Date _____ Day _____

"CHAPTER #05"

- Error Checking:

→ if compiler doesn't support the openmp then we should 1st check.

```
#ifdef _OPENMP
#   include <omp.h>
# endif
```

```
int main() {
#ifdef _OPENMP
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads
#else
    int my_rank = 0;
    int thread_count = 1;
# endif
```

3

Date _____ Day _____

Reduction :

```
global_result = 0;  // shared variable
#pragma omp parallel num_threads(thread_cnt)
{
    double my_result = 0.0;  // private variable
    my_result += Local_trapezoid(double a, double b, int n);

#pragma omp critical  → multiple threads ek sath Result ko race condition.
    global_result += my_result;
}
```

→ Now using reduction

```
global_result = 0;
#pragma omp parallel num_threads(thread_cnt) \
    reduction (+: global_result)
global_result += Local_trapezoid (——);
```

→ Use of subtraction operator (Problematic!)

```
result = 0;
for (i = 1; i <= 4; i++)
    result -= i;
```

↳ result stores -10. →

Date _____ Day _____

But if this loop was performed by two threads then iterations would have been divided.

$T_1 \rightarrow$ result = -3 (-1-2)

$T_2 \rightarrow$ result = -7 (-3-4)

And the subtracting the results of these two would given 4 (-3 - (-7) = 4) which is not corsect it should have been -10 (-3 + (-7) = -10).

→ Use of float or double as reduction variable is also problematic.
As in floats (a+b)+c may not be exactly equal to a+(b+c).

▷ Parallel for :

→ OpenMP can only ||ized the for loops whose no. of iterations are known.

```
for ( ; ; )          or   for(i=0; i< n; i++){
{ ——                        if(——)
}                             break;
                          }
```

There cannot be ||ized.

→ While & Do-while can also be not ||ized.

Date _____ Day _____

Data Dependencies:

```
fib[0] = fib[1] = 1;
#pragma omp parallel for num_threads(thread_ct)
    for (i=2; i<n; i++)
        fib[i] = fib[i-1] + fib[i-2];
```

→ Here the result of fib[2] is dependent on fib[1] & fib[0] therefore there is dependency among loops.

→ OpenMP doesn't check for dependencies among iteration in a loop

→ A data dependent loop gives unpredictable results when parallized by OpenMP.

→ These type of data dependency also called.
loop-carried dependence.
↳ result of one iteration depends on result of another iteration.

```
for (i=0; i<n; i++) {
    x[i] = a + i*h;
    y[i] = exp(x[i]);
}
```

↗ yaha par general dependence hai. loop carried nhi. loop carried mein previous iteration par dependence hoti hai.
like x[i] = x[i-1] + i*h;

Day

Date

▷ Estimating π :

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
double factor = 1
double sum = 0

for (k = 0; k < n; k++) {
    sum += factor / (2*k+1);
    factor = -factor;
}
pi_approx = 4.0 * sum;
```

→ This factor value changed in one iteration will be used by next iteration.

So loop carried dependency

& this should be a private variable or else if it is shared the it would cause race condition.

▷ Corrected version

```
double sum = 0.0;
# pragma omp parallel for reduction(+:sum) \
  private(factor)
  for (k = 0; k < n; k++) {
    if (k % 2 == 0)
      factor = 1.0;
    else
      factor = -1;
    sum += factor / (2*k+1);
  }
pie_approx = 4.0 * sum;
```

→ isse ye previous iterations par dependent nhi hoga like 'k' ki current value par dependent rhega.

Page No.

Date _____ Day _____

Bubble Sort :- Loop Carried Dependences

```
for (listLength=n; listLength>=2; listLength--) {
    for (i=0; i<listLength-1; i++) {
        if (a[i] > a[i+1]) {
            swap(a[i], a[i+1]);
        }
    }
}
```

Outer Loop Dependence :

① Each pass of outer loop depends on completion of previous passes.

Example:

If list starts with [3, 4, 2, 1], after pass-1 list becomes [3, 1, 2, 4]. The 2nd pass should oper. on [3, 1, 2, 4].

Now if 1st & 2nd pass exec simultaneously, the 2nd pass could operate on [3, 4, 2, 1].

Inner Loop Dependence :

① Each inner loop iteration compares & possibly swaps elements, impacting next iteration.

Date

Example:
In a list [3,1,2], starting with i=0, we compare a[0] > a[1] & swaps(a[0], a[1] i.e 3 & 1. [1, 3, 2]. The next comparasion when i=1 a[1] > a[2] & swaps(3 & 2). [1, 2, 3].
Now if i=0 & i=1 iteration runs ||ly then i=1 could incorrectly compare 1 & 2 before i=0 completes it's swap.

⇒ Odd-Even Sort Parallel Program:

```
# pragma omp parallel num_threads (thread_ctr) \
  default (none) shared (a,n) private (i, phase)

for (phase = 0; phase < n; phase++) {
    if (phase % 2 == 0) {
    # pragma omp for
      for (i = 1; i < n; i += 2) {
        if (a[i-1] > a[i])
            swap (a[i-1] > a[i]);
      }
    }
    else
    # pragma omp for
      for (i = 1; i < n-1; i += 2) {
        if (a[i] > a[i+1])
            swap(a[i], a[i+1]);
      }
}
```

// in even phase compare the odd subscripts with left elements

// in odd phase compare odd subs to right element

Date _____ Day _____

Shedule Clause :

Shedule (type, chunkSize)

⇒ Types Of Shedule Clause :

① Static Shedule Type :

The system assigns chunks of chunk_size iterations to each thread in a round-robin fashion.

If we have 12 iteration then each thread will get

T_0 : 0, 3 6 9
T_1 : 1 4 7 10
T_2 : 2 5 8 11

If shedule (static, 2)

T_0 : 0, 1, 6, 7
T_1 : 2, 3, 8, 9
T_2 : 4, 5, 10, 11

If chunk_size is omitted, then it is approximately equal total_iterations / thread_count.

② Dynamic And Guided Type :

In a dynamic sheduling the iterations are broken up into chunks of chunk_size. When one thread finishes a chunk, it request for another.

Day

Date

Benefits:

↳ Dynamic scheduling is effective for loops with variable execution times for each iteration.

↳ Faster threads don't need to wait for slower threads, as they can pick up new chunks when available.

In Guided scheduling in OpenMP is a type of dynamic scheduling where the size of the chunks decreases as more iterations are assigned to the threads.

As chunks are completed, the size of subsequent chunks decs, based on

$$\text{chunk\_size} = \frac{\text{remaining iterations}}{\text{no. of threads}}$$

Benefits:

↳ Larger chunks at begining help handle initial heavy workloads

↳ Useful when workload / iteration reduces gradually throught the loop.

Shedule(guide, 2) → tells OpenMP to start with larger chunks, progressively dec. the chunk size until reaches 2.

Date 1 Day

(3) RUNTIME TYPE :

When we write Shedule (runtime), the system uses the enviornment variable OMP_SCHEDULE to determine at runtime how to shedule loops.
We can set the value of env variable by running following

$ export OMP_SCHEDULE = "static, 1"

Now when we run our program the system takes will take the value from OMP_SCHEDULE and our clause will be modified to

Shedule (static, 1)

~~4) PRODUCER AND CONSUMER~~

5) CACHE, CACHE COHERENCE AND FALSE SHARING :

→ The design of cache takes into consideration the principles of temporal & spatial locality.
"If a processor access a main memory location 'x' at a time 't', then it is likely that at times close to 't', it will access main memory locations close to 'x'."

→ Therefore instead of transferring only 'x' into cache a block of memory is transferred called cache line or cache block.

Day

Thread Safe :

- A function is not thread safe it it give different result when executed // y.

→
```
#pragma omp critical
y = f(x);

double f(double x) {
    #pragma omp critical
    z = g(x)
}
```

This will create deadlock

If thread 1 started exe the 1st critical section then no thread can enter in any critical section so, we it calls the function f(x) & start to execute it has a nested critical section & it cannot enter into it.
It will wait infinitely.

→ We can differentiate the CS by giving names to them

```
#pragma omp critical (one)
y = f(x)

double f(double x) {
    #pragma omp critical (two)
    z = g(x);
```

Page No.

Date _____ Day _____

"PRODUCE CONSUMER"

▷ Message Passing :

→ Threads communicate each other via messages
→ Threads can access same memory space, which allows them to share info.
→ Each thread has it's private queue
→ If one thread want to send msg to another it places msg in other's queue (destination queue).
→ Recieving thread checks its queue.
→ If there is a msg, the thread dequeues it & process it.

```
for (send_msg=0; send_msg < max; send_msg++){
        Send_msg();          // Each thread sents a max
                             amount of msg to other
    Tryrecieve_msg();        thread & checks if it
                             recieves any msg
}
while(!Done()){              // if all msgs are send but
    Try_recieve();           still recieving the msg
}                            from other threads.
```

Day

Date

▷ Sending Message :

→ If two threads tries to simultaneously enque a msg in a destination thread's queue race condition may occur.

