Jonathan Armknecht
CS 324 Sec 2
OpenMP Tasks Short Report

Report

I messed around with where to put the task. I at first tried to have calculating a single pixel as a task but this was much slower than having a task be a row. So I set up the master task to run through the columns and make tasks for the other threads to go through the rows. I have to say I was a bit confused at first because I did not think we had to make a producer-consumer relationship and just run the rows as tasks in the parallel omp call. This made the task code significantly slower than my parallel for loop. I felt like I must be doing something wrong because people on Slack were saying that they were getting about the same times as their parallel for loop. So, I went down to see Brian and he told me that it needed to be a producer-consumer relationship otherwise the point of using tasks was completely wasted.

It was seen that when running both versions on the lab machines that the parallel for loop was slightly better or was the exact same for some number of threads. With my parallel for loop the times in seconds were 44, 23, 17, and 16 for 1, 2, 4, and 8 threads respectively. For my task version the times in seconds were 44, 23, 17, and 16 for 1, 2, 4, and 8 threads respectively. I know it was not needed to run it for 1 thread but I wanted to see how close it was to the parallel for loop. The reason why the times differed or were the same I believe came from if the producer thread became a consumer thread sometimes.

For 8 threads there was 1 producer and 7 consumers. Because there were so many consumers the queue never got big enough for the producer to help those threads nor did the consumers take too much time to do their tasks that the producer came in when it was done making all of the tasks. Thus, instead of 8 threads working together on all of the calculations there were only 7 threads and thus they did not finish as quickly as if there was one more thread.

For 2 and 4 threads the reason why I believe it was the same times is that the since there was 1 producer and 1 consumer or 3 consumers for 4 threads, the consumers couldn't keep up with everything on the queue so the producer finished making all the tasks and then stepped in to help with those tasks. Thus there were a total of 2 threads or 4 threads working on all the tasks, the same amount of threads that were working on the calculations with the parallel for loop.

For 1 thread the reason why it took longer was the there was only 1 producer and 0 consumers or 0 producers and 1 consumer. This is because the thread would be come a consumer when the queue filled up and then a produce when it started to become empty again. Because there was this switching going on it took 1 thread with tasks a bit longer than 1 thread with parallel for loop implementation.

Code

```c
/*
  Jonathan Armknecht CS324 Sec 2 Task Version
  Compute Times for tasks as a row
    1 thread 44 seconds
    2 threads 23 seconds
    4 threads 17 seconds
    8 threads 16 seconds

  all done with running this command prompt
    ./mandelbrot 0.27085 0.27100 0.004640 0.004810 1000 8192 pic.ppm



  This program is an adaptation of the Mandelbrot program
  from the Programming Rosetta Stone, see
  http://rosettacode.org/wiki/Mandelbrot_set
  Compile the program with:
  gcc -o mandelbrot -O4 mandelbrot.c
  Usage:

  ./mandelbrot <xmin> <xmax> <ymin> <ymax> <maxiter> <xres> <out.ppm>
  Example:
  ./mandelbrot 0.27085 0.27100 0.004640 0.004810 1000 1024 pic.ppm
  The interior of Mandelbrot set is black, the levels are gray.
  If you have very many levels, the picture is likely going to be quite
  dark. You can postprocess it to fix the palette. For instance,
  with colorMagick you can do (assuming the picture was saved to pic.ppm):
  convert -normalize pic.ppm pic.png
  The resulting pic.png is still gray, but the levels will be nicer. You
  can also add colors, for instance:
  convert -negate -normalize -fill blue -tint 100 pic.ppm pic.png
  See http://www.colormagick.org/Usage/color_mods/ for what colorMagick
  can do. It can do a lot.
*/

#include <time.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>

typedef unsigned char PIXEL[6];
const static int NUM_THREADS = 8;

int main(int argc, char *argv[])
{
  /* Parse the command line arguments. */
```

```
  if (argc != 8)
  {
    printf("Usage:   %s <xmin> <xmax> <ymin> <ymax> <maxiter> <xres>
<out.ppm>\n", argv[0]);
    printf("Example: %s 0.27085 0.27100 0.004640 0.004810 1000 1024 pic.ppm\n",
argv[0]);
    exit(EXIT_FAILURE);
  }

  /* The window in the plane. */
  const double xmin = atof(argv[1]);
  const double xmax = atof(argv[2]);
  const double ymin = atof(argv[3]);
  const double ymax = atof(argv[4]);

  /* Maximum number of iterations, at most 65535. */
  const uint16_t maxiter = (unsigned short)atoi(argv[5]);

  /* Image size, width is given, height is computed. */
  const int xres = atoi(argv[6]);
  const int yres = (xres * (ymax - ymin)) / (xmax - xmin);

  /* Precompute pixel width and height. */
  double dx = (xmax - xmin) / xres;
  double dy = (ymax - ymin) / yres;

  /* Make a pixel map so when concurrent writing occurs it will be printed out
correctly */
  PIXEL **color = malloc(yres * sizeof(PIXEL *));
  color[0] = malloc(yres * xres * sizeof(PIXEL));

  for (int q = 1; q < yres; q++)
  {
    color[q] = color[0] + (q * xres);
  }

  double x, y; /* Coordinates of the current point in the complex plane. */
  int k;       /* Iteration counter */

  time_t startTime = time(NULL); //for computing the time taken
  printf("Starting computation\n");

  omp_set_num_threads(NUM_THREADS);

#pragma omp parallel private(k, x, y)
  #pragma omp single
    for (int j = 0; j < yres; j++)
    {
```

```
      y = ymax - j * dy;
      #pragma omp task
        for (int i = 0; i < xres; i++)
        {
          double u = 0.0;
          double v = 0.0;
          double u2 = u * u;
          double v2 = v * v;
          x = xmin + i * dx;
          /* iterate the point */
          for (k = 1; k < maxiter && (u2 + v2 < 4.0); k++)
          {
            v = 2 * u * v + y;
            u = u2 - v2 + x;
            u2 = u * u;
            v2 = v * v;
          };
          /* compute  pixel color and write it to file */
          if (k >= maxiter)
          {
            for (int z = 0; z < 6; z++)
            {
              color[j][i][z] = 0;
            }
          }
          else
          {
            color[j][i][0] = k >> 8;
            color[j][i][1] = k & 255;
            color[j][i][2] = k >> 8;
            color[j][i][3] = k & 255;
            color[j][i][4] = k >> 8;
            color[j][i][5] = k & 255;
          };
        }
    }

  time_t endTime = time(NULL);
  printf("Calculations done in %f seconds with %d threads.\nWriting to file. .
.\n", difftime(endTime, startTime), NUM_THREADS);
  /* The output file name */
  const char *filename = argv[7];

  /* Open the file and write the header. */
  FILE *fp = fopen(filename, "wb");
  char *comment = "# Mandelbrot set"; /* comment should start with # */

  /*write ASCII header to the file*/
```

```
  fprintf(fp,
          "P6\n# Mandelbrot, xmin=%lf, xmax=%lf, ymin=%lf, ymax=%lf,
maxiter=%d\n%d\n%d\n%d\n",
          xmin, xmax, ymin, ymax, maxiter, xres, yres, (maxiter < 256 ? 256 :
maxiter));

  //copy color array to the file
  for (int j = 0; j < yres; j++) {
    for (int i = 0; i < xres; i++) {
      fwrite(color[j][i], 6, 1, fp);
    }
  }
  fclose(fp);
  free(*color);
  free(color);
  return 0;
}
```