Parallel Computing
Pthreads

May 23, 2017

Introduction

In this session we will implement a parallel version of an Optical Character Recognition (OCR) program using Pthreads. The parallelization will be achieved using a *task pool* approach. There is one week's time reserved for this. In addition, a *pipeline* approach, and a *work-crew* model will be discussed in this assignment.

The sequential program is entirely prepared, and available. So, you do not have to design and implement image processing algorithms. You should only be concerned with the parallelization of the available code and not with the details of the used signal processing algorithms. Nevertheless, it is a good idea to understand the image processing operations. The following section gives explanations on the sequential program.

OCR: The Sequential Program

The provided code is not intended to be a full-featured OCR program. It only works for non-proportional fonts, like the typewriter font (called `tt`-font in LaTeX). Please, keep in mind that you are doing an exercise in parallel computing, not in signal processing. Even if you want to improve the functionality of the program, so that for instance it can recognize other fonts, please, do not do that in the context of this course. Here, you need to focus on the parallelization.
OCR is the automatic translation of images of printed text (usually captured by a scanner) into machine-editable text. The images you will be working with were obtained by converting the pdf-output produced by LaTeX into images, so we do not have to bother about noisy images, or scanlines which are not perfectly horizontally aligned. The accurate recognition of Latin-script, typewritten text is now considered largely a solved problem. Typical accuracy rates exceed 99%, although certain applications demanding even higher accuracy require human review for errors.

The program can be compiled as follows

```
icc -Wall -o ocr ocr.c -lm
```

A collection of pgm-files is provided for this excercise. One of these is called `alphabet.pgm`. This file contains an image of the characters that will be recognized by the OCR system. Please do not change this file! All the other PGM-files are images of typewritten text. The files `vasalis-*.pgm` are poems from the poet M. Vasalis which (among others) can be found at `www.leestafel.info/Poezie`. These files were produced using LaTeX and a program called `pdftopgm` (with the option `-gray`). Also a large collection of image files with the filenames `asyoulikeit-*.pgm` is provided. It contains images of the complete transcription of the play "*As You Like It*" by William Shakespeare.

Compile the program and run it as follows:

```
ocr vasalis*.pgm
```

If everything works fine, you will get an ASCII transcription of the poems.

The structure of the program is as follows. Initially the alphabet image is read into memory and the characters are segmented into mask images. For each character a mask image is generated. These masks are used for pattern matching in the page images. The pipeline for processing the pages is depicted in Figure 1.

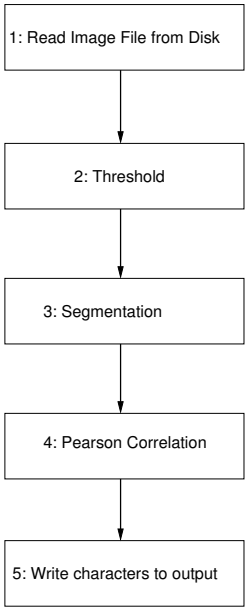

Figure 1: The processing pipeline of the OCR program.

1. `Reading images`: The pipeline starts by reading an input PGM image for each page of text that we want to transcribe. This is performed in the routine `readPGM`. It gets as its sole argument a file name, and returns a grayscale image. You do not have to understand or modify the code of this routine.

2. `Thresholding`: The grayscale image is converted into a binary image by thresholding. All pixel values in the input image that are smaller than a given threshold are replaced by a given small number, e.g. 0, in the output image. All pixel values in the input image that are larger than or equal to the given threshold are replaced by another, larger number, e.g. 1 or 255, in the output image. This is performed by the routine `threshold`. Its first parameter is the threshold value; we use 100. The second parameter is the value that, in the output image, replaces the pixel values of the input image that were smaller than the threshold. The third parameter is the value that replaces the pixel values that were greater than or equal to the threshold. The last parameter is the image on which we apply the operation.

