Performance analysis

1. Please analyse the performances, including the execution time of the serial version from A1, parallel versions with different number of processes, speedup and efficiencies (15 marks).
2. You should provide line plots to represent the results (10 marks).
3. Choose your preferred parallel method and justify your choice (5 marks).

Outline:

1. Analyse the performances

1.1 Serial Version

1.2 Parallel Version

1.2.1 Terminology

1.2.2 Pthreads and OpenMP

1.2.3 Problem scale

2. Line plots

2.1 Time for Pthread and OpenMP

2.2 Speedup for Pthread and OpenMP

2.3 Efficiency for Pthread and OpenMP

3. Preferred parallel method

4. Reference list

1. Analyse the performances

1.1 Serial Version

Considering the serial version of the program, three functions are used: getPGMFile(), blur(), and save(). The structure PGMstructure is used to store the image information.

The specific running time data of the serial program is obtained through GPROF. The blur function accounts for nearly 99% of the time, therefore the focus of analysis and optimization is the blur function.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 98.84      5.94     5.94        1     5.94     5.94  blur
  0.67      5.98     0.04        1     0.04     0.04  save
  0.33      6.00     0.02                             fscanf
  0.17      6.01     0.01        1     0.01     0.01  getPGMFile
```

The pseudo code in the figure below is the final version. Using a custom PGMImage data structure to pass data to the blur function will effectively reduce the time by 20%.

