HPC Lab 3: Report

Rajdeep Pinge 201401103, Aditya Joglekar 201401086

September 13, 2016

Part I

Hardware Details:

- CPU model: AMD A10-5750M APU with Radeon(tm) HD Graphics
- memory information:
  1. L1d cache: 16K
  2. L1i cache: 64K
  3. L2 cache: 2048K
- No. of cores: The CPU has 4 physical cores with 1 thread per core. This means, there are 4 logical cores.
- Compiler: gcc version 5.4.0 20160609
- precision used: We have used double precision wherever possible so as to get more accuracy in the result.

Part II

Q1. Image Warping

a) Write a serial code for twist transformation/ image warping of an image as discussed in the class (lecture-10, case study-3). Write a parallel version of this using openMP. Change the image dimension (same image) and compare the serial vs. parallel implementation.

1 Context:

- Brief description of the problem.

We have been provided with helper code for reading and writing the image data in ppm format. We had to use the twist transformation to warp the image.

- Complexity of the algorithm (serial).

  The algorithm runs over image space and during each iteration performs some computation for the twist in constant time. Hence the serial running time is of O(n^2)

- Possible speedup (theoretical).

  We run the code to find the fraction of parallelizable code

  Fraction of parallelizable code (p) = 0.999929

  In theory also, since the nxn image is traversed using two loops and the instructions of one iteration are not dependent on any other iteration, therefore the loop and hence the whole program can be parallelized.

  Therefore by Amdahl's Law the Theoretical speedup that we should get = No. of processors = 4 in this case.

  Note that this is a problem which is extremely parallel in nature. Each pixel in the output image has to borrow its value from some pixel in the input image whose location is just being calculated by using the rotation and the interpolation. Hence there is absolutely no communication required and so the work can be divided amongst the threads coarsely. Such coarse form of parallelism is a characteristic of many image processing problems.

- Profiling information (e.g. gprof). Serial time.

  granularity: each sample hit covers 2 byte(s) for 7.68

  With reference to Table 1, The warping function has a double loop. In each iteration of that loop, the bilinear interpolation routine is called. The profiling information indicates that most time is spent in this routine funciton. However note that this function has no scope for data parallelism. We do not see any loops or work which could be shared. Hence our efforts must be directed towards parallelizing the for loop in the warping function.

- Optimization strategy.

  We will use the following opimization strategies to parallelize and optimize the processing time.

  1. Since this is an embarrassingly parallel problem we do not need to put any big thought in parallalizing the code. We just put a openmp parallel pragma for over the double loop taking care to define the shared and parallel variables properly.
  2. We can use the collapse clause of the openmp so that it combines the total work to be done in the two 'for' loops and performs them

Table 1: Call-Graph Profiling table

index	% time	self	children	called	name
		0.03	0.10	1/1	main [2]
[1]	100.0	0.03	0.10	1	warping [1]
		0.10	0.00	939093/939093	bilinearinterpolation [3]
[2]	100.0	0.00	0.13		main [2]
		0.03	0.10	1/1	warping [1]
		0.00	0.00	1/1	readPPM [4]
		0.00	0.00	1/1	writePPM [5]
		0.10	0.00	939093/939093	warping [1]
[3]	76.9	0.10	0.00	939093	bilinearinterpolation [3]
		0.00	0.00	1/1	main [2]
[4]	0.0	0.00	0.00	1	readPPM [4]
		0.00	0.00	1/1	main [2]
[5]	0.0	0.00	0.00	1	writePPM [5]

equally among all threads. Collapse actually divides all the pixels among the 4 threads instead of just dividing the rows i.e. the first 'for' loop.

3. We can also use static scheduling as well as the dynamic scheduling

- Problems faced in parallelization and possible solutions.

  It was much harder to get the serial functionality correct in this question. The correct warping effect was achieved only after getting the equations right with theta and the dynamic calculation of the distance from the centre.

  Also if we reference the pixels outside the image it gives a segmentation fault since in terms of programme, we are doing an illegal memory access with respect to heap memory. Hence it was difficult to get the warping of image correctly in the first place.

  Later, we needed to properly implement the bi-linear interpolation in order to get the precision in the warping.

  We also were stumped for sometime because of a regrettable mistake in computing the parallel time. The serial code had used the clock() function to measure the running time. We forgot to use the omp_get_wtime() function in the parallel code. Hence the clock function was returning the cumulative processor time + overhead. We were left wondering how come our parallel code was running slower than the serial code.

2 Hardware details:

We have allready put all the hardware details at the beginning.

