_val** | | | | |
| 14 |  | | | if frequency[**i**] >= 0 | | |
| 15 |  | | | | print ("Data Point ", **i**, ": ") | |
| 16 |  | | | | for **j** ß 0 to frequency[**i**] - 1 | |
| 17 |  | | | | | print ("\*") |
| 18 |  | | | | **end for** | |
| 19 |  | | | **end if** | | |
| 20 |  | **end for** | | | | |
| 21 | **end function** | | | | | |

* 1. Performance

**The complexity of the algorithm (big O) with variable array sizes.**

The sequential code has a time complexity of O(N), where N is the size of the input array, and M is the maximum value in the input array. There are 3 loops in the function 'generateBarChart':

1. The first loop is **finding the maximum value** in the data set which takes **O(N)** time because it iterates through the entire input set.
2. We have utilized two arrays to achieve the frequency count task. If it is done using a nested loop, we would notice the shift in performance from sequential to parallel. This loop **counts the frequency** of each value and the values at the same time from the dataset (the input array) which also takes **O(N)** time.
3. The third loop would be the loop that **displays the bar chart**, this loop iterates from 1 to M with time complexity **O(M),** where M is the maximum value. However, since M is usually much smaller than the size of the input dataset N, it can be considered to have a time complexity of **O (1).**

**The time your sequential code takes when it is executed.**

After running the code 10 times on an array of size 2000, with randomized values ranging from 1 to the max range (2000), as seen in Appendix A the average time taken throughout the 10 runs was **3.36 microseconds**, which proves to be good results but can be further optimized when the sequential code is parallelized instead.

1. OpenMP Algorithm
   1. OpenMP pseudo code

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithm 2. Parallel Bar Chart Generation with OpenMP** | | | | | | | | | |
|  | **Input**: size of array, array of values, number of threads (i.e., num\_threads) | | | | | | | | |
|  | **Output**: Bar Chart, time elapsed of parallel regions execution | | | | | | | | |
|  | **function** generatePBarChart (**size: int,** **values: int**, **num\_threads: int**): **double** | | | | | | | | |
| 1 | Initialization of variable **max\_val** ß **values** [0] | | | | | | | | |
| 2 | Declaration of start, end | | | | | | | | |
| 3 | **start ß** omp\_get\_wtime(); /\* start time \*/ | | | | | | | | |
| 4 | **#pragma omp parallel for reduction(max:max\_val) num\_threads(num\_threads)** | | | | | | | | |
| 5 | for **i** ß 1 to **size** - 1 | | | | | | | | |
| 6 |  | | | if **values**[**i**] > **max\_val** | | | | | |
| 7 |  | | | **max\_val** ß **values**[**i**] | | | | | |
| 8 |  | | | **end if** | | | | | |
| 9 | **end for** | | | | | | | | |
|  |  | | | | | | | | |
| 10 |  | | **frequency** ß allocate an array of integers of size (**max\_val** + 1) initialized with 0 | | | | | | |
|  |  | | | | | | | | |
| **11** |  | | **#pragma omp parallel for num\_threads(num\_threads)** | | | | | | |
| 12 |  | | for **i** ß 0 to **max\_val-1** | | | | | | |
| 13 |  | | | | **#pragma omp atomic** | | | | |
| 14 |  | | | | frequency[**values**[**i**]] ß frequency[**values**[**i**]] + 1 | | | | |
| 15 |  | | **end for** | | | | | | |
| 16 |  | | **endß** omp\_get\_wtime(); /\* end time \*/ | | | | | | |
| 17 |  | | print ("--- Bar chart ---") | | | | | | |
| 18 |  | | for **i** ß 1 to **max\_val** | | | | | | |
| 19 |  | |  | | | | if frequency[**i**] >= 0 | | |
| 20 |  | |  | | | |  | print ("Data Point ", **i**, ": ") | |
| 21 |  | |  | | | |  | for **j** ß 0 to frequency[**i**] - 1 | |
| 22 |  | |  | | | |  |  | print ("\*") |
| 23 |  | | | | | | | **End for** | |
| 24 |  | | | | | **End if** | | | |
| 25 |  | **End for** | | | | | | | |
| 26 | **end function** | | | | | | | | |

