Sequence and Parallel comparison of deadlock avoidance algorithms

Supriya Sahoo, Akriti Kumari, Ankita Bhalavi, Vandana Baidya
Department of Information Technology,
National Institute of Technology Karnataka, Surathkal,
Mangaluru 575025
Supriya6off@gmail.com,akritismhs@gmail.com
ankitabhalavi06@gmail.com
vandanabaidya@gmail.com

Dr. Geetha. V
Department of Information Technology,
National Institute of Technology Karnataka,
Surathkal,
Mangaluru 575025
geethav@nitk.ac.in

***Abstract*-- This paper presents the implementation of the three different algorithms for deadlock avoidance namely the dining philosophers problem, the producer-consumer problem and the reader-writer problem. The deadlock prevention algorithms are used in concurrent programming when multiple processes must acquire more than one shared resource. The algorithms are implemented in sequentially as well as in parallel execution by using OpenMP . The main contribution to our work is to analyze the performance and to compare the efficiency of the parallel and sequential execution in terms of running time and speedup.**

Keywords—Dining philosopher problem ,Reader-writer problem, producer-consumer problem, parallelization, Histogram equalization, OpenMP.

I. INTRODUCTION

In concurrent computing, a deadlock is a state in which each member of a group is waiting for some other member to take action, such as sending a message or more commonly releasing a lock.Deadlock is a common problem in multi- processing systems, parallel computing, and distributed systems, where software and hardware locks are used to handle shared resources and implement process synchronization. To avoid the deadlock, the three algorithms have been implemented in sequential and parallel approach where the parallelization can be done using threads. The main aim is to compare the efficiency of both the parallel and sequential algorithms. OpenMP provides a way for thread parallelism, with shared memory concept. Nowadays, as normal standalone systems also come with dual core, quad core etc, it is beneficial to run the algorithm parallel using threads. The OpenMP provides directives to compiler, for identifying the portion of code or algorithm, which can be run in parallel. For the purpose of comparing the efficiency of the parallel and sequential execution, the three algorithms are implemented in sequential and parallel approach using OpenMP . The paper is structured as follows: Section II provides details on methodology, section III provides the related work and section IV provides details about proposed parallel algorithm and discussion followed by conclusion and reference.

II. METHODOLGY

A. The Dining Philosophers problem

The dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. The problem statement for this algorithm is for example , five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.
Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks. Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.
The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

B. *The Producer-Consumer Problem*

In producer-consumer problem, there are two types of process involved: producer and consumer who are sharing a fixed size buffer . The role of producer is to put the item one a time in the buffer and the role of consumer is to retrieve the item from the buffer. The problem is to make sure that the producer will not try to add the item once the buffer is full and that the consumer will not try to remove the item from an empty buffer .

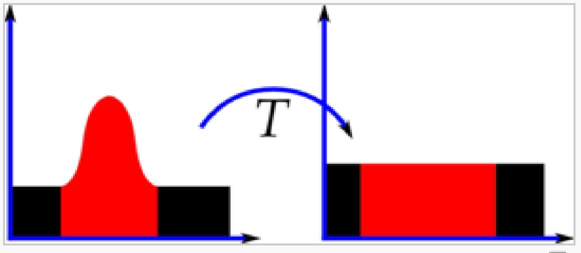
The normal behavior of this problem are (i) Random arrival of petitions to put item in the buffer; (ii) One consumer that immediately after getting an information from the buffer intends to get another one; (iii) Buffer of defined and finite

C. *Reader-Writer Problem*

Readers writer problem is another example of a classic synchronization problem. There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer**is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource.

III. RELATED WORK

We have used the concept of deadlock avoidance ,synchronization and threads to build the histogram equalization algorithm on images .
Histogram equalization is a method for modifying the dynamic range and contrast of an image by altering that image such that its intensity histogram has a desired shape.
The histogram of an image refers to histogram of the pixel intensity values. A mapping is done between the pixel intensity values in the input and output images such that the output image file contains a uniform distribution of intensities (a flat histogram). This technique is used in image comparison processes for detail enhancement and in correction of nonlinear effects.
The probability mass function and the cumulative distribution function of the pixel values in input image are used to determine the pixel values in output image. A transfer function is used to derive the required relation. The histogram and output pixel value computation can be carried out as 2 parallel steps.

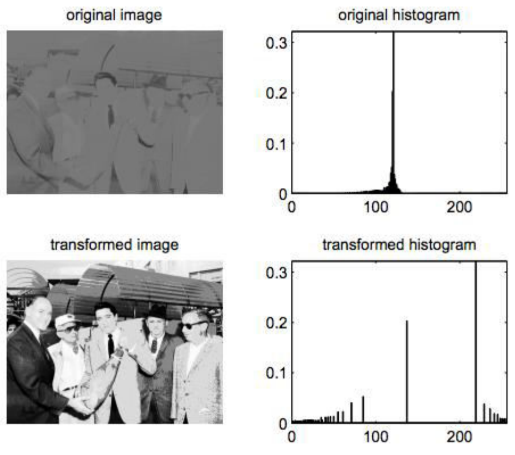


Histograms of an image before and after equalization.

