Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Computing Lab for CSE **2024**

Student: Benedict Armstrong Discussed with: Tristan Gabl

Solution for Project 02

Due date: 25 March 2024, 23:59

1. Computing π with `OpenMP` [20 points]

We used two different ways to parallelize the computation of π. Defining a parallel region using the `#pragma parallel` directive and the calculating the work for each thread 1. The second method is to use the `#pragma parallel for` directive to parallelize the for loop. The benchmark results are shown in Figure 1.

```
#pragma omp parallel
{
    double sum_private = 0.;
    int nthreads = omp_get_num_threads();
    int tid = omp_get_thread_num();
    int i_beg = tid * N / nthreads;
    int i_end = (tid + 1) * N / nthreads;
    for (int i = i_beg; i < i_end; ++i)
    {
    double x = (i + 0.5) * h;
    sum_private += 4.0 / (1.0 + x * x);
    }

#pragma omp critical
    sum += sum_private;
}
```

Listing 1: Parallel computation of π using `OpenMP` and a parallel region.

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

The Mandelbrot set can be parallelized by splitting the image into N equal parts, where N is the number of threads. Each thread then calculates the Mandelbrot set for its assigned part of the image. The pragma directive for parallelizing the Mandelbrot set calculation is shown in Listing 2. The result of a strong scaling benchmark (using 1000 iterations to determine convergence) is shown in Figure 2. The generated Mandelbrot set image is shown in Figure 3.

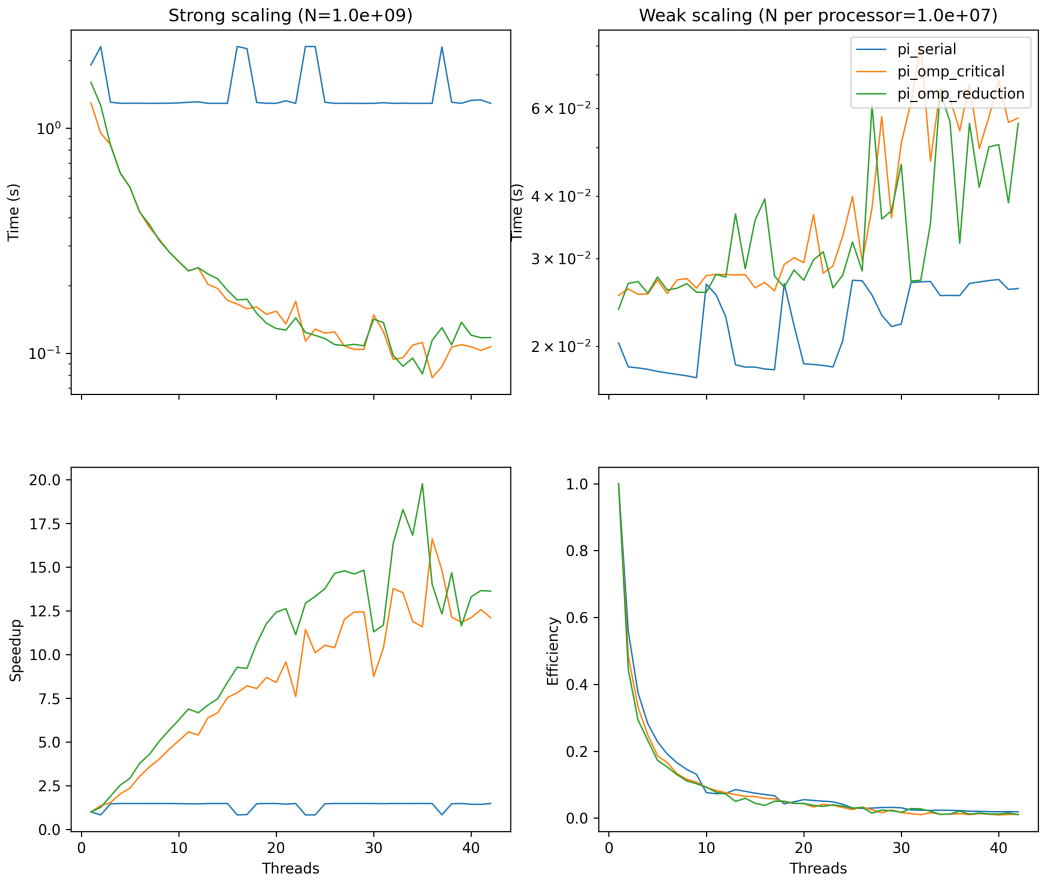

Figure 1: Benchmark results for the π calculation using `OpenMP`.

```
#pragma omp parallel for shared(pPng) \
    private(i, j, x, y, x2, y2, cx, cy) reduction(+ : nTotalIterationsCount)
```

Listing 2: Pragma directive for parallelizing the Mandelbrot set calculation.