* 1. OpenMP performance

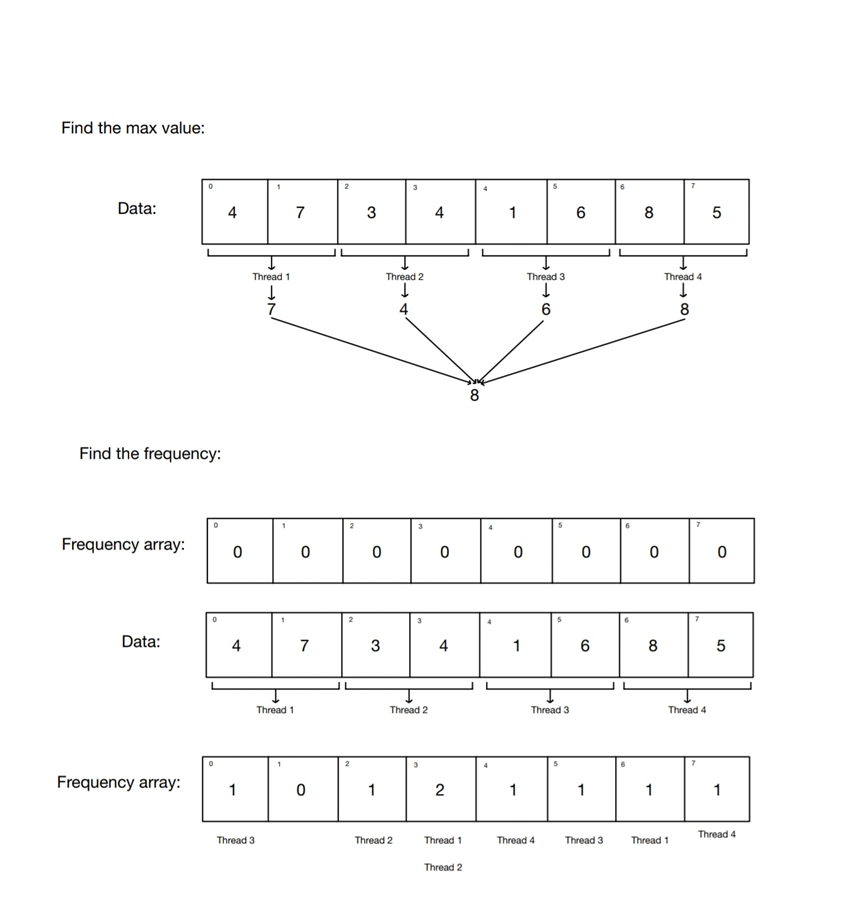


Figure 2. Sample of Work Distribution Among Threads (Number of Threads = 4)

**The complexity of the algorithm (big O) with variable array size and variable number of threads.**

The **OpenMP** code has a time complexity of O(N), where N is the size of the input-

array, M is the maximum value in the input array, and K is the number of threads.

There are 3 loops in the function 'generatePBarChart':

1. The first loop is **finding the maximum value** in the data set which takes O(N) time because the iterations will be divided among the K threads, where k is a constant number. which means **O(N).**
2. The second loop **counts the frequency** of each value in the dataset (the input array) which also takes **O(N)** time, as the M iterations will be divided among the K threads. Assuming the worst-case scenario where M = N.
3. The third loop would be the loop that **displays the bar chart**, this loop iterates from 1 to M with time complexity **O(M)**, where M is the maximum value. However, since M is usually much smaller than the size of the input dataset N, it can be considered to have a time complexity of **O (1)**.

**The time your OpenMP code takes when it is run**

To evaluate the performance of the optimized code, it was executed ten times on an array of size 2000, with randomized values ranging from 1 to the maximum range (2000). As detailed in Appendix A, the average execution time across the ten runs was **0.000487** **seconds**, demonstrating significant optimization compared to the sequential code.

