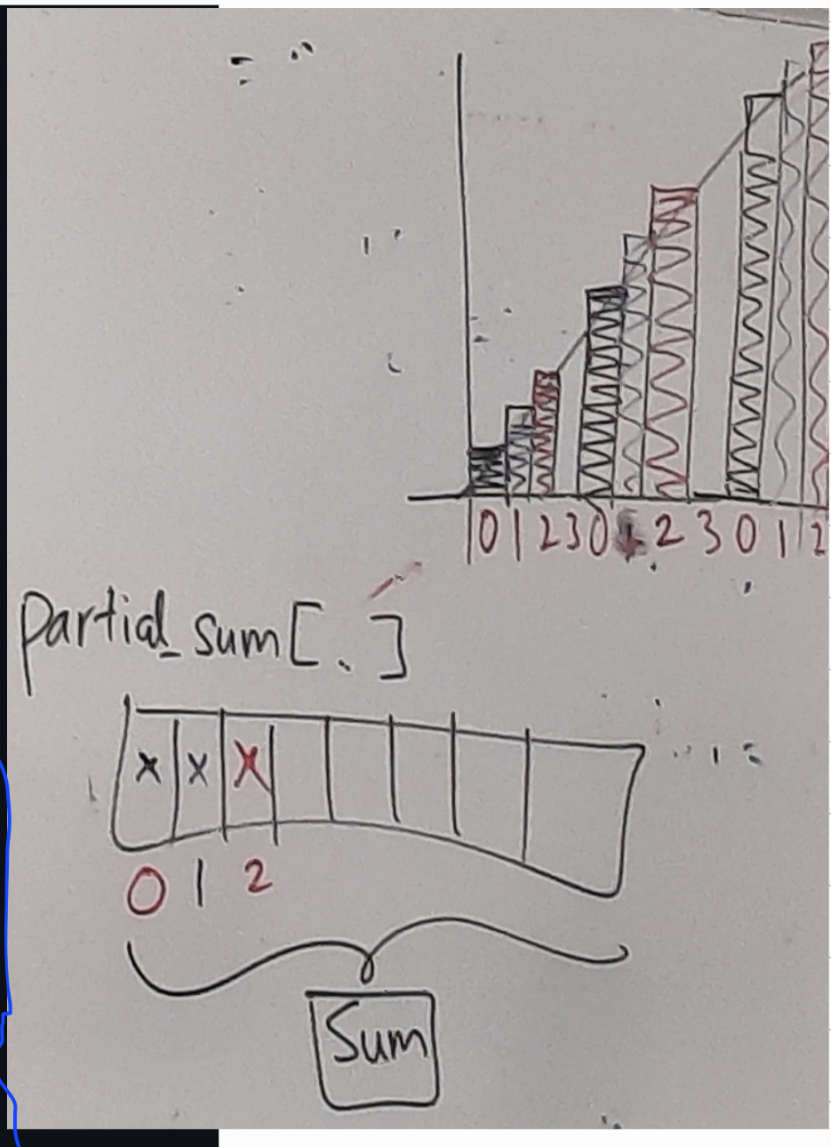
critical region (01_Compute_PI_Critical_Region)

```
int compute_pi_using_partial_sum()
{
    int i;
    double step, pi = 0;
    double partial_sums[NUM_THREADS], sum = 0.0;
    step = 1.0 / (double)num_steps;

    omp_set_num_threads(NUM_THREADS);
    printf("Total OMP threads: %d\n", NUM_THREADS);

    double start_time = omp_get_wtime();
#pragma omp parallel
    {
        int i, id;
        double x;
        id = omp_get_thread_num();

        partial_sums[id] = 0.0;
        for (i = id; i < num_steps; i += NUM_THREADS)
        {
            x = (i + 0.5) * step;
            partial_sums[id] += 4.0 / (1.0 + x * x);
        }
    }
    for (i = 0, pi = 0.0; i < NUM_THREADS; i++)
    {
        sum += partial_sums[i];
    }
    pi = sum * step;
    double end_time = omp_get_wtime();

    cout << setprecision(16) << pi << endl;
    cout << "Work took " << (end_time - start_time) << " seconds\n";

    return 0;
}
```