3. `Segmentation`: This stage consists conceptually of two separate stages.The first stage is implemented by the routine `lineSegmentation`, which converts the binary image into a set of line strips. Line strips are images themselves. Each line strip contains the characters of a single line of text of the binary image. Next, for each line strip the routine `characterSegmentation` splits the line strip into separate characters. These characters are (again) small images. For technical reasons involving the correlator (next stage), the images are 'blurred' using a routine called `distanceBlur`. For each background pixel, it computes the distance to the nearest foreground pixel. This blurring makes the correlator less sensitive for small distortions (shape deformations) of the character images. These images are the input for the next stage.

4. `Pearson correlator`: The correlator gets as its input a blurred image and a set of template images (images of the characters that we want to recognize). The correlator tries to match the input character with all these templates, and returns the template with the best match. Correlators come in many flavors. The one that we chose for this program is called a *Pearson correlator*. The reason for this choice lies in the fact that such a correlator returns values between -1 (anti-correlated) and 1 (perfect match), which eases comparison. For each character the best match is returned to the next stage of the pipeline.

5. `Output`: The best matching ASCII characters are printed.

Exercises

Exercise 1: Pool of independent tasks.
In the `main` routine of the program you will find the loop

```
for (i=1; i<argc; i++) .....
```

This loop iterates over the image files. The body of this loop implements the image processing pipeline described in the previous section[1]. Make a thread routine that executes the body of this

[1]Notice that each image file can be processed independently of the other image files and that the same code is applied to all files. More generally, we can think of a problem in which some code has to be applied to different input

loop, and parallelize the loop by creating a thread for each iteration of the loop. All threads can in principle write output to the screen simultaneously, however this would result in chaos. Find a solution for this problem. Compare the runtime of the original sequential program with the runtime of your threaded version. Give an estimate of the efficiency of the threaded program.

Exercise 2.
Think about how you would achieve parallelization of this program using the Pipeline approach and the Working crew approach respectively. Record how many times the words '*master*' and '*slave*' appear in the play 'As you like it'.
On Nestor you can find the Assessment corresponding to this week's assignment. Answer and submit the questions based on what you have been able to conclude with the help of this assignment.

data. For example, SETI@home is a scientific experiment that uses Internet-connected computers in the Search for Extraterrestrial Intelligence (SETI). One can participate by running a free program that downloads and analyzes radio telescope data. Such a problem is usually referred as a 'pool of (independent) tasks' and it can be parallelized in a straightforward way.

