Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing **2022**

Student: Anthony Bugatto

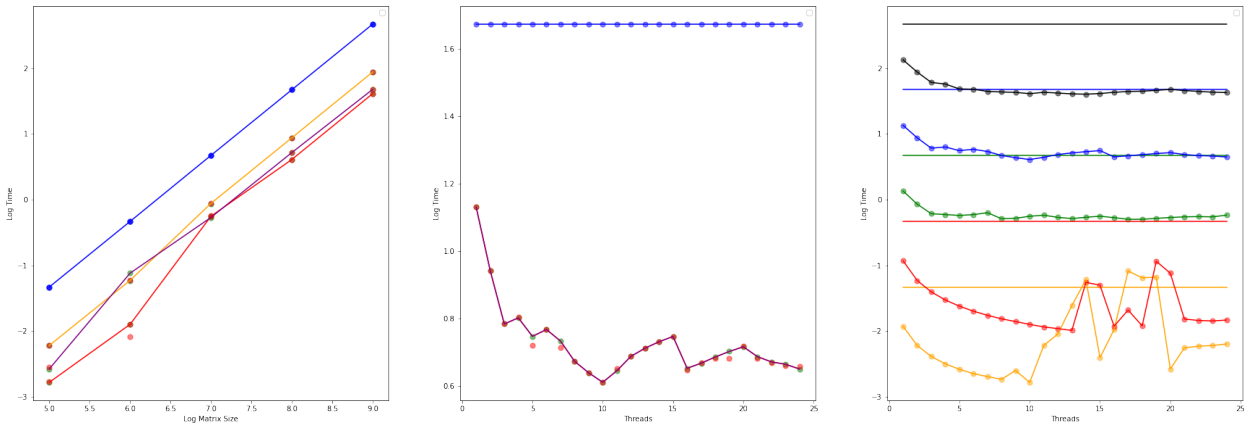
Discussed with: Riccardo Giacometti

Solution for Project 2

Due date: 26.10.2022 (midnight)

1. Parallel reduction operations using OpenMP [10 points]

The parallel dot product was implemented by using the serial, parallel reduction, and parallel critical programs. Though the reduction and critical had much of the same performance (as seen in the middle plot, they both greatly improved on the serial program. On the right we can see the performance of the parallel code with different sized arrays as a function of threads used (scatter) and the serial performance as a line.
On the left we can see the performance as a function of matrix size, which shows us that there is approximately linear scaling with respect to the size.



2. The Mandelbrot set using OpenMP [30 points]

The mandelbrot set is computed by computing the set

$$Mandelbrot(\mathbb{C}) = \{z \in \mathbb{C} | z = z^2 + p, z < 2\}$$

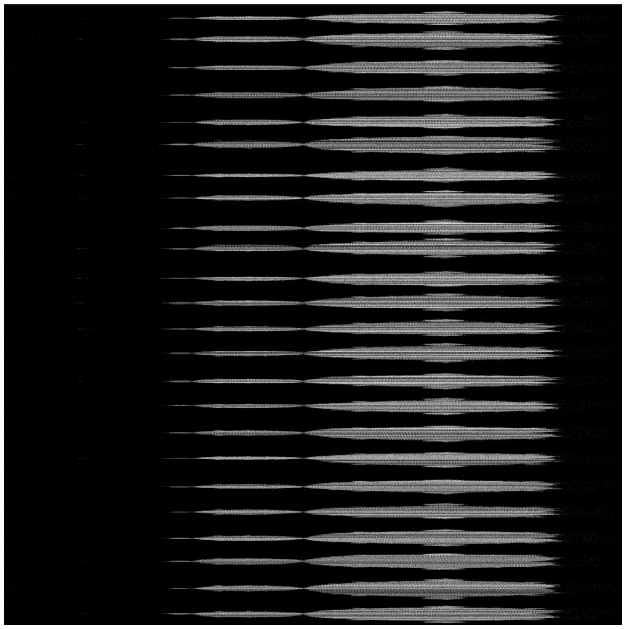
for each point p in the image, sampled uniformly to an image NxN matrix. For each point loop the equation is computed independently of every other point, and is thus embarrassingly parallel. The pseudo code for parallelization in OpenMP is below:

