Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing, Fall 2024
Lecturer: Dr. A. Eftekhari, Prof. O. Schenk
Assistants: M. Lechekhab, D. Vega
D. Santarsiero, J. Palumbo, I. Ecevit

Project 2 – Parallel Programming using OpenMP

Due date: 23 October 2024 at 23:59 (See iCorsi for updates)

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

All tests and simulation results must be run on the compute nodes of the Rosa cluster . However, feel free to try and develop on other available systems (e.g., your workstation or laptop) and compilers (the final code must compile and run on Rosa cluster). You will find all the skeleton source codes for the project on the course iCorsi page.
To help you get started with OpenMP, here is a sample OpenMP program that you can run using interactive jobs. The source code for the sample program is available in `skeleton_codes/hello_omp.cpp`

```cpp
#include <iostream>
#ifdef _OPENMP
#include <omp.h>
#endif

int main() {
#ifdef _OPENMP
    #pragma omp parallel
    {
        if (omp_get_thread_num() == 0) {
            std::cout << "OpenMP threads=" << omp_get_num_threads() << std::endl;
        }
    }
#else
    std::cout << "OpenMP is not available." << std::endl;
#endif
    return 0;
}
```

You can compile the code using using, for example, `g++ -fopenmp hello_omp.cpp -o hello_omp`. Don't forget the `-fopenmp` flag! We recommend using a `makefile` (either provided or from other in-class examples) and adjusting them as needed. While developing/debugging your code, it can be useful to work in an interactive session (you can use the `--reservation=hpc-tuesday` or `--reservation=hpc-wednesday` for better priority) as follows:

```
[user@icslogin01 Test]$ srun --nodes=1 --exclusive --time=00:01:00 --pty bash -i
        srun: job 9737 queued and waiting for resources
        srun: job 9737 has been allocated resources
[user@icsnodeXX Test]$ export OMP_NUM_THREADS=2
[user@icsnodeXX Test]$ ./hello_omp
OpenMP threads=2
```

This allocates an interactive session on 1 node of the Rosa cluster for 1 minutes. The program sets the number of threads for parallel regions according to the value of the environment variable `OMP_NUM_THREADS` which is assigned before executing the code.
Note: Using `--exclusive` grants you exclusive access to the node, meaning that there is no sharing of resources—its all yours! It is very important to use exclusive allocation sparingly, as it restricts others from accessing the node. It is recommended to use, for example, `srun --nodes=1 --time=00:20:00 --pty bash -i` for code compilation and testing. When you are ready to run larger tests and collect results, such as strong scaling results, you would use `srun --nodes=1 --exclusive --time=00:01:00 --pty bash -i`.

1. Parallel reduction operations with `OpenMP` [20 points]

1.1. Dot Product

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

```
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];
  }
}
time_serial = wall_time() - time_start;
cout << "Serial execution time = " << time_serial << " sec" << endl;

long double alpha_parallel = 0;
double time_red = 0;
```

Solve the following tasks (Dot Product):

1. Implement two parallel versions of the dot product using `OpenMP`: (i) one using the `reduction` clause, and (ii) another using the `critical` directive.
2. Perform a strong scaling analysis on the Rosa cluster for both parallel implementations, using different numbers of threads $t = 1, 2, 4, 8, 16$ and 20 for various vector lengths $N = 10^5, 10^6, 10^7, 10^8$ and 10^9 (provide a strong scaling plot using log-axis where applicable). Discuss the observed differences between the implementation that uses the `critical` directive and the `reduction` clause. Keep all output of the code, these results will be used in the next question.
3. Perform an analysis on the parallel efficiency between the two parallel implementations (using the results from the above, provide a parallel efficiency plot using log-axis where applicable). Discuss `OpenMP` overhead and the relation between thread count, parallel efficiency, and workload (in this case the size of the vectors N). At what size of N does it become beneficial to use a multi-threaded version of the dot product?

