National University of Singapore
School of Computing

CS3210 Assignment 1

Parallel Implementation of Gaussian Blur using POSIX Threads & OpenMP

with performance analysis

Syed Abdullah

October 13, 2017

Contents

1 Parallel Implementation using PThreads and OpenMP **2**
- 1.1 Overview . . . 2
- 1.2 Details of Parallel Implementations . . . 2
  - 1.2.1 Analysis of Sequential Implementation . . . 2
  - 1.2.2 General Implementation . . . 2
  - 1.2.3 PThreads Implementation . . . 4
  - 1.2.4 OpenMP Implementation . . . 5
  - 1.2.5 Utilising Intel AVX on both `blur_threads` and `blur_omp` . . . 5
  - 1.2.6 Potential Improvements . . . 5
- 1.3 Empirical Evaluation of Parallel Implementations . . . 6
  - 1.3.1 Tabulation of Results . . . 6
  - 1.3.2 Analaysis of Results . . . 6

2 Performance Analysis on Multicore Systems **7**
- 2.1 Experiment & Results . . . 7
- 2.2 Execution Times on Single Core . . . 7
  - 2.2.1 Comparing T_1(seq) and T_1(threads)/T_1(OMP) . . . 7
  - 2.2.2 Comparing T_1(threads) and T_1(OMP) . . . 8
- 2.3 Speedup using T_1(seq) versus T_1(OMP) . . . 8
- 2.4 Optimal # of Threads for 8 Cores . . . 8
  - 2.4.1 Results & Analysis . . . 9
- 2.5 Threading Overheads . . . 9
  - 2.5.1 Results . . . 10

Chapter 1

Parallel Implementation using PThreads and OpenMP

1.1 Overview

In this report, we will be investigating a parallelisation of Gaussian blur kernel convolution. More specifically, parallelising a Gaussian blur algorithm that utilises a 1D (linear) convolution kernel and applies it on a 1D (flattened) representation of the bitmap.

1.2 Details of Parallel Implementations

Before analysing the performance of the parallel implementations, we first have to understand the rationale behind the implementation that was chosen to improve the execution time of the program for the parallel implementation.

1.2.1 Analysis of Sequential Implementation

We first analyse how the sequential implementation, `blur_seq.c` applies a Gaussian blur. The main function that performs this Gaussian blur is `gaussian_blur` and this is executed three times, one for each colour channel (RGB).

For each convolution, either row-wise or column-wise, a "rolling buffer" consisting of neighbouring pixels is used to calculate the value of the current pixel using the following formula:

$$\texttt{dstPixel} = \sum_{i=0}^{ksize} \texttt{buffer[i]} \times \texttt{kernel[i]} \tag{1.1}$$

Note that `kernel` is a constant array throughout the convolution process. The result of a pixel convolution is written to an output array and eventually, to a new bitmap file.

1.2.2 General Implementation

Both PThreads and OpenMP are libraries to assist with the implementation of shared-memory multiprocessing programs. As both PThreads and OpenMP rely on threads, both implementations would result in similar code and almost similar speedup relative to the sequential program.

Type of Parallelism

Both implementations will utilise the parbegin-parend model and we will focus on data parallelism, as the convolution algorithm that are performed on all pixels are the same, just that there's a different `buffer` involved (refer to Equation 1.1).

Data Distribution

In order for us to speak of data parallelism, we first have to determine a way to distribute the data across all threads. For both implementations, we shall use a **blockwise** distribution of rows/columns in the convolution.

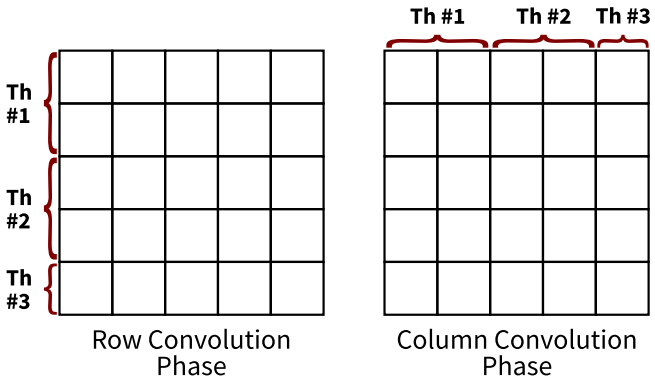

Figure 1.1: Illustration of Blockwise Distribution in Parallel Implementation

The reason for choosing this over other methods, such as **cyclic** or **block cyclic**, is to reduce the amount of communication and coordination between threads. With a blockwise distribution model, the block range that will be worked on by each thread is deterministic across multiple runs.

