Report for computer exercise 1 - Filter in MPI

Anton Karlsson
antka388
931217-7117

1 Code description

An over-view of the two filters used in lab. In both filters, MPI was intiliasized and the master process read the input image, which could then be scattered to the other processes in the communication world. However, they where sent in some different ways due to data dependecies.

Let n denoted the problem size (i.e the number of pixels), and p the number of processors used.

1.1 Blur filter

In the blur filter, each process needed to have access to pixels in a radius around the chunk of its imgae that have been sent from the master process. Along with this and the fact that the image was read as long string, rather than a 2 by 2 matrix, Each process recvied n/p number of rows of pixels that is to be written by the process, along with some extra `radius` number of rows of pixels, from above and below. Note that the processes that had chunks of the middle part of the image recived extra pixels from above and below chunks of images.

After that, each process could apply the blurfilter on its private chunk of pixels, and then the master process gather the result and computed the ouputfile.

1.2 Threshold filter

The threshold filter did not need the overlap of pixels, so the master process can simply scatter n/p rows of pixels to each process. Then, each process can comute the pixel mean from its chunk. After that, the master process perfomes a reduce communication with where it compute the mean for each process, thus obtaining an global threshold parameter, which can brodacsted to all process, which they can use to compute the threshold filter on there individual chunk. From there, individual chunks is gather by the master process, which can then create the output file.

2 Results

The results are splited up into to subsection; one covering the elapsed times and comparing the time when using more processors, the other section covers the images that was produced.

2.1 Computational results

A comparision of effecitenty of the number of processors used is shown in Table 1 for the blur filter, and Table 2 for the threshold filter. The problem size, i.e. the number of pixels used, is denoted by n, and the number of processors used is denoted by p.

p \ n	515788		
1	1.68212		
2	0.933425		
4	1.1206		

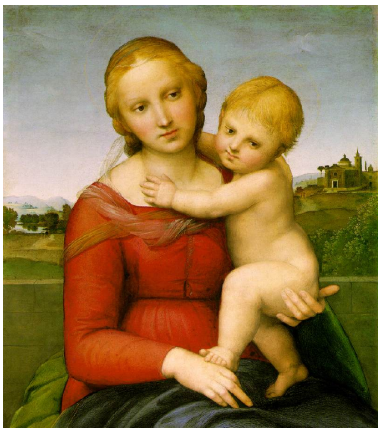
Table 1: Elapsed time when diferrent number of processors was used.

p \ n	515788		
1	1.68212		
2	0.933425		
4	1.1206		

Table 2: Elapsed time when diferrent number of processors was used for the threshold filter

2.2 Filter results

The result of using `im1.png` is shown in Figure 1.



(a) Before filtering.

(b) After the blurfilter with radius $r = 50$.

(c) After the threshold filter.

Figure 1: Image result using the threshold filter and the blurfilter (with a radiues $r = 50$) on `im1.png`. Before and after

Code

Listing 1: blurmain.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stddef.h>
#include <time.h>
#include <unistd.h> // getpid
#include "ppmio.h"
#include "blurfilter.h" //gets also struct pixel
#include "gaussw.h"
#include "mpi.h"
#include <math.h> //ceil()
#include "subroutines.h"

int main (int argc, char ** argv) {
  int radius;
  int xsize, ysize, colmax, yfrom, yto, recvcount, sendcount;
  pixel *localsrc, *src, *sendbacksrc;

  const int root = 0;
  //pixel src[MAX_PIXELS];
  /* struct timespec stime, etime; */
#define MAX_RAD 1000

  double w[MAX_RAD];

  /* Take care of the arguments */

  MPI_Init(&argc, &argv);


  //Get rank of process
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int np;
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  double starttime;

  if (rank == 0){
    starttime = MPI_Wtime();
  }


  int scounts[np],  displs[np], filtercounts[np], sendbackdispls[np];

  //Create a pixel MPI struct
  MPI_Datatype mpi_pixel; /* MPI_Struct of three members */
  const int nitems = 3; /* Decalration of parts for the actual declaration of mpi_pixel
      below */
  int blocklengths[3] = {1,1,1};
  MPI_Datatype types[3] = {MPI_CHAR, MPI_CHAR, MPI_CHAR};
  MPI_Aint offsets[3];
```

```
  offsets[0] = offsetof(pixel, r);
  offsets[1] = offsetof(pixel, g);
  offsets[2] = offsetof(pixel, b);
  MPI_Type_create_struct(nitems, blocklengths, offsets, types, &mpi_pixel);
  MPI_Type_commit(&mpi_pixel);