Appendix: OCR code

```
/* file: ocr.c
 *
 * A simple OCR demo program.
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define FALSE 0
#define TRUE  1

#define FOREGROUND 255
#define BACKGROUND 0

#define MIN(a,b) ((a) < (b) ? (a) : (b))
#define MAX(a,b) ((a) > (b) ? (a) : (b))

#define CORRTHRESHOLD 0.95

/* Data type for storing 2D greyscale image */
typedef struct imagestruct
{
  int width, height;
  int **imdata;
} *Image;

int charwidth, charheight;  /* width an height of templates/masks */

#define NSYMS 75
Image mask[NSYMS];          /* templates: one for each symbol     */
char  symbols[NSYMS] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                       "abcdefghijklmnopqrstuvwxyz"
                       "0123456789-+*/,.?!:;'()";

static void error(char *errmsg)
{ /* print error message an abort program */
  fprintf (stderr, errmsg);
  exit(EXIT_FAILURE);
}

static void *safeMalloc(int n)
{ /* wrapper function for malloc with error checking */
  void *ptr = malloc(n);
  if (ptr == NULL)
  {
    error("Error: memory allocation failed.\n");
  }
  return ptr;
}

static Image makeImage(int w, int h)
{ /* routine for constructing (memory allocation) of images */
  Image im;
  int row;
  im = malloc(sizeof(struct imagestruct));
  im->width  = w;
  im->height = h;
  im->imdata = safeMalloc(h*sizeof(int *));
  for (row = 0; row < h; row++)
  {
    im->imdata[row] = safeMalloc(w*sizeof(int));
  }
  return im;
}

static void freeImage(Image im)
{ /* routine for deallocating memory occupied by an image */
  int row;
  for (row = 0; row < im->height; row++)
  {
    free(im->imdata[row]);
  }
  free(im->imdata);
  free(im);
}

static Image readPGM(char *filename)
{ /* routine that returns an image that is read from a PGM file */
  int c, w, h, maxval, row, col;
  FILE *f;
  Image im;
  unsigned char *scanline;

  if ((f = fopen(filename, "rb")) == NULL)
```

```
  {
    error("Opening of PGM file failed\n");
  }
  /* parse header of image file (should be P5) */
  if ((fgetc(f) != 'P') || (fgetc(f) != '5') || (fgetc(f) != '\n'))
  {
    error("File is not a valid PGM file\n");
  }
  /* skip commentlines in file (if any) */
  while ((c=fgetc(f)) == '#')
  {
    while ((c=fgetc(f)) != '\n');
  }
  ungetc(c, f);
  /* read width, height of image */
  fscanf (f, "%d %d\n", &w, &h);
  /* read maximum greyvalue (dummy) */
  fscanf (f, "%d\n", &maxval);
  if (maxval > 255)
  {
    error ("Sorry, readPGM() supports 8 bits PGM files only.\n");
  }
  /* allocate memory for image */
  im = makeImage(w, h);
  /* read image data */
  scanline = malloc(w*sizeof(unsigned char));
  for (row = 0; row < h; row++)
  {
    fread(scanline, 1, w, f);
    for (col = 0; col < w; col++)
    {
      im->imdata[row][col] = scanline[col];
    }
  }
  free(scanline);
  fclose(f);
  return im;
}

static void writePGM(Image im, char *filename)
{ /* routine that writes an image to a PGM file.
   * This routine is only handy for debugging purposes, in case
   * you want to save images (for example template images).
   */
  int row, col;
  unsigned char *scanline;
  FILE *f = fopen(filename, "wb");
  if (f == NULL)
  {
    error("Opening of file failed\n");
  }
  /* write header of image file (P5) */
  fprintf(f, "P5\n");
  fprintf(f, "%d %d\n255\n", im->width, im->height);
  /* write image data */
  scanline = malloc(im->width*sizeof(unsigned char));
  for (row = 0; row < im->height; row++)
  {
    for (col = 0; col < im->width; col++)
    {
      scanline[col] = (unsigned char)(im->imdata[row][col] % 256);
    }
    fwrite(scanline, 1, im->width, f);
  }
  free (scanline);
  fclose(f);
}

static void threshold(int th, int less, int greater, Image image)
{ /* routine for converting a greyscale image into a binary image
   * (in place) by means of thresholding. The first parameter is
   * the threshold value, the  second parameter is the value in the