While a cyclic or block cyclic distribution would reduce the average idle time for each thread, these distributions do not provide a speedup that is significant enough to warrant extra synchronisation code. As the number of operations (adds, multiplies) are fixed per pixel for a specific ksize, on average, all threads (except a few that might have one less row/column to work on) will have the same execution time. Thus, to minimise complexity in the code, a simple blockwise distribution would lead to a significant speedup.

Inter-thread Communication

As this is a shared memory implementation, there's no need for us to perform any inter-thread communication, apart from synchronisation of threads, which we will cover in the next section. All necessary information exchange will be done through shared variables, without requiring information from other threads.

The kernel used in the convolution process does not change after it is first initialised and it is fixed for all rows/columns. Thus, the `kernel` variable can be shared between all threads, due to its read-only nature, without any consequences.

On top of that, the input/output image array can be shared between threads, as every output pixel is written by only a single thread (i.e. no issues with multiple writes). One problem that might be faced when sharing the output is **cache misses**, especially in the column convolution phase, due to the arrays being row-major. However, transposing the matrix after the row convolution phase would lead to a lot of cache misses, presumably because transposing requires us to read/write to columns. As such, there's no benefit to transposing the array to reduce the cache misses, as the act of transposition itself is equivalent to the cache misses that were experienced when reading/writing column-wise. Thus, the original code leads to minimum amounts of cache misses.

Parallel Execution of Gaussian Blur

```
1  int* kernel;
2
3  // parbegin
4
5  int* buffer;
6
7  row_convolution(rank, start, length);
8
9  barrier(rank);
10
11 column_convolution(rank, start, length);
12
13 // parend
```

Figure 1.2: Sketch of implementation

The parallelisation of Gaussian Blur can be simply done by parallelising the `gaussian_blur` function, more specifically, the two loops that convolves the image row and column wise.

Naively parallelising both loops will not work, as the column convolution can start only after all row convolutions have completed. As such, after the row convolution, we will have to establish a barrier where all threads would have to wait before proceeding on to the column convolution. While it may look as if this implementation may lead to idle threads, this is not a problem that is significant, as we have discussed earlier, the amount of work done by each thread is equal and that the computation time per thread is roughly equal. As such, the variation in runtime is not expected to be large.

1.2.3 PThreads Implementation

The POSIX Threads library only provides us with the ability to create/destroy and synchronise threads. Apart from that, it does not assume a specific method of or provide functionality for data distribution.

Parallel Task

The task that can be parallelised is extracted to a new method `work_on_image`. The shared variables are not passed in as arguments, instead they are defined as global variables.

In order to synchronise the two tasks, row and column convolution, between threads, we shall use a barrier or more specifically, `pthread_barrier_t` in which all threads will wait on after the row phase. This will ensure that the intermediate results are fully populated before any thread works on the columns.

Distribution of Work

As pthreads does not have any "work distribution" functionality, we have to determine the best way to partition the work up based on the data distribution. We can use the following equations to split up the work based on the thread's "rank" (which is supplied by the main thread as an argument).

$$\text{Start Index} = \left\lfloor \frac{N \times i}{P} \right\rfloor \tag{1.2}$$

$$\#\text{ elements} = \left\lfloor \frac{N \times (i+1)}{P} \right\rfloor - \left\lfloor \frac{N \times i}{P} \right\rfloor \tag{1.3}$$

where N is the number of rows/columns, P is the number of threads and i is the rank of the thread (0-indexed).

1.2.4 OpenMP Implementation

OpenMP provides a similar set of functionality as POSIX Threads. However, we can delegate the data distribution to OpenMP itself, using the construct `#pragma omp for`. Unlike in PThreads, there's no need for us to create/join or bookkeep threads in OpenMP, as it is done by OpenMP when we declare sections to be distributed.

With that in mind, we can make minimal modifications to the code without changing the overall structure of the program (as compared to the sequential version). We shall first define the parallel section, using `#pragma omp parallel`, which will encompass nearly the whole `gaussian_blur`, excluding the kernel, as it is constant throughout the whole convolution process, regardless of pixel position/values. Do note that the parallel section wraps the two `#pragma omp for`, as this will allow us to preserve the threads without unnecessary destruction of threads.

Unlike PThreads, we don't need a barrier mechanism as the `#pragma omp for` has an implicit barrier after the loop that is marked.

1.2.5 Utilising Intel AVX on both `blur_threads` and `blur_omp`

As Intel AVX allows us to perform operations on 256-bit registers (assuming standard AVX) and that a `float` (single precision) is 32 bits, we are able to perform the same operation on 8 `float` values at the same time.

