Assignment 3
Microbenchmarks
Syed Hassan Jalil - 11161604
Trisha Anand - 11434104

Building Microbenchmarks

The first task of this labs assignment was to build a microbenchmark suite. We needed to have results for the following microbenchmarks

- Memory Bandwidth for different memory levels
- Arithmetic Operations
- Synchronization for OpenMP

System Configuration for Microbenchmarks

First of all we would like to mention the system these benchmarks were written for and ran on to obtain value. We did this on my personal computer, A Macbook Pro Late 2016 w/o Touchbar.

It has the following configuration

- Intel® Core™ i5-6360U
- 4MB Cache
- 8 GB RAM
- macOS Sierra v 10.12.4

Memory Latency for Different Memory Level

The first benchmark we needed was to measure the performance of different memory levels. We needed to calculate the latency and bandwidth for different memory levels (L1,L2,L3 cache and memory).

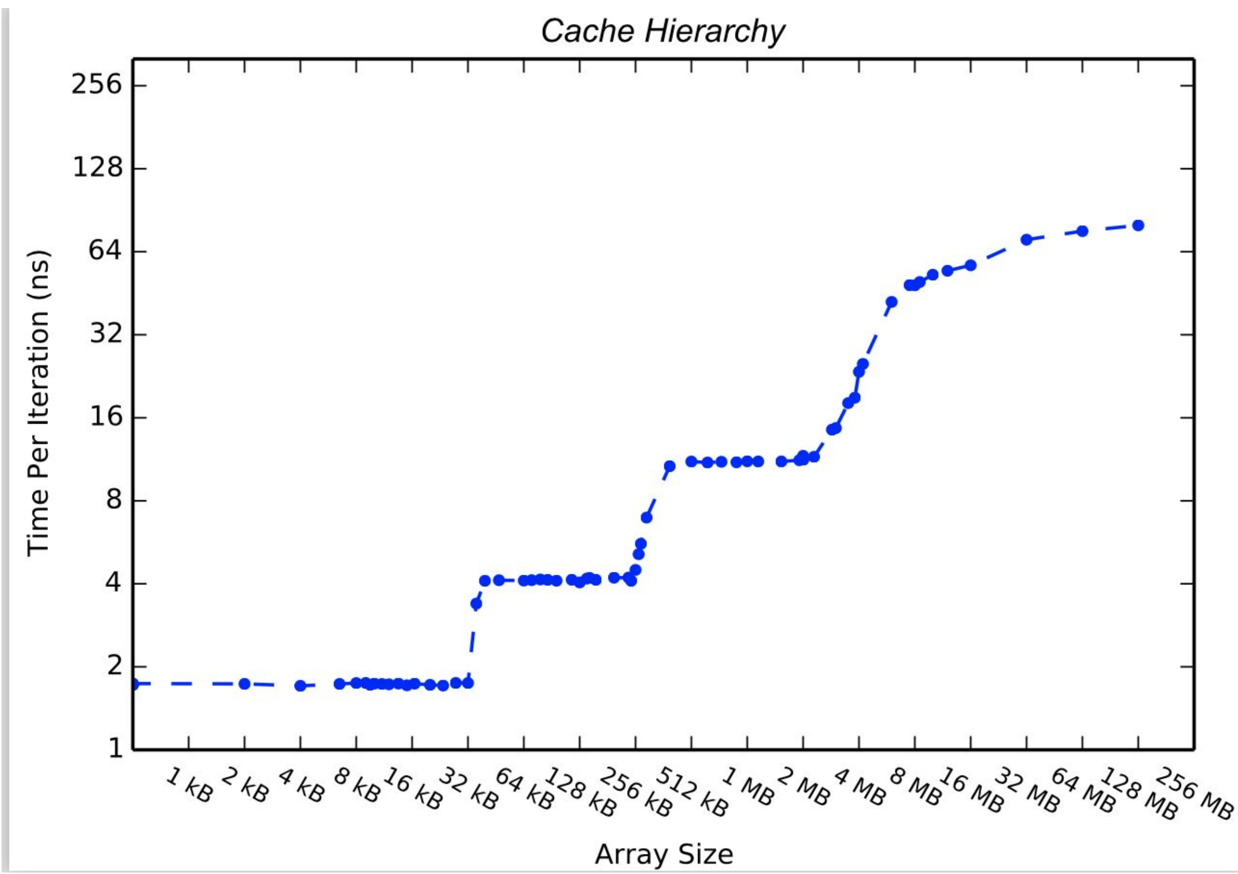
For this purpose, we utilized the help of ccbenchmark suite [1]. This suit can be used to run a number of different benchmarks. Ccbench is a small collection of micro-benchmarks designed to empirically characterize some of the interesting parameters of a processor and its memory system. The microbenchmark that we needed were the ones written for cache. The Microbenchmark uses the following algorithm

[1] https://github.com/ucb-bar/ccbench

```
cctime_t volatile start_time = cc_get_seconds(clk_freq);

   intptr_t idx = 0;

   for (uint32_t k = 0; k < g_num_iterations; k++)
   {
      idx = arr_n_ptr[idx];
   }

   cctime_t volatile stop_time = cc_get_seconds(clk_freq);


Repeat with increase size of strides
```

The microbenchmark reports time for different sizes of strides, and plots a neat graph using the library matplotlib[2].

