Assignment 2

Due Feb 23 by 10pm **Points** 10

Assignment 2 - Parallel Data and Task Management

Due: Tuesday, February 23, at 10 p.m.

Points: 10

Deadline to find your group partner and create your group on MarkUs: Wednesday, February 10, at 10 pm. If you do not have a partner in MarkUs by this deadline, you will not be able to submit this assignment!

Intro to Scinet:

We will use Scinet for this assignment. You should have already received an email with your Scinet login. In that email you see a link to *The short intro to Scine*t document (the document is also uploaded in Quercus). You have to read that document before starting this lab. While you can use your own machine or a lab machine to debug the code, your final code has to be performance engineered and executed on Scinet. We will do all grading on Scinet. For each part, we do provide scripts to help with running your code on Scinet.

Overview

You have to work in groups of two for this assignment. Before you start, you must read section 7 from this pthreads tutorial **(https://computing.llnl.gov/tutorials/pthreads/)** from the Lawrence Livermore National labs. Your task is to implement a series of image processing methods, then parallelize them using data decomposition (partitioning) and the following parallel models: the data parallel model, and the work pool model.

Please log into MarkUs as soon as possible to find your repository and invite your partner, and make sure that you can commit and push to your repo. For all assignments and labs you will be given your repo URL on MarkUs. Make sure to use this repository (which should already be created for you), otherwise MarkUs won't know about it and we won't be able to see your work.

The starter code is available on */home/t/teachcsc367/CSC367Starter/assignments/assignment2/starter_code.tgz* so copy this into your repository and make sure you can do your first commit. Please make sure to read carefully over the code, including the licenses and the instructions from the comments. **To encourage you to avoid versioning things that you won't need to submit, we've included a ".gitignore" file for you in the starter code. Feel free to adjust it for your needs.**

Please note that this assignment is once again, meant to be practical and encourage your critical thinking ability. Therefore, the implementation component is not incredibly time consuming (although you do have to reserve reasonable time for it), but you will likely spend quite a bit of time in analyzing your findings and reasoning about your results.

Image representation

We will be working with images in the pgm format. Feel free to go over the image specification here **(http://netpbm.sourceforge.net/doc/pgm.html)** (don't read the "Plain PGM" section). You don't have to worry too much about the format, as we provide you with code that reads a pgm image and transforms it into an array of pixels in grayscale, and all of your work will be done with this array. If you are curious, this code is in the files *pgm.h* and *pgm.c*.

You will notice that images are represented as a 1-dimensional array in the code, even though intuitively they are 2-dimensional (why do we do this?). It is your job to calculate the corresponding offsets in the 1 dimensional array of pixels in the 2-dimensional image.

Image processing

Image processing techniques are exciting and useful in many situations: to obtain various artistic effects, sharpen blurry photos, perform edge detection, etc.

For this assignment, you will be working with monochrome (greyscale) images. Monochrome images are represented as a two-dimensional array of pixels. Each pixel is stored as a byte and encodes a greyscale color as a value between 0 and 255.

A discrete Laplace operator is used to compute the second derivatives of an image, which can emphasize edges within an image. This is useful in image processing for performing edge detection and various other related applications. The discrete Laplacian filter is a 3 x 3 array, which typically contains a high negative value at the center, surrounded by small positive values. Some variations include opposite signs, to achieve a similar effect.

Here are some examples of two Laplacian filters:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix} \text{ or } \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

A Laplacian filter can be applied to an existing image, by considering each pixel and its surrounding 8 neighbors and multiplying the 9-pixel values with the corresponding values in the Laplacian filter. The sum of the pairwise multiplications is used as the new value of that pixel.

Pixels on the edges and corners of the image are dealt with a bit differently since they do not have all 8 neighbors. Only the valid neighbors and the corresponding filter weights are factored into computing the new value of such a "marginal" pixel.

Intuitively, the Laplacian filter emphasizes sudden changes in pixel values. The weights in the Laplacian filter end up setting a darker tone for areas with low pixel changes, and contrasting white tones for sharp changes which represent edges.

