High-Performance Computing Lab for CSE **2023**

Student: Simon Aaron Menzi Discussed with: Lukas Martin Bühler

Solution for Project 2

Due date: 27. March 2023, 23:59

In this project you will practice memory access optimization, performance-oriented programming, and OpenMP parallelizaton on Euler.

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

1.1. Develop a serial implementation that integrates function $\phi(x)$ over 0, 1.

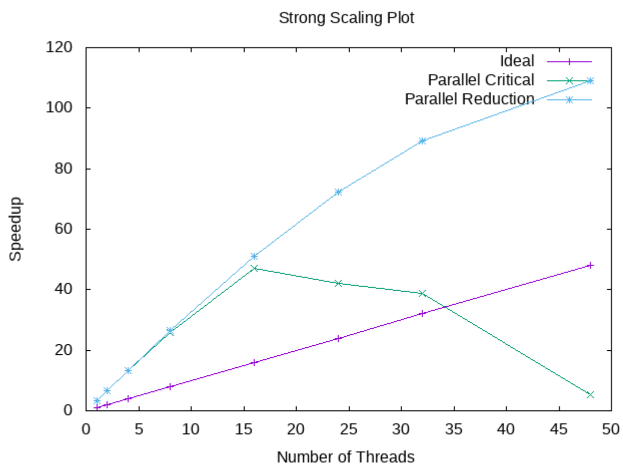
As the hint said that midpoint method would be an appropriate choice as a quadrature rule, this is what was used. The following code implements this.

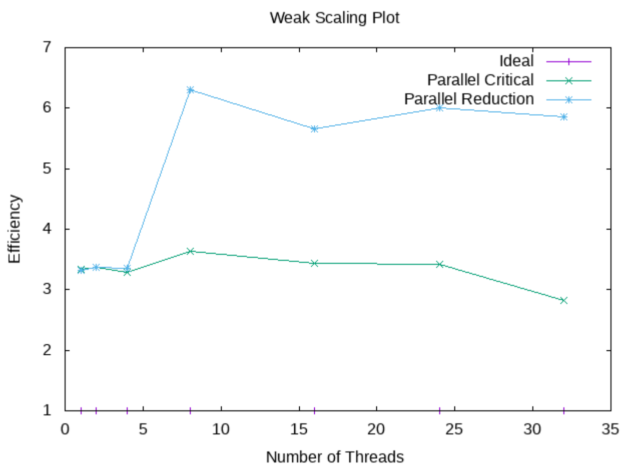
```
double midpoint_arctan_serial(const int &n) {
    double delta_x = 1.0 / n;
    double sum = 0.0;

    for (int i = 0; i < n; i++) {
        double x_mid = (i + 0.5) * delta_x;
       sum += 1.0 / (1.0 + x_mid * x_mid);
    }

    return sum / n;
}
```

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.

For parallelization with the critical directive, a local_sum-variable is used and sum up these local_sums into the sum-variable for better performance. The reduction- x