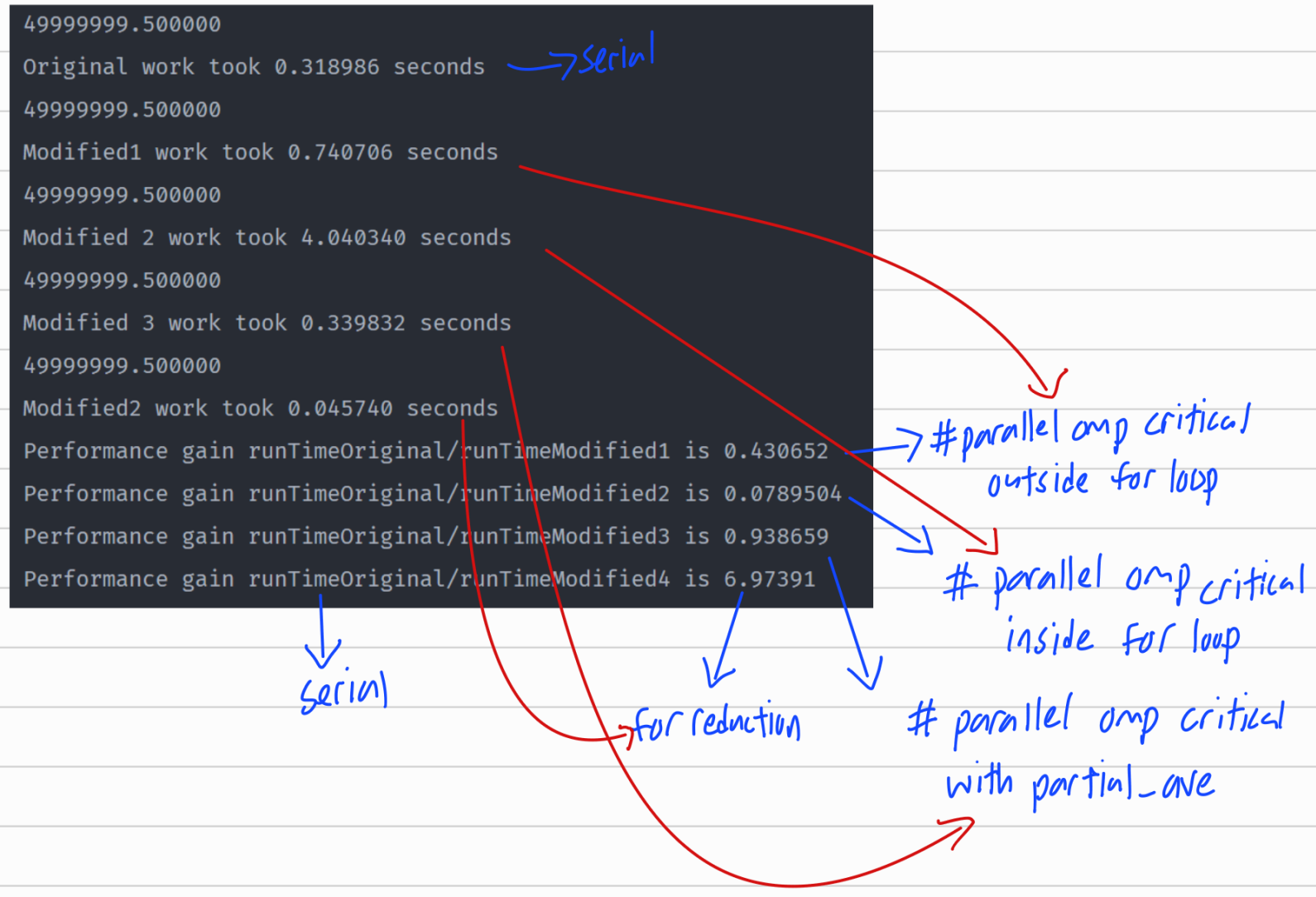
```c
int compute_pi_using_critical_region()
{
    int i;
    double step, pi = 0;
    double sum = 0.0;
    step = 1.0 / (double)num_steps;

    omp_set_num_threads(NUM_THREADS);
    printf("Total OMP threads: %d\n", NUM_THREADS);

    double start_time = omp_get_wtime();
#pragma omp parallel
    {
        int i, id;
        double x;
        id = omp_get_thread_num();

        // partial_sums[id] = 0.0;
        double partial_sum = 0.0;

        for (i = id; i < num_steps; i += NUM_THREADS)
        {
            x = (i + 0.5) * step;
            partial_sum += 4.0 / (1.0 + x * x);
        }

#pragma omp critical(partial_sum)
        {
            sum += partial_sum;
        }
    }
```

critical region that each time only 1 thread is allow to enter

02_Calc_Avg_Serial_VS_OMPParallel_CriticalR_ForReduction

Result

```
49999999.500000
Original work took 0.318986 seconds
49999999.500000
Modified1 work took 0.740706 seconds
49999999.500000
Modified 2 work took 4.040340 seconds
49999999.500000
Modified 3 work took 0.339832 seconds
49999999.500000
Modified2 work took 0.045740 seconds
Performance gain runTimeOriginal/runTimeModified1 is 0.430652
Performance gain runTimeOriginal/runTimeModified2 is 0.0789504
Performance gain runTimeOriginal/runTimeModified3 is 0.938659
Performance gain runTimeOriginal/runTimeModified4 is 6.97391
```