   * output image for pixels that were less than the threshold,
   * while the third parameter is the value in the output for pixels
   * that were at least the threshold value.
   */
  int row, col;
  int width=image->width, height=image->height, **im=image->imdata;
  for (row = 0; row < height; row++)
  {
    for (col = 0; col < width; col++)
    {
      im[row][col] = (im[row][col] < th ? less : greater);
    }
  }
}

static void distanceBlur(int background, Image image)
```

```
{ /* blur the image by assigning to each pixel a gray value
   * which is inverse proportional to the distance to
   * its nearest foreground pixel
   */
  int width=image->width, height=image->height, **im=image->imdata;
  int r, c, dt;
  /* forward pass */
  im[0][0] = (im[0][0] == background ? width + height : 0);
  for (c = 1 ; c < width; c++)
  {
    im[0][c] = (im[0][c] == background ? 1 + im[0][c-1] : 0);
  }
  /* other scanlines */
  for (r = 1; r < height; r++)
  {
    im[r][0] = (im[r][0] == background ? 1 + im[r-1][0] : 0);
    for (c = 1; c < width; c++)
    {
      im[r][c] = (im[r][c] == background ?
                  1 + MIN(im[r][c-1], im[r-1][c]) : 0);
    }
  }
  /* backward pass */
  dt = im[height-1][width-1];
  for (c = width-2; c >= 0; c--)
  {
    im[height-1][c] = MIN(im[height-1][c], im[height-1][c+1]+1);
    dt = MAX(dt, im[height-1][c]);
  }
  /* other scanlines */
  for (r = height-2; r >= 0; r--)
  {
    im[r][width-1] = MIN(im[r][width-1], im[r+1][width-1]+1);
    dt = MAX(dt, im[r][width-1]);
    for (c = width-2; c >= 0; c--)
    {
      im[r][c] = MIN(im[r][c], 1 + MIN(im[r+1][c], im[r][c+1]));
      dt = MAX(dt, im[r][c]);
    }
  }
  /* At this point the grayvalue of each pixel is
   * the distance to its nearest foreground pixel in the original
   * image. Also, dt is the maximum distance obtained.
   * We now 'invert' this image, by making the distance of the
   * foreground pixels maximal.
   */
  for (r = 0; r < height; r++)
  {
    for (c = 0; c < width; c++)
    {
      im[r][c] = dt - im[r][c];
    }
  }
}

static double PearsonCorrelation(Image image, Image mask)
{ /* returns the Pearson correlation coefficient of image and mask. */
  int width=image->width, height=image->height;
  int r, c, **x=image->imdata, **y=mask->imdata;
  double mx=0, my=0, sx=0, sy=0, sxy=0;
  /* Calculate the means of x and y */
  for (r = 0; r < height; r++)
  {
    for (c = 0; c < width; c++)
    {
      mx += x[r][c];
      my += y[r][c];
    }
  }
  mx /= width*height;
  my /= width*height;
  /* Calculate correlation */
  for (r = 0; r < height; r++)
  {
    for (c = 0; c < width; c++)
    {
      sxy += (x[r][c] - mx)*(y[r][c] - my);
      sx  += (x[r][c] - mx)*(x[r][c] - mx);
      sy  += (y[r][c] - my)*(y[r][c] - my);
    }
  }
  return (sxy / sqrt(sx*sy));
}

static int PearsonCorrelator(Image character)
{
  /* returns the index of the best matching symbol in the
   * alphabet array. A negative value means that the correlator
```

```
   * is 'uncertain' about the match, but the absolute value is still
   * considered to be the best match.
   */
  double best = PearsonCorrelation(character, mask[0]);
  int sym, bestsym = 0;
  for (sym=1; sym<NSYMS; sym++)
  {
    double corr = PearsonCorrelation(character, mask[sym]);
    if (corr > best)
    {
      best = corr;
      bestsym = sym;
    }
  }
  return (best >= CORRTHRESHOLD ? bestsym : -bestsym);
}

static int isEmptyRow(int row, int col0, int col1,
                      int background, Image image)
{ /* returns TRUE if and only if all pixels im[row][c]
   * (with col0<=c<col1) are background pixels
   */
  int col, **im=image->imdata;
  for (col = col0; col < col1; col++)
  {
    if (im[row][col] != background)
    {
      return FALSE;
    }
  }
  return TRUE;
}

static int isEmptyColumn(int row0, int row1, int col,
                         int background, Image image)
