COMS20001 Concurrent Computing (2014-15)

Kate Repejova, G403, kr13918@my.bristol.ac.uk,
Maria Marinova, G403, mm13354@my.bristol.ac.uk

Assignment 2 (XC-1A Game of Life Process Farm)
Report

The initial task of this assignment was to develop a small process farm on the XC-1A Development board which would simulate the 'Game of Life' on a provided image matrix by making use of the parallel system.

Functionality implemented:

Our system starts by reading the image from the pgm file in the provided DataInStream function. After the reading is completed, the program waits for a signal from the user to start the game.
Our design makes use of several functions named and described below.

- DataInStream
- DataOutStream
- distributor
- worker
- buttonListener
- visualiser
- showLED

DataInStream: reads the image from the pgm file and sends it to the distributor. Once it is finished, it terminates.

DataOutStream: receives the image from the distributor and outputs it to a pgm file.

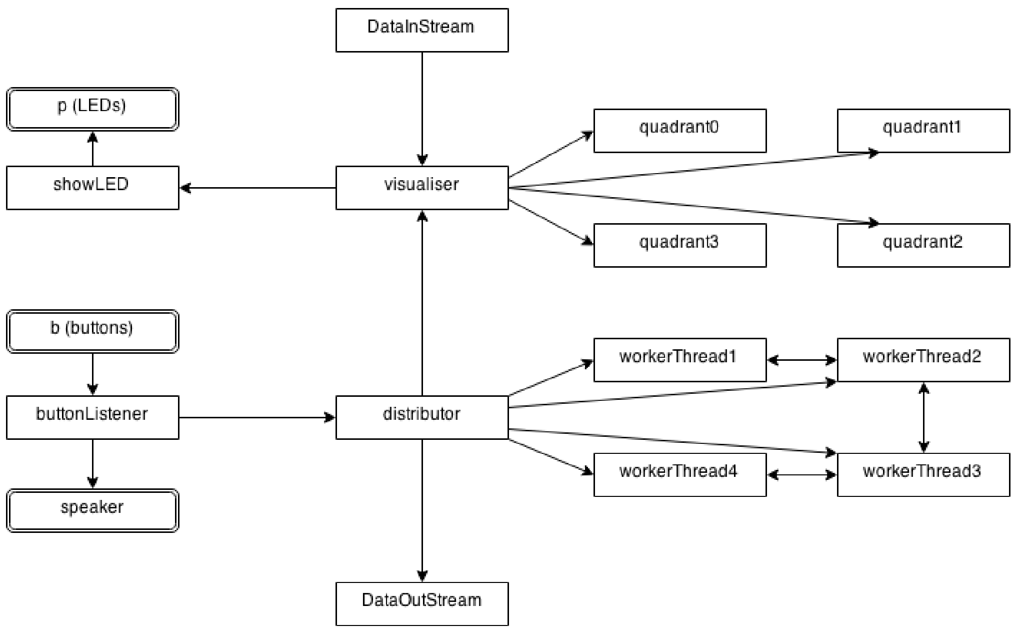
Distributor: Main controlling function. It assigns the parts of an image to workers. In our design, the distributor splits the image into 4 chunks and passes each chunk to a new worker. It also receives the button presses from buttonListener and based on those, tells other processes (visualiser,DataOutStream, workers) what to do.

Worker: Our design makes use of 4 workers to process the image.For the ease of differentiation, each worker has an ID assigned to them. Each worker receives their assigned chunk of the image from the distributor and with respect to their ID, exchanges their top and bottom lines with the previous and next workers accordingly (if such exist - for example, the first worker does not send his top line as there is no worker before him, same applies for the bottom line of the fourth worker). After the inter-worker line exchange is finished, the workers start to calculate flow in the chunks. After updating a chunk, the worker pauses and waits for the distributor to send it a signal to either terminate, send all the information back to the distributor for exporting, or to start another round.

ButtonListener: reads button presses and sends them to the distributor. In case a button to terminate is pressed, it sends the termination signal to the distributor and terminates itself.The button presses are as follows:
A - starts the game
B - pauses/unpauses the game
C - exports the current game state to a pgm file

D - terminates the game.
Visualiser: The visualiser receives information from the distributor and the DataInStream functions and based on it, controls whether to light up the four quadrants of the LED clock. While the game is running, the LEDs on the board visualise the number of cells that are currently alive. The number representation is done in binary numbers, with the least significant bit being displayed on the LED with the smallest value. If the game is paused, it displays the numbers of rounds played.

showLED: The showLED function constantly communicates with the visualiser - if the received information requires shutting them down, it would do so; otherwise it would send the pattern to the LED clock to light them up.