```
1  blur(PGMImage* image)
2      for(i->20){
3          for(row)
4              for(column)
5                  *blur four corners of the image
6                  *if(first/last_row && first/last_column)
7                      original_data/5 -> new_data;
8                  else
9                       original_data/8 -> new_data;
10     }
```

In addition to the above pseudo code, we have also tried many other methods. Not only did the running time not decrease, but it increased by 20% on the basis of the original version. The following are several versions of the experimental results data

Unused struct	Other approach(used struct)	Last Version
4.916	4.268	3.398

Table 1: Three versions of Blur serially (unit: second)

The final version of the Blur function is:

- Traverse the data of the entire image,
- Process the data around the image in turn,
- Processing the intermediate data

In addition to relatively excellent performance, this version is also conducive to parallelization, so it will be used as the basis for subsequent parallelized versions of Pthreads and OpenMP.

Since the above running time will be affected by the quality of the machine, the time complexity is used to measure the serial blur algorithm.

The algorithm has two levels of for loops inside and outside, and for loop iterations of several judgment statements and constant terms, and the problem size is n. But because the for loop of the judgment statement and the constant term does not affect the order of magnitude of the time complexity. Thus, the time complexity is $O(n\text{^}2)$.

1.2 Parallel Version

1.2.1 Terminology:

Thread Number	p