Once a pixel value is updated, you might notice that the new pixel values may end up outside the [0,255] range. When processing an image, the new pixel values must be brought back to the original range by a process called **normalization**. This involves finding the minimum (Min) and maximum (Max), from the new (out-of-bounds) pixel values, then using these to normalize the pixel values back into the [0,255] range. For example, assume that the new pixel values are in the range [-100, 190]. In this case, you must add 100 to all pixels to bring all pixels in the range [0, 290]. Then, you must scale all pixels multiplying with 255/290, to bring all pixels in the [0,255] range.

Your program

You will implement a program which, given an image, it applies a discrete Laplacian filter to produce a new image. You will work with 4 different filters listed below; their respective matrices are already in the starter code provided (*filters.c*).

The program takes the following command line arguments:

- The input image: "-i input.pgm". This is an optional argument and can be ignored if the built-in parameter is used (see below).
- The built-in parameter: "-b image_number". This argument indicates that your program will be run with one of the hardcoded images we provided. The reason for using these images and how to generate them will be explained later, in the testing tips section. Use 1 for a big square image or 2 for a very tall (one column) image. Either -b or -i must be specified.
- The output image: "-o output.pgm". This is the resulting image after applying the filter to the input image (or the built-in image selected). This is an optional argument. If the output image is not specified, then no output image is produced. That is, the program still computes the resulting image in memory but does not write it to a file on disk and its contents will be lost once the program exits. The reason for this will be clear in the testing tips section.
- The number of threads: "-n num_threads". You must vary the number of threads between 1 and the number of cores available on the test machine.
- Timing enabled: "-t toggle_timing". Timing is enabled if toggle_timing is set to 1, and disabled if set to 0. As described later, you will need to time the execution time of your **image processing code**.

- The filter: "-f filter_number"
- The execution method: -m method_number". If the method is SEQUENTIAL, the num_threads parameter can either not be provided or simply ignored.
- The work chunk: "-c chunksize". This argument is optional and is only used when the execution method is WORK_QUEUE. This is described further in Part 3.

We are providing in the starter code a set of filters and run method macros which should be self-explanatory. The filter is one of a set of predefined filters (you will find the relevant macros in the starter code):

1. 3x3 Laplacian
2. 5x5 Laplacian
3. 9x9 Laplacian of Gaussian
4. 1x1 Identity

The method is the mode of execution, either sequential or one of several parallel strategies (described in Part 2 & 3):

1. SEQUENTIAL
2. SHARDED_ROWS
3. SHARDED_COLUMNS_COLUMN_MAJOR
4. SHARDED_COLUMNS_ROW_MAJOR
5. WORK_QUEUE

You will find an enumeration defining those methods in *filters.h*. We have given you code that handles the parsing of the command line arguments in the *main.c* file, so you don't have to change it.

Part 1 - Sequential implementation

For this part, you will be using the SEQUENTIAL method and the predefined filters. This task is as simple as implementing the function *apply_filter2d* in the file *filters.c*.

NOTE: you must especially test your sequential implementation **thoroughly**, with lots of corner cases and check against your own manual calculations on paper, to make sure that your code produces the correct image. Correctness is **crucial** here, especially if you re-use the sequential code for filtering pieces of the image in your parallel implementations!

Part 2 - Data Parallel implementation

You will now write a Data Parallel implementation using pthreads. Each thread is statically assigned a chunk of the data to process. You must partition the data in three ways:

- horizontal sharding (or row partitioning): each thread processes a set of consecutive rows, such that each thread gets a roughly equal number of rows. If the total number of rows is not divisible by the

number of threads, then each thread will be assigned nrows/nthreads rows, except possibly for the last thread, which may get assigned more rows.
- vertical sharding (or column partitioning): similar to horizontal sharding, but instead, each thread processes a (roughly) equal number of consecutive columns. For this partitioning, there are two processing methods: column major and row major. In column-major, a thread processes all the pixels in a column before moving on to its next assigned column. In row major, a thread processes first all the pixels from the first row of every column in its subset of columns, then moves on to the pixels from the second row of its columns, and so on.

