Object Detection Simulation Using Sequential and Parallel Programming: Comparative Analysis

1st Haya Aldossary[‡]
Department of Computer Engineering
College of Computer Science and Information Technology,
Imam Abdulrahman Bin Faisal University.
Dammam 31441, Saudi Arabia
2190002968@iau.edu.sa

2nd Rahaf Yaan Allah[‡]
Department of Computer Engineering
College of Computer Science and Information Technology,
Imam Abdulrahman Bin Faisal University.
Dammam 31441, Saudi Arabia
2200003935@iau.edu.sa

3rd Reema Albrahim[‡]
Department of Computer Engineering
College of Computer Science and Information Technology,
Imam Abdulrahman Bin Faisal University.
Dammam 31441, Saudi Arabia
2200001337@iau.edu.sa

4th Reham Alzahrani[‡]
Department of Computer Engineering
College of Computer Science and Information Technology,
Imam Abdulrahman Bin Faisal University.
Dammam 31441, Saudi Arabia
2200001931@iau.edu.sa

5th Wahbia Saleh[‡]
Department of Computer Engineering
College of Computer Science and Information Technology,
Imam Abdulrahman Bin Faisal University.
Dammam 31441, Saudi Arabia
2200006267@iau.edu.sa

6th Dr. Naya Nagy
Department of Computer Engineering
College of Computer Science and Information Technology,
Imam Abdulrahman Bin Faisal University.
Dammam 31441, Saudi Arabia
nmnagy@iau.edu.sa

***Abstract*—Two of the most widely used programming languages presently are Python and C. High-level interpreted Python is renowned for being easy to comprehend and use. C, on the other hand, is a low-level compiled language that is renowned for its efficiency. Furthermore, three aspects will be the main emphasis of this comparison analysis: efficiency, speedup, and runtime of parallel code for detecting objects, emphasizing task parallelism. The study examines several approaches to dealing with race conditions and addresses using OpenMP for parallelism in C programming. Finally, the outcomes demonstrate that by increasing the number of threads, C with OpenMP achieves substantially faster run times and improved speed-up compared to Python. This benefit arises from the low-level nature of C, which allows for direct hardware access and effective memory management.**

***Index Terms*—object detection, race condition, OpenMP, efficiency, speedup, runtime, parallel, task parallelism.**

I. Introduction

Object detection (OD) is a fundamental part of computer vision that has been used in several real-world applications across various industries, such as medical imaging, autonomous driving, and more. OD relies on image processing programming to extract features and identify objects within an image or a video frame.

Traditionally, OD algorithms were implemented sequentially, where tasks follow a linear execution model which is easier to debug and understand. However, it can be slow when dealing with complex algorithms or large datasets. In addition, sequential processing may not benefit from the hardware's full potential.

‡ These authors contributed equally to this work.

As for parallel programming, it breaks down tasks into smaller units to be executed concurrently on multiple threads. Adapting this technique, which takes advantage of the parallel hardware architecture, can result in improved performance and faster completion time. Although this technique is theoretically faster than a sequential approach, it can be challenging to manage shared resources and ensure data integrity [1].

In this study, an OD algorithm was executed using both sequential and parallel programming in order to conduct a comparative analysis. In the testing process, the algorithm was implemented using Python and the C language to compare which language was a better option for task parallelization. Task parallelism is a method that speeds up overall execution times and increases efficiency by enabling the execution of numerous tasks at once. Furthermore, different race-handling techniques and different numbers of threads were tested to determine which resulted in the optimal solutions. The outcome of this study shows that parallel execution tends to be faster than sequential execution, and Python is a better option than C. To address the mentioned issue, this study aims to conduct a detailed evaluation of the efficiency, speedup, and run-time of OD implemented in both Python and C programming languages. By investigating the effectiveness of parallel computing techniques, this research contributes to the existing literature and enhances our understanding of the potential benefits of parallelization in optimizing OD algorithms.

The key contributions of this paper include:

- Stimulates the implementation of OD in both parallel and sequential models to provide a guideline for the development and optimization of computer vision systems.