As each pixel in the input image is made up of three colour channels, namely Red, Green and Blue, we can perform the convolution on all three colour channels for two consecutive pixels with one AVX operation.

To simplify the algorithm for determining the destination array (and in a way, add future support for alpha channels), we add an Alpha channel for each pixel, so that the two pixels can fit into a single 256-bit AVX register.

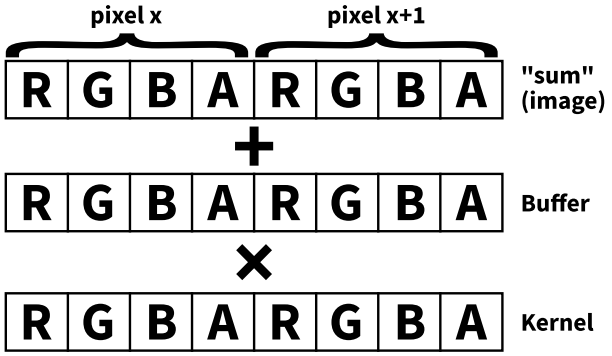

Figure 1.3: Illustration of AVX operations preformed in the `convolve` function

Some changes have to be made to the code to accommodate these new AVX instructions. They are as follows:

- `buffer` must be 8 times larger, so as to accommodate the different colour channels and thus, allowing us to load 8 consecutive values through one AVX instruction
- Ensuring that `buffer` is aligned to 32-bit boundary (using `posix_memalign`) – allows us to use the more efficient `_mm256_load_ps` than `_mm256_loadu_ps`

1.2.6 Potential Improvements

There are some improvements that were thought of, but they were not implemented in the final version of both programs. The reason for doing so is so that there's some precision in the results, such that the speedup can be discussed in greater detail.

One of which is to change the sequential process to use a "rolling sum" instead of the "rolling buffer". However, this makes it such that the parallelism only results in very trivial speedups, especially when compared to an "enhanced" serial implementation of said program.

1.3 Empirical Evaluation of Parallel Implementations

To measure the execution time of the various implementations, we utilised `perf`. All three programs were executed 10 times for every ksize and the lowest execution time across all 10 experiments were taken. As the sigma value does not affect the number of calculations being made, these measurements will only focus on execution time with respect to different kernel sizes. For the record, a sigma (σ) of 20 was used for all experiments.

This experiment can be replicated by running the following Bash file: `task1.sh`

1.3.1 Tabulation of Results

ksize	**Execution Time (s)**			**Speedup**	
	`blur_seq`	`blur_pthreads`	`blur_omp`	`blur_pthreads`	`blur_omp`
50	7.6288	2.1022	2.0845	3.62	3.65
100	11.0743	2.2231	2.2492	4.98	4.92
150	15.5397	2.3733	2.3918	6.54	6.49
200	19.5479	2.5844	2.5950	7.56	7.53
250	23.5189	2.7972	2.8008	8.40	8.39
300	28.5161	3.0293	3.0360	9.41	9.39

Table 1.1: Execution Time & Speedup for both sequential and parallel implementations (`image1.bmp`)

1.3.2 Analaysis of Results

From the results & calculations, we can see that the speedup increases as the kernel size increases. This is expected as with a larger `ksize`, it allows us to utilise more of AVX and ensures that the additional cost of parallelism is offset by the larger amount of computation that needs to be done.

Chapter 2

Performance Analysis on Multicore Systems

The Parallel Gaussian Blur implementations allow us to investigate how the number of cores utilised affects the execution time of shared-memory multiprocessing program. This can be done by physically limiting the number of cores that the program can run on.

To preserve the fairness of this experiment, we shall use a 4096×4096 image (more specifically, `romania.bmp`) and perform the Gaussian Blur with a 1024 kernel size.

2.1 Experiment & Results

The experiment can be replicated by executing the following Bash script: `task2.sh`

Cores	Execution Time (s)		Speedup					
			using T_1(seq)		using T_1(threads)		using T_1(OMP)	
	Threads	OMP	Threads	OMP	Threads	OMP	Threads	OMP
1	262.8703 (seq) 77.1338 (threads)	77.0217	3.4079	3.4129	1	1.0014	0.9985	1
2	40.2916	40.2086	6.5241	6.5376	1.9143	1.9183	1.9116	1.9155
4	21.8560	21.8294	12.0273	12.0419	3.5291	3.5334	3.5240	3.5283
6	17.2432	17.2715	15.2448	15.2198	4.4732	4.4659	4.4667	4.4594
8	14.3463	14.4026	18.3231	18.2515	5.3765	5.3555	5.3687	5.3477

Table 2.1: Tabulation of Execution Time & Speedup for different # of cores with kernel size of 1024 & 64 threads

2.2 Execution Times on Single Core

We observe a difference in T_1 (i.e. one core) values between the sequential, PThread and OpenMP implementations. This difference is especially prominent between the sequential version and any one of the parallel implementations.

