FIT 3042 System Tools and Programming Languages

Assignment 1

Parallel command-line image processing

Worth: 20% of final mark.

Must be completed individually

Hurdle: 40% of assigment marks across assignment 1&2.

Due: Monday 18th April 2011, 5PM.

In this project you will be performing simple image processing operations on grayscale pixmap images in PGM format.

As you probably know, there are two main ways in which images can be represented on a computer. The first, *vector* images, represent images as a collection of lines, shapes, and curves. "Raster" images, or "pixmaps", represent images as a two-dimensional array of "pixels". Each pixel represents one "dot" in the image, and contains a number (or array of numbers) representing the shade or colour of that pixel.

PGM file format

In this project, we will work with *grayscale* images that will be stored as files in the PGM image format. PGM is one of a related family of image formats initially used in the netpbm image processing tools. These tools which includes the libnetpbm library, which allow PGM images to be easily read and written to a file (amongst other things).

It is a very simple file format, as you can see from the example below of a PGM file, and the corresponding rendered image (enlarged):

```
P2
# feep.pgm
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

As you can see, a PGM file is simply a list of numbers representing the number of rows and columns in the image, the number of gray levels, and the pixel values starting from the top left.

Note that the “feep.pgm” is easily human-readable. This is because it is stored as the plain or “ASCII” variant of PGM. The *raw* version, rather than containing whitespace-separated ASCII characters which represent numbers, contains the raw data without whitespace separation. This is not human-readable but is much faster to read and write. Your code should read *both* binary (raw) and ASCII (plain) versions, and write the binary version. By far the easiest way to do this is to use libnetpbm.

Netpbm

Netpbm (http://netpbm.sourceforge.net/) is a huge collection of mainly command-line tools for manipulating pixmap graphics. It is designed very much in the Unix tradition - a collection of single-purpose tools which can write to files, or talk to each other by being “piped” together on the command line. While netpbm contains a wide variety of programs that can read and write most pixmap formats, for intermediate processing, netpbm uses three formats - PBM, PGM, and PPM. If you want (additional) test images, you can use the netpbm tools to convert an image of your choice into PGM format. If you need to convert files between plain and raw PGM, the tool “pnmtopnm” (included with netpbm) will do so (type man pnmtopnm at the command line for instructions). This may be useful if you need to inspect the output.

Obviously, code for reading and writing these files is shared amongst all the netpbm tools; this code is included in a simple, well-documented library called libnetpbm, which you can use for the programs you are to write. Libnetpbm is installed on ra-clay by default, and is packaged for most Linux distributions if you want to install it at home (you may need to install a package called libnetpbm-dev or similar).

A full description of the PGM file format is available by typing “man pgm” at the command prompt on ra-clay and other machines with netpbm installed.

Introduction to Gaussian Blurs

One of the basic operations that can be performed on a pixmap image is to blur it. Blurring can be useful in itself - for instance, judicious blurring of a photo of a face can remove pimples (for you) or wrinkles (for me). But it is also an essential building block in other image operations; for instance, a standard technique for *sharpening* images is the unsharp mask. To perform unsharp masking, first a blurred copy of the image is created. The blurred copy is then subtracted from the original image. The end result is a sharpened image!

The most common technique for digitally blurring an image is called a “Gaussian blur”.

Consider an image *I*, of width *w* and height *h*. *I(r,c)* refers to the pixel at row *r*, column *c* (assume that the rows start from the top and columns from the left, numbered at zero like c or java arrays). If *I’* represents *I* with the Gaussian blur applied, how do we calculate *I’(r, c)*?

Essentially, I’(r,c) is a weighted sum of the surrounding pixels, divided by the sum of the weights. The weights are stored in a *convolution matrix M* which is square, and its side length should be an odd number *m (think about it why it should be odd!!)*.

For example, consider I(1,4) from “feep.pgm”. The pixel, and surrounding pixels, are shown below:

		column		
		3	**4**	**5**
	0	0	0	0
row	**1**	3	3	0
	2	0	0	0

If we we want to apply the convolution matrix *M*

	0.05	0.1	0.05
M=	0.1	0.4	0.1
	0.05	0.1	0.05

Then,
I’(1,4) =

0.05 * 0 + 0.1 * 0 + 0.05 * 0 +

0.1 * 3 + 0.4 * 3 + 0.05 * 0 +
0 .05 *0 + 0.1 * 0 + 0.05 * 0 /
(0.05 + 0.1 + 0.05 + 0.1 + 0.2 + 0.1 + 0.05 + 0.1 + 0.05)

= 1.5 / 1
~= 2 (I don't mind which way you round in this circumstance!)

There are two additional problems you will have to deal with:

Problem 1: the pixel values should be integers between 0 and the maximum value specified for the original image.

Solution: You should round to the nearest integer, ensuring that it stays within the bounds.

Problem 2: what should you do when the pixels lay on the edge of the image, and therefore parts of the convolution matrix "overlay" are outside the image boundary.

You should calculate the weighted sum for the pixels that actually exist (within the image bounds), and calculate the sum of the weights in the convolution for only those pixels.

e.g For I(0,4) from feep.pgm,

I'(0,4) = 0.1 * 0 + 0.4 * 0 + 0.1 * 0 +
0.05 * 3 + 0.1 * 3 + 0.05 * 0 /
(0.1+ 0.4 + 0.1 + 0.05 + 0.1 + 0.05)
= 0.45 / 0.8
~= 0.56
~= 1.

In practice, a larger Gaussian Blur convolution matrix is generally used.

For this project, you should use the following matrix of size 7 (source: Wikipedia):