- Showcase the detailed setup of the experiment, including the number of threads used in parallel execution.
- Discuss the challenges related to synchronization and race conditions in parallel execution and analyze different strategies of race handling to find the optimal method.
- Summarize key findings of the comparative analysis and highlight the importance of parallelization in OD algorithms for real-world applications.

This paper is structured as follows: Section 2 discusses previous studies that deployed parallelization for OD, followed by the gap analysis in Section 3. Moreover, Section 4 illustrates the details and justification of the parallel implementation. The methodology used to carry out the study is presented in Section 5. While Sections 6 and 7 showcase the analysis and results. Section 8 summarizes the key findings and provides a conclusion.

II. Literature Review

Multiple studies have been published addressing the use of parallel programming in OD. This section summarizes earlier research to present the body of literature. A chronological order is used to arrange the summaries.

All of the studies that have been reviewed implemented parallel techniques for OD. Firstly, a study conducted by Kyrkou and Theocharides [2], which developed a flexible parallel hardware architecture for real-time object identification using the AdaBoost model and operated on a unique hardware architecture for real-time OD utilizing a Field Programmable Gate Array (FPGA). Additionally, the FPGA is a kind of integrated circuit that may be configured to carry out specific tasks once it is programmed. AdaBoost is a machine-learning technique that is frequently used for computer vision problems involving object recognition and classification. It builds on several weak classifiers to produce a single strong classifier. The model was trained and tested on a large number of labeled images and data collected from the World Wide Web. Furthermore, the methods used in this study include picture scaling, feature computation, stage computation, integral image calculation, and region identification, including objects of interest. The outcomes of the study show that the architecture was scalable, enabling a range of object types and input picture sizes at fair frame rates. Finally, the main limitation of the research is that the system's power consumption is contingent upon several aspects, such as the dimensions of the input picture, the quantity of down-scaled images produced, the number of search windows, and the number of features within the training set.

Another study that utilized the AdaBoost algorithm by Lai et al. [3], the research aimed to optimize data locality for parallel OD implementation in multi-core systems. Data locality is a fundamental design issue that deals with minimizing the time and resources required to access data. This paper discussed a three-staged parallelization scheme for the Viola-Jones algorithm, which was first designed as a face detection technique. The first stage is resized, which scans the image to check if it needs to be resized into different resolutions using the AdaBoost algorithm, short for Adaptive Boosting, which is an ensemble technique, followed by the integral stage, which evaluates the features. For the last stage, OD is performed to detect the location of an object. The proposed design reduced memory access, which resulted in a performance improvement of 58%. Moreover, the study mentions a limitation regarding the trade-off between granularity and synchronization, where higher parallelism levels result in a lower synchronization overhead.

Next, Ma et al. [4] developed a research paper aimed to improve the computational efficiency of parallel algorithm for OD using a multicore CPU platform with OpenMP in static images. The utilized dataset is sourced from the UIUC Image Database for Car Detection and the CMU/VASC Image Database. Moreover, the hardware used to carry out the experiment was a dual-core Core2 E7400 processor with 2.8GHz frequency and 2GB of memory. The methodology involved a combination of template matching and Fuzzy Support Vector Machine (FSVM), which is a machine learning algorithm that combines the power of support vector machines with fuzzy logic to handle uncertainty and imprecision in data classification tasks, with a focus on parallel processing using OpenMP on a multicore platform. The process includes dividing input images into sub-images, using template matching for initial classification, and refining results with the FSVM classifier. The template matching algorithm is parallelized to enhance computational efficiency. The results of the study indicate that the proposed parallel algorithm significantly improves computational performance compared to traditional serial algorithms. The main limitation of the proposed algorithm lies in its heavy computational burden, particularly evident in the use of FSVM combined with template matching for OD. This could impact its compatibility with general PC platforms.

Nonetheless, Elbahri et al. [5] proposed a novel approach to object detection by utilizing parallel processing. The approach was implemented on the PETS '09 dataset, which contains 795 frames recorded from a campus with ten personas having similar colors moving and walking in and out of the frame. Furthermore, the methodology used is the Orthogonal Matching Pursuit (OMP) algorithm for object recognition and tracing; each moving object is represented by a descriptor containing its appearance feature and its position feature. Afterward, the object is classified and indexed in a parallel manner on a GPU to speed up the computation time. Finally, the proposed parallel implementation has significantly reduced the computation time compared to the state-of-the-art methods, with a speedup of 100x, especially in crowded scenes. thus, making it suitable for real-time applications.