① Serial

```c
double calculate_average_serial()
{
    double ave = 0.0, *A;
    int i;

    A = (double *)malloc(MAX * sizeof(double));
    if (A == NULL)
    {
        printf("Insufficient memory! Can't continue. Terminating the program abruptly.\n");
        return -1;
    }

    for (i = 0; i < MAX; i++)
    {
        A[i] = (double)i;
    }
    double start_time = omp_get_wtime();

    for (i = 0; i < MAX; i++)
    {
        ave += A[i];
    }

    ave = ave / MAX;

    double end_time = omp_get_wtime();

    printf("%f\n", ave);
    printf("Original work took %f seconds\n", end_time - start_time);
    free(A);
    return end_time - start_time;
}
```

MAX = 100,000,000

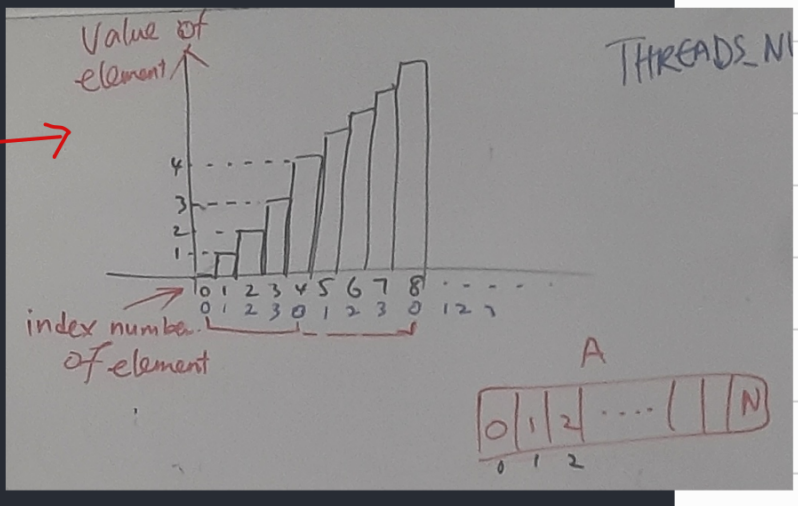
allocate ↓ size

if not enough memory malloc return null

initialize value →

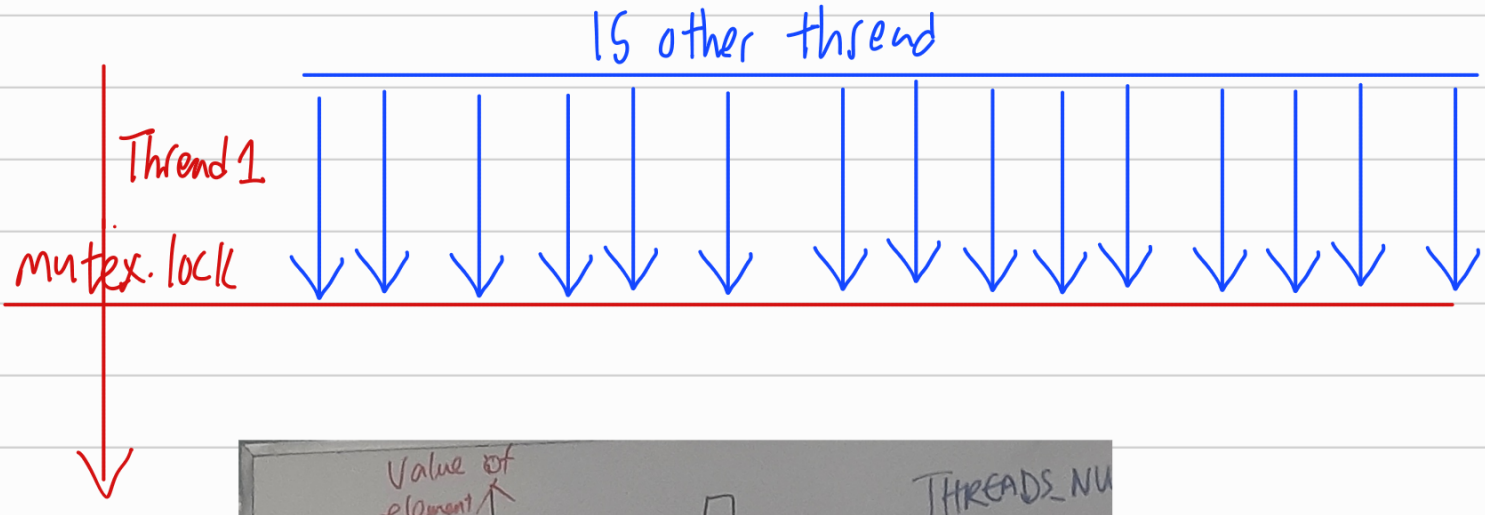
add up 1 - 100,000,000

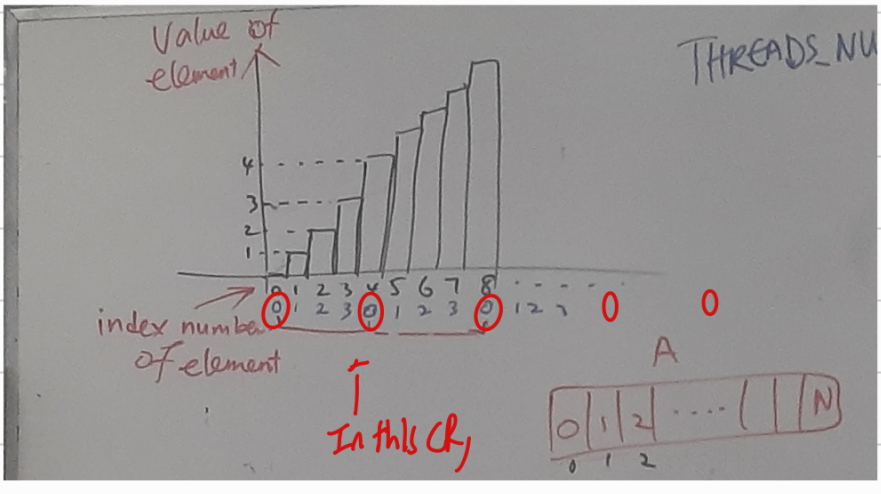
calculate average



② #parallel omp critical outside for loop

```
#pragma omp parallel num_threads(MAX_THREAD)
    {
        int i;
        int id = omp_get_thread_num();

#pragma omp critical // other 15 thread have to wait outside
        for (i = id; i < MAX; i += MAX_THREAD)
        {
            ave += A[i];
        }
    }
```





thread 0, will add up all the value

mutex. unlock