1. MPI Algorithm
   1. MPI pseudo code

|  |  |  |
| --- | --- | --- |
| **Algorithm 3. Parallel Bar Chart Generation with MPI** | | |
| **Input**: number of processors | | |
| **Output**: Bar Chart, time elapsed of parallel regions execution | | |
| **Constant DEFAULT\_SIZE, ITERATIONS** | | |
| **function** main ( ): **int** | | |
| **Initialize MPI environment** | | |
| **Getting the number and rank of processes** | | |
| max\_range**ß** **DEFAULT\_SIZE** | | |
| Initialize double array **iteration\_times**[ITERATIONS] | | |
| for **iteration** **ß** 0 to **ITERATIONS** -1 | | |
| initialize integer array **values**[max\_range] | | |
| If rank is 0 | | |
| Initialize **values** with random values between 1 and **max\_range** | | |
| Print **values** | | |
| **End if** | | |
| **Broadcast dataset to all processes using MPI\_Bcast()** | | |
| **StartTime ß MPI\_Wtime();** | | |
| **elementsPerProcessor ß** max\_range/ number of processors | | |
| **Extra\_elements ß** max\_range % number of processors | | |
| **start ß** rank \* **elementsPerProcessor** + (rank < Extra\_elements ? rank : Extra\_elements) | |
| **end ß** start + **elementsPerProcessor** + (rank < Extra\_elements? 1 : 0) | |
| **LocalMax ß** **values**[start] | |
|  | for i **ß** **start** to **end**-1 | |
|  | If values[**i**] > **localMax** | |
|  | **localMax** **ß** values[i] | |
|  | **End if** | |
|  | **End for** | |
| **Calculate Global maximum** using **MPI\_Reduce** | |
| **Broadcast Global maximum** to all processes using **MPI\_Bcast** | | |
| **Initialize** Integer array **Lcount[globalMax]** | | |
|  | For i **ß 0 to globalMax** - 1 | |
|  | **Lcount** [i] **ß** 0 | |
|  | **End for** | |
|  | for **i** **ß** s**tart** to **end** - 1 | |
|  | Increment **Lcount** [values[i] – 1] | |
|  | **End for** | |
| **Initialize** Integer array **Rcount**[number of processors] | | |
|  | for **i ß** 0 to **number of processors** - 1 | |
|  | Rcount[**i**] **ß** 0 | |
|  | **End for** | |
| **Gather Global maximum** values to **Rcount** on process 0 using **MPI\_Gather** | | |
| **Initialize** Integer array **Gcount**[globalMax \* number of processors] | | |
|  | for **i ß** 0 to **globalMax \*** number of processors **- 1** | |
|  | **Gcount** [**i**] **ß** 0 | |
|  | **End for** | |
| **Gather Lcount values to Gcount** on process 0 using **MPI\_Gather** | | |
|  | **If rank is 0** | |
|  | **Print(“--- Bar Chart for iteration %d ---“, iteration + 1)** | |
|  | **for i ß** 0 to **globalMax - 1** | |
|  | **Print(“Data Point %d: “, i + 1)** | |
|  | **for j ß** 0 to number of processors **- 1** | |
|  | **for k ß 0 to Gcount[j \* globalMax + i] - 1** | |
|  | **Print(“\*”)** | |
|  | **End for** | |
|  | **End for** | |
|  | **End for** | |
|  | **End if** | |
| **EndTime ß** **MPI\_Wtime();** | | |
| **Durationß** **Duration + (Endtime – StartTime)** | | |
| If rank is 0 | | |
| Print Number of iteration and its execution time | | |
| **End if** | | |
| **End for** | | |
| If rank is 0 | | |
| Print average execution time over the number of iterations | | |
| **End if** | | |
| **Finalize MPI environment** | | |
| **End main function** | | |
|  | | |
|  | | |

