MPCS-52060 – Project 3

Image Processing System Performance Experiment Report

1. Introduction

In this project, we created an image processing system, which can support four effects on the images: Blur, Edge Detection, Grayscale, and Sharpen. All the effects except the grayscale is implemented using the convolution calculation. We implemented three different versions for this image processing system. The basic one is the sequential version. In this version, all the images are processed sequentially without any parallel design. The second version is called "pipelines". In this version, we use channels and split the task to different stages. In this version, we processed the different images concurrently using the pipeline to decrease the processing time. We will introduce the details in next part. In the third and last version called "workStealing", we used circular array based dequeue structure to implement work stealing refinement. In this version, the thread can steal work from other thread when idle so that we can increase the efficiency of concurrency design and decrease risk of over-heading. To run different versions of the system, simply use command "**go run editor.go [file_size] [workStealing/pipelines] [threadNum]**". The command structure is as same as the project 1 with only different mode name. (workStealing, and pipelines).

2. Implementation Details

For the "pipelines" implementation, we divided the tasks in two three stages:

1. Create Tasks: in this stage, we will fetch the required data for each image task such as effect data, directories, then the tasks are stored in the "taskChan" Channel for further processing.
2. Process Image: in this stage, we will retrieve the task from "taskChan" Channel and then we will load and process the image. Finally, we will send it to "imgChan" Channel
3. Save Image: in this stage, we will retrieve the images from "imgChan" Chanel and store them using the output path.

All the channels are created with buffer size equals to input thread count size. When distributing the task, we divided the tasks equally using the number of availabilities per channel to ensure that each thread has approximately equal amount of work. We choose the pipeline design since we can easily divide the task to different stages. In the project 1, only the image processing part is implemented concurrently. By adopting the pipeline design, we can let each stage include the File I/O operate independently and concurrently so that the inefficient of one stage will not block the whole task. For example, if the image is ready in stage 1, it can immediately proceed to stage 2 without have to wait for other tasks, leading to increased efficiency. Moreover, the pipeline design can improve the scalability to manage heavy workloads. In the original "parfile" and "parslices" design, the File I/O and task creation can cost significant amount of time if the data volume is large. The pipeline design can increase the system's ability to handle demand spikes without affecting performance.

For the "workStealing" version, there are two important components:

1. We implemented the dequeue structure to support the work stealing. The algorithm of stealing and implementing the dequeue using circular array is from the work of David Chase & Yossi Lev (Dynamic Circular Work-Stealing Dequeue). We re-implemented the algorithm in Go. In this algorithm, we created the dequeue using a dynamic buffer which can shrink or expand based on the number of tasks and a circular array. By using the circular array, we can easily maintain the boundary of our array and resizing as needed. Firstly, we have a pushBottom function. When triggered, the function will add a new task to the bottom of the array. If the buffer is full, it will trigger the expansion of buffer to accommodate more tasks, greatly increased the scalability and space efficiency. Secondly, we have a popBottom function. This function will remove a task from bottom. The function will also shrink based on the number of works. Lastly, we have a steal function. The function will attempt to remove and steal the work from top of dequeue to support the stealing mechanism. We used atomic operation in the

dequeue structure to ensure that multiple threads can safely execute together. We also prevent using lock in this structure to prevent affecting performance of the whole system.

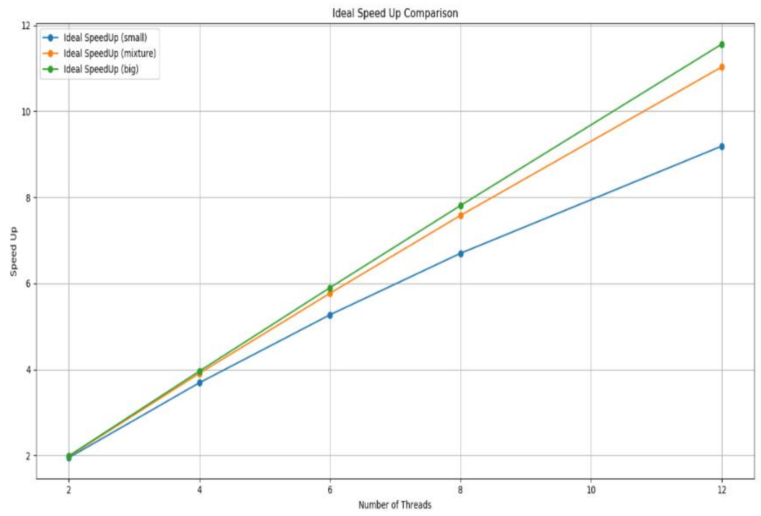
2. RunWorkStealing Function: This is function which distribute task and support work stealing mechanism. The function will let the ideal thread steal work from other thread using the data structure implemented above.