Input and Output:
Serial Program :
Input : A bmp image which is to be modified by applying the equalisation algorithm.
Output : Modified bmp image file.

Parallel Program :
Input : A bmp image which is to be modified by applying the equalisation algorithm and the number of threads
Output : Modified bmp image file.

The image sizes being considered are :
128 x 128
256 x 256
512 x 512
1024 x 1024
2048 x 2048



Applications :
Histogram equalization is often used to achieve better quality images in black and white color scales in medical applications such as digital X-rays, MRIs, and CT scan.
Histogram equalisation enhances the quality of the image and hence has applications in image processing. This technique is also used for enhancement of satellite images.

System Configuration :

CPU MHz	1200.000
Cache size	20480 KB for each processor
cpu cores	8
fpu	yes
fpu_exception	yes
cpuid level	13
cache_alignment	64
address sizes	46 bits physical, 48 bits virtual

IV. PROPOSED PARALLEL ALGORITHM

➢ Description of serial algorithm -

Pseudocode :

Read the image from bmp file into a one-dimensional character array.

```
unsigned char* image = read_bmp(argv[1]);
```

Histogram calculation : For each pixel value calculate its frequency of occurrence and store it in an array - calculation of probability mass function.

```
int* histogram = (int *)calloc(sizeof(int), color_depth);
unsigned char* output_image = malloc(sizeof(unsigned char) * image_size);

for(int i=0;i<image_size;i++) {
    histogram[image[i]]++;
}
```

Normalize the read values by applying a transfer function - calculate cumulative distribution function of the probability mass function and store the values in another one-dimensional array.

```
float* transfer_function = (float *)calloc(sizeof(float), color_depth);

for(int i=0;i<color_depth;i++) {
    for(int j=0;j<i+1;j++) {
        transfer_function[i] += color_depth*((float) histogram[j])/image_size;
    }
}
```

For each pixel in the output image, we store the normalised value for that pixel from the input image.

```
for(int i=0;i<image_size;i++) {
    output_image[i] = transfer_function[image[i]];
}
```

Write the pixel values in the output bmp file.

```
write_bmp(output_image, image_width, image_height);
```

Time Complexity : For each step, that is, reading from the input file, calculating the probability mass function, normalizing using the transfer function and writing to the output bmp file, involves computation on each pixel value and hence the running time of the algorithm is O(n) where n is the number of pixels.

Scope of parallelism :

There are some parts of the code that are inherently serial, for example, reading from and writing to a bmp image needs to be performed serially. This introduces a bottleneck in the performance of the algorithm. However, the portion of the code that deals with the computation of new pixel values from the old pixel values by probability mass function and cumulative distribution

Normalize the read values by applying a transfer function - calculate cumulative distribution function of Similar to the above step, we can calculate transfer the probability mass function and store the values in another one-dimensional array. function can be parallelised.Approximately, 2-3% of the code is inherently serial which means that 97-98% of the code can be executed concurrently as obtained by measuring for different image sizes.

Sections of code that can be parallelised :

The transfer function value is calculated for each pixel the value of which is independent per pixel. Hence we can parallelize this part of the code by dividing the work as equally as possible among the threads.

Also, it can be seen that for each pixel, the value of the histogram array for all the smaller indices at every iteration for each pixel. Hence, this can be optimise and can be done parallely.

Effect of increasing problem size on serial code:

As the problem size increases, the computation increases. Hence, the time taken also increases. It can also be seen that the computations that can be done in parallel without any conflicts increase. This can be considered when we parallelise the code.

➢ Description of parallel algorithm –

Pseudocode :

Read the image from bmp file into a one-dimensional character array.

```
unsigned char* image = read_bmp(argv[1]);
```

Histogram calculation : For each pixel value calculate its frequency of occurrence and store it in an array - calculation of probability mass function. The assignment of pixel values to the one-dimensional array image[] (which represents the input image) can be parallelised since the threads are accessing elements at different indices. However, since the histogram[] array is a shared variable which all the threads are modifying in each iteration of the loop, one needs to ensure race conditions do not occur. Hence, that part of the code is put in the critical section.

```
    int* histogram = (int*)calloc(sizeof(int), color_depth);
#pragma omp parallel for num_threads(n_threads)
    for(int i = 0; i < image_size; i++){
        int image_val = image[i];
#pragma omp critical
        histogram[image_val]++;
    }
```

function in a parallel fashion because
each thread operates on a different set of values. Scheduling is used to for achieving almost equal work to all threads.

```
#pragma omp parallel for num_threads(n_threads) schedule(static,1)
    for(int i = 0; i < color_depth; i++){
        float sum = 0.0;
        for(int j = 0; j < i+1; j++){
            sum += (float)histogram[j];
        }
        transfer_function[i] += color_depth*((float)sum)/(image_size);
    }
```

For each pixel in the output image, we store the normalised value for that pixel from the input image.

```
for(int i=0;i<image_size;i++) {
    output_image[i] = transfer_function[image[i]];
}
```

Write the pixel values in the output bmp file.

```
write_bmp(output_image, image_width, image_height);
```

Time Complexity :
For each step, that is, reading from the input file, calculating the probability mass function, normalizing using the transfer function and writing to the output bmp file, involves computation on each pixel value and hence the running time of the algorithm is O(n) where n is the number of pixels. The steps that are parallelised run in O(n)/p time where p is the number of pixels but the reading and writing steps still take O(n) time.