1.3. Perform a scaling study of your algorithm.

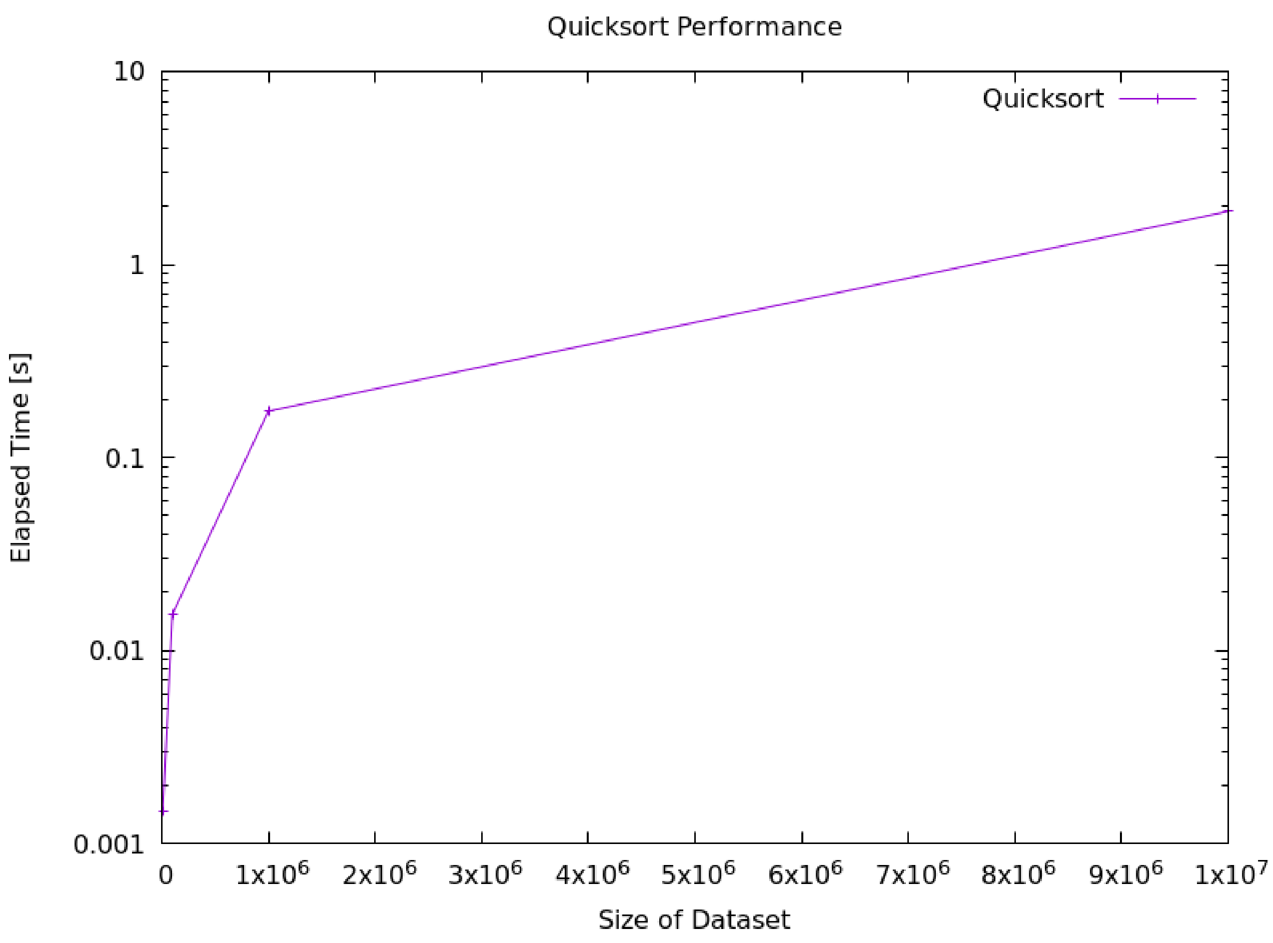




As seen in these plots, the parallel implementations exceed the linear scaling by far. A reason could be that the compiler has an easier time vectorizing and using other tricks with pragma omp directives/clauses in place to get this speedup. This would also explain why both the implementation with the critical directive and the implementation using the reduction clause perform better even with only one thread. It doesn't come as a surprise that paralleling with the reduction scales better than using the critical directive.

