PDC Assignment 2
21K-3153

Q1

```
#define array_size 10
int main(){
    int array1[array_size]        //creating arrays
    int array2[array_size]
    int result[array_size]                    //*

    for(int i=0; i<array_size; i++){
        array[i] = i
        array2[i] = 2*i
    #pragma omp parallel for
        for (int i=0; i<array_size; i++)
            result[i] = array1[i] + array2[i];
        printf("Array1:")
    for //print array1
        printf("Array2")
        //print array2
      #print result array
        return 0
    }
```

after creating 3 arrays, initializing array1 and 2 with values we use "parallel for" to add the values into our 3rd array. "#pragma omp parallel" creates threads and "for" distributes workload in the for loop between threads

Since we have use parallel for, no need to declare a parallel block as it is implicitly understood that threads will execute for loop because of "parallel for" pragma.

day / date:

Static scheduling of reduction:

```
# define array_size 10
int main(){
    int array [array_size]
    int sum = 0

for(int i=0; i< array_size; i++)
    array [i] = i+1;

#pragma omp parallel for schedule (static) reduction(+: sum)
    for (int i=0; i< array_size; i++)
        sum += array [i];

int f(array)
    //print array
printf("\n %d ", sum);
    return 0
}
```

Here, we are initializing an array and summing it's elements in the sum variable.

"omp parallel" to creates threads. "for" to specify that they will run a for the "for loop"

schedule (static) ensures that iterations will be performed by threads cyclically. for eg thread 0 has iteration 1, thread 1 has iteration 2 and so on. Assume 4 threads, thread 0 will do iteration 4, thread 1 will do iteration 5 and so on.

The reduction clause makes the sum variable private to each threads, which prevents the variable being changed by something else and ensures consistency

day / date:

③ dynamic scheduling reduction

```
#define size 10
int main()
    int array1 [size]
    int array2 [size]
    int result [size]

for (i < size)
    array1[i] = i
    array2[i] = i*2

#pragma omp parallel for schedule(dynamic)
    for (int i=0; i<size; i++)
        result[i] = array1[i] + array2[i];

printf("Array1")
//print array1
printf("Array2")
// print array2
printf("Result")
// printf result

return 0
}
```

3 arrays created, two initialized. Each element of the 2 arrays is summed into 3rd array. pragma omp 'parallel' to create threads for to specify workload of for loop.

Dynamic scheduling makes it so that when one threads finishes it's task, it communicate with other threads to find an iteration which is not being worked on, and execute that iteration. However it is expensive.

day / date:

(4) int main()

```
int main(){
    int array[size]
    int sum=0

for (i<size)
    array[i]=i+1

#pragma omp parallel shared(sum)
{
    int local_sum=0
    #pragma omp for
      for (i<size)
        local_sum += array[i]

      #pragma omp critical
        {
          sum += local_sum
        } //end of critial
} //end of parallel

    printf("sum")
  return 0
}
```

initialized array and summing it. "omp parallel" to create threads. "shared (sum)" shares the global variable sum between threads. every thread sums their part of the array in local_sum variable. one thread at a time then accesses critical section and update the shared sum variable.

day / date:

⑤ Collapse clause

```
#pragma omp for collapse(2)private (i,k,j)
  for (k=kl, k<=ku, k+=ks)
      for (j=jl; j<=ju; j+=js)
          for (i=il; i<=iu; i+=is)
              bar (a,i,j,k)
```

J and k are part of the loop construct, so they are collapsed into 1 loop. i is not part of the loop construct thus is executed in every iteration of collapsed J and k

⑥ #pragma omp sections

```
#pragma omp section
    { task() }
#pragma omp section
    { taskB() }
```

this allows for non iterative parallel tasks.

⑦ Vector add and multiply using sections

```
int main()
  int vectorA[size], vector B[size], result add [size], result mul[size]
  for (i< size)
      vectorA[i] = i+1
      vector B[i] = i*2

#pragma omp parallel sections
  #pragma omp section
  {
      for a(i < size)
          result add[i] = vector A[i] + vector B[i]
  }
```

day / date:

```
#pragma omp section
{
    for (i<size)
        resultmul[i] = vectorA[i] * vectorB[i]
}

print resultadd array
print resultmul array
return 0
}
```

- this splits ~~parallel sections~~ ~~to~~ ~~do~~

this splits code into different sections to be parallelize by different threads

day / date:

(8) print message using sync clause

```
int main() {
    omp_set_num_threads (4);
#pragma omp parallel
    {
        printf("%d", omp_get_thread_num());
#pragma omp barrier

#pragma omp single
        {
            printf(" barrier complete");
        }

        printf("%d print second message", omp_get_thread_num());
    }
```

omp parallel to create threads.
omp barrier to wait for each thread to print 1st message
omp single for only 1 thread to print message
then all threads print second message

day / date:

(7) ~~synchronizing~~ critical (2) generating prime with critical

```
{ int is_prime (int num)
{
    if (num <= 1) return 0
    if (num == 2) return 1

    if (num % 2 == 0) return 0
    for (i = 3 , i*i <= num, i += 2)
        if ( num % i == 0)
            return 0

    return 1 }
int main ()
    int start=2
    int end=100
  #pragma omp parallel for
    for ( i = start , i <= end ; i++)
        if (is_prime (i))
            #pragma omp critical
                printf ("%d is prime", i)
        }
    }
    return 0;
}
```

omp parallel for creates threads that run is_prime check. If is_prime true, one thread at a time can access critical section and print the prime number.