2.2.1 Comparing T_1(seq) and T_1(threads)/T_1(OMP)

This section will investigate the difference between the sequential program and both parallel implementations. This is done as both PThreads & OMP implementations are almost identical to each other. We can see that $T_1(\text{seq}) >> T_1(\text{threads})/T_1(\text{OMP})$. The reason why the threaded version runs faster may be due to various factors:

- AVX implementation – vectorization of the convolution would mean that there's less processor instructions executed

- Almost no contention – as there's only one thread performing the convolution, there's no other threads that may "invalidate" the cache (especially in column convolution). Thus, this lowers the time needed to read/write to cache due to the lowered number of cache refreshes
- Gain in AVX > Loss to parallel overheads – the AVX code may have resulted in a speedup that offsets the loss of speedup due to the parallel overheads introduced

2.2.2 Comparing T_1(threads) and T_1(OMP)

The value of T_1(threads) is approximately equal to T_1(OMP). One possible reason as to why is due to how both parallel implementations results in the same parallel system, with minute differences. Both implementations utilise data parallelism, shared memory and threads. As such, since the underlying OS/Machine architecture remains unchanged, both of them will execute the task in the same manner.

Thus, while there might be differences in values between T_1(threads) and T_1(OMP), these differences provide us with inconclusive results and trends and they can be better attributed to factors like CPU load and task scheduling in the OS level.

2.3 Speedup using T_1(seq) versus T_1(OMP)

We can observe that the speedups with respect to T_1 (seq) is larger than T_1 (OMP).

- Speedup w.r.t T_1 (seq) – measuring performance gain of **both** the improved implementation (i.e. AVX and cache aligned memory) and from the actual data parallelism itself
- Speedup w.r.t T_1 (OMP) – a measure of performance gain/loss of varying the number of threads working on the problem, without taking into consideration the improvements made from the serial version

2.4 Optimal # of Threads for 8 Cores

When running a parallel program, we need to ensure that the optimal speedup is achieved. One way to achieve this is to have multiple processes work on the problem in parallel, which in the case of single-machine multicore systems, would mean increasing the # of thread executing the parallel program.

While this might seem like a good idea, there are limitations, especially once the number of threads exceed the number of physical/logical (dependent on architecture) cores. When $n_{threads} >> n_{cores}$, each thread has to compete for a limited number of cores (PUs). The overhead from context switching, albeit small for threads, may outweigh the speedup of finely distributing the problem to more threads.

One simple experiment that can be carried out to determine the optimal number of threads is to measure the execution time of both the threaded and OMP versions, while varying the number of threads used. We shall increase the number of threads up till the point where we don't observe an increase (or even, a decrease) in speedup relative to T_1. The reason why we don't have to continue is precisely because of physical limitations and diminishing returns in speedup when $n_{threads} >> n_{cores}$. From there, we can determine the optimal number, which is one that leads to the maximum speedup possible (global maximum).

2.4.1 Results & Analysis

Threads	Time (s)		Speedup	
	`blur_omp`	`blur_threads`	`blur_omp`	`blur_threads`
1	76.7663	77.2407	1	1
2	40.2040	40.1530	1.9094	1.9236
3	27.9776	27.8092	2.7438	2.7775
4	21.8167	25.7769	3.5186	2.9965
5	20.2149	20.0989	3.7975	3.8430
6	17.8992	17.6287	4.2888	4.3815
7	16.0142	15.9221	4.7936	4.8511
8	14.3186	14.2773	5.3612	5.4100
9	14.7023	14.6101	5.2213	5.2867
10	14.6641	14.6778	5.2349	5.2623
11	14.4955	14.4954	5.2958	5.3286
12	14.5491	14.4723	5.2763	5.3371
13	14.4315	14.5230	5.3193	5.3185
14	14.4847	14.4467	5.2997	5.3465
15	14.3723	14.3857	5.3412	5.3692
16	14.3328	14.3139	5.3559	5.3962

Table 2.2: Execution Time with varying number of threads on both PThreads and OpenMP

The results can be better visualised in a graph form.