after unlock threas 1 continue add their own value

Thread 0 → add up 0 value → Thread 1 → add up 1 value
... → Thread 15 → add up 15 value.

(3) # parallel omp critical inside for loop

```
#pragma omp parallel num_threads(MAX_THREAD)
    {
        int i;
        int id = omp_get_thread_num();

        for (i = id; i < MAX; i += MAX_THREAD)
        {
#pragma omp atomic // or critical // ave is shared so need to protected by critical region (protect by MAX times)
            ave += A[i];
        }
    }
```

ave += (only 1 value)

mutex.unlock()

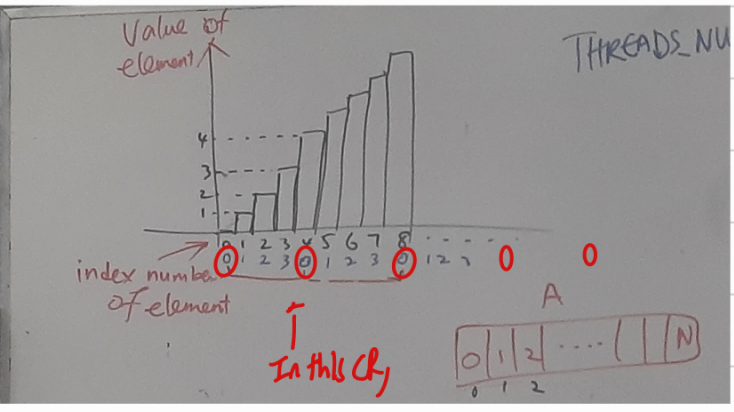
Thread 0 → add 1 value → Thread 1 → add 1 value
... → Thread 100,000,000 → add 1 value

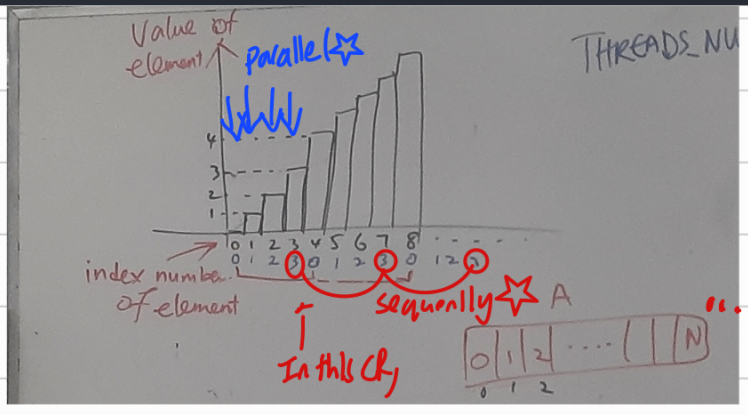
↓

really really slow

4 # parallel omp critical with partial_ave

```
#pragma omp parallel num_threads(MAX_THREAD)
    {
        int i;
        int id = omp_get_thread_num();
        double partial_ave = 0.0;

        for (i = id; i < MAX; i += MAX_THREAD)
        {
            partial_ave += A[i];
        }

#pragma omp atomic // only 16 threads will go into this critical region (Number of threads access to critical region is reduced)
        ave += partial_ave;

        // #pragma omp critical(sum) // same as atomic but run faster
        //         ave += partial_ave;
    }
```