Algorithm 1 Parallel Mandelbrot in OpenMP

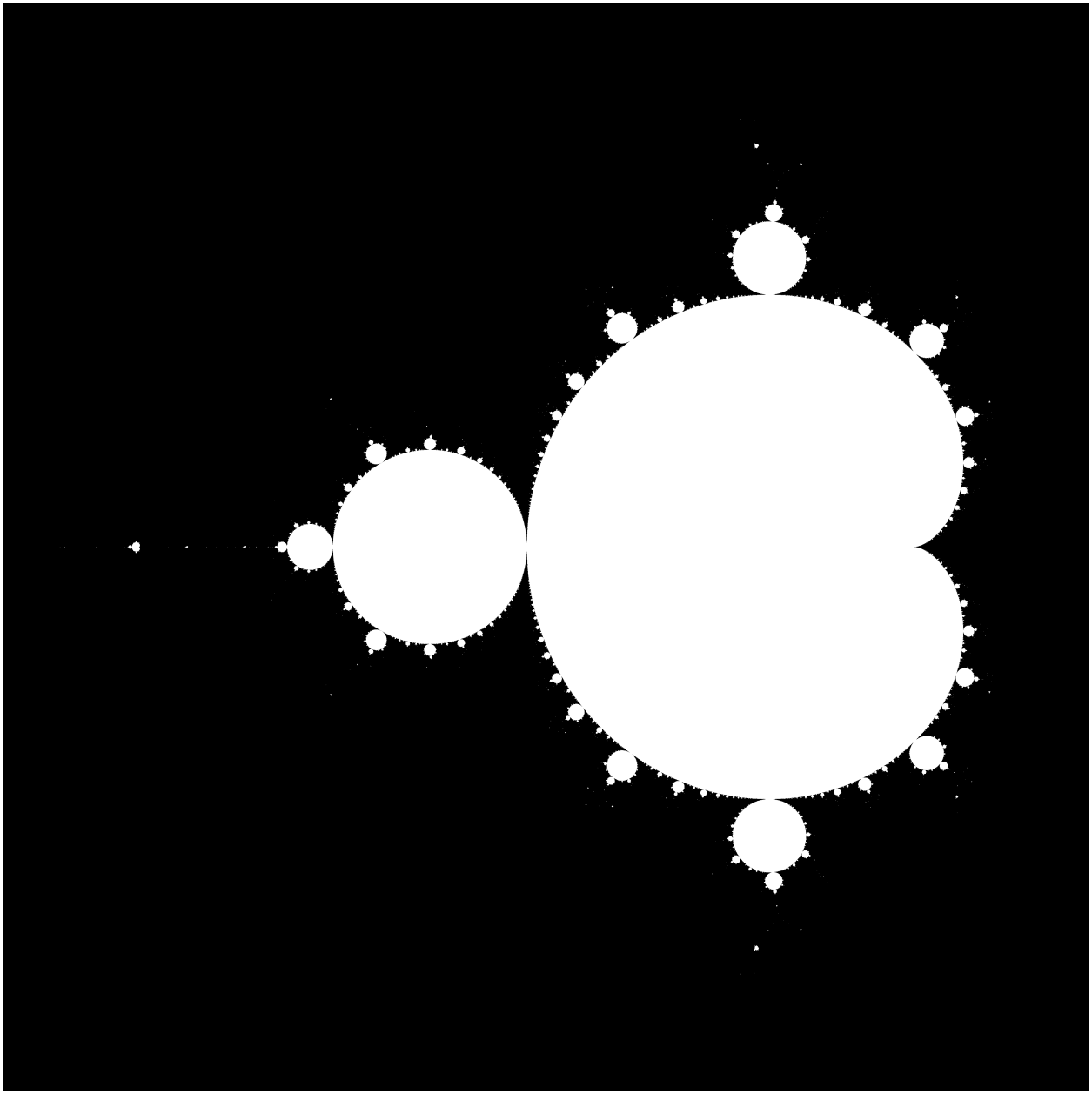
for $\{j = 0; j < N; j++0\}$ **do**
  #pragma omp parallel for private(cx)
  for $\{i = 0; i < N; i++0\}$ **do**
    $p_x = MIN\_X + \Delta x * i$
    $x, y = p_x, p_y$
    for $\{; (x2 + y2 < 4) n < MAX\_ITERS; n = n + 1\}$ **do**
      $y = 2 * x * y + p_y$
      $x = x^2 - y^2 + p_x$
    end for
    $setPixel(i, j, n)$
    $cx = cx + \Delta x$
  end for
  $cy = cy + \Delta y$
end for

2.1. Mandelbrot Bugs

OpenMP parallelizes the indices by giving every pth row or column to each thread when the pragma is used before both loops. Because of this, if the indices print out sequentially then the result will be p copies of the total image stacked on top of eachother, as the indices have been misnamed in each local computation. This can be remedied by putting the pragma after the first loop, which ensures that the indices have the correct labels and thus print out in order on the image.