{ /* returns TRUE if and only if all pixels im[r[col]
   * (with row0<=r<row1) are background pixels
   */
  int row, **im=image->imdata;
  for (row = row0; row < row1; row++)
  {
    if (im[row][col] != background)
    {
      return FALSE;
    }
  }
  return TRUE;
}

static void makeCharImage(int row0, int col0, int row1, int col1,
          int background, Image image, Image mask)
{ /* construct a (centered) mask image from a segmented character */
  int r, r0, r1, c, c0, c1, h, w;
  int **im = image->imdata, **msk = mask->imdata;
  /* When a character has been segmented, its left and right margin
   * (given by col1 and col1) are tight, however the top and bottom
   * margin need not be. This routine tightens the top and
   * bottom marging as well.
   */
  while (isEmptyRow(row0, col0, col1, background, image))
  {
    row0++;
  }
  while (isEmptyRow(row1-1, col0, col1, background, image))
  {
    row1--;
  }
  /* Copy image data (centered) into a mask image */
  h = row1 - row0;
  w = col1 - col0;
  r0 = MAX(0, (charheight-h)/2);
  r1 = MIN(charheight, r0 + h);
  c0 = MAX(0, (charwidth - w)/2);
  c1 = MIN(charwidth, c0 + w);
  for (r=0; r<charheight; r++)
  {
    for (c=0; c<charwidth; c++)
    {
      msk[r][c] = ((r0 <= r) && (r < r1) && (c0 <= c) && (c < c1) ?
                   im[r-r0+row0][c-c0+col0] : background);
    }
  }
  /* blur mask */
  distanceBlur(background, mask);
}

static int findCharacter(int background, Image image,
          int row0, int row1, int *col0, int *col1)
```

```
{ /* find the bounding box of the left most character
   * with column>=col0 in the linestrip given by row0 and row 1.
   * Note that col0 is an input/output parameter, while
   * col1 is a strict output parameter.
   * This routine returns TRUE if a character was found,
   * and FALSE otherwise (at end of line).
   */

  /* find fist column which has at least one foreground (ink) pixel */
  int width=image->width;
  while ((*col0 < width) &&
         (isEmptyColumn(row0, row1, *col0, background, image)))
  {
    (*col0)++;
  }
  /* find column which is entirely empty (paper) */
  *col1 = *col0;
  while ((*col1 < width) &&
         (!isEmptyColumn(row0, row1, *col1, background, image)))
  {
    (*col1)++;
  }
  return (*col0 < *col1 ? TRUE : FALSE);
}

static void characterSegmentation(int background,
                                  int row0, int row1, Image image)
{ /* Segment a linestrip into characters and perform
   * character matching using a Pearson correlator.
   * This routine also prints the output characters.
   */
  int match, col1, prev=0, col0=0;
  Image token = makeImage(charwidth, charheight);
  while (findCharacter(background, image, row0, row1, &col0, &col1))
  {
    /* Was there a space ? */
    if (prev > 0)
    {
        int i;
        for (i = 0; i < (int)((col0-prev)/(1.1*charwidth)); i++)
        {
          printf (" ");
        }
    }
    /* match character */
    makeCharImage(row0, col0, row1, col1, background, image, token);
    match = PearsonCorrelator(token);
    if (match >= 0)
    {
      printf ("%c", symbols[match]);
    } else
    {
      printf ("#%c#", symbols[-match]);
    }
#if 1
    {
      /* Mark bounding box of character.
       * This can be turned on for debugging pruposes.
       * In combination with writePGM it is possible
       * to write the 'segmnented' image to a file.
       */
       int **im=image->imdata;
       int r, c;
       for (r = row0; r < row1; r++)
       {
         for (c = col0; c < col1; c++)
         {
           if (im[r][c] == background)
           {
             im[r][c] = 128;
           }
         }
       }
   }
#endif
    col0 = prev = col1;
  }
  freeImage(token);
  return;
}

static int findLineStrip (int background, int *row0, int *row1,
                          Image image)
{ /* find the first line strip that is encountered