3. Experiment Design

We used three image sets with same content but different sizes for this performance experiment. The images are classified as small, mixture, and large. To test the performance of the image processing system, we performed five times of experiment by running the benchmark-proj3.sh test script: we observed and recorded the best result among the five data output. To run the test script, simply using the command: [sbatch benchmark-proj3.sh]. The main test logic is included in the performanceTest.py located in benchmark directory. For each experiment, the test script will run small, mixture, and big images in "pipelines" and "workStealing" mode using thread number from two to twlve. After all the experiment completed. All the performance data are recorded and included in the generated graph as the experiment output by the test script. To obtain and calculate the ideal speed up for the experiments, we recorded the sequential execution time and sequential part execution time of the program (the sequential part execution time is the execution time of the part of program before entering the pipelines.go or workStealing.go). This part of the program is expected to be fully sequential. We assume that the "pipelines.go" or "workStealing.go" part is fully parallelized and obtain the parallel fraction using the following formula:

$$\frac{\text{sequential } time - sequential\ part\ time}{\text{sequential time}}.$$

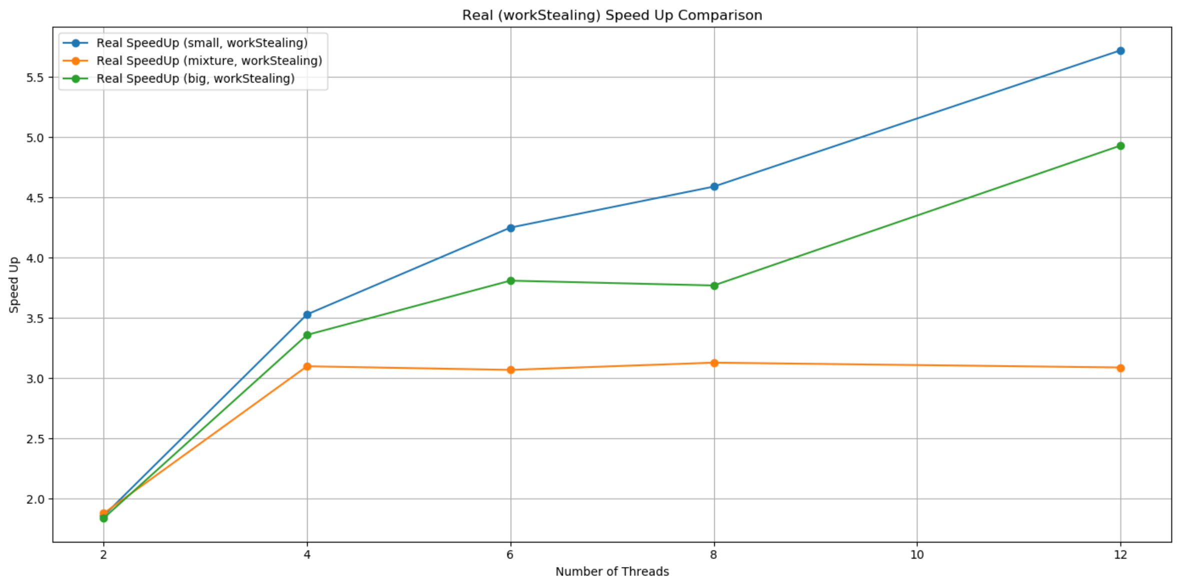
We then used Amdahl's law $\frac{1}{1-p+\frac{p}{n}}$ (for p is parallel fraction, n is thread numbers) to calculate the ideal speed up based on the parallel fraction obtained and thread numbers. All these steps are implemented automatically by the testing script. The following graph and table show the ideal speed up calculated for different experiment cases.