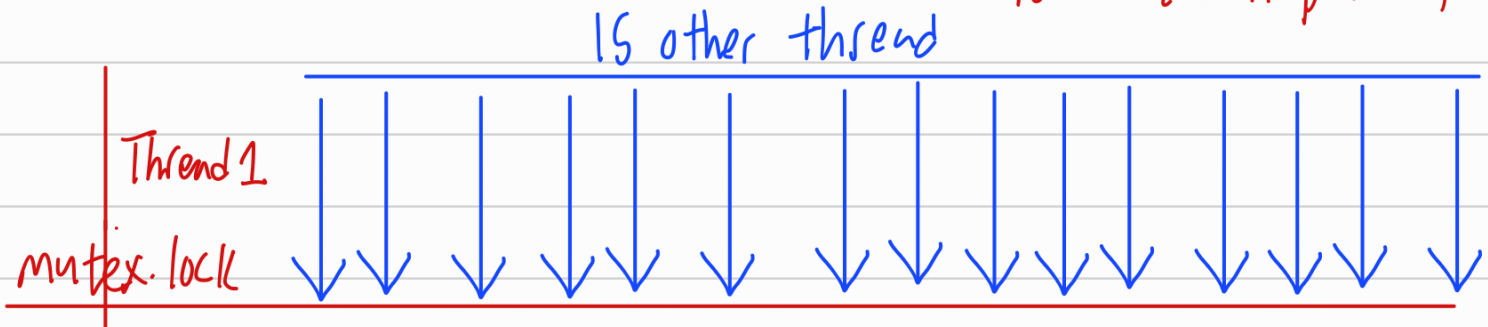




thread 0, will add up all the value

thread 3, will add up all the value

16 thread run parallel



ave += partial_ave

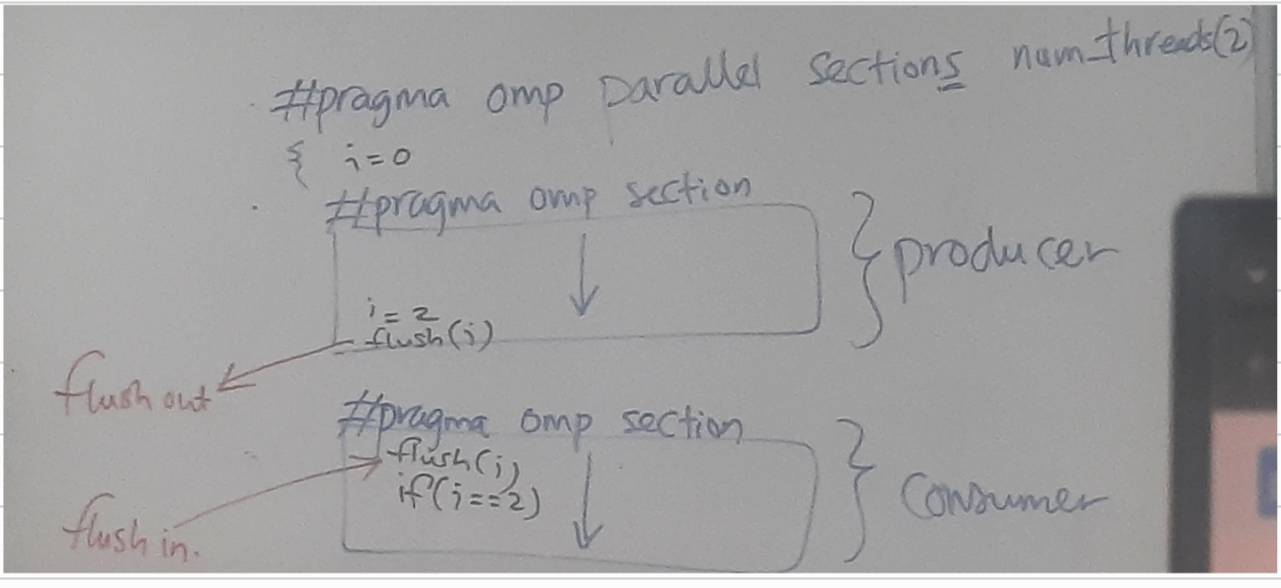
mutex. unlock

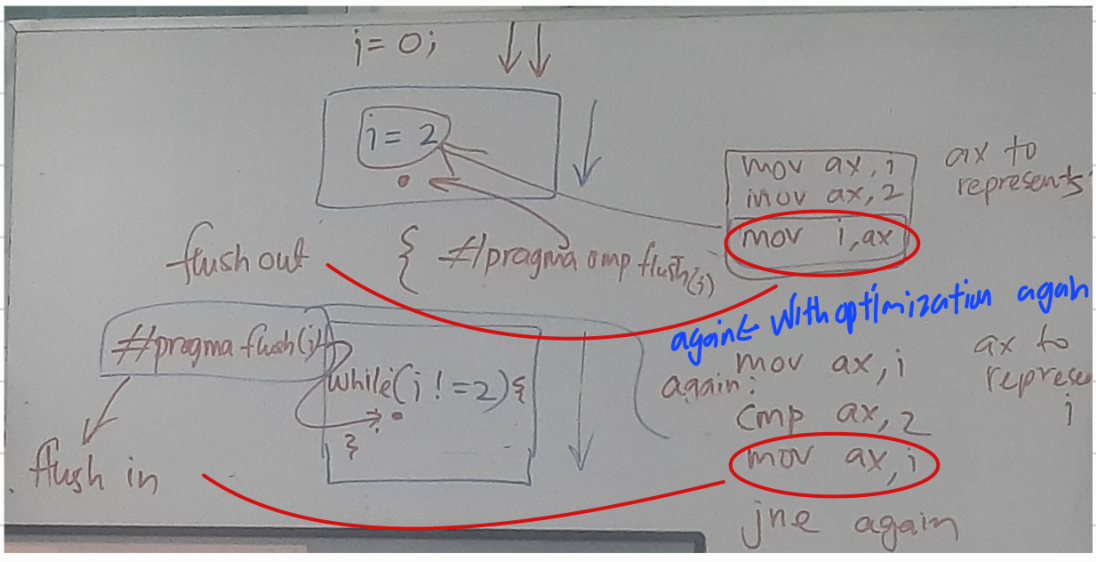
after unlock thread 1 continue add their own value

Thread 0 → add up partial_ave → Thread 1 → add up partial_ave
... → Thread 15 → add up partial_ave

03_Producer_Consumer_FlushInOut.cpp

```
#pragma omp parallel sections num_threads(2)
    {
#pragma omp section
        {
            printf("Thread %d: ", omp_get_thread_num());
            read(&data);
#pragma omp flush(data)
            flag = 1;
#pragma omp flush(flag)
            //  Do more work.
        }

#pragma omp section
        {
            while (!flag)
            {
#pragma omp flush(flag)
            }
#pragma omp flush(data)

            printf("Thread %d: ", omp_get_thread_num());
            process(&data);
            printf("data = %d\n", data);
        }
    }
    return 0;
}
```





