Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE, Spring Semester 2021
Lecturer: Dr. R. Käppeli, Prof. O. Schenk
Assistants: A. Dolfini, L. Gaedke-Merzhaeuser
T. Holt, R. Janalik, I. Labarca, D. Pasadakis,
G. Pollak, M. Sudwoj

Project 2
Parallel Programming using OpenMP
Due date: 26 March 2021 (midnight)

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP [10 points]

The file *dotProduct/dotProduct.cpp* contains a serial version of a C++ code that computes the dot product $\alpha = a^T \cdot b$ of two vectors $a \in \mathbb{R}^N$ and $b \in \mathbb{R}^N$. Below is a code snippet:

```
    // serial execution
    // Note that we do extra iterations to reduce relative timing overhead
    time_start = wall_time();
    for( int iterations=0; iterations<NUM_ITERATIONS; iterations++) {
        alpha=0.0;
        for( int i=0; i<N; i ++) {
            alpha += a[i] * b[i];
        }
    }
```

Solve the following tasks:

1. Implement a parallel version of the dot product (i) using the OpenMP *reduction* clause, and (ii) using the OpenMP *parallel* and *critical* clauses.

2. Run the serial and both parallel versions on the Euler cluster and measure their execution times. Make a performance scaling chart for the serial version and both parallel versions using various number of threads from $p = 1, ..., 24$ for various vector lengths of $N = 100,000$, $N = 1,000,000$, $N = 10,000,000$, $N = 100,000,000$, and $N = 1,000,000,000$. In addition to the performance scaling, plot the parallel efficiency for all these dot product benchmarks.

3. Please summarize your observations at what size of N it would be beneficial to use a multi-threaded version of a dot product on one compute node of Euler cluster .

Hint: Next let's build the code. You have to load the `gcc` module.

```
[user@eu-login]$ module load new
[user@eu-login]$ module load gcc/6.3.0
[user@eu-login]$ make
```

Execute the code using 1 thread:

```
[user@eu-login]$ bsub -n 1 -W 00:10 ./dotProduct
```

or t threads:

```
[user@eu-login]$ export OMP_NUM_THREADS=t
[user@eu-login]$ bsub -n t -W 00:10 -R "span[ptile=t]" ./dotProduct
```

Note that you need `-R "span[ptile=t]"` to make sure all cares are allocated on the same compute node.

2. The Mandelbrot set using OpenMP [30 points]

Write a sequential code in C to visualize the Mandelbrot set. The set bears the name of the "Father of the Fractal Geometry," Benoît Mandelbrot. The Mandelbrot set is the set of complex numbers c for which the sequence $(z, z^2 + c, (z^2 + c)^2 + c, ((z^2 + c)^2 + c)^2 + c, (((z^2 + c)^2 + c)^2 + c)^2 + c, \ldots)$ does not approach infinity. Mandelbrot set images are made by sampling complex numbers and determining for each whether the result tends towards infinity when a particular mathematical operation is iterated on it. Treating the real and imaginary parts of each number as image coordinates, pixels are colored according to how rapidly the sequence diverges, if at all. More precisely, the Mandelbrot set is the set of values of c in the complex plane for which the orbit of 0 under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ remains bounded. That is, a complex number c is part of the Mandelbrot set if, when starting with $z_0 = 0$ and applying the iteration repeatedly, the absolute value of z_n remains bounded however large n gets. For example, letting $c = 1$ gives the sequence $0, 1, 2, 5, 26, \ldots$ which tends to infinity. As this sequence is unbounded, 1 is not an element of the Mandelbrot set. On the other hand, $c = -1$ gives the sequence $0, -1, 0, -1, 0, \ldots$ which is bounded, and so -1 belongs to the Mandelbrot set.
The set is defined as follows:

$$\mathcal{M} := \{c \in \mathbb{C} : \text{the orbit } z, f_c(z), f_c^2(z), f_c^3(z), \ldots \text{ stays bounded}\}$$

where f_c is a complex function, usually $f_c(z) = z^2 + c$ with $z, c \in \mathbb{C}$. One can prove that if for a c once a point of the series $z, f_c(z), f_c^2(z), \ldots$ gets farther away from the origin than a distance of 2, the orbit will be unbounded, hence c does not belong to $\mathcal{M}$. Plotting the points whose orbits remain within the disk of radius 2 after `MAX_ITERS` iterations gives an approximation of the Mandelbrot set. Usually a color image is obtained by interpreting the number of iterations until the orbit "escapes" as a color value. This is done in the following pseudo code:

```
for all c in a certain range do
        z = 0
        n = 0
        while |c| < 2 and n < MAX_ITERS do
                z = z^2 + c
                n = n + 1
        end while
        plot n at position c
end for
```

