Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE, Spring Semester 2023
Lecturer: Dr. R. Käppeli, Prof. O. Schenk
Assistants: Tim Holt, Malik Lechekhab, Dimosthenis Pasadakis, Xiaohe Niu, Niall Alexander Siegenheim, Julien Schmidt, Jitin Jami

Project 2
Parallel Programming using OpenMP
Due date: 27 March 2023, 11:59pm (midnight)

This project will introduce you to parallel programming using OpenMP.

1. Shared memory π-calculation using OpenMP [20 points]

With

$$\phi(x) = \frac{1}{1+x^2}$$

we have

$$\int \phi(x)dx = \arctan(x).$$

Hence, π can be calculated through integration of $\phi(x)$: http://mathworld.wolfram.com/InverseTangent.html .

1. Develop a serial implementation that integrates function $\phi(x)$ over $[0, 1]$.
2. Parallelize your application using OpenMP. Please implement two different versions using both the *critical* directive and the *reduction clause* in order to ensure the correct summation order.
3. Perform a scaling study of your algorithm.
   - Use OMP_NUM_THREADS in order to start your application with different thread counts.
   - Calculate the achieved speed-up and provide interpretations. Please perform weak and strong scaling studies.
     Weak scaling: if you double the number of threads you also double the problem size.
     Strong scaling: You keep the problem size constant but increase the number of threads.
   - Does the version using *critical* or the one using *reduction* perform/scale better? If so, why do you think this is the case?

Hint: Use the mid-point rule for integration: Split the unit-interval into n equal sized sub-intervals with length $h = \frac{1}{n}$. For each mid-point x_i of each sub-interval, calculate the value of function $\phi(x_i)$. Afterwards, sum up all function values. In the end multiply the result with $4 \times h$. Explain why this method works.

2. Quicksort using Task-Concept of OpenMP 3.1 [20 points]

1. Parallelize the quicksort implementation given in *quicksort.c*. Please employ the task-concept of OpenMP 3.1. Use the final clause for stopping the parallelization of the recursion at a sufficient level of the recursion. *Hint: Try to approach the sufficient level of the recursion for the final clause scientifically, i.e. make sure that you can repeat your results, e.g. by measuring multiple times and taking the average.*
2. Examine the scalability (strong scaling) for different problem sizes and plot your results.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE, Spring Semester 2023
Lecturer: Dr. R. Käppeli, Prof. O. Schenk
Assistants: Tim Holt, Malik Lechekhab, Dimosthenis Pasadakis, Xiaohe Niu, Niall Alexander Siegenheim, Julien Schmidt, Jitin Jami

3. The Mandelbrot set using OpenMP [20 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 $(z, z^2 + c, (z^2 + c)^2 + c, ((z^2 + c)^2 + c)^2 + c, (((z^2 + c)^2 + c)^2 + c)^2 + c, \dots)$ 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 $0, 1, 2, 5, 26, \dots$ 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 $0, -1, 0, -1, 0, \dots$ which is bounded, and so -1 belongs to the Mandelbrot set.
The set is defined as follows:

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

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 $z, f_c(z), f_c^2(z), \dots$ 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);
```

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE, Spring Semester 2023
Lecturer: Dr. R. Käppeli, Prof. O. Schenk
Assistants: Tim Holt, Malik Lechekhab, Dimosthenis Pasadakis, Xiaohe Niu, Niall Alexander Siegenheim, Julien Schmidt, Jitin Jami

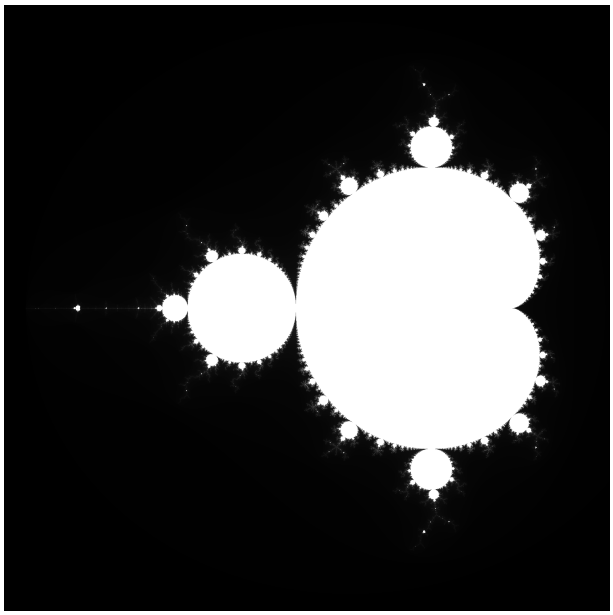

Figure 1: The Mandelbrot set

```
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*:

```
      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.
      // TODO
      // >>>>>>>> CODE IS MISSING

      // <<<<<<<< CODE IS MISSING
      // 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();

  // print benchmark data
  printf("Total time:                 %g millisconds\n",
         (nTimeEnd - nTimeStart) / 1000.0);
  printf("Image size:                 %ld x %ld = %ld Pixels\n",
```

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`. Perform benchmarking for a strong scaling analysis of your implementation and provide a plot for your results as well as a discussion.

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]$ salloc --exclusive
[user@eu-g5-048-2]$ export OMP_NUM_THREADS=t
[user@eu-g5-048-2]$ ./mandel_omp
```

In case the module `new` is not loadable, you are using the software stack with LMOD modules. You can either keep using this stack without loading the module `new` or switch to the software stack [1] with Environment Modules using the command

```
$ lmod2env
```

4. Bug hunt [10 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?

5. Parallel histogram calculation using OpenMP [15 points]

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

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

// Write results
```

[1] Find more information about Euler software stacks on `https://scicomp.ethz.ch/wiki/New_SPACK_software_stack_on_Euler`

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]$ salloc --exclusive
[user@eu-g5-048-2]$ export OMP_NUM_THREADS=t
[user@eu-g5-048-2]$ ./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.

6. Parallel loop dependencies with OpenMP [15 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]$ salloc --exclusive
[user@eu-g5-048-2]$ ./recur_seq
```

Here we use the same trick as in Exercise 5 to get enough memory and schedule our job in the *normal* queue.

Additional notes and submission details

Submit **all 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 a detailed Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to Moodle .

- Your submission should be a zip or tar archive, formatted like project_number_lastname_firstname.zip / .tgz. It must contain:
  - all the source codes of your OpenMP solutions.

  - Makefiles. If you have modified them, make sure they still build the sources correctly. We will use them to grade your submission.
  - project_number_lastname_firstname.pdf, your write-up (report) with your name.
  - these formats and naming convention, please. Not following these instructions leads to more busy work for the TA's, which makes the TA's sad...

- Submit your archive file through Moodle .