This part consists of implementing part of the function *apply_filter2d_threaded* in *filters.c*.

For now, only handle the cases where the method parameter is one of: SHARDED_ROWS, SHARDED_COLUMNS_COLUMN_MAJOR, SHARDED_COLUMNS_ROW_MAJOR.

You must now profile your code and determine where most of the execution time is spent. If you have modularized your code well, it should be easy to prove that the code which does the image processing is where most of the computation time is spent.

Part 3 - Work Pool implementation

You will now write a Work Pool implementation using pthreads. This is specified as the WORK_QUEUE method number. In the WORK_QUEUE method, the image is divided in square tiles of size chunk x chunk (where chunk is the command line argument described as the work chunk earlier). All the tiles (work chunks) are statically placed in a queue at the start of the program. A task has the granularity of one tile. That is, each thread will take one tile at a time from the queue and process it before proceeding to grab another tile. You must implement this abstraction **efficiently**.

Note: Your implementation must ensure that accesses to shared resources are synchronized. Keep in mind that although your program may be run in sequential mode (that is, using a work pool with 1 thread), you should still use locking where necessary.

This part consists of implementing the remaining part of the function *apply_filter2d_threaded* in *filters.c*, in other words, you should handle the case where method is WORK_QUEUE.

General parallel guidelines

In the parallel implementations, you have to consider the fact that normalization cannot be done until all threads know the Min and Max pixel values. We suggest structuring your computation into the following steps:

- In the first stage, while applying the filters on a data partition, threads must also calculate a partial Min and Max pixel value from their assigned data partition. Consider having each of the threads store their partial Min and Max values into separate elements of a shared global array.

- Once the filter is applied by all threads and each thread has calculated their local Min and Max values, all threads must synchronize to ensure that everyone has finished this step. This can be achieved by using a *pthread_barrier* construct (check out the documentation for further details).

- Next, all threads calculate the global Min and global Max values using the other threads' local Min and Max. Once the threads all have the global Min and Max, the next step involves all threads performing the normalization on their assigned data partitions.

Part 4 - Data collection and analysis

Once you have implemented all the parts of the assignment and verified the correctness of the output, you must measure the time taken in each of the previous parts, and collect all the necessary data points. We have already provided time measurements in the starter code, inside main.c. It's up to you to design meaningful experiments using this timing functionality.

You should also collect other data using perf tools.

You must plot your results visually (graphs!) and analyze your results.

You must automate running the experiments, collecting the data and generating the graphs. You need to write a script (e.g. Python) that invokes your C program, performs all measurements, and produces all the graphs and other necessary data that you use to draw conclusions that you describe in your report. We have provided a sample script perf_student.py to get you started. You must also describe your data collection programs and scripts in your report.

*(Please note that the python scripts that help with generating the graphs is only made **available in the starter code after 6pm of Wednesday Jan 29th** to not overlap with the grace period of Assignment 1)*

In a report (you must name this file Report.pdf, see the following section), you must describe your implementation, your experiments and include the graphs, then analyze your results, draw conclusions and report your findings. It is important to show insight into what you are measuring and explain the performance of various strategies under certain conditions and inputs, explain why different methods perform in a certain way, etc. For example, you should discuss how well your algorithms scale and how various considerations discussed in class play a factor into the results. You should keep your report reasonably concise (e.g., 5-10 pages is ok, 40 pages is rather excessive), assuming a reasonable font size and readable figures.

To help guide your observations, we are recommending the following experiments and plotting the corresponding data points. You are welcome to run extra experiments and investigate further, but your report should at least describe your results and analyze your findings for the following experiments.

- Run the sequential algorithm as well as the parallel methods for a given image. For each run, vary the number of threads from 1 to the total number of cores by doubling the number of threads (i.e.,., 1, 2, 4, 8.). We want the experiments to be on up to 8 threads which is the number of physical cores. Consider checking the L1 cache misses and explaining why and how these factor into your results.