```
Send_msg() {
    mesg = random();
    dest = random % thread_count;
    #pragma omp critical
        Enqueue(queue, dest, my_rank, mesg);
```

▷ Recieving Message :

→ Only the thread to which the queue belongs can dequeu the msg.

→ We uses enqueue variable & dequeue variable to keep the track of no. of mesgs enqued & dequed.

→ This is done so that we can get queue_size.

queue_size = enqueue - dequeue

→ No conflicts occur when we simultaneously enque or deque msg.

Date _____ Day _____

▷ Termination Detection :

- If ~~others thread exe~~

```
Done (){
1       queue_size = enqueue - dequeue
2       if (size == 0 ) return TRUE
3       else            return FALSE.
}
```

If a thread 'U' executes line 1 and at same time thread 'V' sends a msg then thread U cant recieve it as it will return ~~false else~~ TRUE as queue_size == 0.

To avoid this we do

```
if (size == 0 && done_sending == thread_count)
        return TRUE;
else
        return FALSE;
```

▷ Startup:

→ Master thread get command line arguments & allocate array of msg queue to each thread.

→ If a thread recieves a msg before completing it's memory allocation for queue it would crash the program.

→ We should create an explicit barrier so that no thread could send msgs before complete allocation of memory.

Page No. ☐

Date _____ Day _____

- Atomic directive can only protect critical section that consist of a single C assignment statement

```
x++
--x
x+=(y++);   → this remains unprotected.
```

- Locks:

```
omp_lock_t lock;
omp_init_lock(&lock);
omp_set_lock(&lock);
omp_unset_lock(&lock);
omp_destroy_lock(&lock);
```

Date

Day

" OpenMp Parallel Programming "

- Parallel Region :

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

void main() {

  omp_set_num_threads(4); // this will create 4 threads. If we donot write this default=2.
  int id;

  #pragma omp parallel   // jitne andar scope code k baar chalega // k.
  {
    id = omp_get_thread_num();
    printf("Thread %d : Hello World \n", id);
  }

  return 0;

}
```

parallel

```
gcc -fopenmp hellocoolol.c
./a.out
```

Date _____ Day _____

- Parallel For Loop:

```
#pragma omp parallel for
for(_____){
```

⇒ This loop will execute in ||.

```
}
```

or

```
#pragma omp parallel
{
    #pragma omp for
    for(_____){
    }
}
```

```
#pragma omp parallel
for(_____){
{
}
```

→ Is tarha krenge to loop ki iteration d
nhi hongi like has thread mein saar
iterations chalengi.

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

void main(){
    int i, sum=0;
```

Date _____ Day _____

```
int thread_sum[4];
omp_set_num_threads(4);

#pragma omp parallel
{
    int ID = omp_get_thread_num();
    thread_sum[ID] = 0;

#pragma omp for        // ye loop ki iteration khud ni 25,25 mein divide ho jayegi.
    for(int i=1; i<=100; i++)
        thread_sum[ID] += i;

}
for(i=0; i<4; i++)
    sum += thread_sum[i];

}
```

⇒ Shared & Private Variables.

All variables declared outside // region are shared among all.

Variables inside // region are private to each thread. Each thread has a local copy for it.

Iteration variable are by default private.

Shared: sum, thread_sum[], private: i, ID

value of private variable is unspecified at beginning of
|| block & also after completion of || block

Date _____ Day _____

Explicit rule:

```
#pragma omp parallel for shared(n, a)
for(int i=0; i<n; i++)
    int b = a+i;
```

#

```
int a,b,c,n
#pragma omp parallel for default(shared) private(a,b
for(_____){
    || a & b are private
    || c and n are shared
```

```
int n = 10;
std::vector<int>vector(n);
int a = 10;
#pragma omp parallel for default(none) shared(n, vector,a
for(_____)
{       }
```

yeh likhne ki vajase hamein shared waale explicitly batane pareng.

Critical Section:

```
#include <omp.h>
int cont = 0;
void myFunc(){
    #pragma omp critical
```

Day

Barriers :

It is the point in the execution of a program where threads wait for each other.
No thread is allowed to continue until all threads reach barrier.
Used to achieve sync.
As soon as one thread reaches barrier then all thread in the team must reach the barrier

```
#pragma omp barrier
```

OpenMp function

① void omp_set_num_threads(4)
int omp_get_num_threads() → ret no. of threads in current team
int omp_get_max_threads() → Ret max no. of threads that could be used to form a new team using a ll construct without a num_threads clause.
int omp_get_thread_num() → Ret ID of current thread
int omp_get_num_procs() → ret no. of processors available to the program

Day

Date

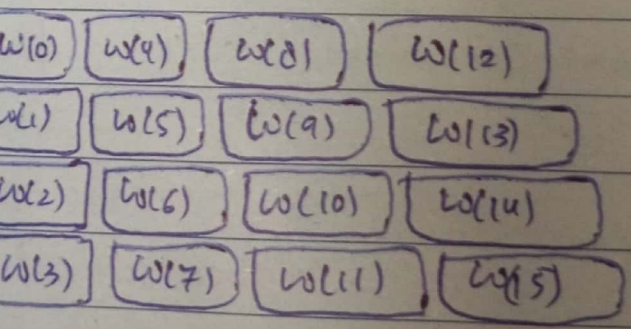
int omp_in_parallel() → Ret true if the call to routine is enclosed by an active // region

Parallel For Loops : Shedulling.

→ If working of each thread increase after each iteration then the last thread will have lot of work to do.

→ Therefore we need to distribute / load balance the work.