Similarly to [3], Rakhimov et al. [6] have utilized real-time object detection. this paper aims to use the OpenMP library to parallelize parts of the OD operations in the video stream, improving real-time processing speed in the absence of GPUs. Moreover, an eighteen-frames-per-second camera was employed in the study as an input dataset. Then, by utilizing the YOLO (You Only Look Once) algorithm method, the image is divided into grid systems, with each grid having the ability to identify items within itself. The YOLO algorithm was selected in this study due to its effectiveness in identifying objects. Consequently, using OpenMP technology greatly increases the processing speed for real-time object identification on multi-core CPUs. Additionally, it made it possible to process a large number of the camera's frames and instantly identify items in the matching image frame. Lastly, in real-time object detection, the paper does not discuss the trade-offs between processing speed and accuracy.

III. Literature Gap Analysis

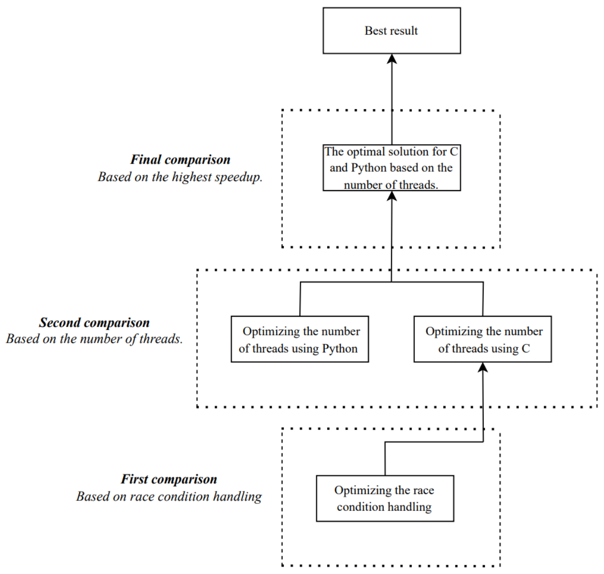
The study [2], operated on FPGA, which is a unique hardware architecture for real-time OD programmed for a specific task. On the contrary, [3], [6] utilized CPU for task parallelism in OD, while the [5] project utilized GPU. The choice of CPU or GPU depends on various factors, such as the specific task requirements, hardware availability, and performance goals. In the [6] paper, CPU was chosen due to its suitability for the task at hand, and the code was designed to effectively distribute the workload among multiple threads to achieve efficient and accurate object detection. Similarly to [6], the authors in [5] utilized a CPU along with a GPU to achieve computational efficiency in computer vision. Moreover, a literature gap emerges as these studies collectively lack a specific analysis of the programming languages C and Python regarding race conditions, thread management, and their influence on OD efficiency. The existing research focused on parallelization techniques, hardware architectures, and performance improvements but does not delve into language-specific considerations crucial for developers building OD models. Additionally, there is a deficiency in comprehensive comparisons between C and Python in terms of their impact on efficiency, runtime, and speedup. Bridging this gap, the project aims to provide practical recommendations for developers aiming to build OD models, outlining the advantages and limitations of using C or Python concerning race conditions, thread management, and overall performance.

IV. Details and justification of the parallel implementation

Task parallelism, as this project demonstrates, is a common technique for parallelizing the execution of tasks that could be divided into smaller, independent subtasks [7]. This is utilized in this work due to the code's ability to parallelize the detection of specific objects across several threads. Task parallelism is typically used to distribute the workload across multiple threads within a CPU. However, in some cases, task parallelism can also be used with GPUs to achieve even higher levels of parallelism, as explained in [2], [5]. Furthermore, the shared variables, such as (detected objects with race handling) and (confidence source with race handling), should each have a private copy of their own, so each thread completes a specific task and detects objects independently of the others. As a result of many threads executing the algorithms concurrently, the detection process can be completed in parallel. Through the effective parallel execution of the object detection task, this method ensures accurate results while dispersing the workload among multiple threads to improve performance.