Here the cache size does not matter much since there is no reuse of the image data. The reuse ratio is 0 and hence no use of calculating cache hit ratio.

3 Input. Output

The input for both the serial and parallel codes is a ppm image.

The output should be suitably warped image in ppm format.

4 Parallel overhead time:

(openmp version on 1 core vs serial without openmp)

- CASE I:

  Image size = 1024 x 1024

  Both serial and parallel code give the same warped image

  Serial code: Time taken = 0.331304 s

  Parallel code (1 thread): Time taken = 0.341747 s

  Overhead = 0.341747 - 0.331304 = 0.010443 s (3.15

- CASE II:

  Image size = 512 x 512

  Both serial and parallel code give the same warped image

  Serial code: Time taken = 0.073487 s

  Parallel code (1 thread): Time taken = 0.078641 s

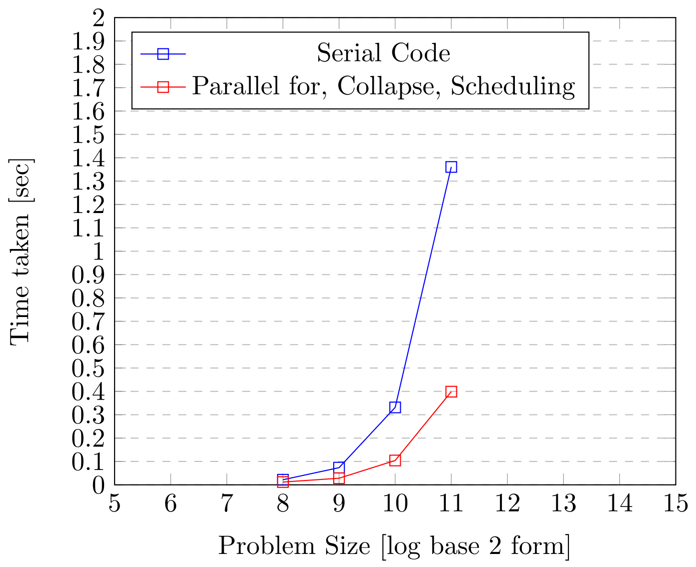
  Overhead = 0.078641 - 0.073487 = 0.005154 s (7.01

5 Problem Size vs Time (Serial, parallel) curve:

We have taken the number of cores equal to 4 throughout.

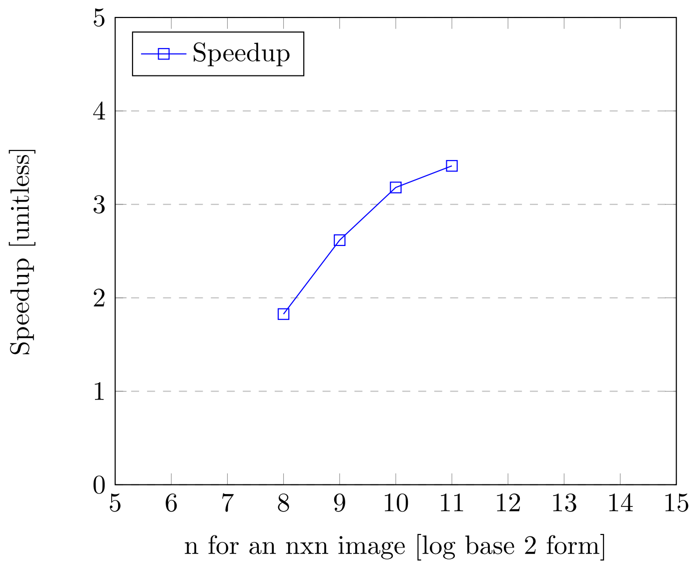
The above graph shows that when the image size is less, the amount of time taken for warping is close in both cases since the overhead time of the parallel code is comparable to the total time of serial code. As the size increases, the processing part increases and the communication decreases comparatively. Hence the time taken by the parallel code in comparison to the serial code decreases