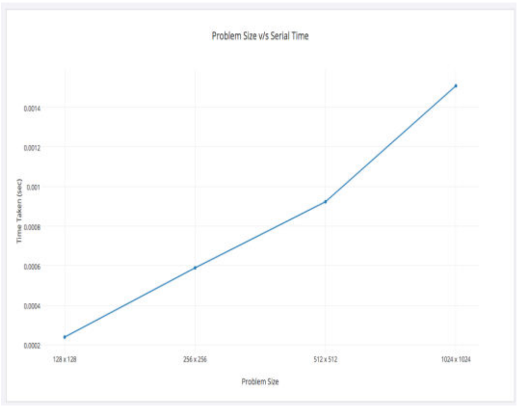
Strategy of parallelization :
Looking at the serial implementation of the algorithm, it can be seen that reading from and writing to a bmp file needs to be done serially. Even if we attempt to do it in parallel, there would not be much difference in terms of performance since overhead due to false sharing and cache coherency would increase as well.
However, there are two loops - one for calculating probability mass function and the other for calculating the cumulative distribution function which do not have any loop-carried or data dependency and only involve calculating new values from independent pixel values of the input image. Hence, we apply different optimisation strategies to achieve this.
Scheduling can also be used to divide work among the threads in such a way that there is minimum overhead due to false sharing among the threads and work distribution is as uniform as possible.

V. RESULTS AND DISCUSSIONS :

For each input image, we have compared the performance of the program with number of threads equal to 2, 4 and 8 threads. Value for higher number of threads have not been mentioned since the performance was lesser as compared to the results with lesser number of threads.
We first compared the performance of static scheduling by varying the chunk size from 2, 4 to 8. As can be seen, the best result is obtained for chunk size = 8. Then we ran the same program with dynamic scheduling and chunk size = 8.

We have computed the values for different image sizes but only included the best results here. The output have been tabulated below:

Image Dimensions	Time taken for serial program
128 x 128	0.000239
256 x 256	0.000589
512 x 512	0.000923
1024 x 1024	0.001509
2048 x 2048	0.000373



VI. CONCLUSION :

Time Taken by serial implementation is always more than that in parallel implementation. It shows that parallel implementation decreases time complexity. And as the number of increases, the parallel time taken also decreases.

The speedup obtained is maximum for static scheduling with chunk size = 8. Here the work done by all the threads is equal, thus static scheduling works the best. The speedup in dynamic scheduling with chunk size = 8 is similar to speedup in static scheduling with chunk size = 4. This is because, dynamic scheduling is done at run-time, thus there is some overhead as compared static scheduling. In our problem the threads have same amount of computations and thus static scheduling works the best and the most optimum chunk size is 8.

As the problem size is increasing the speedup is also increasing, but it is not reaching a maximum speedup it should achieve. Maximum speedup achieved is only 6 with 8 threads. Upto 4 threads the speedup is almost equal to the number of threads which shows that the serial fraction of code is not much significant. After 4 threads, as the number of threads are increasing the speedup is not achieved as expected. This is because after 4 threads the parallel overhead becomes more significant and it increases. Thus speedup is not achieved as expected.

Scope of further improvement :

Scalability: With increase in the image size, the computations and the parallel overhead increases. In our program we are making histogram arrays which are of same size as

image size. Thus, a large amount of memory is used in order to execute the program.

In shared memory system, there might be the memory constraints and thus the image size cannot be increased after a certain point. This problem can be solved using MPI. Using MPI would provides scalability to our problem.

The histogram array can be stored in different memories of different processors. As the computations are independent of each other in our program, these computations can be done independently on the processor. The communication between the processors in the program is needed only during reading and writing the image.

VII. REFERENCES :

[1]https://en.wikipedia.org/wiki/Histogram_equalization

[2]https://veprit.com/photography-guide/using-histogram/what-is-image-histogram

[3] D.G. Lowe, "Distinctive image features from scale-invariant key points," International Journal of Computer Vision, vol. 60, 2004, pp. 91-110.

[4] Y. Ke and R. Sukthankar, "PCA-SIFT: A more distinctive representation for local image descriptors,"In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol. 1, 2004, pp. 511-517.

[5] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," IEEE Trans. PAMI, 2005, pp. 1615-1630.

[6] "Histograms: Construction, Analysis and Understanding" from http://quarknet.fnal.gov/toolkits/new/histograms.html