**5.2 MPI performance**

**The complexity of the algorithm (big O) with variable array sizes.**

The MPI code has a time complexity of O(N/K), where N is the size of the input array, M is the maximum value in the input array, and K is the number of threads. There are several parts in the code:

1. The first part is a loop for **finding the local maximum in each processor** which takes **O(N/K)** time because each processor iterates through the data in its range.
2. The next part of the code **reduces the maximum of the processes** to get the global maximum and then **broadcasts the global maximum** to all the processors which both takes **O(log(K))**
3. After that a loop is used to **initialize a local counter** which takes **O(M)**
4. In the next part each processor **count the occurrences of each value in its local range** which takes **O(N/K)**
5. The next part **initializes receive counter array** for the size of the processors in **O(K)**
6. The next part **gathers global max values** from all the processors in complexity of **O(log(K))**
7. The next part **initializes a global counter array** which takes **O(M\*K)**
8. Finally, the **local count is gathered** to array global count which takes **O(log(K))**

**The time your MPI code takes when it is executed.**

After running the code 10 times on an array of size 2000, with randomized values ranging from 1 to the max range (2000), as seen in Appendix A the average time taken throughout the 10 runs was **0.006796 seconds.**

1. Result and discussion

As shown in Figure 3, The line graph compares the performance of three different bar star generation algorithms implemented in various ways, including sequential execution, OpenMP parallelization with 2, 4, 8, and 16 threads, and MPI parallelization with 2, 4, 8, and 16 processors. The performance of each algorithm was measured over 10 iterations with a dataset of size 2000.

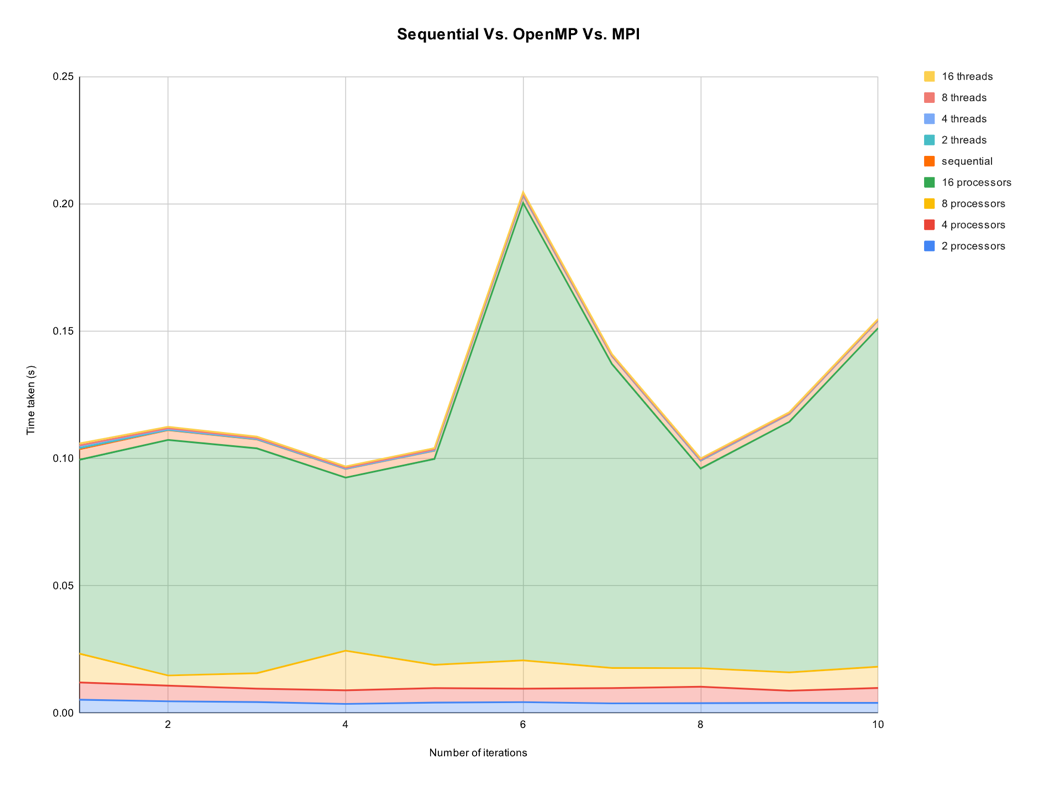


Figure 3. Performance Comparison of Bar Star Generation Algorithms - Sequential vs. OpenMP and MPI Parallelization.