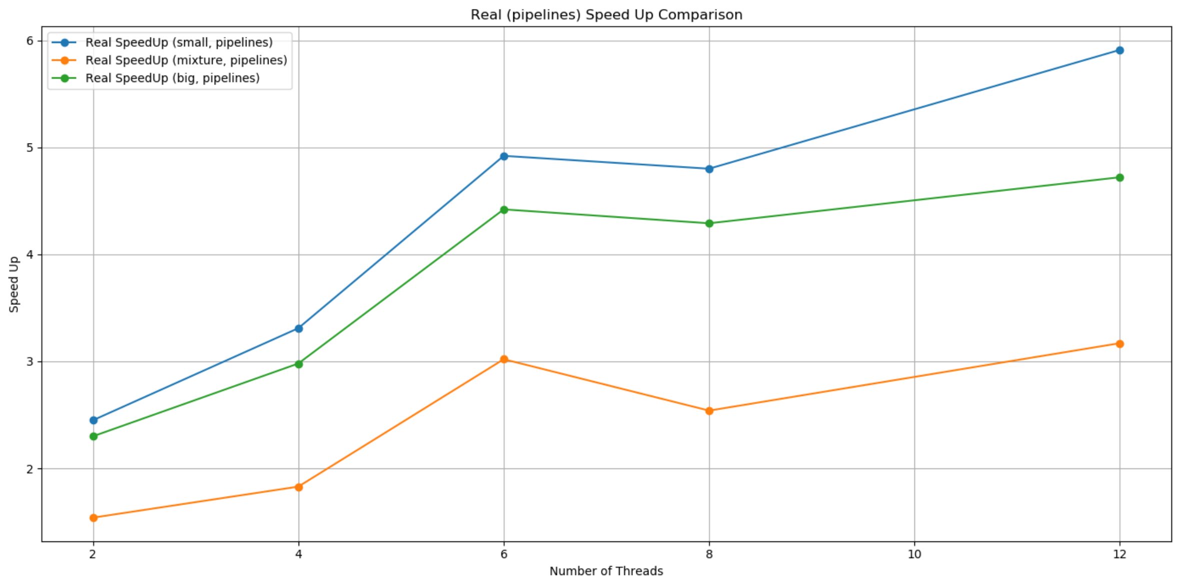


4. Experiment Result

The experiment result of the "pipelines" and "workStealing" mode cases are showed in the following graphs. From the graphs below, we can conclude that the size is a significant effect when performing parallel computing. The big size speed up is the most significant, following by the small size image set. The linear relationship between speedup and number of threads is less significant on mixture size image set.

This pattern appeared in both the "pipelines" and "workStealing" mode. Moreover, all the experiment cases have speed up less than the ideal speed up.





5. Experiment Analysis

From the experiment results, the possible reason causing the real speed up less than the ideal speed up is the parallel fraction. In the ideal situation, we assumed that the parallel part of program is full paralleled; however, in our program implementation, there is multiple part of the program is implemented sequentially. For example, in the work Stealing mode, we only implemented the concurrent design when processing the task, causing lower parallel fraction. All these sequentially implemented part in our parallel design is not calculated and included when obtaining the ideal speed up and calculate ideal parallel fraction, causing the real speed up significant less than the ideal speed up. There also exists some bottleneck and hotspot in the sequential part of the system. For example, when reading, loading, and saving the copy of image. All file I/O is implemented sequentially in the work stealing mode. The incorrect concurrency design of the File I/O could cause serious race condition or catastrophic incident to the system. We make the File I/O different stages in the pipeline mode and use channels to implement concurrency design. This is one of the reasons that the pipelines mode performs better in some testcases than the work stealing mode. We also note that size of pictures can affect the performance of parallel design. This is an expected result. The parallel design will effectively distribute the large amount of work when processing the large size image. However, when the thread numbers increase, each thread will still get enough work to process when dealing with the big image set, for the small image set, the same implementation may cause over heading leading to waste of computation power and decrease in performance. When dealing with mixture dataset, the system will have to frequently wait for the big size part of image to process; while the small size part of the image will still suffer from over heading, leading to a significantly decrease in performance. From the experiment result, we can see that the "pipelines" mode performed slightly better since the parallel fraction of this version is higher. Moreover, in the pipeline design, one task will not need to wait until other task to finish to enter the next stage. This can increase the performance and avoid possible over heading and bottleneck. Additionally, the use of channels in pipelines to manage data flow and task states helps in minimizing the delays associated with task handoffs and synchronization, which further enhances performance especially with larger and more complex datasets.

Overall, the pipelines approach provides a more continuous and managed workflow, which appears to be more suitable for tasks requiring complex and staged data processing like image processing workflows. The work-stealing approach, while highly flexible and dynamic, tends to suffer from efficiency losses due to the overhead associated with managing and balancing a diverse set of tasks across multiple workers.

6. Citations

Dynamic Circular Work-Stealing Dequeue by David Chase & Yossi Lev
https://www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-dequeue.pdf