Speedup	$S = \frac{T_{Serial}}{T_{Parallel}}$
Efficiency	$E = \frac{S}{p}$
Overhead Time	$T_{overhead}$

1.2.2 Pthreads and OpenMP.

Parallelizing serial programs with shared memory using Pthreads and OpenMP. The tasks in the Blur function are evenly distributed among the Threads, and no extra workload is added for each thread. Ideally, when p threads are used, the running speed is p times the serial speed, which is Linear Speedup.

According to Figure [1] and Figure [2], the values of $T_{parellel}$, S and E depend on the number of threads p. And speedup will not achieve linear speedup except for one thread.

The extra cost also needs to be paid attention to. Many parallel programs divide the tasks of serial programs, and increase the overhead required for parallelism between threads.

Pthread used in Figure1 and OpenMP used in Figure2 are shared memory parallel programs, therefore the parallel overhead of the two is similar. Because of the mutual exclusion relationship between threads, when the thread number is increasing, the efficiency decreases.

Figures below shows the time, speedup and efficiency of Pthread and OpenMP.

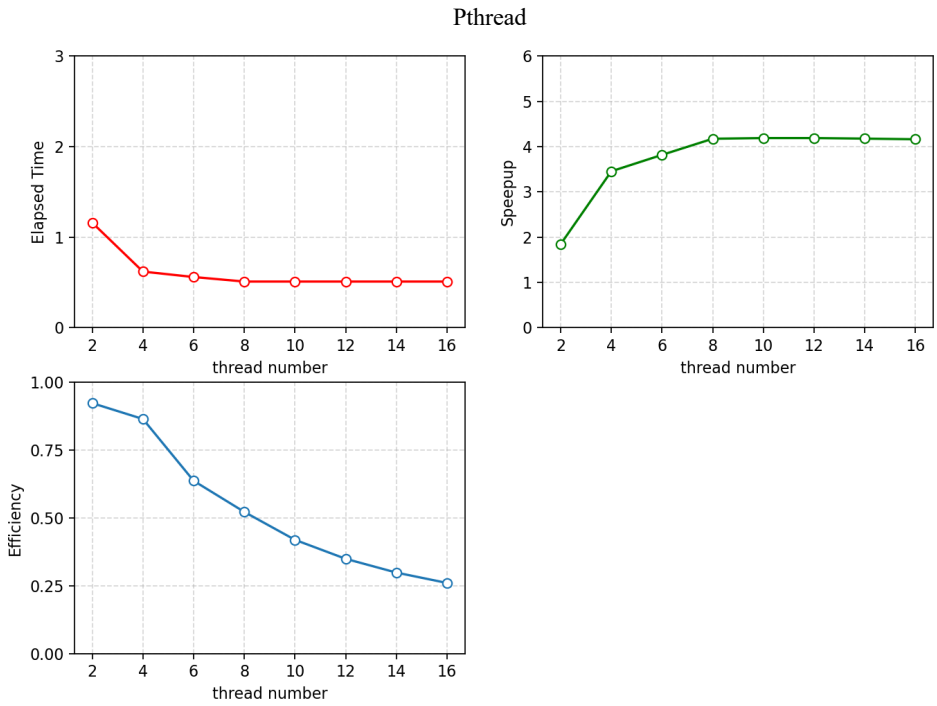

Figure 1: Elapsed time, Speepup and Efficiency for PThread

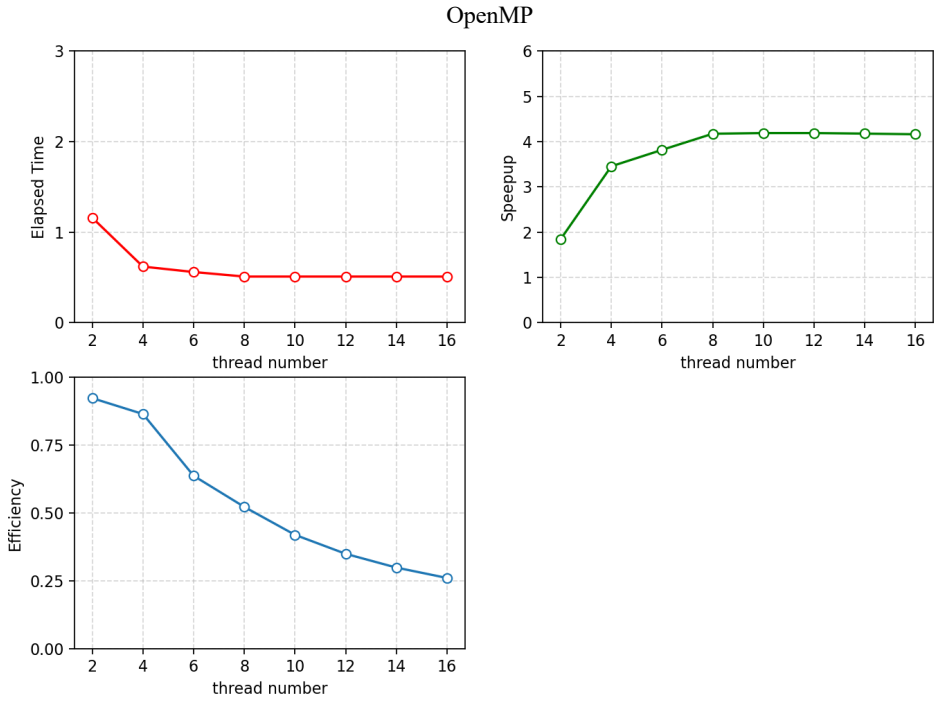

Figure 2: Elapsed time, Speepup and Efficiency for OpenMP

In addition, there are some tasks that cannot be parallelized in the program. According to Amdahl's Law, there are parts with a ratio of r in serial programs that cannot be parallelized, and the speedup that can be achieved is close to $\frac{1}{r}$ [1].