2.2. Mandelbrot Plot



2.3. Runtime Statistics

We can see that the parallel mandelbrot is roughly ten times faster.

```
Total time:                551944 millisconds
Image size:                4096 x 4096 = 16777216 Pixels
Total number of iterations: 113610974266
Avg. time per pixel:       32.8984 microseconds
Avg. time per iteration:   0.00485819 microseconds
Iterations/second:         2.05838e+08
MFlop/s:                   1646.7
```

```
[stud07@icsnode27 mandel]$ ./mandel_omp
Total time:                68904.8 millisconds
Image size:                4096 x 4096 = 16777216 Pixels
Total number of iterations: 113610939011
Avg. time per pixel:       4.10705 microseconds
Avg. time per iteration:   0.000606498 microseconds
Iterations/second:         1.64881e+09
MFlop/s:                   13190.5
```

3. Bug hunt [15 points]

3.1. Bug 1

When this program is runs it outputs a compile time error. This is caused by a statement being between the omp for pragma and the for loop. Once moved the program works.

3.2. Bug 2

This program returns the same thread number in every thread. This is because the tid is set as a global variable instead of a private variable. Once private(tid) is added to the pragma the thread numbers are unique for each thread.

3.3. Bug 3

This code is blocked by the barriers. To fix this the barriers could be removed so that the output is given whenever each thread finishes instead of just the first two.

3.4. Bug 4

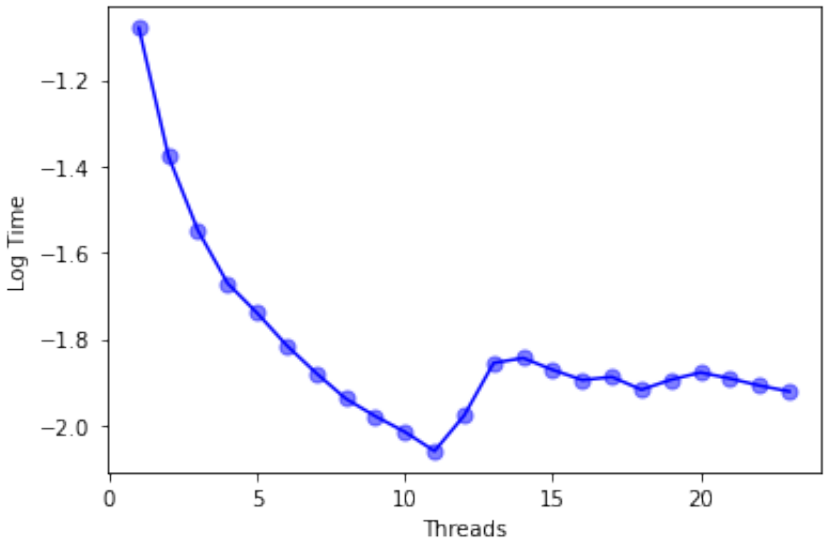
This code throws a segmentation fault. This is caused by the stack size being too small and could be fixed by modifying the omp stack size variable.

3.5. Bug 5

This code deadlocks. To fix this the threads could initialize and then wait for the other to finish, producing a program in which one thread can acquire both locks.

4. Parallel histogram calculation using OpenMP [15 points]

The histogram calculation is parallelized by adding a pragma omp parallel to the distribution update function. However, the distribution must be critical since we are accumulating the same memory locations from different threads, thus we add a reduction for the distribution. We can see that there is good improvement for the first 12 or so threads but after that it tapers off, likely due to the overhead of the distribution being a critical section in so many threads. However, it is also likely that with larger matrices we would see more improvement at larger thread sizes.



5. Parallel loop dependencies with OpenMP [15 points]

This sequential function is clearly not parallelizable but can be modified to be faster with OpenMP for a cost of extra computation. The sequential function is computing the sequence

$$S_{n+1} = S_n * up, S_0 = c$$

$$op[n] = S_{n+1}$$

while we can parallelize this version at the cost of having to recompute n of the multiplications each loop within the exponent:

$$op[n+1] = S_0 up^n$$

The code was parallelized by adding a pragma omp for with first private taking in the base and up variables from the global and a last private updating the S_N value at the end of the loop. We can see that as the size of the array gets large (bottom to top) there is an improvement in efficiency due to the number of total computations becoming greater than the overhead of redoing some in parallel.