PC-2016/17 Course Project Template

Kai Schewina
kai.schewina@fu-berlin.de

Abstract

An integral image is an algorithm to sum up values in a table, such that each value is the sum of all entries left and above to it. We investigate a sequential and a parallel implementation of the algorithm in Python using the multi-processing library. The parallel implementation is achieved by using data parallelism for calculating the sum over the two axes of the integral image. We present an experiment with multiple setups and their respective run-times as well as speedup. As expected, the larger the data-set, the higher the speedup than can be achieved by parallelization.

Future Distribution Permission

1. Introduction

An integral image is an algorithm to sum up values in a table, such that each value is the sum of all entries to the left and above. It finds application in areas such as object detection, e.g. the famous Viola-Jones object detection algorithm by [1]. This is due to its property, that with only accessing four values of the integral image, any sum of elements in a rectangular shape can be found.

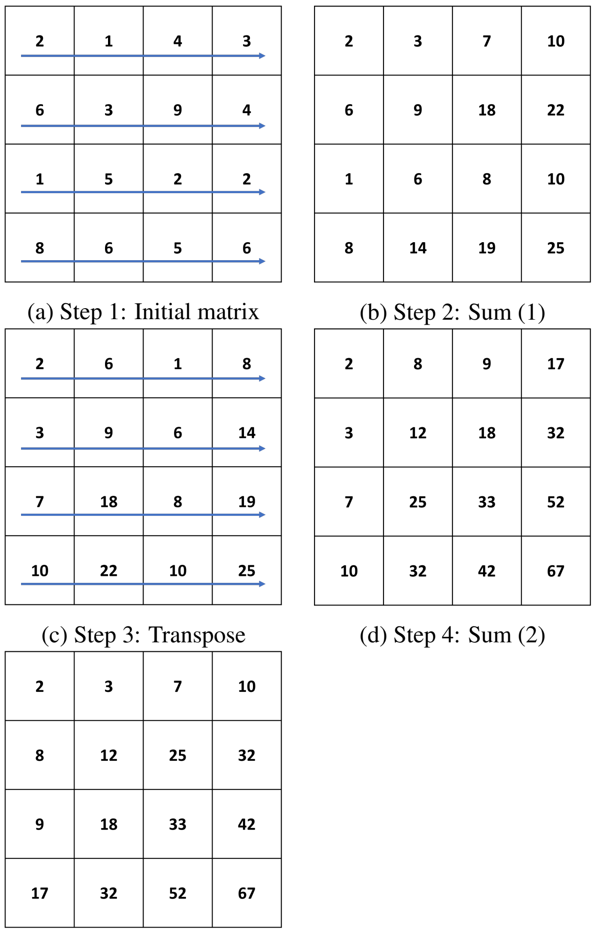
1.1. Formula

To calculate the value I of a point with the coordinates x and y, you sum up all values to the left and up. This is defined as follows:

$$I(x,y) = \sum_{\substack{x' \le x \\ y' \le y}} i(x', y')$$

2. Design

We can implement the task by calculating the cumulative sum over each row, transposing the matrix, calculating the cumulative sum over the rows again (that are the former columns) and transposing the matrix back to its original state. The process is visualized in Figure 1



(a) Step 1: Initial matrix

(b) Step 2: Sum (1)

(c) Step 3: Transpose

(d) Step 4: Sum (2)

(e) Step 5: Final matrix

Figure 1: Process Visualization

As a general approach we apply the multiprocessing library in Python. The data generation process is realized by creating a 2-dimensional list with random integers between 1 and 20. For reproducibility, a seed is set.

```
def create_data(x, y):
    random.seed(42)
    data = [[random.randint(1, 20) for i in
        range(y)] for j in range(x)]
    return data
```

The function that is being parallelized is the sum_chunk function, that iterates over all rows in a chunk (or the whole dataset in the sequential version) and calculates the cumulative sum for each element.

```
def sum_chunk(chunk):
    for f in chunk:
        sum = 0
        for index, e in enumerate(f):
            sum += e
            f[index] = sum
    return chunk
```

Finally, the 2-d list is transposed by use of zip and map.

```
def transpose_matrix(matrix):
    return list(map(list, zip(*matrix)))
```

The main function then executes the aforementioned algorithm. It sums the rows and transposes the result, followed by again summing and transposing the matrix. The sequential version is displayed below.

```
def main(size, data):
    data = sum_chunk(data)
    transposed = transpose_matrix(data)
    transposed = sum_chunk(transposed)
    final = transpose_matrix(transposed)
```

For parallelization a data-parallelism approach was used, by splitting the data into c chunks and passing it to the sum_chunk function, using the pool.map function.

```
def main(data, size, cpus):
    with mp.Pool(processes=cpus) as pool:
        chunk\_size = int(len(data)/cpus)
        chunks = [data[i:i + chunk\_size]
            for i in range(0, len(data),
            chunk\_size)]
        data\_row\_summed = pool.map(sum\
            _chunk, chunks)
        ...
```

3. Experiments

We test multiple setups to evaluate the performance. For the amount of cores 2, 4 and 6 were tested on a Ryzen 5500U with 16 GB of DDR4 RAM. For the data, quadratic matrices of size 2,400, 4,800 and 9,600 were tested out.

Parameter	Values
n	[2,400, 4,800, 9,600]
c	[2, 4, 6]

Table 1: Parameter Grid

The detailed runtimes can be found in the appendices. To account for cache warm-up, all experiments are conducted ten times and the average is taken.

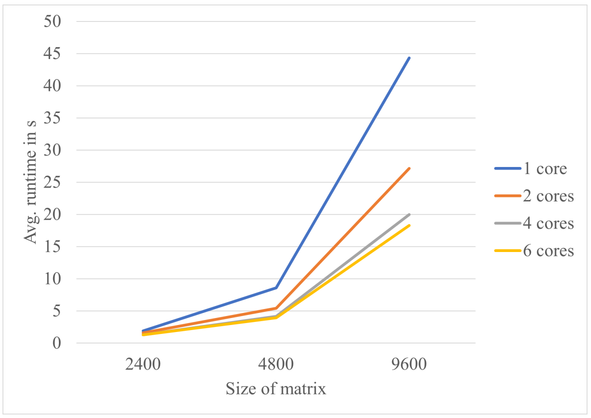


Figure 2: Runtime depending on size and cores

Above we display the runtime depending on the size of the quadratic matrix and the cores used. For a tabular format see the Appendix. Even for smaller data-sets there is a slight improvement in execution speed, that becomes larger with the data size.

log(n)	cores		
	2	4	6
2400	1.20	1.45	1.48
4800	1.59	2.06	2.19
9600	1.63	2.22	2.42

Table 2: Speedup

Finally, we present the speedup for the different setups in Table 2. Even for smaller data-sets we can see a speedup of 1.20. The larger the data-set and the more cores are used, the higher the speedup. Still, there is diminishing returns for usage of more cores.

References

[1] P. Viola and M. J. Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

4. Appendix

size	cores			
	1	2	4	6
2400	1.88	1.57	1.30	1.27
4800	8.59	5.40	4.17	3.92
9600	44.31	27.14	19.98	18.29

Table 3: Feature dimension m=3