Graph to plot the variation in time compared to the image size in log form



6 Problem size vs. speedup curve.

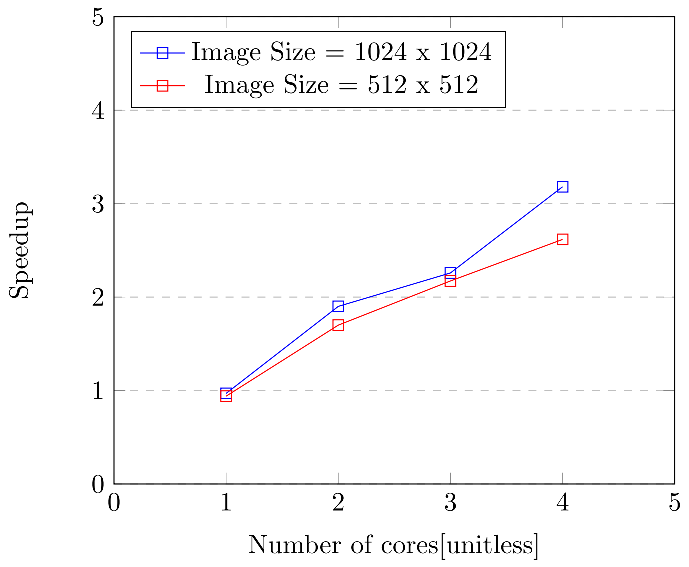
Graph to plot the variation in speedup compared to the size of the image (in log form)



A point to be noted. Here the speedup is increasing with increase in problem size. As the image size increases, the amount of processing increases but the communication overhead does not increase with the same rate. Hence granularity and speedup increase. However, the rate of change of speedup with problem size is decreasing towards the end. This means that the problem is not scalable as speedup is not able to keep pace with increase in problem size.

7 No. of cores vs. speedup curve for a couple of problem sizes.

Graph to plot the variation speedup compared to the number of cores



It is clear that as the number of cores increase, the speedup increases. It is clear that when the problem size is different the change in speedup is different. Observe that the red graph approximately linear but bends downwards towards the end indicating that the overhead increases faster than the processing as the number of cores increase. The blue graph should have been linear as well but for the unfortunate kink at 3 cores.

8 Measure performance in MFLOPS/sec.

According to the problem, we have total 87 Floaing Point Operations per cycle of for loop and we have rows*columns no. of such cycles. Therefore we have total 87*rows*columns FLOP. We measure the time taken to execute those many operations.

$$performance = No.ofMFLOP/time_taken(sec)$$

For parallel code, performance = 8773.058088 MFLOPS
For serial code, performance = 2751.430404 MFLOPS

The performance of the parallel code is more than thrice the serial code. But theoretically it should have been 4 times! The reason that the perfomance is 3 times is because of the pipeline stalls. Since the operations contain trigonometric operations, each of them takes different amount of time in the CISC architecture and hence the length of the pipeline differs as mentioned in the lecture. Also since only execute phase of the pipeline is used, there is no optimal use of pipeline and hence the performance optimization is limited.

Part III
Q2. Image Filtering

9 Context:

- Brief description of the problem.

  We have to use the median filter to modify each pixel of the image by replacing it with median value of RGB components from its surrounding square stensil defined by half-width h.

- Complexity of the algorithm (serial).

  The algorithm runs over the whole nxn image and for each pixel iterates over its surrounding stensil h^2 times. Therefore, the complexity is O($n^2 * h^2$).

  Assuming h is much smaller than total image, and is constant for each iteration of outer two loops, we can assume the complexity to be O(n^2).

- Possible speedup (theoretical).

  We run the code to find the fraction of parallelizable code

  Fraction of parallelizable code (p) = 0.999694

  In theory also, since the nxn image is traversed using two loops and the instructions of one iteration are not dependent on any other iteration, the loop and hence the whole program can be parallelized.

  Therefore by Amdahl's Law the Theoretical speedup that we should get = No. of processors = 4 in this case.

- Profiling information (e.g. gprof). Serial time.

  Flat profile:

  Each sample counts as 0.01 seconds.

  In Table 2, we have taken the flat profiling information since it is easy to understand. It shows that programme spends most of the time in partition and quick-sort functions which are needed to find the median. For the time being, we don't have any specific method of parallelizing

Table 2: $Flat_Profiling Table$

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
85.96	2.73	2.73	27676308	0.00	0.00	partition
8.06	2.98	0.26	786432	0.00	0.00	quick_sort
5.69	3.16	0.18	1	0.18	3.16	filter
0.32	3.17	0.01	1	0.01	0.01	writePPM
0.00	3.17	0.00	1	0.00	0.00	readPPM

the sorting algorithms. Hence we look at the other functions. The filter function which has 4 for loops and actually calls the quick-sort takes the next significant time and can be parallelized.

- Optimization strategy.

  We will use the following opimization strategies to parallelize and optimize the processing time.

