# Parallel Programming
# Assignment #1: OpenMP Smoother

Yifan Ge

February 21, 2013

## 1 Problem 1

Evaluate the functions `smoothSerialXY` and `smoothSerialYX` that iterate in column major and row major order respectively for kernels of size $3 \times 3$, $5 \times 5$, $7 \times 7$.

1. **What is the relative performance of the two functions. Give a runtime number (average of 20 trials) for both.**
   `smoothSerialYX` show better performance over `smoothSerialXY`.

| Running time (sec) | `smoothSerialXY` | `smoothSerialYX` |
|---|---|---|
| $3 \times 3$ | 5.4520186 | 4.3357826 |
| $5 \times 5$ | 11.9414182 | 10.8491422 |
| $7 \times 7$ | 21.8240202 | 20.5165304 |

Table 1: Performance comparison

2. **Explain the performance difference.**
   The performance difference is due to memory layout of array used in C. In C, arrays are stored in row major order. As the data stored in consecutive cache or memory spaces can be fetched at the same time, `smoothSerialYX` takes the advantage of this and these data can be used in the inner loop. While the data need in the inner loop of `smoothSerialXY` are not stored consecutively, more memory fetches are needed, and the data in the cache need to be written back to memory more frequently.

## 2 Problem 2

Create a parallel for loop implementation in the function `smoothParallelYXFor`. This should consist of adding an OpenMP parallel for directive around the outer loop of `smoothSerialYX`.

1. **Generate speedup plots for 1, 2, 4, 8, 16 and 32 threads. You will have to specify the number of threads to run using `omp_set_num_threads`.**
   Figure 1 shows the speedup for 1, 2, 4, 8, 16 and 32 threads.

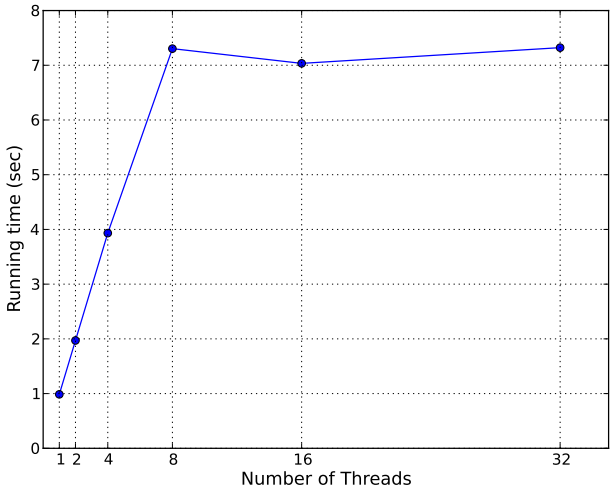


Figure 1: Speedup of `smoothSerialYX` for different number of threads

2. **Describe the results. What type of speedup was realized? When did speedup tail off? Why? What hardware did this run on and how did that influence your result.**
   With the number of threads increasing from 1 to 8, a linear speedup is realized. When the number of threads goes beyond 8, the speedup begins to tail off. This is because I am running this program on a 8-core machine. The optimal number of threads is 8. When the number of threads reaches 8, the processors have already been fully utilized, and the speedup cannot be improved more.

3. **Estimate the value of $P$ in Amdahl's law based on your results for 1 to 8 threads.**

For the number of threads ranging from 1 to 8, the result shows a linear speedup, so the value of $P$ is close to 1. Specifically, when the number of threads equals 8, the speedup is 7.30. Using the equation of Amadahl's Law:

$$Speedup = \frac{1}{1 - P + \frac{P}{S}}$$

Here $Speedup = 7.30$ and $S = 8$, we can get $P = 0.986$.

# 3 Problem 3

Create a parallel for loop implementation in the function `smoothParallelXYFor`. This should consist of adding an OpenMP parallel for directive around the outer loop of `smoothSerialXY`.

1. **Generate speedup plots for 1, 2, 4 and 8 threads.**
   Figure 2 shows the speedup of `smoothParallelXYFor` for different number of threads.

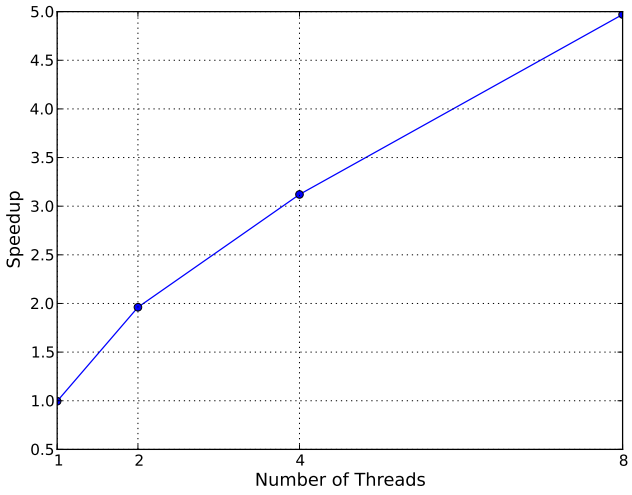


Figure 2: Speedup of `smoothParallelXYFor` for different number of threads

2. **Qualitatively compare the speedup of this code to the $YX$ parallel version. Explain the differences. *Note:* this question is about the relative speedup, not the relative performance of the serial versions.**

The speedup of `smoothParallelXYFor` is relatively lower than the speedup of `smoothParallelYXFor`.
As the memory spaces accessed are not consecutive in `smoothParallelXYFor`, they cannot be fetched together into the cache line. With more threads working and fetching data into the cache line, the data in the cache need to be write back to the memory more frequently than `smoothParallelYXFor`. Moreover, there is higher possibility that different threads are working on consecutive memory spaces, and this can causes more inferences which lowers the speedup.