#ifdef DEBUG
  // DEBUGGING
  {
    int i = 0;
    char hostname[256];
    gethostname(hostname, sizeof(hostname));
    printf("PID %d on %s ready for attach\n", getpid(), hostname);
    fflush(stdout);
    while (0 == i)
      sleep(1);
  }
#endif

  /* read file */
  if (rank == root)
    {
      if (argc != 4) {
        fprintf(stderr, "Usage: %s radius infile outfile\n", argv[0]);
        exit(1);
      }
      radius = atoi(argv[1]);
      if((radius > MAX_RAD) || (radius < 1)) {
        fprintf(stderr, "Radius (%d) must be greater than zero and less then %d\n",
            radius, MAX_RAD);
        exit(1);
      }

      src = (pixel*) malloc(MAX_PIXELS * sizeof(pixel));
      if(read_ppm (argv[2], &xsize, &ysize, &colmax, (char *) src) != 0)
        exit(1);
      if (colmax > 255) {
        fprintf(stderr, "Too large maximum color-component value\n");
        exit(1);
      }

      int len = xsize*ysize*sizeof(pixel);
      src = realloc(src, len);
      if (src == NULL)
        {
          fprintf(stderr,"Pixel was not allocated correctly (== NULL)\n");
        }

      //Chop up the scr file into np equizised pices
      chopup(np, xsize, ysize, radius, scounts, displs,
             filtercounts, sendbackdispls);

      get_gauss_weights(radius, w);
    }

  //MPI_Bcast(&chunkSize, 1, MPI_INT, root, MPI_COMM_WORLD);
```

```
MPI_Bcast(&xsize, 1, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&ysize, 1, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&radius, 1, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&scounts, np, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&displs, np, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&filtercounts, np, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&sendbackdispls, np, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&w, sizeof(w)/sizeof(w[0]), MPI_DOUBLE, root, MPI_COMM_WORLD);

recvcount = scounts[rank];
localsrc = (pixel *) malloc(scounts[rank] * sizeof (pixel));

/* if (rank == root) fprintf(stderr,"Process %d is sending data to all other processes\
    n", root); */
//Might be sending to much data, but should not matter.
if (MPI_Scatterv(src, scounts, displs, mpi_pixel, localsrc,
                 recvcount, mpi_pixel, root, MPI_COMM_WORLD) != MPI_SUCCESS) {
  /* fprintf(stderr, "FAILURE!\n"); */
}/*  else { */
/*   fprintf(stderr, "SUCCESS!\n"); */
/* } */


yfrom = 0;
yto = scounts[rank]/xsize;

/* fprintf(stderr, "Process %d is calling filter\n", rank); */
/* clock_gettime(CLOCK_REALTIME, &stime); */
blurfilter(xsize, yfrom, yto, localsrc, radius, w);
/* clock_gettime(CLOCK_REALTIME, &etime); */

/* printf("Filtering took: %g secs for processs %d\n", (etime.tv_sec  - stime.tv_sec) +
    */
/*     1e-9*(etime.tv_nsec  - stime.tv_nsec), rank) ; */

//Return result from slaves to master

if (rank == 0)    {
    sendbacksrc = localsrc;
  }
else if (rank > 0)     {
    sendbacksrc = localsrc + radius*xsize;
  }

MPI_Barrier(MPI_COMM_WORLD);
sendcount = filtercounts[rank];
MPI_Gatherv(sendbacksrc, sendcount, mpi_pixel,
            src, filtercounts, sendbackdispls, mpi_pixel,
            root, MPI_COMM_WORLD);

/* write result */
if (rank == root)
  {
    printf("Writing output file\n");

    if(write_ppm (argv[3], xsize, ysize, (char *)src) != 0)
      exit(1);
```

```
    }

  if (rank==0){
    double endtime = MPI_Wtime();
    printf("Elaplsed time: %g np: %d: num.pixels: %d\n",
           endtime-starttime, np, xsize*ysize);
  }

  //Free the MPI_Data_types created.
  free(mpi_pixel);
  mpi_pixel = NULL;

  //Free pointers
  free(src);
  src = NULL;
  //  free(sendbacksrc);// can not free
  free(localsrc);
  localsrc = NULL;