3. Bug hunt [10 points]

1. The first bug is a compile-time bug. The `#pragma` directive must be followed by a for loop. In this case we have a `tid = omp_get_thread_num()` statement immediately after the `#pragma`. This is not allowed. The fix is to move the `tid` assignment into the for loop.

2. There are a couple of errors in the code. The variable `tid` should be made explicitly private as every thread is writing to it. In the last `for` loop the total sum should be marked as a reduction variable (using the `reduction(+:total)` clause). Also by default the second loop will not spawn any new threads as the option `OMP_NESTED` is set to `FALSE` by default (see IBM OpenMP documentation).

4. Parallel histogram calculation using `OpenMP` [15 points]

The histogram calculation can be easily parallelized using a one line compiler directive. The code is shown in Listing 3. The strong scaling benchmark results are shown in Figure 4 and the speedup

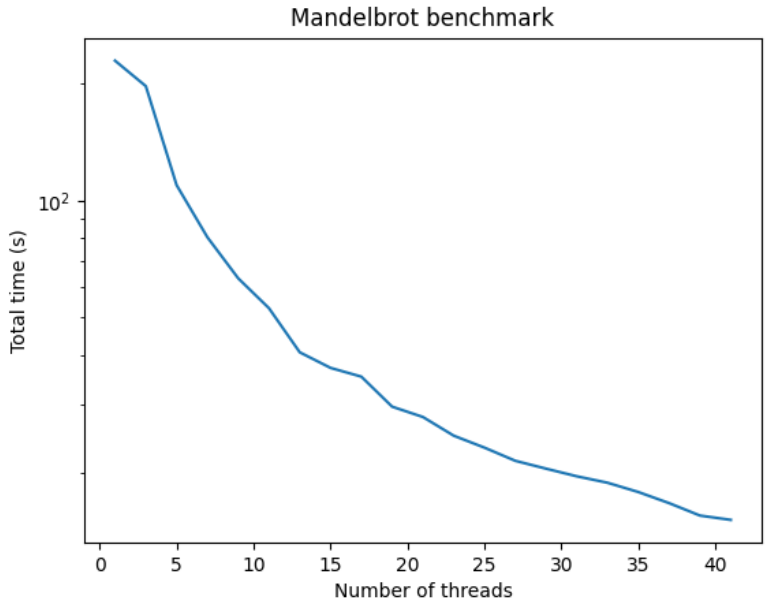

Figure 2: Benchmark results for the Mandelbrot set calculation using `OpenMP`.

results are shown in Figure 5.

```
#pragma omp parallel for reduction(+ : dist[ : BINS])
```

Listing 3: Parallel histogram calculation using `OpenMP`

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

To parallelize the loop with dependencies we split up the loop into N equal parts, where N is the number of threads. We then calculate the first element of each thread's partition

$$S_i = Sn * up^{i*chunk\_size}$$

where i is the thread number. Each thread then calculates the rest of the assigned elements. Figure 6 shows the benchmark results and Figure 7 shows the speedup results.

6. Quicksort using `OpenMP` tasks [20 points]

Quicksort can be easily parallelized using tasks. We create a task for each recursive call to the quicksort function. We then wait for all tasks to finish before returning. This is done by adding a `taskwait` directive after the recursive calls. The only complication is defining a minimum task size to prevent the creation of too many tasks. The code is shown in Listing 4.

After implementing the quicksort function using tasks I benchmarked the code using a strong scaling analysis. The results are shown in Figure 8.

