CS 133 Homework 2
(Dennis Zang, 704766877)

1.Given an integer array a[ ] of N elements of value between 1 to m as the input, please write an efficient OpenMP function to generate the histogram h for array a[ ] such that h[i] is the number of elements in a with value i (1 <= i <= m). The function header is: void histogram(int *a, int *h). In addition, you can use constant variables N and m in your function directly. Is there a possibility for race condition in your implementation? If so, how do you handle it?

```
#include <omp.h>
using namespace std;

#define N 1000
#define m 100

void histogram(int *a, int *h)
{
	for(int i = 0; i < m; i++)
	{
		h[i] = 0;
	}
	#pragma omp parallel for schedule(guided) num_threads(8) reduction(+:h[:m])
	for(int i = 0; i < N; i++)
		h[a[i]]++;
}

//There is a race condition when dealing with the counter for each value in h. To handle this race condition, we would need to
//make incrementing each individual counter a critical section. The easiest way to do so would be to add a reduction for each
//element in array h as shown above (with the clause "reduction(+:h[:m])").
```

2.Please write an OpenMP program to compute the numerical value of the integration of the function sqrt(x)/(1+x^3) between 0 and 1 using 16 threads. Your goal is to make it as efficient as possible. Please present two ways to deal with possible race conditions and compare their efficiency.

```
#include <cmath>
#include <vector>
#include <iostream>
#include <omp.h>
using namespace std;

#define A 0.0
#define B 1.0
#define STEPS 10000

void integral1()
{
	float area = 0;
	float stepSize = (B - A) / STEPS;
	#pragma omp parallel for schedule(guided) num_threads(16) reduction(+:area)
	for(int step = 0; step < STEPS; step++)
	{
		area += stepSize * sqrt(A + (step * stepSize))/(1 + pow((A + (step * stepSize)), 3));
	}
	cout << area << endl;
}

void integral2()
{
	float area = 0;
	vector<float> stepAreas(STEPS, 0);
	float stepSize = (B - A) / STEPS;
	#pragma omp parallel for schedule(guided) num_threads(16)
	for(int step = 0; step < STEPS; step++)
	{
		stepAreas[step] = stepSize * sqrt(A + (step * stepSize))/(1 + pow((A + (step * stepSize)), 3));
	}
	#pragma omp parallel for schedule(guided) num_threads(16) reduction(+:area)
	for(int step = 0; step < STEPS; step++)
	{
		area += stepAreas[step];
	}
	cout << area << endl;
}

//Above are two implementation of finding the integral of sqrt(x)/(1+x^3) usin Riemann sums. "integral1()" does this by
//immediately summing up each rectangle partition and adding it to a shared variable (a critical section dealt with using
//a reduction). The total time (user time + system time from "time ./<program name>") was about 11ms.
//"integral2()" works differently in that it allocates a variable for each rectangle to calculate the area of (buffering them
//using a vector), so that no race conditions would be an issue when calculating each area. However, we would then still need
//to add all of the sub-partitions (a critial section dealth with a reduction again). The total time for "integral2()" was 13ms.
//Not surprisingly, "integral1()" was faster since it only had to go through a loop 0 to STEPS - 1 once, and didn't have to spend
//extra time to dynamically allocate memory.
```

3.Given the following OpenMP program (segment) running on four CPU cores using four threads, assuming that the computation of function f(i, j) takes one minute on a single CPU core, and we ignore that scheduling overhead. You are asked to experiment with different scheduling methods. Please estimate the completion time under each of the following schedule schemes:

(a) default scheduling

The fastest time this would run in would be 15 seconds (assuming f(i, j) has equal execution time uniformly distributed over all i and j). However, if this is not the case (i.e. f(i, j) increases as i and j increases), then the execution time would likely be much longer than 15 seconds.

(b) schedule (dynamic, 2)

This would most likely run longer than 15 seconds, as we cannot guarantee that the workload of each "f(i, j) is equal. If f(i, j) increases as i and j increases, then dynamically scheduling tasks as the tasks get longer would make it more likely that, when there are less iterations left over than cores, some cores would go unutilized. This is especially as bad as the chunk size has a size of 2.

(c) schedule (guided, 1)

This would probably run slightly over 15 seconds (if each chunk allocated to each core each time is about equal, which is very likely).

4.How one may use large memory bandwidth to hide high memory latency? Please outline two ways and discuss the additional resources needed.

(method 1) (prefetching)

One way to use large memory bandwidth to hide high memory latency would be to load as much memory that will be needed in the future ahead of time (i.e. in a for loop, we can load the necessary data from memory for as many iterations as possible to decrease the number of times we need to load from memory). Additional resources needed for this option would be a large cache size to store this memory, and that the data is memory is contiguous (spatial locality).

(method 2) (multithreading)

Another way to do this would be to have multiple threads work independently from one another, and load data from memory for their own use. Like prefetching, we load a lot of data at once, but each loaded data partition will be immediately processed. The resources required for this would be a large enough cache size allocated to each thread (so cache misses won't be to large).

5.Given the baseline processor described in Lecture 5 (1Ghz clock frequency with two multiply-add units), assuming that it has a cache of 32KB with a cache line size of 64B (8 words), if we want to perform the dot-product of two integer vectors of length 1024 each, what is the best memory layout to achieve the highest performance? (Note that each integer is 1 word = 4B.)

A good memory layout to use would be to interleave the two integer vectors in memory (i.e. for vectors a[1024] and b[1024], we can have a[0], b[0], a[1], b[1], etc... in memory), so that we can conveniently load 2 integers that are needed for a given operation immediately (8 word load). This way, we can conveniently load up the contiguous memory incrementally and store them contiguously in the 32KB cache. In addition, we can also deallocate the unneeded memory in the cache a lot more easily if it is also stored contiguously.

6.Given the processor in #5, if we want to multiply two integer matrices of 1024x1024 each, please compute the best tile size, and estimate the peak performance (in terms of GOP/sec) for the tiled matrix multiplication program discussed in Lecture 5.

tile size: $3b^2 < 32000 \rightarrow b \sim= 103$

peak performance: (# of operations) / (time) = (326 GOP/sec)

# of operations: f + m = 2.20E9

time: ((f * tf) + (m * tm)) = 6.74sec

$f = 2n^3 = 2(1024^3) = 2.15E9$

$m = (1024 / 103)(1024^2) + (1024 / 103)(1024^2) + 2(1024^2) + (2(1024 / 103) + 2) * (1024^2) = 4.59E7$

tf = 1ns

tm = 100ns