

Assignment 3 (Week 8-12: “Concurrent Filtering” on XC-1A board)

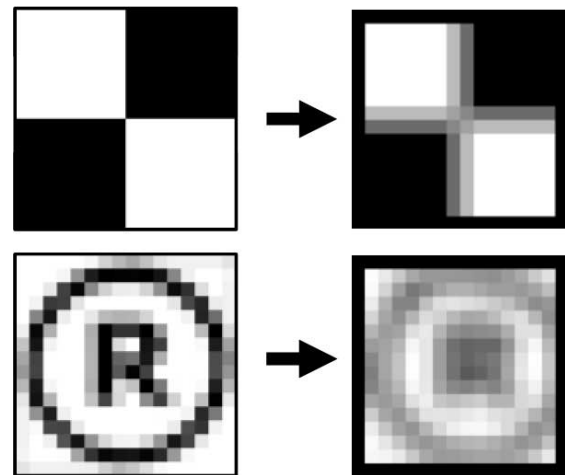
- This assignment is the last assessed piece of coursework in the unit.
- It is to be completed in pairs. (report any change to the team structure to the course director BEFORE starting your assignment)
- It is worth **30% of the unit mark** (i.e. 60% of the coursework component).
- **Submission:** Every student is required to upload their full piece of work (incl all XC source files and the concise PDF report) as a single **ZIP file to SAFE before 23:59:59, Mo 21st Jan 2013**. Make sure you submit it early enough (not last minute!) to avoid upload problems. (Each member of a team has to upload an identical copy of the teams work.)
- **Assessment:** You will present your submitted program (using XC-1A boards) in the labs on **Tue 22nd, Thu 24th and Fri 25th Jan 2013**. You will need to attend these lab sessions to get a mark. At these labs, we will ask you questions about your work and you will be able to showcase the merits of it.
- Do not attempt to plagiarise or copy code between teams etc. It is not worth it, be proud of your own work! We will ask you questions about your work in the labs - so you must understand the code your team developed in any case.

Your Task:

Introduction. Today, media streams (video, audio etc.) are routinely manipulated on-the-fly (e.g. decompressed, filtered, transcoded etc) in order to support versatile and responsive multimedia applications on devices ranging from mobiles to servers. To achieve sufficient performance, different parts of the stream are often processed simultaneously employing parallel hardware. Organising an efficient communication structure between the different functional parts of such a concurrent system is critical to provide smooth application performance...

Task. Your task is to design and implement a simple concurrent image filter on the **XMOS XC-1A board**. Your application should be able to blur grey scale images (a matrix of bytes) by farming the processing of different portions of the image out to different threads that perform the blur operation. Results are then collected from these worker threads and assembled into an output image stream.

An image is a matrix of values (representing ‘picture elements’ or ‘pixels’). Blur may be understood as replacing every pixel by the average value of the pixel and its eight neighbouring pixels in the image matrix. The effect of image blur is illustrated on two example images (the boundary pixels are set to 0 (black) since not all neighbouring pixels are defined):



Skeleton Code. To help you along, you are given a very simple code skeleton that:

- 1) reads an image (from a PGM image file) using a thread *DataInStream* that produces a stream of pixel values (describing the image matrix left to right, line by line) sent on an XC channel;
- 2) writes a stream of pixel values from an XC channel to an image (PGM image file) using a function *DataOutputStream*;

This example code operates on example images of size 16x16 pixels only. (It needs to be extended/changed to achieve high marks.) Feel free to change any of the skeleton code or add skeleton code from any previous assignment.

Process Control and User Feedback.

In your application use buttons on the XC-1A board for control:

Button A: starts reading and processing of an image

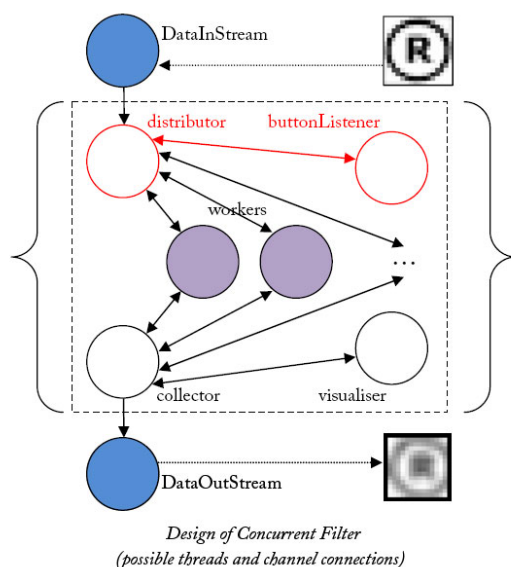
Button B: processing is paused and restarted by pressing Button B again

Button C: gracefully terminates your program (all threads shut down)

Use the LED clock on the board as visual feedback that indicates the progress of processing.

Building the Concurrent Filter.

To perform the filtering, your program should implement a 'distributor thread' that sends different parts of the image to different 'worker threads' that blur a part of the image and send the result to a 'collector thread' that assembles the processed parts of the final image. The illustration below visualises this, feel free to add threads or channels to augment the layout:



For the subsystem **distributor thread interacting with the buttonListener thread** (hiding all other events and interactions) provide a basic transition diagram and a CSP specification.

Your Report.