   * when parsing scanlines from top to bottom starting
   * from row0. Note that row0 is an input/output parameter, while
   * row1 is a strict output parameter.
   * This routine returns TRUE if a line strip was found,
```

```
   * and FALSE otherwise (at end of page).
   */
  int width=image->width, height=image->height;
  /* find scanline which has at least one foreground (ink) pixel */
  while ((*row0 < height) &&
         (isEmptyRow(*row0, 0, width, background, image)))
  {
     (*row0)++;
  }
  /* find scanline which is entirely empty (paper) */
  *row1 = *row0;
  while ((*row1 < height) &&
         (!isEmptyRow(*row1, 0, width, background, image)))
  {
    (*row1)++;
  }
  return (*row0 < *row1 ? TRUE : FALSE);
}

static void lineSegmentation(int background, Image page)
{ /* Segments a page into line strips. For each line strip
   * the character recognition pipeline is started.
   */
  int row1, row0=0, prev=0;
  while (findLineStrip(background, &row0, &row1, page))
  {
    /* Was there an empty line? */
    if (prev > 0)
    {
        int i;
        for (i = 0; i < (int)((row0 - prev)/(1.2*charheight)); i++)
        {
          printf ("\n");
        }
    }
    /* separate characters in line strip */
    characterSegmentation(background, row0, row1, page);
    row0 = prev = row1;
    printf ("\n");
  }
#if 0
  /* You can enable this code fragment for debugging purposes */
  writePGM(page, "segmentation.pgm");
#endif
}

static void constructAlphabetMasks()
{ /* Construct a full set of alphabet symbols from the file
   * 'alphabet.pgm'. The symbols are represented in the
   * template images mask[].
   */
  int row0, row1, col0, col1, sym;
  Image alphabet = readPGM("alphabet.pgm");
  threshold(100, FOREGROUND, BACKGROUND, alphabet);

  /* first pass: only used for computing bounding box of largest
   * character in the alphabet. */
  charwidth = charheight = 0;
  row0 = sym = 0;
  while (sym < NSYMS)
  {
    if (!findLineStrip (BACKGROUND, &row0, &row1, alphabet))
    {
      error("Error: construction of alphabet symbols failed.\n");
    }
    charheight = MAX(charheight, row1 - row0);
    col0 = 0;
    while ((sym < NSYMS) &&
       (findCharacter(BACKGROUND, alphabet, row0, row1, &col0, &col1)))
    {
      /* we found a token with bounding box: (row0,col0)-(row1,col1) */
      charwidth = MAX(charwidth, col1-col0);
      sym++;
      col0 = col1;
    }
    row0 = row1;
  }

  /* allocate space for masks */
  for (sym=0; sym<NSYMS; sym++)
  {
    mask[sym] = makeImage(charwidth, charheight);
  }
  /* second pass */
  row0 = sym = 0;
  while (sym < NSYMS)
  {
    findLineStrip (BACKGROUND, &row0, &row1, alphabet);
```

```
    col0 = 0;
    while ((sym < NSYMS) &&
       (findCharacter(BACKGROUND, alphabet, row0, row1, &col0, &col1)))
    {
       makeCharImage(row0, col0, row1, col1,
                     BACKGROUND, alphabet, mask[sym]);
#if 1
       {
         /* You can enable this code fragment for debugging purposes */
         char filename[16];
         sprintf(filename, "template%02d.pgm", sym);
         writePGM(mask[sym], filename);
       }
#endif
      sym++;
      col0 = col1;
    }
    row0 = row1;
  }
  freeImage(alphabet);
}

int main(int argc, char **argv)
{
  int i;
  Image image;

  if (argc < 2)
  {
    fprintf(stderr, "Usage: %s page1.pgm page2.pgm...\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  /* process alphabet image */
  constructAlphabetMasks();

  /* process pages */
  for (i = 1; i < argc; i++)
  {
    printf ("\n--%s----------------------------------\n", argv[i]);
    image = readPGM(argv[i]);
    threshold(100, FOREGROUND, BACKGROUND, image);
    lineSegmentation(BACKGROUND, image);
    freeImage(image);
  }

  /* clean up memory used for alphabet masks */
  for (i = 0; i < NSYMS; i++)
  {
    freeImage(mask[i]);
  }
  return EXIT_SUCCESS;
}
```