We ran this microbenchmark for our machine and got the following plots.



[2] https://matplotlib.org

From this plot we can clearly see the sizes of the different levels of Cache

L1 cache is 32 KB
L2 cache is 512 KB
L3 cache is 4MB

The microbenchmark also gives values for latency of all these cache

Memory Level	Latency (ns)
L1	1.69622
L2	4.06417
L3	10.8003
Main Memory	70.2747

Memory Bandwidth for Different Memory Level
We can calculate the memory bandwidth using the latency from the above mentioned test. The test also shows the size of data that was fetched from each level of memory to calculate the average latency. Using the latency values and the size of data, we can calculate the Bandwidth in MB/s

Memory Level	Size of Data	Bandwidth (MB/s)
L1	8kB	4603176.474754
L2	64kB	15369435.825716
L3	1 MB	740720.165181

Arithmetic Operations
For arithmetic Operations we decided to take help from stream benchmarks [3]. Stream is a very simple benchmark tools that help us with different arithmetic operations. By default stream can be used to get the time for addition operations, and triad operations (A+ scalar*B). We modified the code to find the time for simple multiplication. The algorithm for this microbenchmark is as following

[3] stream benchmark osu

```
for (k=0; k<NTIMES; k++)
{
     times[0][k] = mysecond();
     for (j=0; j<STREAM_ARRAY_SIZE; j++)
           c[j] = a[j]+b[j];
     times[0][k] = mysecond() - times[2][k];

     times[1][k] = mysecond();
     for (j=0; j<STREAM_ARRAY_SIZE; j++)
           a[j] = b[j]*c[j];
     times[1][k] = mysecond() - times[3][k];
}
```

This microbenchmarks ensures that the values are calculated by using them in the next step, in the end it compares them to the expected results to make sure the test ran correctly (in our case this is false because we have replaced the triad operation with multiplication).

This benchmark is run for an array size of 10000000, hence it performs the computations 10000000 times. Moreover, we run it for a total number of 50 iterations to average the results. Running the microbenchmark we get the following results

Operation	Time for 10000000 execution (s)	Time for one execution(ns)
Addition	0.025242	2.5242
Multiplication	0.024994	2.499

<u>Synchronization for OpenMP</u>

The Last microbenchmarks we wanted to run were for openMP. We wanted to know the overhead of creating Parallel For loops and the overhead of using critical section. For this purpose we decided to use EPCC OpenMP micro-benchmark suite[4]. This suite provides a whole host of different benchmarks for openMP but we were interested only in those benchmarks which played a role in our code. They are as follows :

[4] https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite

- Overhead of creating Parallel For Loop
- Critical Section

It uses the following algorithm to calculate the values

```
//Test for Parallel For loop over head
void testpfor() {
    int i, j;
    for (j = 0; j < innerreps; j++) {
#pragma omp parallel for
        for (i = 0; i < nthreads; i++) {
            delay(delaylength);
        }
    }
}

//Test For Critical Section Over head
void testcrit() {
    int j;
#pragma omp parallel private(j)
    {
        for (j = 0; j < innerreps / nthreads; j++) {
#pragma omp critical
            {
                delay(delaylength);
            }
        }
    }
}

//delay adds a predetermined amount of delay (pause) and then the time is subtracted from
the total time.
```

We get the following results by running the microbenchmark

Task	Time (ms)
Parallel For	0.069919
Critical Section	0.090688

Matrix Multiplication

In this part of the assignment, we take the microbenchmark values and apply them to the models we made for our matrix multiplication. We have two version of matrix multiplication, the sequential version, and the parallel version.

Sequential Version

The model for the sequential version is as following

T =(3N1 + 3xN1M2 +5xN1M2M1)*t_addition + (N1M2 + N1M2M1)*t_mult

We injected the values into this model, and compared our results to those of the execution time

Size	Model Time (s)	Actual Time (s)
100x100	0.01522147326	0.000957823
500x500	1.8925216863	0.070920248
1000x1000	15.130079172	0.568554688

Parallel Version

The Model for parallel version is
T = (((N1 + 2*N1M2 + 5*N1M2M1)/k)*t_addition)+ (((2*N1 + N1M2 + N1M2M1)/k)*t_mul)

Size	**Mode Time (s)**	**Actual Time (s)**
100x100	0.00759	0.000001843
500x500	0.945945305	0.000049236
1000x1000	7.563777461	0.000366058

As its clearly evident our model times are much higher than the actual times. We think that this may be due to compiler optimizations and super scalar operations, but we do not think that should entirely justify a difference of order 10^2. We have rechecked our model multiple times to make sure it is correct and cross checked the implementation to ensure it is performing the calculations, but we can not seem to find what is wrong with our model. Hopefully we can discuss this in the upcoming lecture.