For the work queue implementation, the length of the tile is set to be equal to the number of threads used. That is, the chunk will be N x N, where N is the number of threads. Remember that your implementation uses synchronization regardless of N. In fact, observing the locking overhead when N=1 is an interesting aspect to consider.

- Test the work pool method for a variety of chunk sizes. For example, using N threads, you could measure the time depending on chunk size, by changing the chunk size: 1x1, 2x2, 4x4, 8x8, 16x16, 32x32, etc. Plot a separate line for each N between 1 and the number of cores on the same graph.
- Now vary the filter size for each of the methods (including the sequential one). Keep the number of threads constant to an N = 8, i.e. the number of physical cores, and (for work pool) use a chunk size of N x N. You may additionally vary these if you wish, or if you plan to gain further insights.
- Vary the images used for testing using the built-in ones, and any additional images you may wish to create yourself. Consider special cases that might help you gain certain insights into the differences between various methods (e.g., when would some of them likely perform better than others). Keep in mind that you must analyze your results in the report and discuss your findings.

Report

You must write a report documenting your implementation, displaying your results in a meaningful way, and analyzing your findings. Name it "Report.pdf". In your report, you should present your implementation, the experimental setup and results. You should discuss what you noticed, draw conclusions and analyze the tradeoffs, if any. The report should be written in a scientific manner (clear structure, clear description of your approach, results, findings, etc., and should use technical writing instead of colloquial terminology or phrases).

Keep in mind that presenting your experimental findings and observations to a technical audience is an important skill to develop as a computer scientist.

Testing tips

For this assignment, here are a few tips to help you get a better experimentation environment:

1. You should use the Scinet server to run your experiments.
2. A job file named with *run-job-a2.sh* is provided that you can use for running the *perf_student.py* on the Scinet compute node. You can execute *run-job-a2.sh* by typing:
   ./run-job-a2.sh.sh
   You should be able to see some graphs by running the scripts but the graphs will be correct once you finish the TODOs properly.
3. Measuring architectural events using perf tools cannot be done accurately for the part you intend to measure, if your program is also writing the output image to a file, or doing any serious IO (including printing timing measurements). This is why it is possible to disable writing the output to a file and collecting the timing. In other words, when using perf tools don't use -o and use -t 0. Consider

carefully what you are measuring and how your program's execution parameters can impact your tests.

4. Since reading an image from disk might affect your architectural counter measurements too, you should consider using the hardcoded images we provide as part of the starter code, or generate your own in a similar fashion. Read the *run-job-a2.sh* script in order to learn how to generate those images.
5. For each experiment, you must take the average of 10 runs. Measure the standard deviation of your timing results across the 10 runs, and (this takes a lot of work to code, so it is just a suggestion) consider adding error bars on your measurements, for clarifying if the difference between two test results is significant or whether your results are very noisy.
6. Use small scale experiments or contrived examples, if you must test a specific behaviour. Check the correctness of your code though (including when making substantial code changes), because an incorrect result will render any performance measurement meaningless. For instance, you can create a small image by hand, apply one filter by hand (it's just basic arithmetic operations, after all), and compare the result with what your code produces. **Again, you *must* check on paper on a small example that your sequential algorithm works correctly.**

Submission

You will submit your code on MarkUs under your *Assignment2* directory (do not create this manually, it should be created for you when you log into your MarkUs web interface). Be sure to make it clear how to run your code with various configurations, in your report! You must also submit any files required to build your program (including any header files (optional), a *Makefile* (mandatory!), etc.). Do not submit executables or object files!

Our autotesting script will invoke *make main* to test the correctness of your code, that is, if it produces the correct output image. In other words, *make main* MUST generate your binary file (the starter code already does that for you). Furthermore, you must provide a *make run* target (see the provided Makefile), which will generate ALL graphs included in your report. It is ok if *make run* takes roughly one hour to run.