V. Methodology

The methodology outlines the analysis for comparing programs implemented in Python and C in terms of evaluating speedup, efficiency, and runtime. Firstly, the C program is analyzed alone by implementing a critical compiler directive, reduction clause, or atomic compiler directive and comparing these solutions based on their runtime. Secondly, the best-performing C program's threads are tuned to further find the fastest-performing one. As for the Python program, the number of threads is changed to find the fastest-performing program. Finally, both the resulting C and Python programs from the previous analysis steps are compared as shown in the following figure.


Best result
Final comparison
Based on the highest speedup.
The optimal solution for C and Python based on the number of threads.
Second comparison
Based on the number of threads.
Optimizing the number of threads using Python
Optimizing the number of threads using C
First comparison
Based on race condition handling
Optimizing the race condition handling

Methodology flow.

Performance measures:

Speedup: The ratio of execution time of a sequential program to the execution time of a parallel program.

$$Speedup = \frac{Tsequential}{Tparallel} \tag{1}$$

Efficiency: The ratio of the speedup to the number of processors used.

$$Efficiency = \frac{Speedup}{NumberofProcessors} \tag{2}$$

Runtime: The total time taken by each parallel program to complete its execution.

VI. Comparison and analysis

The code contains two shared variables, (detected objects with race handling) and (confidence source with race handling). Since they are declared as a global variable and they are updated by the threads, both could result in race problems if several threads attempt to update the shared variables at the same time. Managing these race conditions effectively is essential to ensuring accurate and consistent results. Three measures were used to evaluate each approach's performance: run time, efficiency, and speedup. In the three performance measures, the run time calculates how long the program takes to run both with and without handling racing conditions. Furthermore, efficiency quantifies the effectiveness with which the available resources (threads) are employed. The speedup shows the difference in speed between the parallel and sequential parts of the program.

- Race condition handling analysis

This analysis examines three different strategies: reduction, critical section, and atomic with eight threads, in order to handle race conditions. The aim of these approaches is to provide synchronization algorithms that allow safe updates to shared variables in a parallel region. Thus, it can be concluded from these analytic statements that, when managing the race condition in the provided code, using the critical solution yields the best run time, efficiency, and speedup.

	Runtime	Speedup	Efficiency
Reduction	0.81s	1.31	0.16
Atomic	1.70s	0.95	0.12
Critical	0.82s	1.61	0.20

Race condition handling strategies

- Number of threads optimization analysis for C programming language

The critical solution yielded the best solution in comparison to the atomic and reduction solutions; thus, the next step of the analysis is to employ the critical solution for further assessments. Specifically, the number of threads is changed to 1, 2, 4, and 8 for the program. Obviously, the runtime for the sequential program that runs on one thread is seemingly less than the parallelized program due to the cost of thread creation and destruction. However, the performance measure is also based on the code's speedup too, hence, only the multi-threaded critical section programs will be analyzed in this stage. Firstly, the speedup of the programs that run on 2, 4, and 8 cores was calculated, and the efficiency was calculated. The findings indicate that the program that runs on 8 threads is the fastest, most efficient program with the highest speedup compared to the programs running on 2 and 4 threads.

	Runtime	Speedup	Efficiency
1 thread	0.89s	1.31	1.31
2 threads	0.66s	1.47	0.74
4 threads	0.69s	1.32	0.33
8 threads	0.82s	1.61	0.20

Number of threads optimization on C code

- Number of threads optimization analysis for python programming language

The lock constructed in Python code yielded the solution to the race condition; thus, the next step of the analysis is to employ the lock constructed for further assessments. Similarly to the C program, the number of threads is tested at 2, 4, and 8 for the lock solution. The obtained speedup for 2, 4, and 8 cores was calculated, and efficiency was also calculated. The results indicate that the program running on two threads achieves a notable speedup of 0.67 and an efficiency of 0.23, indicating effective parallelization. As the number of threads increases to 4 and 8, the speedup and efficiency decline, suggesting diminishing returns and potential overhead. From these observations, the program running on 2 cores achieves the fastest runtime, the highest efficiency, and the highest speedup compared to the programs running on 4 and 8 cores.

