ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE

FS 2024
Dr. R. Käppeli,
Dr. A. Schell, M. R. Herde,
C. Humbel, A. Mazzoleni,
N. A. Siegenheim

Project 2

Due date: 25 March 2024, 11:59pm (midnight)

In this project, we will use `OpenMP`. If `OpenMP` is new to you, we highly recommend the LLNL tutorial[1]. See also the Moodle page for further `OpenMP` resources. For reference, we also recommend Chapters four to eight of the book [1].

For the whole project, the simulations must be run on the Euler (phase 2) nodes with AMD EPYC 7763 CPUs (for which we have privileged access during the lab hours), and we will use the new software stack (LMOD Modules) using the GNU Compiler Collection:

```
[user@eu-login-39 ~]$ module load gcc
[user@eu-login-39 ~]$ gcc --version
gcc (GCC) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

However, feel free to try and develop on other available systems (e.g., your workstation or laptop) and compilers, but please make sure to document them in your report if you include results.

You find all the skeleton source codes for the project on the course Moodle page. A sample `OpenMP` program and job script are provided there in the `hello_omp` directory.

While developing/debugging your code, it can be useful to work in an interactive session as follows:

```
[user@eu-login-39 ~]$ srun --ntasks=1 --cpus-per-task=4 --mem-per-cpu=1024 \
                           --constraint=EPYC_7763                          \
                           --time=01:00:00 --pty bash
srun: Job step's --cpus-per-task value exceeds that of job (4 > 1). Job step may
↪  never run.
srun: job 50078034 queued and waiting for resources
srun: job 50078034 has been allocated resources
[karoger@eu-a2p-409 ~]$
```

This allocates an interactive session on an Euler VII (phase 2) node with up to 4 cores and 4GB of memory for one hour[2].

1 Computing π with `OpenMP` [20 points]

From elementary calculus we know that

$$\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x = 4\,\arctan(x)\Big|_0^1 = \pi.$$

We can approximate the integral with the midpoint rule as

$$\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x \approx \sum_{i=1}^{N} f(x_i)\,\Delta x$$

where the integration interval $[0,1]$ is uniformly partitioned into N subintervals of size $\Delta x = \frac{1}{N}$ and subinterval centers $x_i = (i+\frac{1}{2})\Delta x$ $(i = 1,\ldots,N)$. A serial implementation is provided in the skeleton code `pi/pi_serial.c`. The goal of this task is to parallelize the serial implementation using `OpenMP` and to study its performance:

[1] https://hpc-tutorials.llnl.gov/openmp/
[2] We ignore the warning that the job might never run, it should.

1. Parallelize the serial implementation using `OpenMP`. Implement two different versions using both the `critical` directive and the `reduction` clause.
2. Perform a weak and strong scaling study of your implementations and interpret the results in your report. In particular, discuss the observed differences between the version using `critical` directive or the `reduction` clause.

2 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 Fig. 1 is contained in the rectangle $-2.1 \le \Re(c) \le 0.7$, $-1.4 \le \Im(c) \le 1.4$. 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.

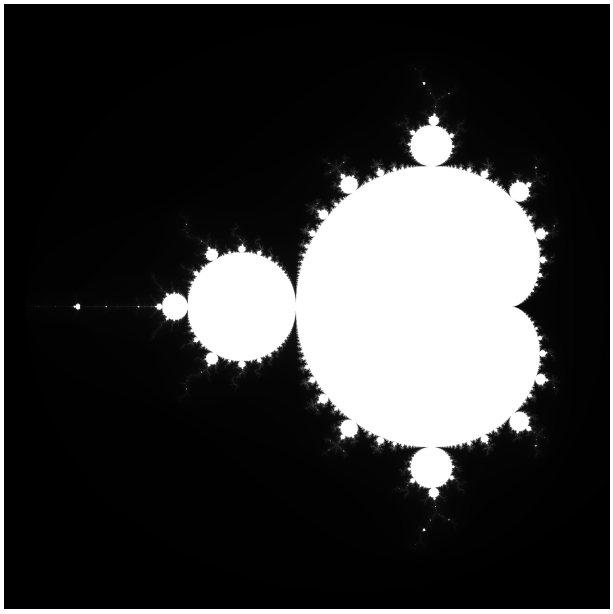

Figure 1: The Mandelbrot set