The entire Mandelbrot set in Figure 1 is contained in the rectangle $-2.1 \leq \Re(c) \leq 0.7$, $-1.4 \leq \Im(c) \leq 1.4$. To create an image file, use the routines from *mandel/pngwriter.c* found in the git repository like so:

```
#include "pngwriter.h"
png_data* pPng = png_create (width, height); // create the graphic
// plot a point at (x, y) in the color (r, g, b) (0 <= r, g, b < 256)
png_plot (pPng, x, y, r, g, b);
png_write (pPng, filename); // write to file
```

You need to link with `-lpng`. You can set the RBG color to white $(r, g, b) = (255, 255, 255)$ if the point at (x, y) belongs to the Mandelbrot set, otherwise it can be $(r, g, b) = (0, 0, 0)$

```
// plot the number of iterations at point (i, j)
int c = ((long) n * 255) / MAX_ITERS;
png_plot (pPng, i, j, c, c, c);
```

Record the time used to compute the Mandelbrot set. How many iterations could you perform per second? What is the performance in MFlop/s (assume that 1 iteration requires 8 floating point operations)? Try different image sizes. Please use the following C code fragment to report these statistics.

```
printf ("Total time:  %g millisconds\n", (nTimeEnd-nTimeStart)/1000.0);
printf ("Image size:  %ld x %ld = %ld Pixels\n", IM_WIDTH, IM_HEIGHT, (IM_WIDTH*IM_HEIGHT));
printf ("Total number of iterations: %ld\n", nTotalIterationsCount);
```

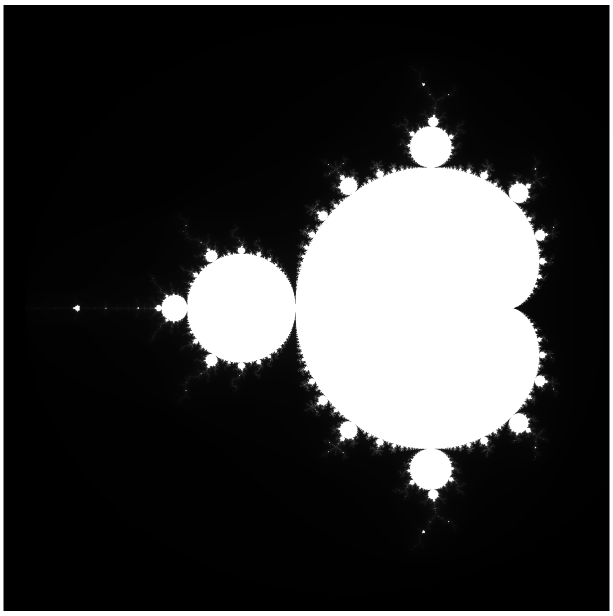
Figure 1: The Mandelbrot set

```
printf ("Avg. time per pixel: %g microseconds\n", (nTimeEnd - nTimeStart)/((IM_WIDTH * IM_HEIGHT));
printf ("Avg. time per iteration: %g microseconds\n",(nTimeEnd-nTimeStart)/nTotalIterationsCount);
printf ("Iterations/second: %g\n", nTotalIterationsCount/(nTimeEnd-nTimeStart)*1e6);
printf ("MFlop/s: %g\n", nTotalIterationsCount * 8.0 / (nTimeEnd-nTimeStart));
```

Solve the following problems:

1. Implement the computation kernel of the Mandelbrot set in *mandel/mandel_seq.c*:

```
        for (j = 0; j < IMAGE_HEIGHT; j++)
        {
                cx = MIN_X;
                for (i = 0; i < IMAGE_WIDTH; i++)
                {
                        x = cx;
                        y = cy;
                        x2 = x * x;
                        y2 = y * y;
                        // compute the orbit z, f(z), f^2(z), f^3(z), ...
                        // count the iterations until the orbit leaves the circle |z|=2.
                        // stop if the number of iterations exceeds the bound MAX_ITERS.
                        // >>>>>>>> CODE IS MISSING

                        // <<<<<<<< CODE IS NISSING
                        // n indicates if the point belongs to the mandelbrot set
                        // plot the number of iterations at point (i, j)
                        int c = ((long) n * 255) / MAX_ITERS;
                        png_plot (pPng, i, j, c, c, c);
                        cx += fDeltaX;
                }
                cy += fDeltaY;
        }
        unsigned long nTimeEnd = get_time ();
```

2. Count the total number of iterations in order to correctly compute the benchmark statistics. Use the variable `nTotalIterationsCount`.