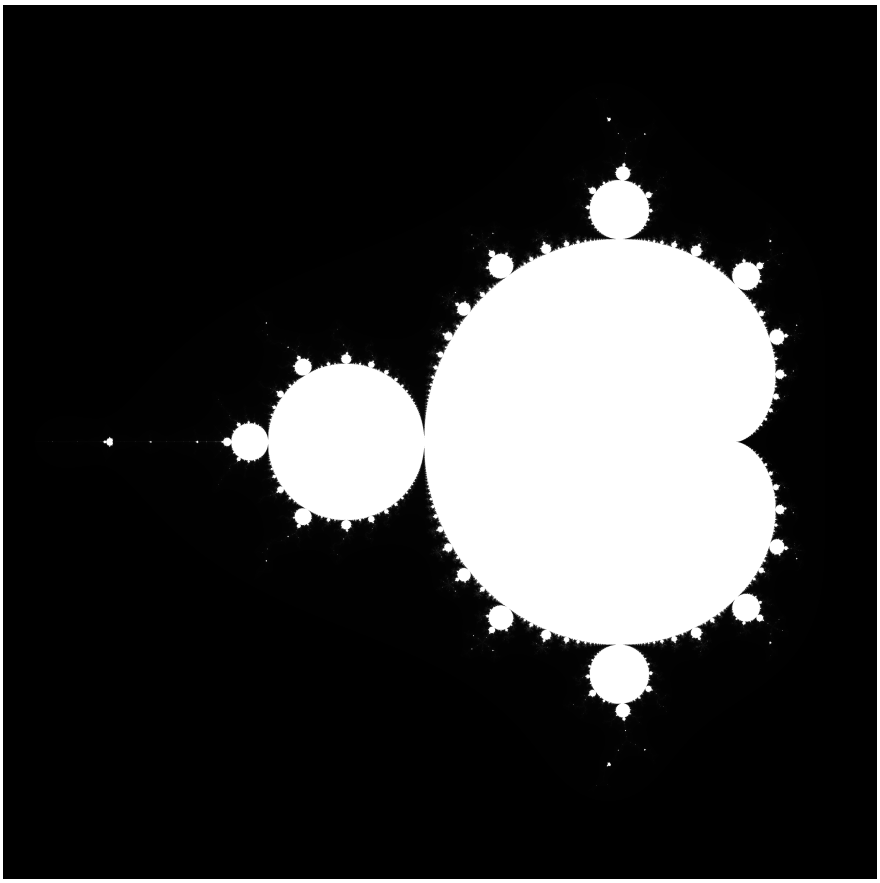

Figure 3: Mandelbrot set

```
  #pragma omp task shared(data) firstprivate(right) final(right < MIN_SIZE)
  quicksort(data, right);

  int t = length - left;
#pragma omp task shared(data, left) firstprivate(t) final(t < MIN_SIZE)
  quicksort(&(data[left]), t);

#pragma omp taskwait
```

Listing 4: Recursion of the quicksort function using tasks

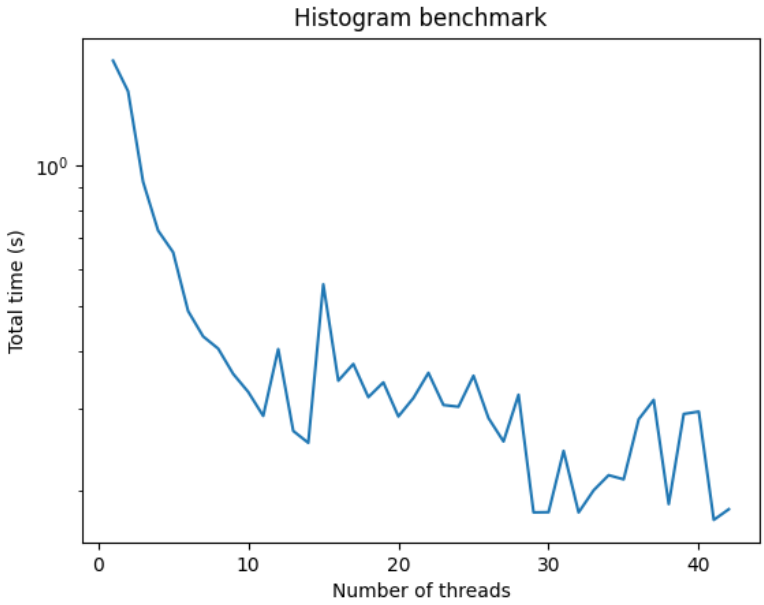

Figure 4: Benchmark results for the histogram calculation using `OpenMP`.

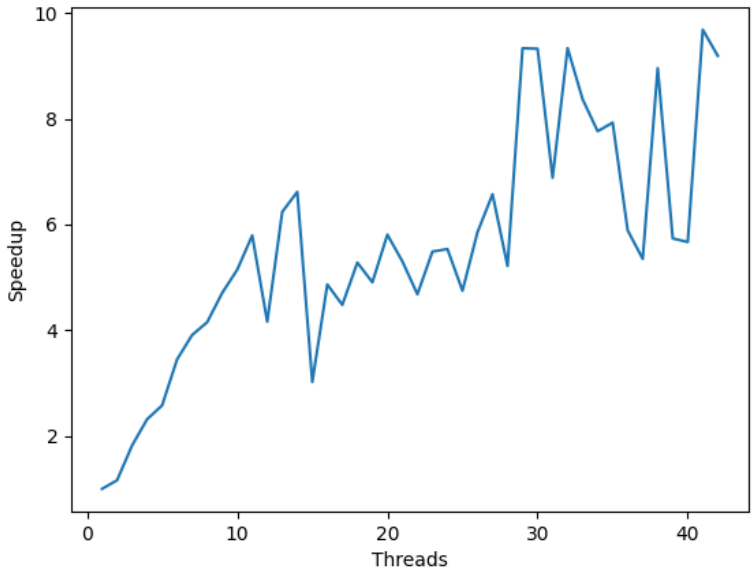

Figure 5: Speedup results for the histogram calculation using `OpenMP`.