withoptimization and no flush in flushout

i=0

i=2

section 1

mov ax,i
mov ax,2
mov i,ax

#pragma omp flush(i) flushout

while(i!=2){
}

with optimization

again:
mov ax,i
cmp ax,2
mov ax,i
jne again

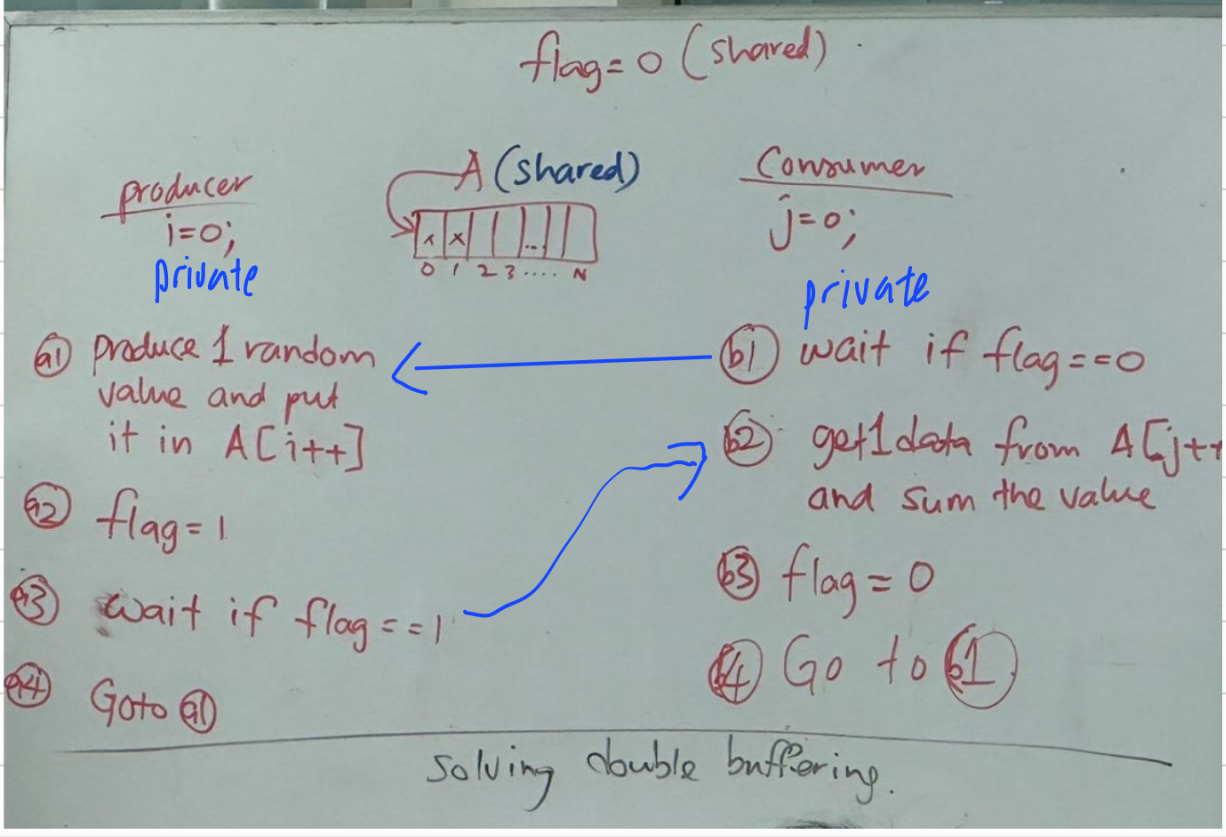
compare

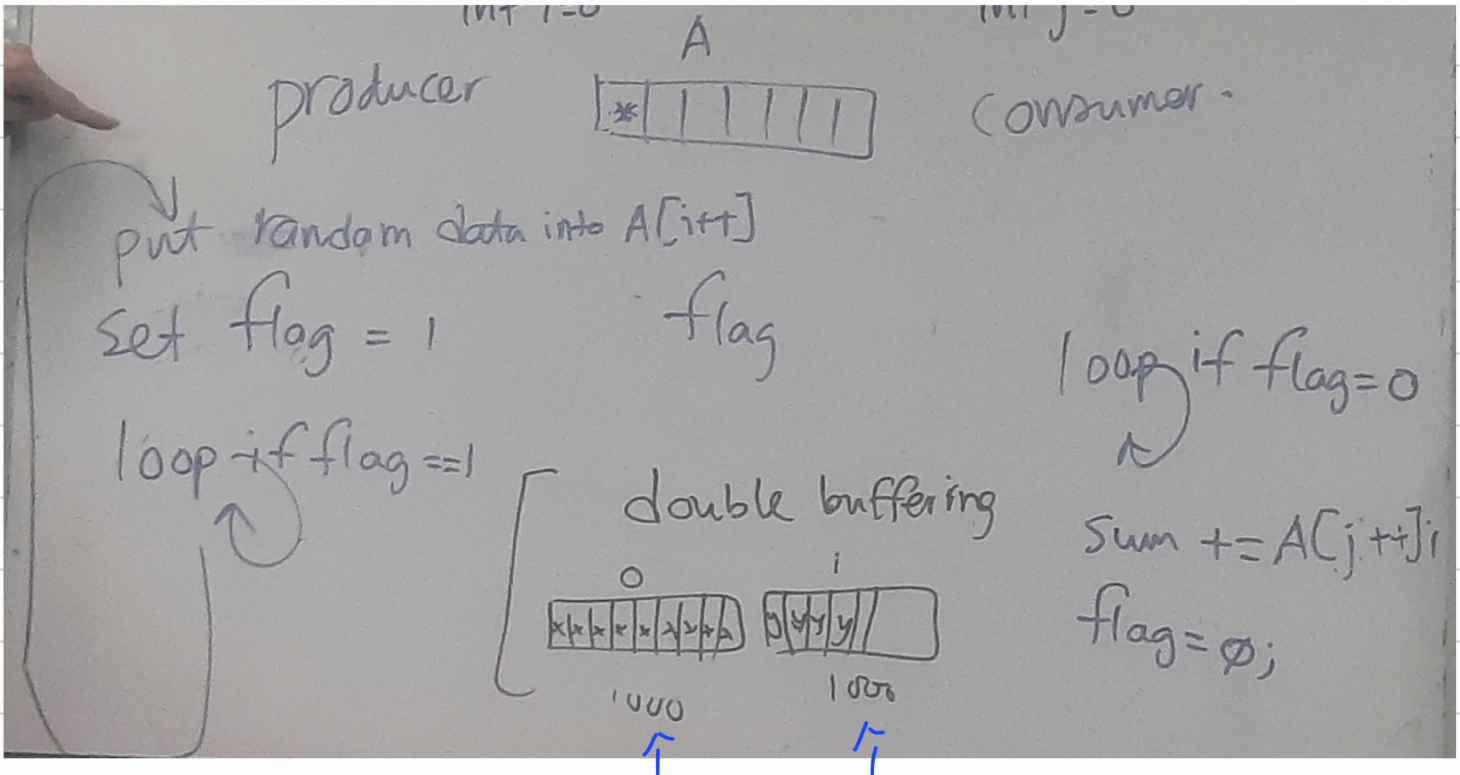
#pragma omp flush(i) -flush in

if do not have flush in, it always had the stale/outdated i

04_Producer_Consumer.cpp

```
#pragma omp parallel sections num_threads(2)
    {
#pragma omp section
        {
            int i = 0;
            // Producer: produce 1 random value and put it in A[i++]
            // until all values are produced
            while (i < N)
            {
                if (flag == 0)
                {
                    A[i] = (double)rand(); // produce a random value
                    i++;                   // move to the next index
                    flag = 1;              // signal the consumer
#pragma omp flush(flag)                    // Ensure visibility of flag
                }
            }
        }

#pragma omp section
        {
            int j = 0;
            // Consumer: sum the array when producer has placed data
            while (j < N)
            {
                if (flag == 1)
                {
                    sum += A[j]; // consume the value
                    j++;         // move to the next value
                    flag = 0;    // signal the producer
#pragma omp flush(flag)          // Ensure visibility of flag      You,
```


flag=0 (shared)
producer
i=0;
private
A (shared)
0 1 2 3 .... N
Consumer
j=0;
private
a1 produce 1 random value and put it in A[i++]
a2 flag=1
a3 wait if flag==1
a4 Goto a1
b1 wait if flag==0
b2 get 1 data from A[j++] and sum the value
b3 flag=0
b4 Go to b1
Solving double buffering.