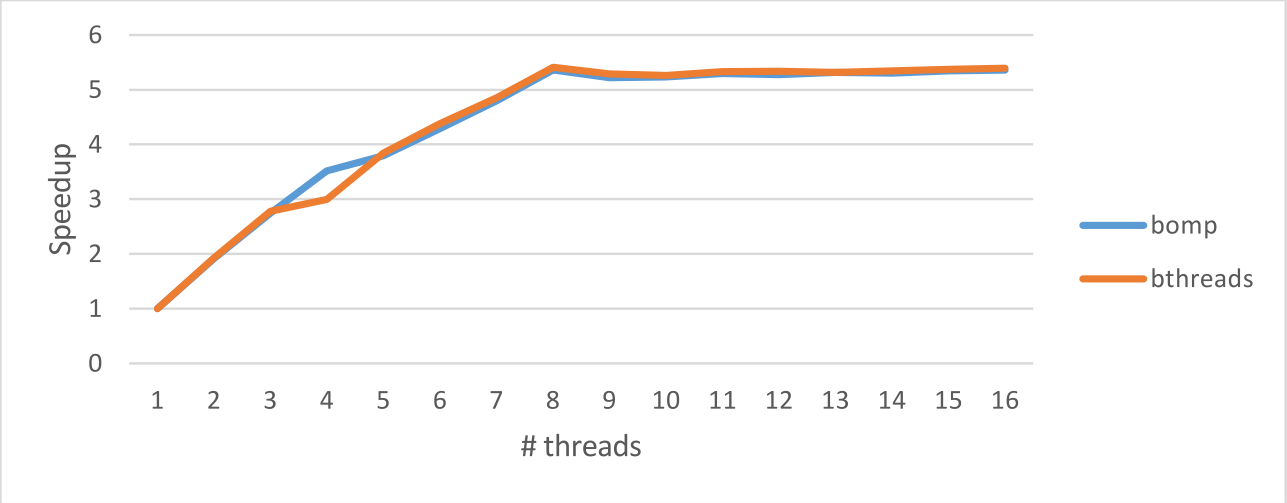

Figure 2.1: Speedup with respect to number of threads (both implementations)

We can see that the maximum/peak speedup that can be achieved (relative to T_1) is when the number of threads is 8. This corresponds to the number of logical cores in the machine, as the Intel i7-2600 CPU has 4 physical cores, but each core is able to "act" as 2 cores, due to HyperThreading.

Thus, we can conclude that the optimal number of threads is the total logical cores count of all CPUs in a machine.

2.5 Threading Overheads

Parallelising a program introduces significant costs in the form of parallel overheads, which facilitates the parallelism. In this case, it is the overhead due to the thread management and synchronisation (create/join).

To calculate the thread overhead, we can utilise the equation for calculating parallel efficiency:

$$E_p = \frac{T_1}{p \times T_p} \tag{2.1}$$

Thus, if the parallel implementation is **as efficient as** the sequential implementation, $E_p = 1$. Since we assume that our parallel implementation will not be as efficient, that is to say, $E_p < 1$. We can consider T_{pi} as the ideal execution time for p threads. Thus,

$$T_1 = p \times T_{pi} \implies T_{pi} = \frac{T_1}{p} \tag{2.2}$$

(as $E_{pi} = 1$) and we can say that $T_p = T_o + T_{pi}$, where T_o is the overhead for p threads. Through manipulating the given equations, we can get the following equation

$$T_o = T_p - \frac{T_1}{p} \implies T_{o1} = \frac{T_o}{p} \tag{2.3}$$

where T_{o1} is the average thread overhead per thread

2.5.1 Results

Program (Threads)	T_1	T_p	T_o	T_{o1}
`blur_threads` (2)	77.2407	40.1530	1.5326	0.7663
`blur_omp` (2)	76.7663	40.2040	1.8208	0.9104
`blur_threads` (8)	77.2407	14.2773	9.4497	0.5777
`blur_omp` (8)	76.7663	14.3186	9.5207	0.5903
`blur_threads` (16)	77.2407	40.1530	9.4863	0.5928
`blur_omp` (16)	76.7663	40.2040	9.5349	0.5959

Table 2.3: Thread overhead calculations for 2, 8 and 16 threads

Using our knowledge gained from the previous experiment, to get the most optimal value for the thread overhead, we shall use the number of threads where the speedup is the greatest. As such, using a thread count of 8, we get an average thread overhead of around 0.58 seconds per thread.

The thread overhead increases as we go beyond the peak speedup and that is consistent with our knowledge that there's no longer any benefit in adding additional threads. However, the total overhead increases as there's one more thread to perform such overhead operations on.