Parallel Histogram

We had a parallel implementation of histogram program. We decided to make a very simple application in C. It calculates greyscale histogram, taking into account 256 different greyscale shades. We do not read actual images in our application, but just initialize them randomly in code. We used openMP to parallelize the code.

We use the following basic kernel to calculate the histogram

```
#pragma omp parallel for shared (histo) private (i,j)
	for (i = 0; i < width; i++) {
		for (j = 0; j < length; j++) {
			index = image[i*(length) + j];
			/* START CRITICAL REGION */
			#pragma omp critical
			{
				histo[index] += 1;
			}
			/* END CRITICAL REGION */
		}
	}
}
```

As we can see from the above code, the array histo is shared between all threads, and since accessing it concurrently could lead to race condition, we have added it inside the critical section, so that only one thread can access it at a given time. This fixes the potential race condition, but unfortunately also makes this part of the code sequential.

Performance Model

We would like to make a basic performance model of this.
Let w= width of image in pixels and h=height of image in pixels
And let N be the total number of threads.

We assume that all threads are equal in performance. The load is perfectly balanced and all the threads run for the exact same time. In a more realistic scenario, we would want to calculate the maximum time taken by each thread.

T= t_compute+t_commuinicate + t_ompOverhead
Since there is no communication involved,
t_communicate =0

t_compute = t_addition + t_multiplication
Number of Additions = (w + 2(w*h))/N + w*h
Number of Multiplication = (w*h)/N

Overhead=(w/N)*t_pFor + (w*h)t_critical

T = (((w + 2(w*h))/N + w*h)*t_addition) + (((w*h)/N)*t_multiplication) + ((w/N)*t_pFor + (w*h)*t_critical)

Where t_addition is time for one addition operation and t_multiplication is time for one multiplication operation t_pFor is the overhead of parallel For loop and t_critical is the overhead of creating critical section

Testing the Model

We test the model on the same machine we ran the microbenchmarks on. We use 4 openMP threads as the machine has 4 cores.

Size	Actual Time (s)	Model Time(s)	Error
100x100	0.063623	0.00095280158	98.5%
500x500	1.057123	0.0237838179	98.09%
1000x1000	4.071925	0.0951171608	97.7%

As expected, we have very high error percentage. This is because our model is very basic, and we haven't taken into account the t_memory, which is the time it takes to access memory and fetch data which is not found in cache. Since memory access is quite slow as compared to the overhead and computation time, the errors are quite high. Our error percentage remains consistent across the three image sizes which reinforces the fact that our model is accurate in its limited scope, but misses some essential values.

Statistical Performance Model

For statistical performance model, we would need a large set of data to create an adequate model. The performance of histogram depends on number of factors :

Size of Image
The run time of the algorithm directly depends on the size of image, the bigger the image is the longer the program will run to calculate the histogram.
Distribution of color
The number of colors and their distribution in an image also plays a role in the time it takes to calculate histogram. An image with a single solid colors results in longer execution time than an image in which colors are more distributed. This is because when the image has a single color, all threads try to update the same bin location and hence have to wait for other threads to release the lock.

So, for our statistical model we have to impose some limitation. It is not possible to come up with a model which caters to all the sizes of image and different distributions of colors. So for our model, we assume that the images have a normal distribution of colors. We do not have images with just solid colors, or gradient of few colors only, we assume that the Colors are normally distributed throughout the image. This is based on the assumption that most images in real world will follow this pattern.

Moreover our model will only be based on CPU, since our application is parallelized using openMP

Under these given conditions, We would need the following things

- A large input set of images, varying in pixel sizes from 10x10 pixel to all the way up to 10000x10000 pixels (uniformly increasing).
- Access to a large variety of testing machines, with the latest hardware, we would like to use the following processors
  - Intel® Core™ i7-7700K Processor
  - Intel® Core™ i5-7600K Processor
  - AMD R7 1800X
  - AMD R5 1600X

- 16 GB DDR4 2,666MHz RAM Across all systems
- A Linux based operating system (preferably CentOS v6.6)

We would then write a script, that runs the code with varying number of openMP threads from 2 up to the number of physical cores on the CPU for each image and write the resulting time in a CSV. This process will be repeated 5 times and the average values will be calculated.
This should then give us enough data to make a model from. Its best to represent the data in the form of a scatter plot (one plot per hardware configuration) for ease of understanding.
We can also use linear regression on the collected data to predict the time taken for image sizes that have not been tested or CPU that have not been tested. With this extensive amount of input data, our model should be able to predict the performance quite accurately.