You need to submit a CONCISE (max 4 pages) report which should contain the following sections:

Functionality and Design

*Outline what functionality you have implemented, which problems are solved by your implementation and how your program is designed to solve the problems. Give a basic CSP specification of the **distributor thread interacting with the buttonListener thread** here (hiding all other system events and interactions).*

Tests and Experiments

Describe briefly the experiments you carried out, provide a selection of appropriate results and images. This must be done at least for the two example images provided and for an example of your own choosing (showing the merit of your system). List the important factors for virtues and limitations of your system as indicated by the results.

Critical Analysis:

Discuss the performance of your program with reference to the results obtained and indicate ways in which it might be improved. (Make sure your name, course, and email address appears on page 1 of the report.)

Further Task Details and Assessment Guideline.

The marker will take into account your lab presentation, your report and your source code. Find below a guideline for assessment. However, this is only a guideline and submissions will be marked on an individual basis.

To pass the assignment you need to implement a working, concurrent system that blurs an image and submit a report that discusses your implementation and results to a basic standard.


For a mark above 45 make sure you submit a clean, well documented piece of code that uses multiple worker threads.

For a mark above 50 make sure your CSP specification is well formed and reflects your actual subsystem.

For a mark above 55 also implement the correct button and LED behaviour as well as a graceful shutdown of all threads.

For a mark of merit above 60 also implement a system that can process images larger than 256x256 pixels.

For a mark of upper merit above 65 use a timer to measure the throughput/processing speed of your system (e.g. between entering the distributor and ready to leave the collector) when blurring an image 20 consecutive times (apply blur to already blurred image) and on different inputs (e.g. small vs. Large images, different image dimensions etc). Experiment with system parameters (e.g. synchronous vs. asynchronous channels etc) and draw conclusions about performance properties of your system in your report.

A top first class mark (70+) is reserved for excellence. Here we would look for work that implements a more complex system allowing, for instance, 1) dynamically adapting the strategy of distributing image parts to workers based on previous performance measurements, or 2) a theoretical extension where you show using CSP and FDR2 that larger parts of your system are deadlock free and livelock free, or your own ideas how this system can be turned into something much more sophisticated...

```

////////////////////////////////////
//
// COMS20600 - WEEKS 9 to 12
// ASSIGNMENT 3
// CODE SKELETON
// TITLE: "Concurrent Image Filter"
//
////////////////////////////////////

typedef unsigned char uchar;

#include <platform.h>
#include <stdio.h>
#include "pgmIO.h"

#define IMHT 16
#define IMWD 16

////////////////////////////////////
//
// Read Image from pgm file with path and name infname[] to channel c_out
//
////////////////////////////////////
void DataInStream(char infname[], chanend c_out)
{
    int res;
    uchar line[ IMWD ];

    printf( "DataInStream:Start...\n" );

    res = _openinpgm( infname, IMWD, IMHT );
    if( res )
    {
        printf( "DataInStream:Error openening %s\n.", infname );
        return;
    }

    for( int y = 0; y < IMHT; y++ )
    {
        _readinline( line, IMWD );
        for( int x = 0; x < IMWD; x++ )
        {
            c_out <: line[ x ];
            //printf( "-%4.1d ", line[ x ] ); //uncomment to show image values
        }
        //printf( "\n" ); //uncomment to show image values
    }

    _closeinpgm();
    printf( "DataInStream:Done...\n" );
    return;
}

////////////////////////////////////
//
// Start your implementation by changing this function to farm out parts of the image...
//
////////////////////////////////////
void distributor(chanend c_in, chanend c_out)
{
    uchar val;

    printf( "ProcessImage:Start, size = %dx%d\n", IMHT, IMWD );

    //This code is to be replaced - it is a place holder for farming out the work...
    for( int y = 0; y < IMHT; y++ )
    {
        for( int x = 0; x < IMWD; x++ )
        {
            c_in >: val;
            c_out <: (uchar)( val ^ 0xFF ); //Need to cast
        }
    }
    printf( "ProcessImage:Done...\n" );
}

```

```

////////////////////////////////////
//
// Write pixel stream from channel c_in to pgm image file
//
////////////////////////////////////
void DataOutputStream(char outfname[], chanend c_in)
{
    int res;
    uchar line[ IMWD ];

    printf( "DataOutputStream:Start...\n" );

    res = _openoutpgm( outfname, IMWD, IMHT );
    if( res )
    {
        printf( "DataOutputStream:Error opening %s\n.", outfname );
        return;
    }
    for( int y = 0; y < IMHT; y++ )
    {
        for( int x = 0; x < IMWD; x++ )
        {
            c_in :> line[ x ];
            //printf( "+%4.1d ", line[ x ] );
        }
        //printf( "\n" );
        _writeoutline( line, IMWD );
    }

    _closeoutpgm();
    printf( "DataOutputStream:Done...\n" );
    return;
}

//MAIN PROCESS defining channels, orchestrating and starting the threads
int main()
{
    char infname[] = "D:\\test.pgm"; //put your input image path here
    char outfname[] = "D:\\testout.pgm"; //put your output image path here
    chan c_inIO, c_outIO; //extend your channel definitions here

    par //extend/change this par statement to implement your concurrent filter
    {
        DataInStream( infname, c_inIO );
        distributor( c_inIO, c_outIO );
        DataOutputStream( outfname, c_outIO );
    }

    printf( "Main:Done...\n" );

    return 0;
}

```