1)
RAW dependency for every addition to c[i][j]
(read then add, then write)

```
transformations:
	loop permutation:
		applicable for j and k to improve performance
		hurts performance if permute i (thus not applicable for i)
	loop distribution:
		not applicable
	loop fusion:
		not applicable
	loop shifting:
		not applicable
	loop unrolling:
		unrolling in k can help perfomance
			-fewer reads and writes of c[i][j]
		unrolling otherwise won't considerably
		help (same number of accesses to c[i][j])
	loop stip mining
		applicable. if done in i or k can improve
		locality
	loop unroll and jam
		applicable to k and j (hurts locality more in i)
	loop tiling
		applicable by the same reasoning as strip mining
		can do in both i and k to improve locality.
```

2)
```
void histogram(int *a, int *h){
	#pragma omp parallel for
	for(int i = 0; i < m; i++){
		h[i] = 0;
	}
	#pragma omp parallel for
	for(int i = 0; i < N; i++){
		h[a[i]]++;
	}
}
```

Yes, there is a race condition due to a[i] can equal a[j]
for i != j. Therefore, a RAW hazard exists. Can solve
this by locking individual entries in h.

```
void histogram(int *a, int *h){
	omp_lock_t histo_locks[m];
	#pragma omp parallel for
	for(int i = 0; i < m; i++){
		h[i] = 0;
		omp_init_lock(&histo_locks[i]);
	}
	#pragma omp parallel for
	for(int i = 0; i < N; i++){
		omp_set_lock(&histo_locks[a[i]]);
		h[a[i]]++;
		omp_unset_lock(&histo_locks[a[i]]);
	}
}
```

3)
Attempt 1:
```
	#include <math.h>
	#include <stdio.h>
```

```
double integral(int number_steps){
	const double increment = 1.0 / number_steps;
	double sum = 0;
	#pragma omp parallel for num_threads(16)
	for(int i = 1; i < number_steps; i++){
		double x = increment * i;
		double y = increment * (i-1);
		double midpoint = (x + y) / 2;
		sum += (sqrt(midpoint)/(1 + midpoint*midpoint*midpoint)) * (x - y);
	}
	return sum;
}

int main(){
	printf("%f\n", integral(1048576));
}
```

Adding to sum introduces a race condition (RAW hazard).
One solution is to use a critical section for adding to the sum:

Attempt 2:

```
#include <math.h>
#include <stdio.h>

double integral(int number_steps){
	const double increment = 1.0 / number_steps;
	double sum = 0;
	#pragma omp parallel for num_threads(16)
	for(int i = 1; i < number_steps; i++){
		double x = increment * i;
		double y = increment * (i-1);
		double midpoint = (x + y) / 2;
		#pragma omp critical
		sum += (sqrt(midpoint)/(1 + midpoint*midpoint*midpoint)) * (x - y);
	}
	return sum;
}

int main(){
	printf("%f\n", integral(1048576));
}
```

Another solution is to use indepedent accumulators then use a reduction
to sum them all together.

Attempt 3:

```
#include <math.h>
#include <stdio.h>

double integral(int number_steps){
	const double increment = 1.0 / number_steps;
	double sum = 0;
	#pragma omp parallel for reduction(+:sum) num_threads(16)
	for(int i = 1; i < number_steps; i++){
		double x = increment * i;
		double y = increment * (i-1);
		double midpoint = (x + y) / 2;
		sum += (sqrt(midpoint)/(1 + midpoint*midpoint*midpoint)) * (x - y);
	}
	return sum;
}
```

```c
        int main(){
                printf("%f\n", integral(1048576));
        }
```

Using a reduction is the more efficient of the two methods. This is due to critical blocking all threads attempting to access the sum (all but one of them) until it's available. Because accumulators in a reduction are private, reduction is much faster.

4) Expect Alice to finish first.
The cost of long tasks is absorbed by running shorter tasks concurrently with it. For instance, if one task takes a really long time and all the others a short amount of time, a single processor can run that very long task while the others finish all the shorter ones. For the same scenario in Bob's order, no processor could immediately grab that very long running process.
Bob's order delays the scheduling of long processes, reducing the available parallelism when those processes finally get scheduled.

5) a) 30 minutes
   b) 26 minutes
   c) 21 minutes