```
#pragma omp parallel for shedule(static, 1)
for(int i=0; i<16; i++)
    w(i);
}
```

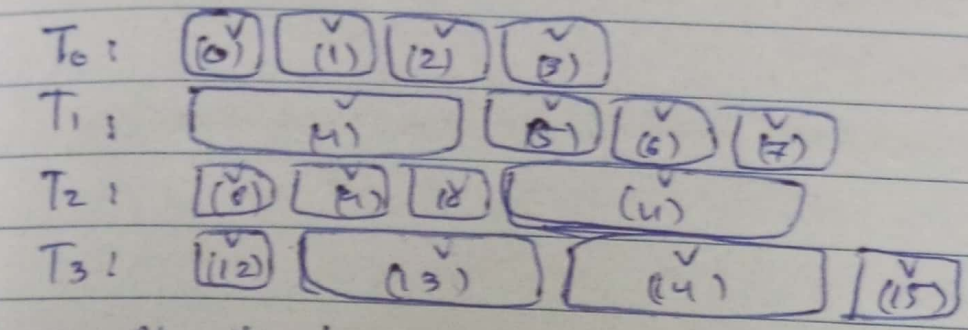


"1" indicates that we assign 1 iteration to each thread before switching to next thread.

Date _____ Day _____

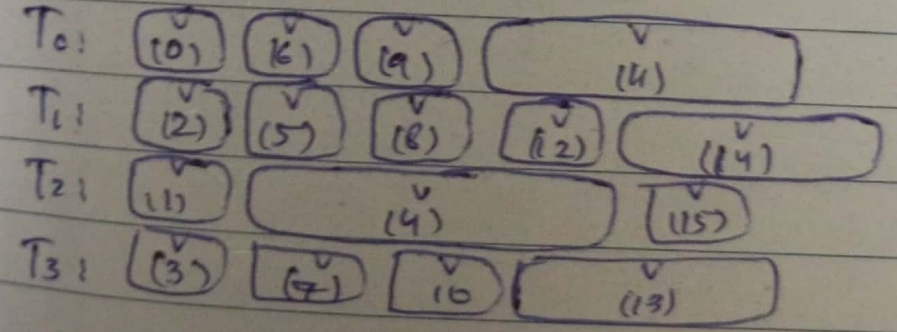
- Dynamic Shedulling:

→ Kabhi Kabar kuch threads ~~bohh~~ boht jaldi execution khatam kodeti hai or kuch ~~bohh~~ boht der se finish hoti hai.



▲ Her iteration k alag size hai.

```
#pragma omp parallel for shedule (dynamic, 1)
for(int i=0; i<16; i++)
    v(i)
```



Dynamic Shedulling assigns iterations of loop to threads at runtime, distributing a specified chunk of iterations to each thread. After a thread finishes its chunk, it gets a new one allowing for better load balancing, especially when iterations have varying execution time.

Page No.

NOTE

Day

Date

⊙ Reduction:

↳ Reduction is used to combine variables from diff threads into a single result after ll execu.

↳ Specifying operator, defines how individual thread results will be combined (eg. + for sum, * for product).

↳ This ensures that each thread work on a private copy of the variable, avoiding direct access to shared mem during computation.

```
int sum=0;
#pragma omp parallel for reduction(+:sum)
for(int i=1; i<=10; i++)
    sum+=i;

printf("Sum : %d ", sum);
return 0;
```

Peeche example mein jo humne thread_sum[4] ka array banaya phir har thread ke liye use use kra or last mein combine kraa ye saara kaam reduction kr deta hai.

Date _____ Day _____

Sections:

↳ Section is used to divide a program into independent code blocks, each of which can be executed by different threads concurrently.

↳ It allows // execution of non-loop tasks.

```
#pragma omp parallel sections num_threads(3)
{
  #pragma omp section
  {
    printf("Task by thread: %d", omp_get_thread_num());
  }

  #pragma omp section
  {
    printf("Hello world");
  }

  #pragma omp section
  {
    printf("My World");
  }
}
```