# COL380

## Introduction to
## Parallel & Distributed Programming

# Parallel Construct

```
#pragma omp parallel \
    if(boolean) \
    private(var1, var2, var3) \
    firstprivate(var1, var2, var3) \
    default(private | shared | none) \
    shared(var1, var2) \
    copyin(var1, var2) \
    reduction(operator:list) \
    num_threads(n)
  {
  }
```

- Implicit barrier at the end, Implicit flush
- Cannot branch in or out
- No side effect from clause: must not depend on any ordering of the evaluations
- Upto one if and *num_threads* clauses
- num_threads must be a +ve integer

# Variable Scope

```
int size;
int numProcs = omp_get_num_procs();
#pragma omp parallel num_threads(numProcs)
{
    size = getProblemSize()/numProcs;
    int tid = omp_get_thread_num();
    doTask(tid*size, size);
}

void doTask(int start, int count)
{ // Each thread’s instance has its own activation record
     for(int i = 0, t=start; i< count; i++; t++)
         doit(t);
}
```

## Private clause

```
int tid, size;
int numprocs = omp_get_num_procs();
#pragma omp parallel num_threads(numProcs) private(tid)
{
    size = getProblemSize()/numProcs;
    tid = omp_get_thread_num();
    doTask(tid*size, size);
}

void doTask(int start, int count)
{ // Each thread's instance has its own activation record
     for(int i = 0, t=start; i< count; i++; t++)
         doit(t);
}
```

# Flush Directive

- #pragma omp flush (var1, var2)
  - ➡ Stand-alone, like barrier
  - ➡ Only directly affects the encountering thread
  - ➡ List-of-vars  ensures that any compiler re-ordering moves all flushes together
  - ➡ implicit:
    - barrier, atomic, critical, locks

# Parallel Loop

```
#pragma omp parallel for
    for (i= 0; i < N; i++) {
        blah …
}
```

- Num of iterations must be known when the construct is encountered
  - ➡ Must be the same for each encountering thread
- Compiler puts a barrier at the end of parallel for
  - ➡ But see nowait

# Parallel For Construct

- #pragma omp for \
  - ➡ private(var1, var2, var3) \
  - ➡ firstprivate(var1, var2, var3)\
  - ➡ lastprivate(var1, var2) \
  - ➡ reduction(operator: list) \
  - ➡ ordered \
  - ➡ schedule(kind[,chunk_size])\
  - ➡ nowait
  - ➡ Canonical For Loop

Once per thread, not once per iteration

Original not corrupted

Last iteration's value

No break

- same loop control expression for all threads in the team.
- At most one schedule, nowait, ordered clause
- chunk_size must be a loop/construct invariant, +ve integer
- ordered clause required if any ordered region inside

# Ordered Directive

```
#pragma omp ordered
{
}
```

- ➡ Binds to inner-most enclosing loop
- ➡ The structured block executed in loop sequential order
- ➡ The loop must declare the ordered clause
- ➡ Each thread must encounter only one ordered region

# Tasks

- Encountering thread creates a task
  - ➡ Code, data, environment ..
- ‘Some’ thread of the team executes the task
  - ➡ Scheduling points
    - Start, End, taskwait Barrier

# Task Construct

```
#pragma omp task \
    if(boolean) \
    untied \
    default(shared | none) \
    private(list) \
    firstprivate(list) \
    shared(list) \
    depend(modifier:list)
{
}
```

- Cannot branch in or out
- No side effect from clause: must not depend on any ordering of the evaluations
- Upto one if clause

```
#pragma omp taskwait
#pragma omp taskgroup
#pragma omp taskyield
```

# Traversal By Tasks

```
struct node { node *left, *right; Data *data;};
extern void process(node* );
void traverse(node* p)
{
    if (p->left)
        #pragma omp task // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        traverse(p->right);
    process(p);                    #pragma omp taskwait
}
```

# Recursive Sum

```
float seq_sum(const float *a, int n)
{
  return (n == 0)? 0.
       : (n == 1)? *a
       : seq_sum(a, n/2) +
         seq_sum(a + n/2, n - n/2);
}
```

```
#pragma omp parallel
#pragma omp single nowait
sum = par_sum(a, n);
```

```
float par_sum(const float *a, int n)
{
    if (n <= SMALL)
       return seq_sum(a, n);
    float x, y;

    #pragma omp task shared(x)
    x = parallel_sum(a, n/2);
    #pragma omp task shared(y)
    y = parallel_sum(a + n/2, n - n/2);
    #pragma omp taskwait
    x += y;

    return x;
}
```

Subodh Kumar

## reduction Clause

- reduction (<op>: <variable list>)
    - + Sum
    - * Product
    - & Bitwise and
    - | Bitwise or
    - ^ Bitwise exclusive or
    - && Logical and
    - || Logical or
- Add to parallel for
    - OpenMP creates a loop to combine copies of the variable
    - The resulting loop may not be parallel