	Runtime	Speedup	Efficiency
1 thread	0.51s	-	-
2 threads	0.67s	0.45	0.23
4 threads	1.40s	0.19	0.05
8 threads	1.67s	0.12	0.01

Number of threads optimization on Python code

VII. Result and Discussion

In this analysis, an evaluation of how well an object detection program performed in the Python and C programming languages was examined. For C, OpenMP was used, and multiple approaches were provided to address the race condition, focusing on critical sections, reduction, and atomic operations. In Python, locks were used to handle the race condition. The study also tested the impact of varying the number of threads on the performance of both languages. After analyzing the results, it was determined that using the C programming language and employing a critical section with eight threads was the optimal strategy to address the race condition. Python performed the best with two threads out of all the tested values. The study used runtime, speedup, and efficiency as performance metrics. However, the main comparison was made based on speedup, as it is a vital factor in determining the effectiveness of parallelization strategies. In conclusion, C with OpenMP and critical sections demonstrated higher speedup than Python with locks in executing an object detection program and handling race conditions. The choice between Python and C depends on the specific requirements of the object detection task, considering factors such as resource availability, development ease, and performance.

	Number of threads	Runtime	Speedup	Efficiency
Python	2	0.67s	0.45	0.23
C	8	0.82s	1.61	0.20

Final comparison between Python and C, based on efficiency

VIII. Conclusion

In conclusion, a key component of computer vision is OD, which makes it possible to recognize and extract things from images or videos. In this project, a comparative analysis was carried out by executing the OD algorithm through both sequential and parallel programming. In the testing process, the algorithm was implemented using Python and C language to compare which language is a better option for parallelization. Moreover, in order to take advantage of parallel hardware design, parallel programming divides jobs into smaller pieces that can be processed concurrently. Moreover, the benefits of parallel programming were highlighted by the faster completion times achieved by parallel execution. Afterwards, the parallel C language code was compared using several methods for handling race conditions, such as reduction, atomic operations, and critical sections. Consequently, it has been found that the critical solution with eight threads produced the best outcomes through examination of speedup, and efficiency. Critical sections enable more complicated processes to be carried out safely by offering a higher degree of control and flexibility in managing access to shared variables. In addition, the Python code was tested with a comparison of the three metrics when handling race circumstances with a lock. The investigation showed how The C code performed better than the Python code, according to an analysis of the three performance measures that were carried out after testing the Python code and comparing the outcomes to those of the C language code.

References

[1] C. G. Kim, J. G. Kim, and D. H. Lee, "Optimizing image processing on multi-core cpus with Intel Parallel Programming Technologies," Multimedia Tools and Applications, vol. 68, no. 2, pp. 237–251, 2011. doi:10.1007/s11042-011-0906-y.

[2] Kyrkou, C. and Theocharides, T. (2011a) 'A flexible parallel hardware architecture for AdaBoost-based real-time object detection', IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 19(6), pp. 1034–1047. doi:10.1109/tvlsi.2010.2048224.

[3] B.-C. C. Lai, C.-H. Chiang, and G.-R. Li, "Data locality optimization for a parallel object detection on embedded multi-core systems," 2011 IEEE 2nd International Conference on Software Engineering and Service Science, 2011. doi:10.1109/icsess.2011.5982381

[4] Y. Ma, W. Wu, and Q. He, "Algorithm for Object Detection using Multi-Core Parallel Computation ." Elsevier Ltd. Selection and/or peer-review under responsibility of international ICMBPE committee, 2012.

[5] Elbahri, M., Taleb, N., Kpalma, K., and Ronsin, J. (2016). Parallel algorithm implementation for multi-object tracking and surveillance. IET Computer Vision, 10(3), 202–211. https://doi.org/10.1049/iet-cvi.2015.0115

[6] rakhimov2021parallel, Parallel implementation of real-time object detection using OpenMP, Rakhimov, Mekhriddin and Elov, Jamshid and Khamdamov, Utkir and Aminov, Shavkatjon and Javliev, Shakhzod,2021 International Conference on Information Science and Communications Technologies (ICISCT),2021,IEEE

[7] Streit, K., Doerfert, J., Hammacher, C., Zeller, A., and Hack, S. (2015). Generalized task parallelism. ACM Transactions on Architecture and Code Optimization, 12(1). https://doi.org/10.1145/2723164