It can be seen from Figure 1 and Figure 2 that the parallelization programs of the two APIs both reach the upper limit of S when P=8.

The upper limit of Pthread S is 5, that is, the proportion of programs that cannot be parallelized by **Pthread** is about **20%.** The upper limit of OpenMP's S is 4, that is, the proportion of programs that cannot be parallelized when using **OpenMP** parallelization is about **25%.**

1.2.3 Problem scale

The values of $T_{parellel}$, S and E depend not only on the number of threads p, but also on the scale of the problem. When the scale of the problem grows, S and E increase at the same time. According to:

$$T_{parellel} = \frac{T_{serial}}{p} + T_{overhead}$$

As the scale of the problem increases, threads have more tasks to perform, and the relative time for coordinating work between threads is reduced. $T_{overhead}$ grows more slowly than T_{serial}, consequently both S and E will increase. And according to Gustafson's Law, as the scale of the problem increases, the proportion of the non-parallelizable part of the program also decreases [2].

Figure 3 and Figure 4 are Speedup and Efficiency using parallelized programs under different problem scales. When the scale of the problem increases from **512x512 to 4000x4000**, the non-parallelizable ratio of Pthread decreases from **25%** to **20%** (the upper limit of speedup increases from **4.0** to **5.0**). The non-parallelizable ratio of OpenMP decreases from **40%** to **25%** (the upper limit of speedup has been increased from **2.5** to **4.0**)

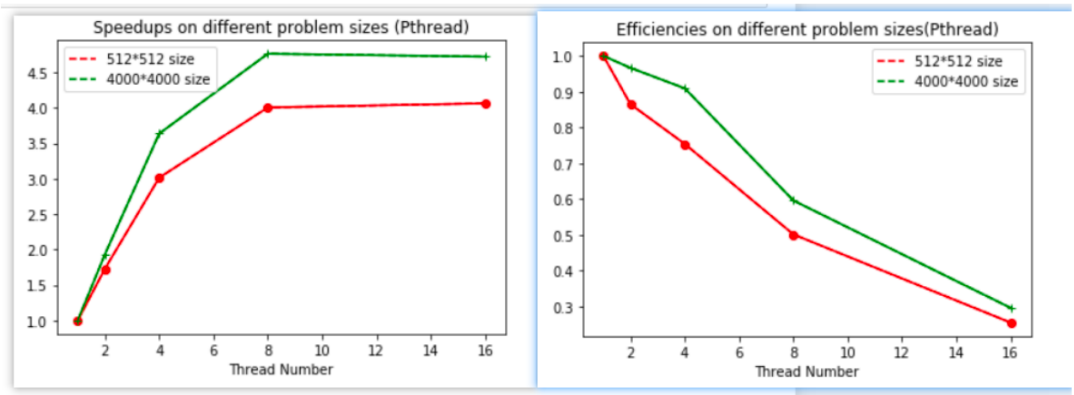

Figure 3: Pthread

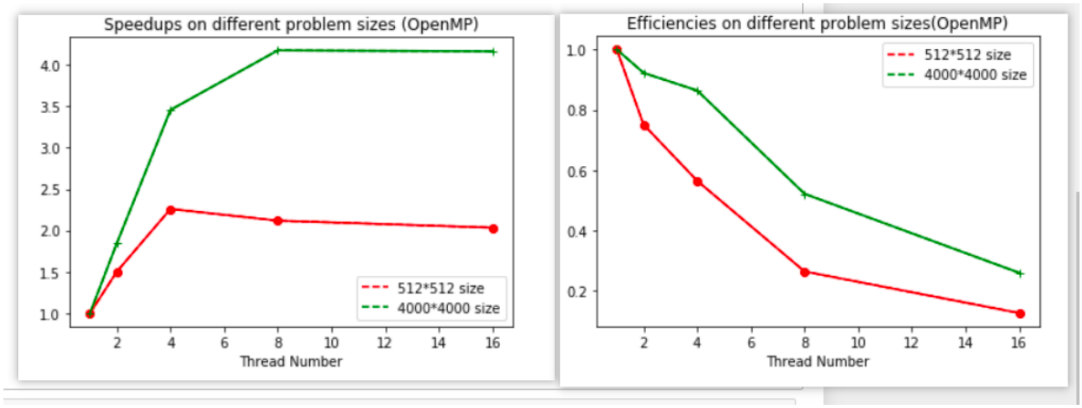

Figure 4: OpenMP

2. Line plots