  1. Since this is an embarrassingly parallel problem we do not need to put any big thought in parallalizing the code. We just put a openmp parallel pragma for over the double loop taking care to define the shared and parallel variables properly.
  2. We can use the collapse clause of the openmp so that it combines the total work to be done in the two for loops and performs them equally among all threads. Here also we can collapse the outer two loops. It is not possible to collapse the inner two loops which move on stencil since there is inter-dependency between the loop variables.
  3. We can also use static scheduling as well as the dynamic scheduling
  4. We can try to find a different code which makes use of the cache reuse efficiently for example, image tiling.

- Problems faced in parallelization and possible solutions.

  The code had four fold nesting of loops. The outer two run over the image space and the inner two loops are to calculate the median. We decided not to parallelise the inner two loops. So the threads will divide the pixels amongst themselves and for each pixel the median will be calculated serially.

  The important thing to bear in mind was the declaration of private and shared variables. If you get this wrong the code obviously does not work correctly.

  For each pixel, the neighbouring pixel values are stored in 3 r,g,b arrays. It is important to make these arrays private. We had this bug in the code. Took some time to figure it out.

10 Hardware details:

Since there is possibility of cache reuse due to the stencil moving over the same pixels many times, it is important to get to know the cache line size of the system. It can be obtained by typing in the following commands: getconf LEVEL1_DCACHE_LINESIZE, getconf LEVEL2_CACHE_LINESIZE

For level1 cache it is = 64 bytes, for level2 cache also it is = 64 bytes.

11 Input parameters. Output.

The input parameter for both the serial and parallel codes is a ppm image and the output is the median filtered image.

The output is same in both the serial and parallel cases.

An interesting observation: As we increase the size of the image without increasing the half-width, the image tends to get clearer because of the closeness of all the pixels and the abundance of pixels in the given area. Hence to get the same blurring we need the increase the stencil size and hence the half-width.

12 Parallel overhead time:

(openmp version on 1 core vs serial without openmp)

- CASE I:

  Image size = 256 x 256, half-width = 3

  Serial code: Time taken = 1.409908 s

  Parallel code (1 thread): Time taken = 0.906250 s

  Overhead = 0.906250 - 1.409908 = -0.503658 s

- CASE II:

  Image size = 512 x 512, half-width = 3

  Serial code: Time taken = 5.679317 s

  Parallel code (1 thread): Time taken = 3.597656 s

  Overhead = 3.597656 - 5.679317 = -2.081661 s

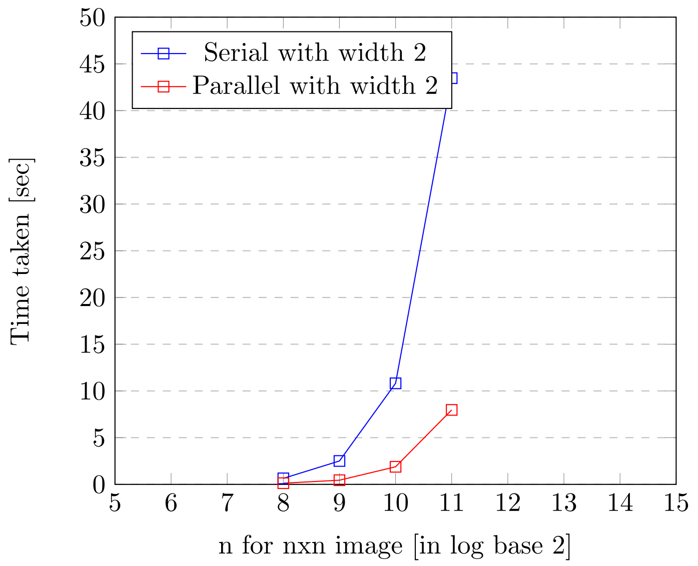
Here even with a single core we are getting some speedup. This may be because this is a shared memory environment and when the code is parallelized, the amount of available cache increases, which reduces the data access time and hence we are able to get super-linear speedup.

13 Problem Size vs Time (Serial, parallel) curve:

We have taken the number of cores equal to 4 throughout.

As expected, the time taken increases as the image size increases. Note here that the time taken is very high as compared to the time for warping in previous questions for the same image sizes. This is because of high amount of data access in the median filter.

