Programming Assignment #1: Basic Image Processing

Due Monday, April 24

In this assignment, you will implement three image analysis and manipulation routines. Of course, our interest is in analyzing the potential parallelism available in the application, not on the image analysis itself, so the algorithms are simplified. Programming assignments #2 and #3 will involve more substantial applications to stress your parallelizing skills.

For this assignment, you will be parallelizing each application with *both* pthreads and OpenMP. On future assignments, you may choose between the two, but for this assignment you must use both. In the process, you should get a working knowledge of both models and develop insight as to their pros and cons.

Make sure you read the syllabus regarding late days, honor code, etc. Note: you may work alone or in groups of two.

Deliverables

1. Code (25% of grade): To submit your code, run

```
/afs/ir.stanford.edu/class/cs315a/bin/submit pa1
```

from the directory containing your submission. All files in the current directory and any subdirectories will be copied to our repository. Please **do not submit our images** back to us; they are large and we already have them. Please include a `README` with the names and SUNet IDs of all group members and a description of the directory structure of your submission (what answers are where, etc.). You must **submit by 11:59 p.m.** on the due date.

2. Writeup (75%): Answer the questions in this document, making appropriate graphs. A significant part of parallel programming is the analysis of program performance, so after you write correct code, you usually need to spend just as long tuning the code for proper performance. We will rely upon your written report to grade your performance-tuning efforts and your analysis of the causes of any problems that may occur. For later programming assignments, much of this analysis should come from your insights about the hardware architecture as well as from the program code itself.

When reporting timing information, **include the input filename and machine name**.

Submit your writeup on paper in the homework box by Darlene's office, Gates 408 **(before 5 p.m.)** or electronically along with your code (PDF, PS, or HTML; no Word documents please). Please include names and SUNet IDs of all group members.

The Environment

You must compile and run your applications on Sun multiprocessor machines using both pthreads and OpenMP. At Sweet Hall, there are many such machines available: the `saga` (1–22) and `elaine` (1–43) machines all have two processors, the two `tree` machines have eight, and `junior` has 16. In order to keep the load on the larger machines reasonable as deadlines approach, we recommend that you compile and debug your applications on the 2-processor machines before moving to the larger machines only for final performance runs. `junior` has been reserved for exclusive use by another class from 6 p.m.–6 a.m., so you can only use it during the day.

Timing Runs

After debugging your programs on the smaller machines (`saga` and `elaine`), you'll need to acquire timed runs to compute speedups. Make sure you use the same machine for *all* runs of one problem (e.g., use the same machine for Problem 1B for 1, 2, 4, 8, and 16 processors). You may vary the machine between problems.

Use `junior` or `bigpun` for timing runs. `bigpun` has a batch system so only one program will be running at once; this makes the reported times more accurate. Watch for announcements regarding the use of `bigpun`.

Use the `Makefile.omp` and `Makefile.pthread` provided. You can change optimization values, but don't remove library or include information. In order to use the Sun compiler, you must set the appropriate environment variables by executing `source pa.csh` in csh or `. pa.bash` in bash, located in `/usr/class/cs315a/PA/`. If you are interested in reading more about the Sun compiler, we refer you to the man page or the manual, which is available on the internet:

Sweet Hall: `http://docs.sun.com/app/docs/coll/771.6`

If you choose to use other machines, with other compilers, the exact method of using these features may vary considerably. If you choose to use a different kind of machine, we will not support you with any unusual compiler setup that may be necessary.

We provide you with shell programs, `P1/pa1-pN.c`, which read in one or more JPEG files (specified from the command line) into memory buffers, call user functions, and then write selected buffers back out to one or more new JPEG file(s). The program also times the user function (over one or more runs) and reports the elapsed execution time, for performance evaluation. With this starting point, you should first write a sequential version of each program to use for timing runs. Then, parallelize your sequential code. Use the supplied JPEG photographs of varying sizes.

Execute the programs by typing

```
./pa1 <# runs> <# procs> <input.jpg> [input2.jpg] <output.jpg>
```

You can specify the number of times you want to run each command and the number of processors to be used. You can copy the necessary files from the following directory on the Leland system:

```
/usr/class/cs315a/PA/PA1/
```

A Note on Performance Times: Because of effects like interference from other users, OS paging, I/O, and the like, you should be very careful interpreting timing information. You can try making several runs until the average time per run stabilizes. Use the "# of runs" option to do this, or do it manually or with run scripts. This assignment's image analysis benchmarks lend themselves well to this kind of analysis, because runs should typically take few seconds to a few minutes, so multiple runs should still take a reasonable amount of time.

Problem 1. Blur Filter

For the first problem, you will perform a simple yet common image operation: blurring. This is making each pixel in the image a weighted average of itself and its neighbors. Edges are smeared out by this averaging process, resulting in a soft-focus effect. Each pixel of the output image is produced by the following formula three times for every pixel (once each for the red, green, and blue color components):

$$\text{output}(x,y) = \frac{\sum_{i=-(r-1)}^{r-1} \sum_{j=-(r-1)}^{r-1} \text{input}(x+i, y+j)(r - |i|)(r - |j|)}{\sum_{i=-(r-1)}^{r-1} \sum_{j=-(r-1)}^{r-1} (r - |i|)(r - |j|)}$$

Please note that the pixels are composed of three 8-bit unsigned chars (R, G, B), with a range of 0–255 each. Your intermediate results will require larger integers, however, to avoid overflow. Through the final division, your output pixels should also be confined to the 0–255 range, or you will get some strange results.