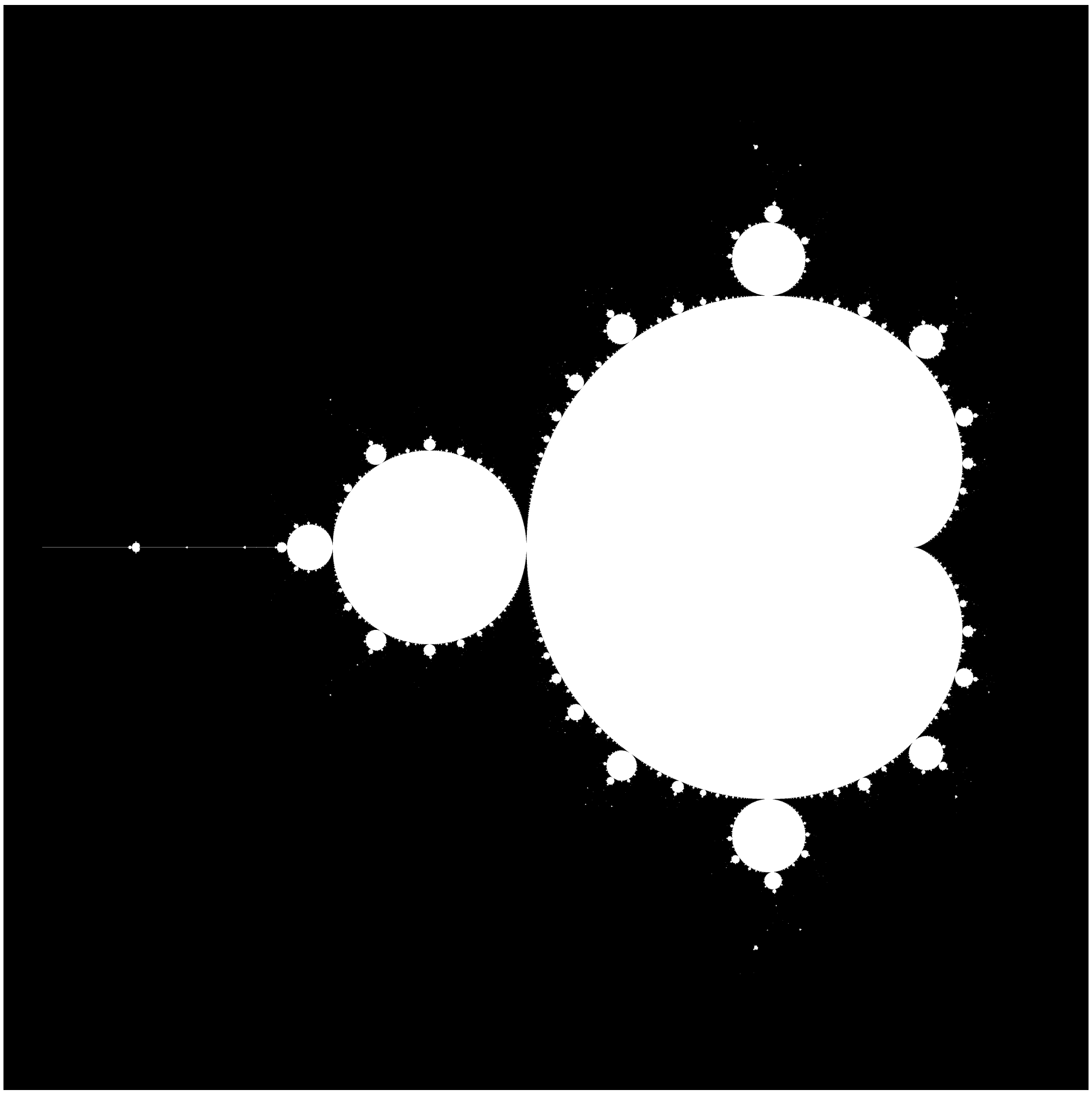
2. Quicksort using Task-Concept of OpenMP 3.1 [20 points]

2.1. Examine the scalability (strong scaling) for different problem sizes and plot your results.



As one can see in the plot, the parallelization does not scale well. One must investigate further in the implementation why this is the case.

3. The Mandelbrot set using OpenMP [20 points]



3.1. Implement the computation kernel of the Mandelbrot set in mandel/mandel seq.c

The serial implementation is straight forward, as the pseudo-code is given.

```
while (n < MAX_ITERS && x2 + y2 < 4.0) {
    x_tmp = x2 - y2 + cx;
    y = 2.0 * x * y + cy;
    x = x_tmp;
    x2 = x * x;
    y2 = y * y;
    ++n;
    ++nTotalIterationsCount;
}
```

3.2. 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.

For the parallelization, some variables like cx and cy have to be moved inside of the for loops. Variables x, y, x2, y2, cx, cy need to be private. The same is true for n but it is initialized inside the parallel region instead.

4. Bug hunt [10 points]

4.1. Bug 1

The problem is that the pragma omp for directive is not followed up with a for loop.

4.2. Bug 2

The problem is variables being shared across all threads without proper synchronization resulting in race conditions and therefore wrong results.

4.3. Bug 3

A barrier does not result in proper output, as the threads will just wait before the barrier for the other threads.

4.4. Bug 4

As the hint suggests, the problem lies in the stacksize being to small for a large number of threads.

4.5. Bug 5

In this code, we can encounter the problem of a deadlock as two different threads could wait for there respective lock to be freed.

5. Parallel histogram calculation using OpenMP [15 points]

Due to bugs and time constraints, this problem has not been properly tested and examined.