You may NOT modify the *main.c* file, nor the *make main* or *make pgm_creator* targets in the Makefile. You must include at least the files *filters.c, filters.h, main.c, Makefile, pgm.c, pgm_creator.c, pgm.h*. An ideal submission would only modify *filters.c* and add the scripts for graph generation. Do NOT include the files related to the hardcoded images (*very_{big,tall}_sample.{c,h}*).

Aside from your code, you must submit the report documenting your implementation, presenting the results, and discussing your findings. When discussing your approach, feel free to also describe any problems encountered and workarounds, what isn't fully implemented (or doesn't work fully), any special design decisions you've taken or optimizations you made (as long as they conform to the assignment specs!), etc.

Additionally, you must submit an *INFO.txt* file, which contains as the first 2 lines the following:

- your name(s)
- your UtorID(s)

If you want us to grade an earlier revision of your assignment for whatever reason (for example, for saving some grace tokens if you had a stable submission before the deadline, tried to add new functionality after the deadline but broke your submission irreparably), then you may specify the git hash for the earlier revision you want marked.

As a general rule, by default we will always take the last revision before the deadline (or last one after the deadline, up to your remaining unused grace tokens), so you should **not** be including a line with the git commit hash, except in the exceptional circumstances where it makes sense. So in general, please avoid using this option and just make sure that the last revision (either before the deadline if you submit on time, or up to a subset of your remaining grace tokens if you submit late) is the one you want graded.

Finally, whether you work individually or in pairs with a partner, you **must** submit a *plagiarism.txt* file, with the following statement:

"All members of this group reviewed all the code being submitted and have a good understanding of it. All members of this group declare that no code other than their own has been submitted. We both acknowledge that not understanding our own work will result in a zero on this assignment, and that if the code is detected to be plagiarised, severe academic penalties will be applied when the case is brought forward to the Dean of Arts and Science."

A missing INFO.txt file will result in a 10% deduction (on top of an inherent penalty if we do not end up grading the revision you expect). **Any missing code files or Makefile will result in a 0 on this assignment**! Please reserve enough time before the deadline to ensure correct submission of your files. No remark requests will be addressed due to an incomplete or incorrect submission!

Again, make sure your code compiles without any errors or warnings.

Code that does not compile will receive zero marks!

Checklist

Make sure you have:

- Implemented the sequential version of the code and checked with a small example on paper to see if it produces the correct output pixel values.
- Implemented the parallel versions of the code:
  - Sharded rows
  - Sharded columns, column major
  - Sharded columns, row major
  - Work queue
- Written your make run target, which will invoke scripts that generate the graphs in your report.
- NOT modified make main and make pgm_creator.

- NOT modified main.c.
- Commited your:
  - source code files (including headers)
  - scripts for the graphs
  - Makefile
  - INFO.txt file
  - plagiarism.txt file
  - Report.pdf file

Marking scheme

We will be marking based on correctness (90%), and coding style (10%). Make sure to write legible code, properly indented, and to include comments where appropriate (excessive comments are just as bad as not providing enough comments). Code structure and clarity will be marked strictly!

Once again: code that does not compile will receive 0 marks! More details on the marking scheme:

- Sequential implementation: 5%
- Sharded parallel implementation: 30% (10% each)
- Work-queue implementation: 15%
- Report: 40% (including testing scripts)
- Code style and organization: 10% (code organization and modularity (if applicable), code readability, reasonable variable names, avoid code duplication, appropriate comments where necessary, proper indentation and spacing, etc.)
- **Negative deductions (please be careful about these!):**
  - **Code does not compile: -100%** for *any* mistake, for example: missing source file necessary for building your code (including Makefile, header files, etc.), typos, any compilation error, etc
  - **No plagiarism.txt file: -100%** (we will assume that your code is plagiarised and you wish to you withdraw your submission, if this file is missing)
  - **Missing or incorrect INFO.txt: -10%**
  - **Warnings: -10%**
  - **Extra output: -20%** (for any output other than what is required in the handout)
  - **Code placed in other subdirectories than indicated: -20%**