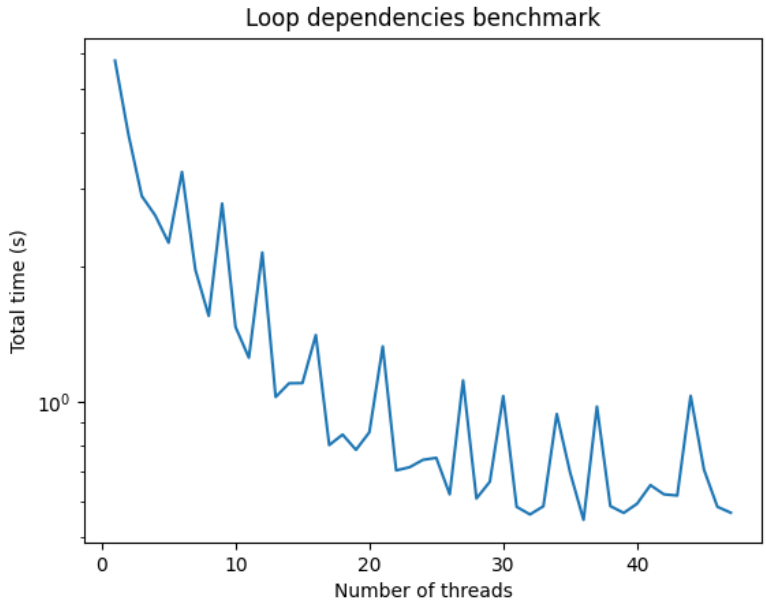

Figure 6: Benchmark results for the loop dependencies calculation using `OpenMP`.

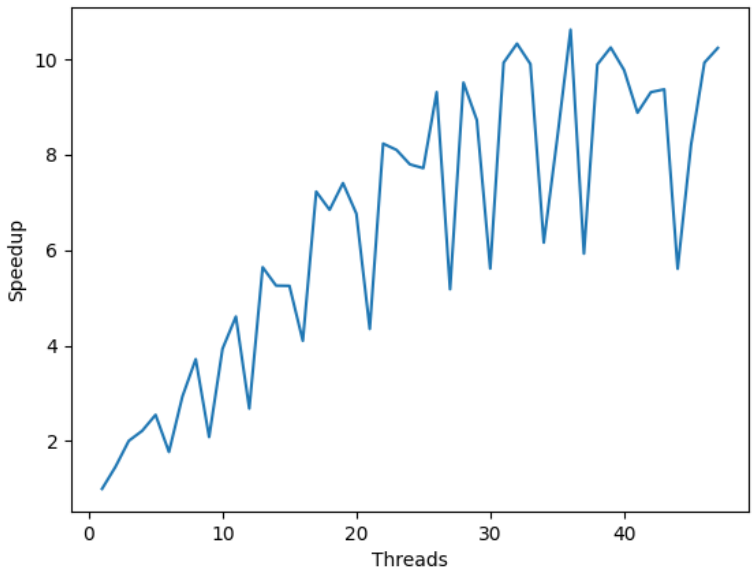

Figure 7: Speedup results for the loop dependencies calculation using `OpenMP`.

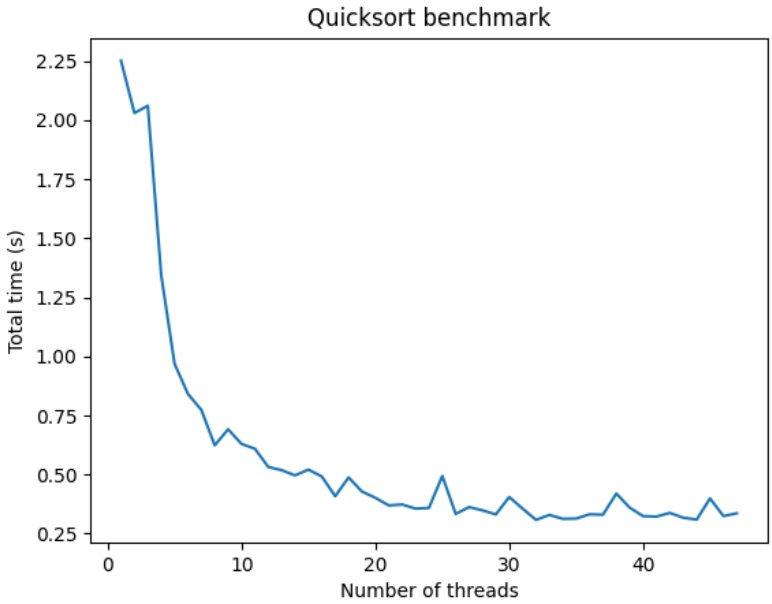

Figure 8: Benchmark results for the quicksort calculation using `OpenMP` tasks.