  MPI_Finalize();

  return(0);
}
```

Listing 2: thresmain.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <stddef.h>
#include "mpi.h"
#include "ppmio.h"
#include "blurfilter.h"
//#include "thresfilter.h"
#include <math.h>

int main (int argc, char ** argv) {
  int xsize, ysize, colmax, root = 0, numrow, rowrest;
  uint sum, i, psum, nump, globalsum;
  pixel *src, *localsrc;
  //pixel src[MAX_PIXELS];
  struct timespec stime, etime;

  int np, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  int scounts[np], displs[np];

  //Create a pixel MPI struct
  MPI_Datatype mpi_pixel; /* MPI_Struct of three members */
  const int nitems = 3; /* Decalration of parts for the actual declaration of mpi_pixel
      below */
  int blocklengths[3] = {1,1,1};
  MPI_Datatype types[3] = {MPI_CHAR, MPI_CHAR, MPI_CHAR};
  MPI_Aint offsets[3];
  offsets[0] = offsetof(pixel, r);
  offsets[1] = offsetof(pixel, g);
  offsets[2] = offsetof(pixel, b);
  MPI_Type_create_struct(nitems, blocklengths, offsets, types, &mpi_pixel);
  MPI_Type_commit(&mpi_pixel);

  /* Take care of the arguments */
  if (rank == root)
    {
      if (argc != 3) {
        fprintf(stderr, "Usage: %s infile outfile\n", argv[0]);
        exit(1);
      }

      src = (pixel *) malloc(sizeof(pixel) * MAX_PIXELS);

      /* read file */
      if(read_ppm (argv[1], &xsize, &ysize, &colmax, (char *) src) != 0)
        exit(1);

      numrow = ceil(ysize/(double)np);
      rowrest = np*numrow % ysize;
```

```
      src = realloc(src, sizeof(pixel) * xsize*ysize);


      if (colmax > 255) {
        fprintf(stderr, "Too large maximum color-component value\n");
        exit(1);
      }

      //chop up
      for (i = 0; (int) i < np-1; ++i)
        {
          scounts[i] = xsize*numrow;
          displs[i] = scounts[i]*i;
        }
      {
        scounts[np-1] = xsize*(numrow - rowrest);
        displs[np-1] = scounts[0]*(np-1);
      }

    }

  MPI_Bcast(&xsize, 1, MPI_INT, root, MPI_COMM_WORLD);
  MPI_Bcast(&ysize, 1, MPI_INT, root, MPI_COMM_WORLD);
  MPI_Bcast(&numrow, 1, MPI_INT, root, MPI_COMM_WORLD);
  MPI_Bcast(&rowrest, 1, MPI_INT, root, MPI_COMM_WORLD);
  MPI_Bcast(&scounts, np, MPI_INT, root, MPI_COMM_WORLD);
  MPI_Bcast(&displs, np, MPI_INT, root, MPI_COMM_WORLD);

  localsrc = (pixel *) malloc(sizeof(pixel) * scounts[rank]);
  MPI_Scatterv(src, scounts, displs, mpi_pixel,
               localsrc, scounts[0], mpi_pixel, root, MPI_COMM_WORLD);

  if (rank == 0)
    printf("Has read the image, calling filter\n");

  clock_gettime(CLOCK_REALTIME, &stime);

  //thresfilter(xsize, ysize, src);

  if (rank == np-1)
    {
      nump = xsize * (numrow - rowrest);
    }
  else
    {
      nump = xsize * numrow;
    }

  // Do this for all processes, then send result and calculate global avg
  for(i = 0, sum = 0; i < nump; i++) {
    sum += (uint)localsrc[i].r + (uint)localsrc[i].g + (uint)localsrc[i].b;
  }

  sum /= nump;

  MPI_Reduce(&sum, &globalsum, 1, MPI_UNSIGNED,
             MPI_SUM, root, MPI_COMM_WORLD);
```

```
  globalsum /= np;

  MPI_Bcast(&globalsum, 1, MPI_UNSIGNED, root, MPI_COMM_WORLD);

  // Do this again for all processes
  for(i = 0; i < nump; i++) {
    psum = (uint)localsrc[i].r + (uint)localsrc[i].g + (uint)localsrc[i].b;
    if(globalsum > psum) {
      localsrc[i].r = localsrc[i].g = localsrc[i].b = 0;
    }
    else {
      localsrc[i].r = localsrc[i].g = localsrc[i].b = 255;
    }
  }

  clock_gettime(CLOCK_REALTIME, &etime);

  printf("Filtering took: %g secs for process %d\n", (etime.tv_sec  - stime.tv_sec) +
         1e-9*(etime.tv_nsec  - stime.tv_nsec), rank) ;

  //gather local src:es to src.
  MPI_Gatherv(localsrc, scounts[rank], mpi_pixel, src, scounts, displs,
              mpi_pixel, root, MPI_COMM_WORLD);

  if (rank == root)
    {
      /* write result */
      printf("Writing output file\n");

      if(write_ppm (argv[2], xsize, ysize, (char *)src) != 0)
        exit(1);
    }

  //free pointers
  free(src);
  src = NULL;

  MPI_Finalize();

  return(0);
}
```