```
// print benchmark data
printf("Total time:                 %g seconds\n",
       (time_end - time_start));
printf("Image size:                 %ld x %ld = %ld Pixels\n",
       (long)IMAGE_WIDTH, (long)IMAGE_HEIGHT,
       (long)(IMAGE_WIDTH * IMAGE_HEIGHT));
printf("Total number of iterations: %ld\n", nTotalIterationsCount);
printf("Avg. time per pixel:        %g seconds\n",
       (time_end - time_start) / (double)(IMAGE_WIDTH * IMAGE_HEIGHT));
printf("Avg. time per iteration:    %g seconds\n",
       (time_end - time_start) / (double)nTotalIterationsCount);
printf("Iterations/second:          %g\n",
       nTotalIterationsCount / (time_end - time_start));
// assume there are 8 floating point operations per iteration
printf("MFlop/s:                    %g\n",
       nTotalIterationsCount * 8.0 / (time_end - time_start) * 1.e-6);
```

Solve the following problems:

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

```
// do the calculation
cy = MIN_Y;
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.
    int n = 0;
    // 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;
}
```

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.

3 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 in you report (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` [15 points]

The following code fragment computes a histogram `dist` containing 16 bins over the `VEC_SIZE` values in a large array of integers `vec` that are all in the range $\{0, \ldots, 15\}$:

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

You find the sequential implementation in `hist/hist_seq.cpp`. Parallelize the histogram computations using `OpenMP` (skeleton code is provided in `hist/hist_omp.cpp`). Report runtimes for the original (serial) code, the 1-thread and the N-thread parallel versions. Document and discuss the strong scaling (i.e., keeping the size `VEC_SIZE` of the large array fixed at its original value) behaviour in your report.

Hint: False sharing can strongly affect parallel performance (see, e.g., [1, Sec. 7.2.4]).

Note: The code requires roughly 4 GB of memory to hold the large array of integers `vec`. If we ask the scheduler for enough memory and only one core, it would give our job to a *bigmem* queue, for which we don't have privileged access. Here we use a little trick: We allocate at least 4 cores (even in the sequential case) and just 1 GB memory per core (see `hist/run_hist_seq.sh`). Then the job is scheduled in a *normal* queue.

5 Parallel loop dependencies with OpenMP [15 points]

Parallelize the loop in the following piece of code `recursion/recur_seq.c` (in the provided skeleton sources) 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. Comment on your parallelisation briefly in the report.

6 Quicksort using OpenMP tasks [20 points]

1. Parallelize the quicksort[3] implementation given in `quicksort/quicksort.c` using `OpenMP` tasks. 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, plot your results and discuss the observed prformance in your report.

Additional notes and submission details

Submit **all the source code files** together with your used build files (e.g., `Makefile`(s)) and other scripts (e.g., batch job scripts) in an archive file (tar or zip) and summarize your results and observations for all sections by writing a detailed LaTeX report. Use the LaTeX template from the webpage and upload the report as a PDF to Moodle.

- Your submission should be a tar or zip archive, formatted like `project_number_lastname_firstname.zip/tgz`. It must contain:
  - All the source codes of your solutions.
  - Build files and scripts. If you have modified the provided build files or scripts, make sure they still build the sources an run correctly. We will use them to grade your submission.
  - `project_number_lastname_firstname.pdf`, your report with your name.
  - Follow the provided guidelines for the report.
- Submit your archive file through Moodle.

Please follow these instructions and naming conventions. Failure to comply results in additional work for the TAs, which makes the TAs sad...

References

[1] Georg Hager and Gerhard Wellein. Introduction to high performance computing for scientists and engineers. *Chapman & Hall/CRC Computational Science*, July 2010. URL: http://dx.doi.org/10.1201/EBK1439811924, doi:10.1201/ebk1439811924.

[3] https://en.wikipedia.org/wiki/Quicksort