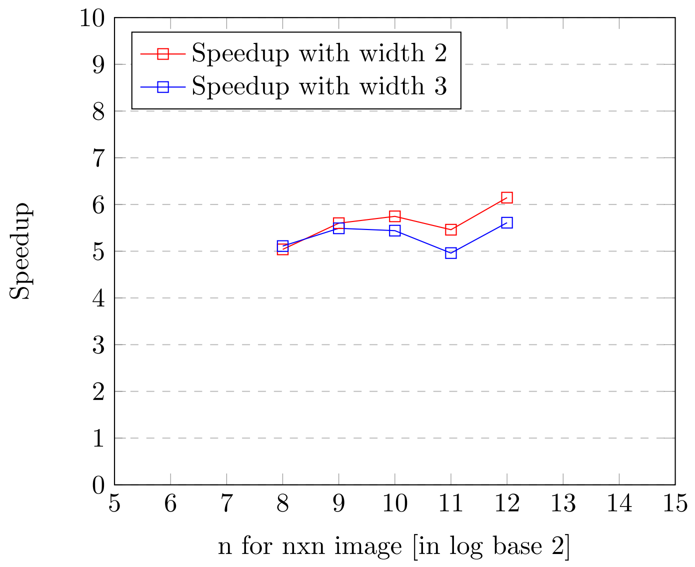
Graph to plot the variation in time compared to the image size in log scale



14 Problem size vs. speedup curve.

A point to be noted. Here the speedup is increasing with increase in problem size however, the rate of change of speedup with problem size is decreasing towards the end. The indicates the change in speedup in powers of 10 in the end is decreasing. This means that the problem is not scalable as speedup is not able to keep pace with increase in problem size

Graph to plot the variation in speedup compared to the image size in log scale

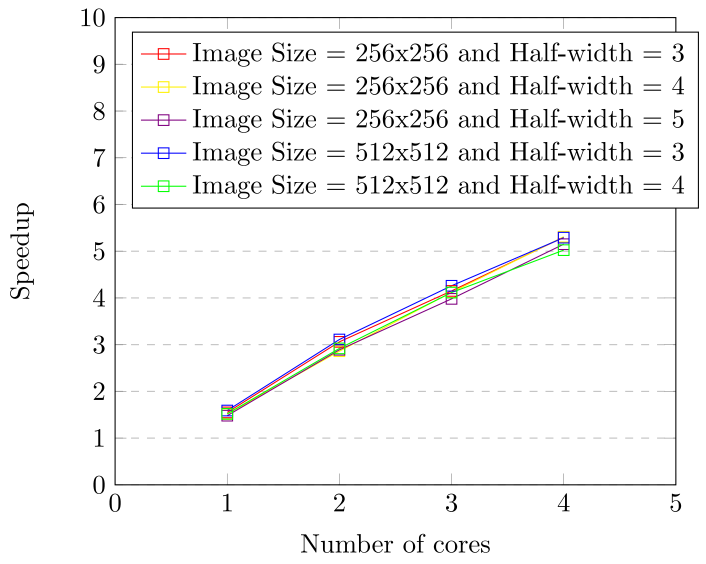


15 No. of cores vs. speedup curve for a couple of problem sizes.

Notice that between the blue and red plots, where the image size has been changed, the speedup curves are almost identical. Normally we expect that with increase in input size, the savings in processing time mask the overhead. But in this case we speculate that since the code is embarassingly parallel, the threads are already working so well for size 256x256, ie the overhead is already negligible, so that even with size 512x512, the gain is not very much.

The same logic applies to when we change the half width. Even if greater half width entails more computation, the embarassingly parallel nature keeps the speedup almost the same.

Graph to plot the variation speedup compared to the number of cores



16 Measure performance in MFLOPS/sec. For 512X512 and half-width=3

According to the problem, we have total 6 Floaing Point Operations per cycle of for loop and we have k such cycles. Therefore we have total 6*k FLOP. We measure the time taken to execute those many operations.

$$performance = No.of MFLOP/time_taken(sec)$$

For parallel code, performance = 9.119507 MFLOPS

For serial code, performance = 1.715637 MFLOPS

We got super linear speedups for this problem. ie for our quad core processor, we are used to getting a maximum speedup of 4x. The reason we were getting such speedups is that when we access the neighbours of a pixel and then move onto the next, many neighbours are common between the two. Hence there is a lot of cache reuse due to spatial locality in this problem. Also we are using 4 bytes of memory for each pixel and at a time we load 7 such pixels in a row taking 28 bytes which is well within the cache line length of 64 bytes. Whenever we load any of the pixels within these 64 bytes, due to memory alignment rule and due to the cache table property, (only 1 logical addresses for a given physical address), the same 64 bytes must be loaded in memory. But those are already in memory which amounts to very high cache hit ratio and cache reuse ratio. The amount of cache available also increases since the number of processors increase. Hence the cache hit ratio increases and memory access time drastically decreases.