```
0.00000067  0.00002292  0.00019117  0.00038771  0.00019117  0.00002292  0.00000067
0.00002292  0.00078633  0.00655965  0.01330373  0.00655965  0.00078633  0.00002292
0.00019117  0.00655965  0.05472157  0.11098164  0.05472157  0.00655965  0.00019117
0.00038771  0.01330373  0.11098164  0.22508352  0.11098164  0.01330373  0.00038771
0.00019117  0.00655965  0.05472157  0.11098164  0.05472157  0.00655965  0.00019117
0.00002292  0.00078633  0.00655965  0.01330373  0.00655965  0.00078633  0.00002292
0.00000067  0.00002292  0.00019117  0.00038771  0.00019117  0.00002292  0.00000067
```

This is available from Moodle as part of the “assign1_resources.tgz” tar achive (see below).

Your tasks!

Step 1:
Write a program `pgmrotate`, which takes two command-line arguments `infile` and `outfile`.

`infile` is the name of a PGM file (may be binary or ASCII). `outfile` is the name of the output file, and should contain a well-formed PGM (binary) file which is the result of rotating the image in infile 180 degrees.

You may and are strongly recommended to use functions from libnetpbm, and the C and Unix standard libraries, but no other third-party libraries are to be used.

Maximum 8 marks.

Step 2:
Write a program `pgmblur`. `pgmblur` takes precisely two command line arguments, `infile` and `outfile` (and should exit with an error if two command line arguments are not present). `infile` is a name of a PGM file. `outfile` should be a PGM (binary) file that contains the result of applying the Gaussian blur convolution matrix supplied in the introduction.

You may (and should) reuse code from `pgmrotate`, either through copy-paste or (if you choose) a source file which contains shared functions that is linked into both executables.

You may and are strongly recommended to use functions from libnetpbm, and the C and Unix standard libraries, but no other third-party libraries are to be used.

Maximum 16 marks.

Step 3: Write a program `pgmblur2`, which performs exactly the same function as pgmblur. However, `pgmblur2` should spread the processing across two Unix processes. Half the rows (why rows?) should be handled by one process, half by the other. POSIX shared memory (as demonstrated in lectures) should be used for inter-process data sharing. Synchronization can be done using an appropriate mechanism - POSIX semaphores (as discussed in lectures) can be used, but may not be necessary (wait() may be enough for you).

You may use file handling functions from libnetpbm, and the C and Unix standard libraries (including librt, to use the POSIX semaphores) , but no other external libraries are to be used.

Once you have it working, compare the performance of your `pgmblur` and `pgmblur2` on the test images on this assignment page on Moodle. Write a short (250-500) word report (in text format) describing what you found. What was the performance difference? What did you expect, and if the difference was not as you expected, why? Would it be worth extending further to make the code run on more processors simultaneously, if they were available? Include your report as text as part of the README file (see below).

Maximum 20 marks

README file

You should prepare a short README file (in text format) containing:

Your name and student number.

A brief description of the stage you got up to in the project (for instance, if part 2 works, but part 3 compiles but does not run properly, say so).

A brief description of the coding standard you used (formatting etc). If you adopted an external coding standard (or parts of it), note that standard.

Your benchmarking report, if you completed stage 3 successfully.

Submission file

You must submit the necessary C source files, your README file, and a makefile that compiles and links your three executable files. Your Makefile must use gcc as your compiler, and compile all code using the -Wall compiler flag.

You may choose to compile your code as C99 if you wish (using the -std=c99 compiler flag).

If a part of your code does not compile (for instance, for stage 3), you should submit the source file **but modify your Makefile so that it does not attempt to compile that executable**.

Your code will be compiled and run for assessment on ra-clay, on which netpbm is already installed. You may develop your code on other machines, or your own personal machine, but you are responsible for fixing any incompatibilities!

Your programs should perform basic error checking expected of a command-line Unix utility, and fail gracefully (with an error message) if there is a problem. This includes if the input file is not a properly-formed PGM file!

You should submit a single compressed `tar` archive containing:

- your source code files
- your Makefile
- your README file
- nothing else

It should uncompress in the *current* directory (ie your tarfile shouldn't create a subdirectory).

If you add the following lines to your Makefile:

```
# "make clobber" removes executables and object files!!
clobber:
	rm -f pgmrotate pgmblur pgmblur2 *.o

# creates tarfile for submission.

dist:
	make clobber
	tar czvf ../3042_assignment1.tgz .
```

executing "`make dist`" in your assignment directory will create the appropriate tarfile in the directory *above* your assignment directory, which you can then submit.

If you want to use some other mechanism for creating your tar archive, read the documentation for tar.

To check that your archive contains the necessary files, move it to a newly created directory and type "t`ar xzvf assignment1.tgz`" at the command line, which will extract all the files.

Moodle submission process

You will submit your assignment online on the Moodle subject page. Remember, submit the tarfile containing ALL the files as above!

If you make a mistake with your submission, you can submit again.

You must also complete the "Assignment 1 electronic plagiarism statement" on Moodle. If you do not do this YOUR ASSIGNMENT WILL NOT BE MARKED!

Special consideration

If a student faces exceptional circumstances (serious illness or injury, family emergency etc) that prevent them completing the assignment, they may apply for special consideration according to the policy and procedure on the Faculty website:

http://www.infotech.monash.edu.au/resources/student/equity/special-consideration.html

Other Late submissions

Students must contact the Lecturer if they do not submit the assignment by the deadline. Penalties will apply.

Marking criteria

Your assignments will be marked on:

- completeness of submission.
- correctness of submission (program output, compliance with requirements as described in this document).
- Quality of implementation of the algorithm as described.
- Coding style (documentation, formatting, good coding practices).
- Quality of benchmarking report.

Sample files and matrix data:

There is a tar archive “assign1_resources.tgz” on the FIT3042 Moodle page which contains:

- matrix.txt - the convolution matrix you are expected to use, in text format (you will probably want to reformat to insert it in your program).
- several sample images, including one pair of images showing the result of blurring.

Obviously, you will want to download this!

Other resources:

The GNU Image Manipulation Program (http://www.gimp.org) is free, runs on Windows and Mac as well as Linux, and supports reading and writing PGM files. You may find it useful if you wish to create additional sample files. You can also create ASCII samples directly in a text editor. GIMP will also be very useful for viewing the outputs from your program, to make sure that it looks OK!