# Reduction

```
int sumint(int data[]) {
    int sum;
    #pragma omp parallel reduction(+:sum)
        sum = partial_sum(data, omp_get_thread_num());
    return sum;
}
```

# Reduction

```
int sum2d(int data[N][N]) {
    int sum;
    #pragma omp parallel for reduction(+:sum)
    for (int i=0; i<N; i++)  {
        for (int j=0; j<N; j++) {
            sum += data[i][j];
        }
    }
    return sum;
}
```

# Critical Section

```
#pragma omp critical (accessBankBalance)
{
}
```

- A single thread at a time
  - through all regions of the same name
- Applies to all threads
- The name is optional
  - Anonymous = global critical region

## #pragma omp barrier

- Stand-alone
- Binds to inner-most parallel region
- All threads in the team must execute
  - they will all wait for each other at this instruction
  - Dangerous:

```
if (! ready)
#pragma omp barrier
```

- Same sequence of work-sharing and barrier for the entire team

# Atomic Directive

```
#pragma omp atomic
    x++;
```

read/write/update/capture

- Light-weight critical section
- Only for some bas
    - ➡ x binop= expr (nc
    - ➡ x++
    - ➡ --x
    - ➡ v = x++

| Thread 0 | Thread 1 |
|---|---|
| // Produce data<br>data = 42;<br><br>// Set flag to signal Thread 1<br>#pragma tmp atomic write<br>    flag = 1; | // Busy-wait until flag is signalled<br>#pragma omp atomic read<br>    myflag = flag<br>while (myflag != 1) {<br>  #pragma omp atomic read<br>    myflag = flag<br>}<br>// Consume data<br>printf(data=%d\n”, data); |

## Atomic?

```
int sum2d(int data[N][N]) {
    int sum = 0;
    #pragma omp parallel for
    for (int i=0; i<N; i++)  {
        for (int j=0; j<N; j++) {
            #pragma omp atomic
            sum += data[i][j];
        }
    }
    return sum;
}
```

- Error due to non-deterministic ordering of mutually visible event
- Data races:
    - ➡ RW
        - Reading of an object potentially overla ... ad
    - ➡ WW
        - Potentially overlapping writes of an obj

```
Transfer(from, to, amt)
{
    #pragma omp atomic read
    fbal  = from.balance;
    if (fbal < amt) return ERROR;
    #pragma omp atomic update
    to.balance += amount;
    #pragma omp atomic update
    from.balance -= amount;
    return OK;
}
```

# Lock

- Lock is an abstract datatype, supports three operations:
- new creates a new lock
  - ➡ Initially “open” by default
- acquire “closes” a lock if open
  - ➡ blocks if closed, until someone else opens it
- release opens a lock

# Re-entrant Lock

- new creates a new lock with no current holder and a count of 0
- acquire blocks *if held by someone different* from the caller
  - ➡ If caller is the holder, increment count
  - ➡ If not already held by a different holder, caller gets a hold with count = 1
- release sets the current holder to “none” if the count is 0.
  - ➡ Otherwise, decrement count

# Helper Functions

- void omp_init_lock (omp_lock_t *);
  - ➡ void omp_destroy_lock (omp_lock_t *);
- void omp_set_lock (omp_lock_t *);
- void omp_unset_lock (omp_lock_t *);
- int omp_test_lock (omp_lock_t *);
- nested lock versions:
  - ➡ e.g., omp_set_nest_lock(omp_test_lock_t *);

```
DoX()
{
    lock(X)
    Operate Exclusively()
    unlock(X)
}
```

```
DoXPlus
{
    lock(X)
    if(not setup X)
        Do(X)
    unlock(X)
}
```

## Efficiency Issues

- Minimize synchronization
  - ➡ Avoid BARRIER, CRITICAL, ORDERED, and locks
  - ➡ Use NOWAIT
  - ➡ Use named CRITICAL sections for fine-grained locking
  - ➡ Use MASTER (instead of SINGLE)
- Parallelize at the highest level possible
  - ➡ such as outer FOR loops
  - ➡ keep parallel regions/tasks large

# Efficiency Issues

- FLUSH is expensive
- LASTPRIVATE has synchronization overhead
- Thread safe malloc/free are expensive
- Reduce False sharing
  - ➡ Careful design of data structures
  - ➡ Use PRIVATE

# Avoiding Errors

- Try to
  - Avoid nested locks
  - Release locks religiously
  - Avoid “while true” (especially, during testing)
- Be careful with
  - ➡ Non thread-safe libraries
  - ➡ Concurrent access to shared data
  - ➡ IO inside parallel regions
  - ➡ Differing views of shared memory (FLUSH)
  - ➡ NOWAIT