Here are some examples of radial blur matrices (note that the 0 case is not computed using the above formula):

$$r = 0, 1 \text{ uses } (1)$$

$$r = 2 \text{ uses } \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

$$r = 3 \text{ uses } \begin{pmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{pmatrix}$$

There are a couple of subtle details for you to think about. Note that the weights and final division factor can be pre-calculated before the loop to reduce work within the loop body itself. However, this doesn't work within r pixels of the edges of the image, when you will be summing over some pixels that don't exist.

For these pixels, you must only run the summations (in both numerator and denominator) over pixels that actually exist. As a result, the value of the denominator will be reduced near the edges of the image. We recommend you use small loops before or after your main filter loop specially optimized for these pixels.

1A) Parallelize the outer loop in both pthreads and OpenMP. Run both versions with a blur radius r = .05×max(picture height, picture width) pixels. In both versions, try blocking the parallel outer loop on an interleaved basis (divided one row/column at a time) and a chunked basis (divided up with blocks of 1/N of the rows/columns to each processor). While we recommend putting your pthread and OpenMP versions of the program in different files, these simpler variations can probably be handled easily using some strategic `#define`s. Measure and analyze the speedup on 2 processors for **papak.jpg,** Then, using the best performing OpenMP and pthreads versions, measure and analyze the speedup on 2 processors for **trythis.jpg and trythat.jpg** (for reasonable radii, use r=64 for both trythis and trythat). Which ones are better/worse, and why?

1B) Take the best blur filter implementation from 1A from each programming model and test it with 4, 8, and 16 processors **on papak.jpg**. Plot the speedup you obtain (Y) against the number of processors (X). How close is it to the "optimal" linear speedup? If it is poor, why?

Problem 2. Histogram Analysis

Another common image manipulation function is to count the number of pixels in an image that are light, dark, midtones, etc. This is useful in determining how under- or over-exposed a photograph might be, for example. For this, we need to accumulate the number of pixels that exist at each level. Accumulate four different values: a red histogram (0–255), green histogram (0–255), blue histogram (0–255), and sum histogram (R+G+B = 0–765) for each pixel. For a simple uniprocessor version of the code, you would store the values in histogram variables like these:

```
int rHist[256], gHist[256], bHist[256]; /* RGB histogram buckets */
int sHist[766];                         /* R+G+B histogram buckets */
```

Using the following code for each pixel:

```
rHist[pixel_red]++;
gHist[pixel_green]++;
bHist[pixel_blue]++;
sHist[pixel_red + pixel_green + pixel_blue]++;
```

When complete, you should save your results to a file with the following loop (which is commented out at the bottom of `pa1-p2.c`), running on one processor:

```
for (i = 0; i < 256; i++)
  fprintf(outputFile, "%3u: R:%8u G:%8u B:%8u S0:%8u S1:%8u S2:%8u\n",
    i, rHist[i], gHist[i], bHist[i], sHist[i], sHist[i+256],
    (i+512 < 776 ? sHist[i+512] : 0));
```

Use bigdaddy.jpg for timing runs.

2A) First allocate your four histograms as a shared memory object in both OpenMP and pthreads protected by locks for each histogram bucket; on groups of 32 histogram buckets; and on R, G, B, and sum histograms (for a total of 6 combinations). How well do they speed up on 2 processors? Which ones are better/worse, why? Pick the best version for each programming model and report and analyze speedups for 4, 8, and 16 processors.

2B) Now allocate private histograms for each processor with a final reduction to one set of histograms at the end (which you can choose to do serially or in parallel). (Note that locks are not necessary as only one processor will be working on its private histograms at a time.) How well does this speed up on 2, 4, 8, and 16 processors? Plot this with the best results from 2A on the same graph and contrast.

2C) Live dangerously by turning off your locks in the shared histogram code (if you did this right, you can just `#define` them to nothing!) and re-run the 2 processor runs without them. Do you notice much difference in the final results when compared with the correct results?

Problem 3. Variable Radius Blurring

You can use a blur filter to blur less interesting background information but leave the foreground material untouched. One way to do this is by using a variable-radius blur filter that averages more pixels in some parts of the image than others. For this purpose, we will read in two input files: the first is a standard RGB photograph, while the second is a grayscale “image” at the same resolution that stores the blur radii (in terms of 0–255 pixels, instead of brightness values) we would like to use for each corresponding pixel in the photograph. Just modify your code from Problem 1 to look up its blurring radius, r, at each pixel from the second “image” buffer, move the weight calculations into the loop (since they’re varying, it’s a lot harder to calculate them in advance), and you should have working code.

Use papak.jpg and papak-vrad.jpg for timing runs.

3A) Using the best static scheduling method from Problem 1 (i.e., interleaved vs. blocked), implement variable blurring on OpenMP and pthreads. Analyze the speedups for 2, 4, 8, and 16 processors.

3B) Now switch to dynamically tasked parallel loops. Using both OpenMP and pthreads, try having processors dynamically grab blocks of 1, 16, and 128 rows at a time. How well do these speed up on 2, 4, 8, and 16 processors? Plot this with the results from 3A on the same graph and contrast.