Overall, the OpenMP parallelization algorithm with a higher number of threads has achieved the best performance among the sequential and MPI when tested on large datasets of size 2000 for this specific bar star generation algorithm. While MPI parallelization has limitations in terms of scalability in this context.

For the MPI parallelization, the execution times increase as the number of processors grows. This suggests that, in this specific case, the parallelization with MPI might not be as effective, potentially due to increased communication overhead or other factors. Specifically, the time increases dramatically when moving from 8 to 16 processors. (refer to Appendix 1)

On the other hand, the OpenMP parallelization demonstrates a clear advantage, with execution times decreasing as the number of threads increases to a certain limit (that it doesn’t exceed the number of cores in the machine). This indicates good scalability, and the algorithm benefits significantly from shared-memory parallelism. The execution times are notably lower compared to MPI, especially with 16 threads.

The sequential execution times serve as a baseline for comparison. The parallel implementations, particularly the OpenMP, outperform the sequential version, highlighting the effectiveness of parallelization in enhancing the algorithm's performance on large datasets.

1. Conclusion and future work

In conclusion, this project has successfully generated an efficient bar chart through both sequential programming and parallel programming by using MPI & OpenMP, the sequential code worked as the foundation for the parallel code as well as giving a base for performance evaluation comparison among all three codes, the sequential code of the program is functional however it's not as efficient as using parallel programming, in the future we hope to hybridize the parallelism using both OpenMP and MPI to optimize the performance, as well as exploring the integration of the code with different data visualization techniques such as scatter plots, test the codes on bigger data sets, and decrease the communication overhead in MPI and explore more optimization techniques such as cache optimization to further optimize the sequential and parallel codes.

References

1. Parhami, B.: Introduction to parallel processing: Algorithms and architectures. Kluwer Academic, Dordrecht (2002).
2. Message Passing Interface, [https://hpc.nmsu.edu/discovery/mpi/introduction/.](https://hpc.nmsu.edu/discovery/mpi/introduction/) Last accessed 2023/10/27.

**Appendix A**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sequential** | **OpenMP** | | | **MPI** | | |
| **The strategy used with a dataset size = 2000, number of iterations = 10** | | | | | | |
| 1. Sequential | | | | | | |
| Screenshot of Output | | | | | | |
| A screenshot of a computer  Description automatically generated | | | | | | |
| 1. OpenMP | | | | | | |
| Screenshot of Output | | | | | | |
| 2 Thread | 4 Thread | 8 threads | | | 16 threads | |
|  |  |  | | |  | |
| 1. MPI | | | | | | |
| Screenshot of Output | | | | | | |
| 2 processors | 4 processors | | 8 processors | | | 16 processors |
|  |  | |  | | |  |

|  |  |
| --- | --- |
| Sequential |  |
| OpenMP |  |
| MPI |  |

**Appendix B**

Table 1. Task Distribution

|  |  |
| --- | --- |
| **Task** | **Student** |
| Introduction and problem definition | Rzan Almagoshi |
| Sequential code | Mona Alzamil |
| Sequential performance | AlAnoud AlYousef |
| OpenMP code | Amal Almuarik , AlAnoud AlYousef |
| OpenMP performance | Ghadah Tawhari |
| MPI code | Ghadah Tawhari, Rzan Almagoshi, Norah Alwabel |
| MPI performance | Mona Alzamil |
| Result and discussion | Amal Almuarik |
| Conclusion and future work | Norah Alwabel |
| Review | All |