Communicating Sequential Processes (CSP)

The distributor and the button listener communicate via flag values: the game enters a state (start the game, pause/unpause the game, shut down the game or save image) depending on the pressed button, and sends it to the distributor. The distributor then acts as a farmer and splits the work between the four workers.

Process BUTTONS with alphabet α(BUTTONS) = {*start*, *save*, *pause*, *shutdown*} = B;
Events: start, save, pause, shutdown

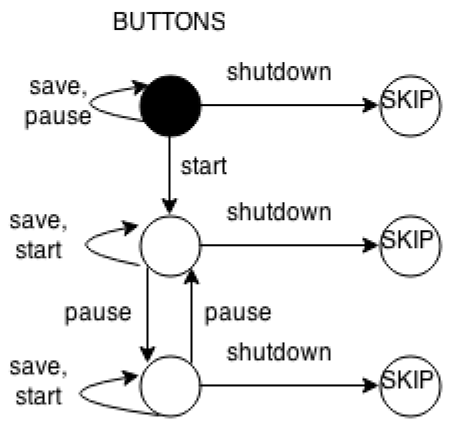
Process description:
BUTTONS = *start* -> RUNNING | *save* -> BUTTONS | *pause* -> BUTTONS | *shutdown* -> SKIP
RUNNING = *pause* -> PAUSED | *save* -> RUNNING | *start* -> RUNNING | *shutdown* -> SKIP
PAUSED = *pause* -> RUNNING | *save* -> PAUSED | *start* -> PAUSED | *shutdown* -> SKIP

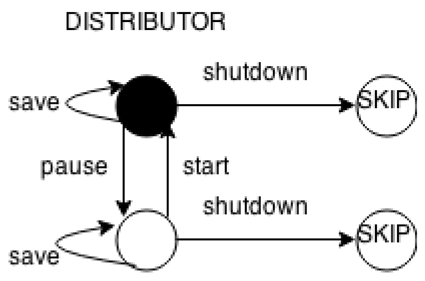
Process DISTRIBUTOR with alphabet α(DISTRIBUTOR) = {*start*, *save*, *pause*, *shutdown*} = B;
Events: start, save, pause, shutdown

Process description:
DISTRIBUTOR = *shutdown* -> SKIP | *save* -> DISTRIBUTOR | *pause* -> PAUSE
PAUSE = *shutdown* -> SKIP | *save* -> PAUSE | *start* -> DISTRIBUTOR

We have: BUTTONS $_B||_B$ DISTRIBUTOR

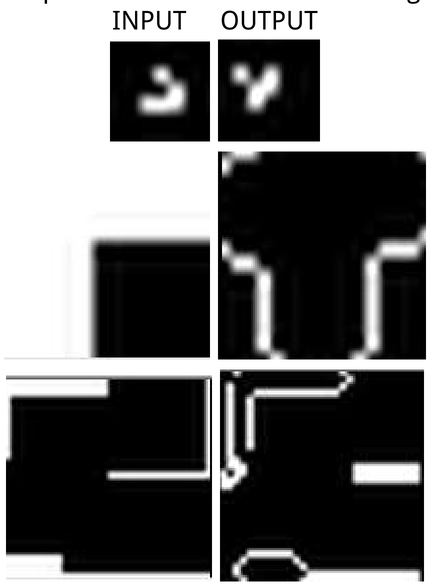
Some traces:
traces (*shutdown* -> SKIP) = { < shutdown , √ > }
traces (DISTRIBUTOR) = { < shutdown , √ >, < save , shutdown , √ >, < save , pause , start , shutdown , √ >, … }





Tests and experiments:

We carried out multiple experiments on the images provided as well as creating our own images for testing purposes. Here are some examples of the evolution of the image:



The main limitation we faced was memory constraint. Since storing the images externally would slow the program down, we are storing images on the board. On-board memory is limited, though, therefore we could only process images up to 360*360 pixels.

Critical analysis:

The performance of the program while running 100 consecutive rounds on a 16x16 pixel image was 250 ms. The biggest image that could be processed was about 360x360 pixels.

There is a number of functionalities that could have been improved and/or implemented to get a better result, and these are:

- for processing bigger images, we could have implemented compression, and/or store the images externally. This would, however, affect the speed of the processing and slow it down significantly (more than a thousandfold).
- another improvement would be to implement processing images that are not of size divisible by four. Currently, it is possible to process images of size that are multiples of 4 only.