2.1 Time for Pthread and OpenMP

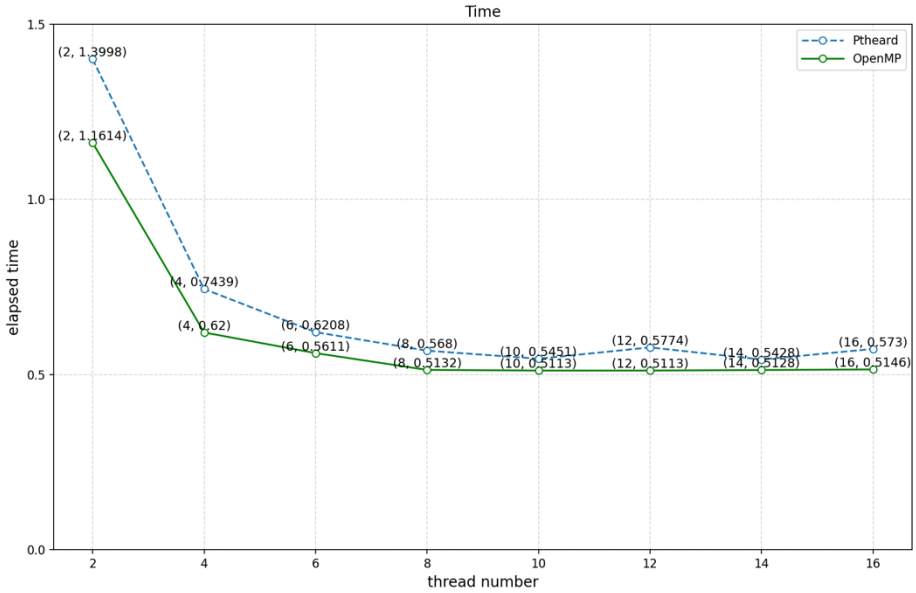

Figure 4

2.2 Speedup for Pthread and OpenMP

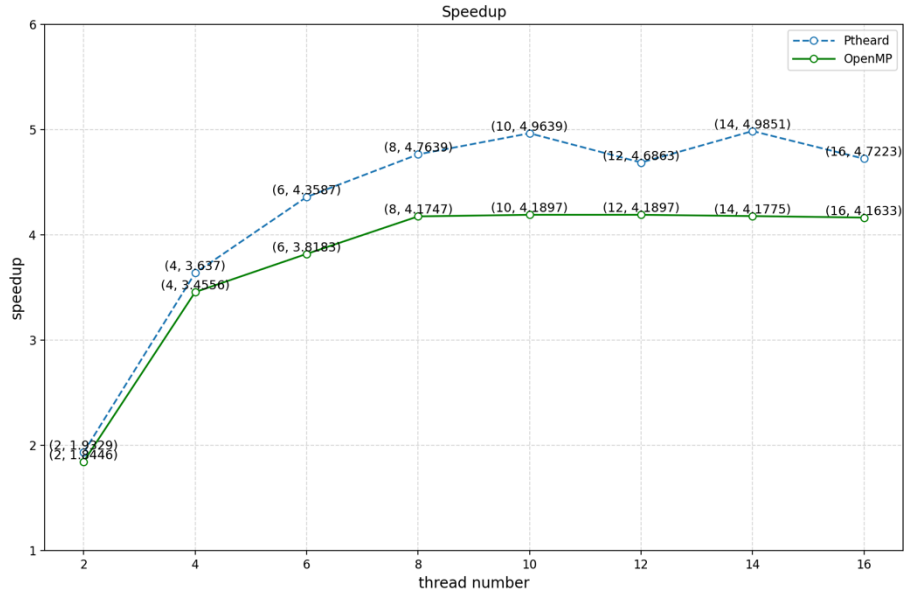

Figure 5

2.3 Efficiency for Pthread and OpenMP

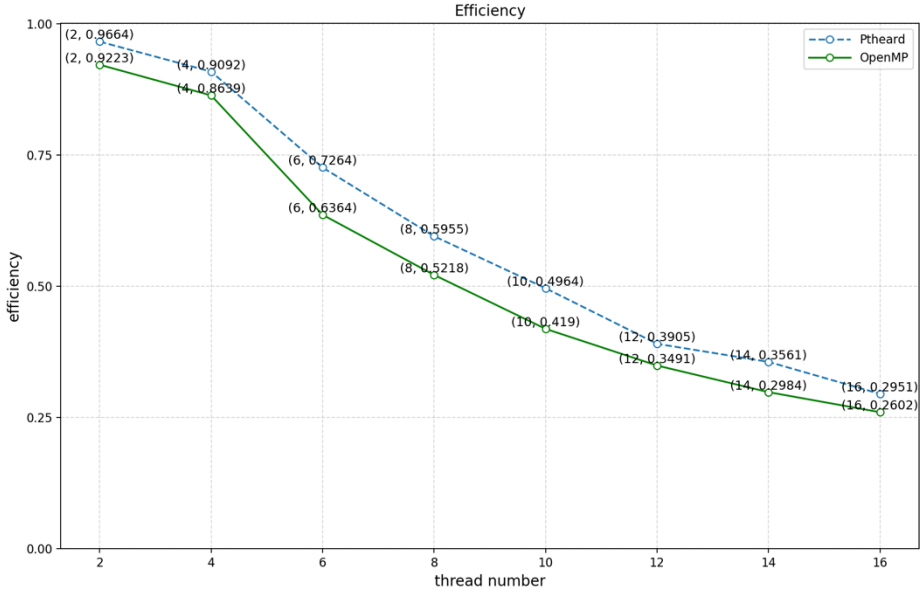

Figure 6

3. Preferred parallel method

Pthread is our preferred choice for this problem.

The approach is to treat the PGMFile structure as a shared memory space that all threads can access together, so that each process evenly obtains a part of the data in the PGMFile and blurs it, and finally stores the result in the array new_data for parallelization.

For OpenMP, it has much higher level, is more portable and doesn't limit you to using C, and it is also much more easily scaled than pthreads. One specific example of this is OpenMP's work-sharing constructs, which let you divide work across multiple threads with relative ease. But Pthread also has advantages such as fine control of thread management, and specific problems are analyzed in detail.

According to the comparison images of Pthread and OpenMP Speedup and Efficiency in the above Figure5 and Figure6, it can be seen that when the number of threads increases, the Speedup and Efficiency of Pthread are significantly greater than that of OpenMP. Thus, our team's final choice is Pthread.

4. Reference list:

[1] Krishnaprasad, S., 2001. Uses and abuses of Amdahl's law. Journal of Computing Sciences in colleges, 17(2), pp.288-293.

[2] McCool, Michael D.; Robison, Arch D.; Reinders, James (2012). "2.5 Performance Theory". Structured Parallel Programming: Patterns for Efficient Computation. Elsevier. pp. 61–62. ISBN 978-0-12-415993-8.