# 4 Problem 4

Create a function `smoothParallelForCoalesced` that coalesces the two for loops in to a single loop.

1. **Generate speedup plots for 1, 2, 4 and 8 threads.** Figure 3 shows the speedup of `smoothParallelForCoalescesd` for 1, 2, 4 and 8 threads.

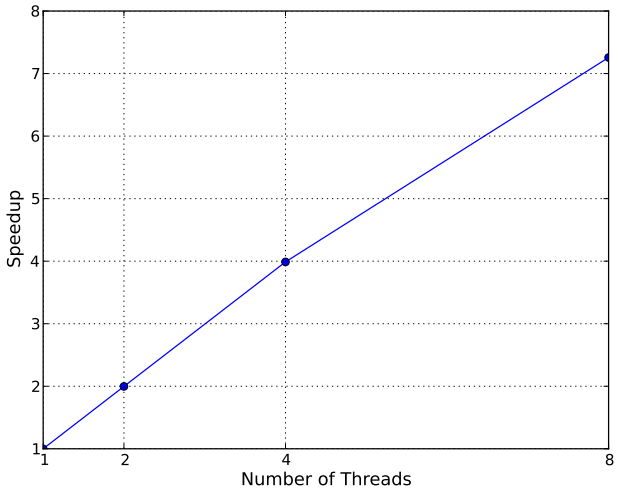


Figure 3: Speedup of `smoothParallelForCoalesced`

2. **Did coalescing the loop improve absolute performance or speedup? Explain why or why not (for both absolute performance and speedup.)**

No for both absolute performance and speedup. As for the absolute performance, `smoothParallelForCoalesced` uses the row major order to manipulate the array, which is no different from `smoothParallelYXFor`. So the way memory is accessed is the same, which leads to no absolute performance improvement.
As for the speedup, there is still no improvement. This is because we have made both `x` and `y` private in `smoothParallelYXFor` and the variables used are loop independent. So even though the loops are flattened, the ways OpenMP making the computations parallel are the same.

# 5 Problem 5

Write and compare the following two programs for computing the smoothed value of two arrays. Program #1 uses two invocations of `smoothParallelYXFor`. Program #2 merges the two loops, evaluating both arrays in a single parallel OpenMP for loop.

1. **Generate speedup plots for 1, 2, 4 and 8 threads.**
   Figure 4 shows the speedup of using two invocations of `smoothParallelYXFor`. Figure 5 shows the speedup of merging the two loops.

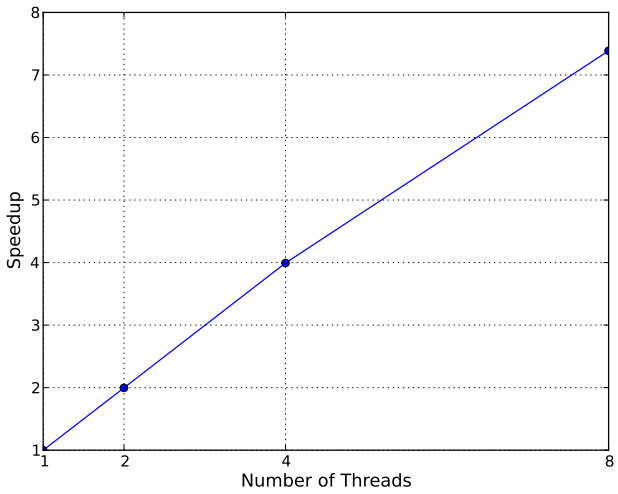


Figure 4: Speedup of Program #1

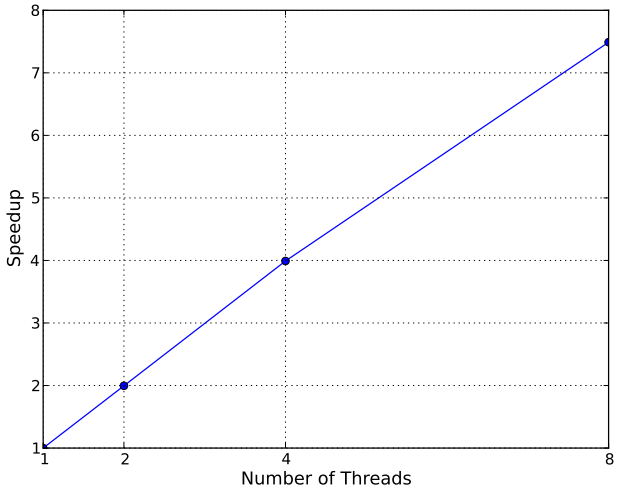


Figure 5: Speedup of Program #2

| Running time (sec) | Program #1 | Program #2 |
|---|---|---|
| Threads=1 | 22.0034534 | 21.87734 |
| Threads=2 | 11.0218692 | 10.9635202 |
| Threads=4 | 5.5110594 | 5.4814878 |
| Threads=8 | 2.978827 | 2.9210408 |
| Threads=16 | 3.0273454 | 2.9817048 |
| Threads=32 | 3.0069572 | 2.9419442 |

Table 2: Performance comparison of Program #1 and Program #2

2. **What is the relative performance of the two programs. What benefit does loop merging have on this code? Explain why.**

   Table 2 shows the comparison of the running time of Program #1 and Program #2. We can see the performance of merging the two loops is relatively slightly better than invoking `smoothParallelYXFor` twice.
   The benefits of merging the loops instead of calling the function twice are:

   (a) The intermediate results like `y * dim + x` can be reused. This reduces the cost to computing these values twice.

   (b) Merging the loops can have less startup costs.