3. Parallelize the Mandelbrot code that you have written using OpenMP. Compile the program using the GNU C compiler (`gcc`) with the option `-fopenmp`. Compare the timings of the parallelized program to those of the sequential program using a meaningful graphical representation.

Hint: Next let's build and execute the code with t threads. You have to load the `gcc` module.

```
[user@eu-login]$ module load new
[user@eu-login]$ module load gcc/6.3.0
[user@eu-login]$ make
[user@eu-login]$ export OMP_NUM_THREADS=t
[user@eu-login]$ bsub -n t -W 00:10 -R "span[ptile=t]" ./mandel_omp
```

3. Bug hunt [20 points]

You can find in the code directory for this project a number of short OpenMP programs (*bugs/omp_bug1_1-5.c*), which all contain compile-time or run-time bugs. Identify the bugs, explain what is the problem and suggest how to fix it (there is no need to submit the correct modified code).

Hints:

1. *bug1.c:* check `tid`
2. *bug2.c:* check shared vs. private
3. *bug3.c:* check barrier
4. *bug4.c:* stacksize! http://stackoverflow.com/questions/13264274
5. *bug5.c:* locking order?

4. Parallel histogram calculation using OpenMP [20 points]

The following code fragment calculates a histogram with 16 bins from a random sequence with a normal distribution stored in an array vec:

```
    time_start = wall_time();
    // Compute histogram
    for(long i = 0; i < VEC_SIZE; ++i) {
        dist[vec[i]]++;
    }
    time_end = wall_time();
```

Parallelize the histogram computations using OpenMP. You can find the serial C example code in *hist/hist_seq.c*. Report runtimes for the original (serial) code, the 1-thread and the N-thread parallel versions. Does your solution scale? If it does not, make it scale! (It really should!)

Hint: Next let's build and execute the code with t threads. You have to load the `gcc` module.

```
[user@eu-login]$ module load new
[user@eu-login]$ module load gcc/6.3.0
[user@eu-login]$ make
[user@eu-login]$ export OMP_NUM_THREADS=t
[user@eu-login]$ bsub -n 16 -W 00:10 -R "rusage[mem=2048]" -R "span[ptile=16]" ./hist_omp
```

Here we use a little trick. We need more than 4 GB of memory. If we ask the scheduler for enough memory and only one core, it would give our job to a *bigmem* queue, where the waiting times are very long. Instead we can allocate more cores and less memory per core. And then the job is scheduled in a *normal* queue.

5. Parallel loop dependencies with OpenMP [20 points]

Parallelize the loop in the following piece of code *recursion/recur_seq.c* in the repository using OpenMP:

```
double up = 1.00001;
double Sn = 1.0;
double opt[N+1];
int n;
for (n=0; n<=N; ++n) {
   opt[n] = Sn;
   Sn *= up;
}
```

The parallelized code should work independently of the OpenMP schedule pragma that you will use. Please also try to avoid – as far as possible – expensive operations that might harm serial performance. To solve this problem you might want to use the firstprivate and lastprivate OpenMP clauses. The former acts like private with the important difference that the value of the global variable is copied to the privatized instances. The latter has the effect that the listed variables values are copied from the lexically last loop iteration to the global variable when the parallel loop exits. Please report the scaling of your solution.
Next let's build and execute the code. You have to load the `gcc` module.

```
[user@eu-login]$ module load new
[user@eu-login]$ module load gcc/6.3.0
[user@eu-login]$ make
[user@eu-login]$ bsub -n 16 -W 00:10 -R "rusage[mem=2048]" -R "span[ptile=16]" ./recur_seq
```

Here we use the same trick as in Exerrise 4 to get enough memory and schedule our job in the *normal* queue.

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to Moodle .

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your OpenMP solutions.
  - your write-up with your name project_number_lastname_firstname.pdf,
- Submit your .tgz through Moodle .