1.2. Approximating π

From elementary calculus we know that for

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

We can approximate the integral with the midpoint rule as

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

where the integration interval $[0, 1]$ is uniformly partitioned into N subintervals of size $\Delta x = \frac{1}{N}$ and subinterval centers $x_i = (i + \frac{1}{2})\Delta x$ $(i = 1, \ldots, N)$.

The file `pi/pi.cpp` serves as a template (it does not implement the `C/C++` program that approximates π). Your goal is to implement both a serial version of the approximation and a parallelized one using `OpenMP`.

Solve the following tasks (Approximating π):

1. Implement serial and parallel versions of the π approximation method shown, using any parallelization scheme you prefer. Discuss your choice. Use a fixed value of $N = 10^{10}$.

2. Show the speedup between your parallel implementation and the serial version for $t = 1, 2, 4$,and 8 (include the plot using log-axis where applicable). Discuss the concept of speedup in relation to strong scaling and its connection to parallel efficiency (you don't need to create a strong scaling or parallel efficiency plot here; just discuss it).

2. The Mandelbrot set using `OpenMP` [20 points]

Write a sequential code in `C/C++` to visualize the Mandelbrot set. The set bears the name of the "Father of Fractal Geometry," Benoit Mandelbrot. The Mandelbrot set is the set of complex numbers c for which the sequence $(z, z^2 + c, (z^2+c)^2+c, ((z^2+c)^2+c)^2+c, (((z^2+c)^2+c)^2+c)^2+c, \dots)$ tends toward 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. 2 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 provided sources 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)
```

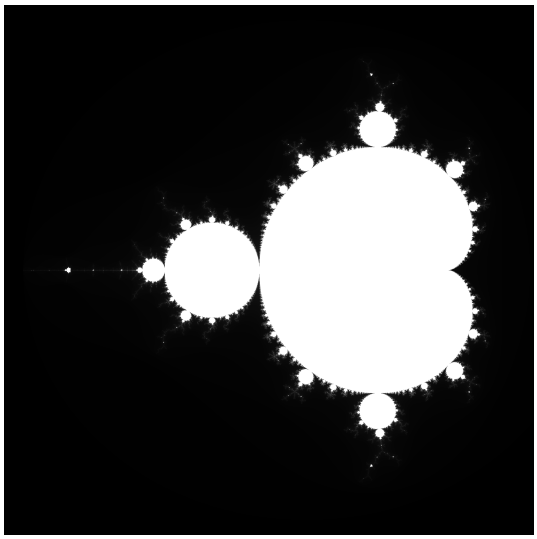

Figure 1: The Mandelbrot set

```
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 $(r, g, b) = (0, 0, 0)$

```
      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.

```
// 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);

png_write(pPng, "mandel.png");
```

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 [15 points]

You can find in the code directory for this project a number of short OpenMP programs (*bugs/omp_bug1-5.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` [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 behaviour in your report (i.e., keeping the size `VEC_SIZE` of the large array fixed at its original value).

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

5. Parallel loop dependencies with `OpenMP` [15 points]

Parallelize the loop in the following code snippet from `loop-dependencies/recur_seq.c` (available in the provided 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. Quality of the Report [15 Points]

Each project will have 100 points (out of 15 point will be given to the general written quality of the report).

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 iCorsi .

- 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 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 write-up with your name.
  - Follow the provided guidelines for the report.

- Submit your .tgz through iCorsi .

Code of Conduct and Policy

- Do not use or otherwise access any on-line source or service other than the iCorsi system for your submission. In particular, you may not consult sites such as GitHub Co-Pilot or ChatGPT.
- You must acknowledge any code you obtain from any source, including examples in the documentation or course material. Use code comments to acknowledge sources.
- Your code must compile with a standard-configuration C/C++ compiler.

References

[1] Georg Hager and Gerhard Wellein. Introduction to high performance computing for scientists and engineers. *Chapman & Hall/CRC Computational Science*, July 2010.