A
producer
consumer
put random data into A[i++]
set flag = 1
flag
loop if flag==1
loop if flag=0
double buffering
0
1
Sum += A[j++];
flag=0;
producer produce
consumer consume
once done, switch (front/back buffer)
consumer consume what producer just produce
producer produce empty slot

Atomic vs Critical

Slower, flexible

```
#pragma omp critical
{
    ____
    ____
    ____
    ____
}
```

allow multiple operation

count arithmetic operation

Faster, restricted

```
#pragma omp atomic
b t=3
```

+ - * / %

only single arithmetic operation

```
int atomic_exercise_no_atomic()
{
    float x[1000];
    float y[10000];
    int index[10000];
    float sum1, sum2;
    int i;

    for (i = 0; i < 10000; i++)
    {
        index[i] = i % 8;
        y[i] = 0.0;
    }
    for (i = 0; i < 1000; i++)
        x[i] = 0.0;
    atomic_example_no_atomic(x, y, index, 10000);

    sum1 = Sum_array(1000, x);
    sum2 = Sum_array(10000, y);

    printf(" The sum is %f and %f\n", sum1, sum2);
    return 0;
}
```

0 1 2 3 4 5 6 7 0 1 2 3 ....

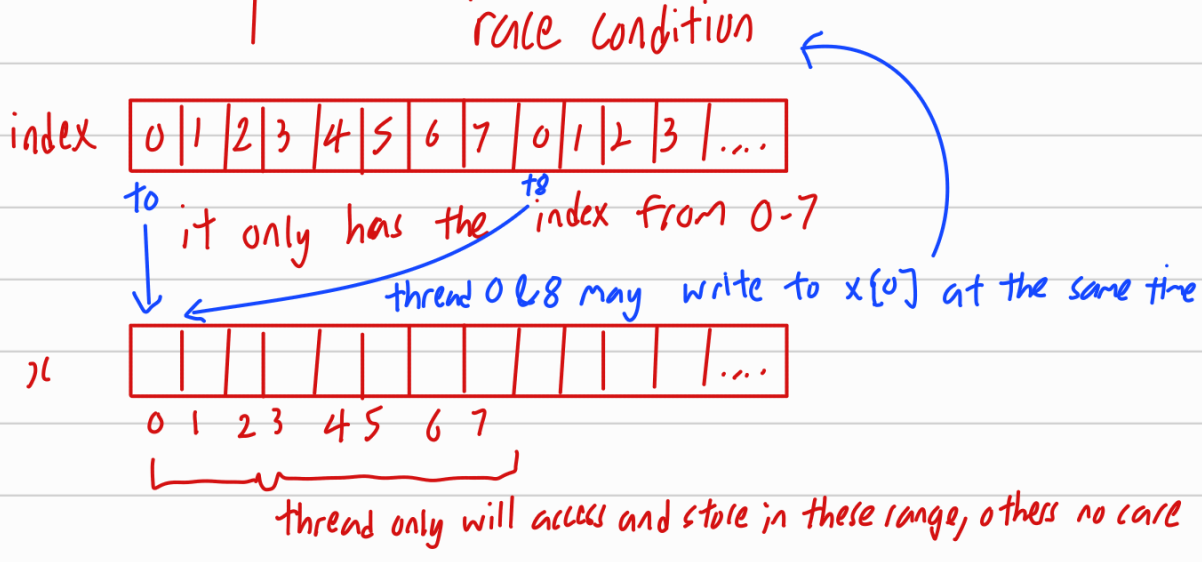
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ....

for 10000

for 1000

```c
void atomic_example_no_atomic(float *x, float *y, int *index, int n)
{
    int i;

#pragma omp parallel for shared(x, y, index, n) num_threads(16)
    for (i = 0; i < n; i++)
    {
        x[index[i]] += work1(i);
        y[i] += work2(i);
    }
}
```

```c
float work1(int i)
{
    return 1.0 * i;
}

float work2(int i)
{
    return 2.0 * i;
}
```




```c
void atomic_example_with_atomic(float *x, float *y, int *index, int n)
{
    int i;

#pragma omp parallel for shared(x, y, index, n) num_threads(16)
    for (i = 0; i < n; i++)
    {
#pragma omp atomic
        x[index[i]] += work1(i);
        y[i] += work2(i);
    }
}
```

→ only allow one thread access x[0]

↳ EG thread 0 → write first then thread 8 write