

# COMP 364 / 464

## High Performance Computing

**OpenMP:**  
task parallelism

# Computing Terminology

Terms	Definition
• NUMA	• Non Uniform Memory Access. In SMP systems with multiple CPUs, access time to different parts of memory may vary
• Affinity	• Propensity to maintain a process or thread on a hardware execution unit
• SMP	• Symmetric Multi-Process(ing/or). Single OS system with shared memory
• OpenMP	
– Directive	• Comment statement (F90) or #pragma (C/C++) that specifies parallel operations and control
– Construct	• The lexical extent that a directive controls
– Region	• All code controlled by a directive– lexical extent + content of called routines
• Runtime	• Code or a library within an executable that interacts with the operating system and can control code execution

# OpenMP Constructs

## OpenMP Language “extensions”

Parallel Control  
Structures

- governs flow of control in the program

parallel  
directive

Parallel Control  
worksharing

- distributes work among threads

do  
for  
sections  
single  
construct

Control  
Single Task

- assigns work to a thread

task  
directive

Data  
Environment

- specifies variables as shared or private

shared  
private  
reduction  
clause

Synchronization

- coordinates thread execution

critical  
atomic  
barrier  
directive

Runtime  
Environment

- runtime functions
- environment variables

```
omp_set_num_threads()  
omp_get_thread_num()  
OMP_NUM_THREADS  
OMP_SCHEDULE
```

- **scheduling**  
static, dynamic, guided

Directive Sentinels: “!\$omp” and “#pragma omp” not shown.

# OpenMP Synchronization

```
InitializeQueue( allWorkItems );
#pragma omp parallel
{
    bool notEmpty = true;
    while (notEmpty)
    {
        workItemType workItem;

        // Pop a task off the queue one thread at a time.
        #pragma omp critical
        {
            notEmpty = PopQueue(&workItem);
        }

        // Do some work (if necessary)
        if (notEmpty)
            foo(workItem);
    }
}
// All threads block (or sync) here.
```

# OpenMP Tasks

```
List *head = InitializeListItems( allItems ); // Serially
#pragma omp parallel shared(default)
{
    #pragma omp single nowait
    {
        List *p = head;
        for (List *p = head; p != NULL; p = p->next) {
            #pragma omp task
            doSomething (p); // p is private to each task.
        }

        #pragma omp taskwait

        List *p = head;
        for (List *p = head; p != NULL; p = p->next) {
            #pragma omp task
            doSomethingElse (p);
        }
    }
} // All threads block (or sync) here.
```

# OpenMP Recursive Tasks

```
void doSomething (Node *p)
{
    if (p->size > thresholdSize)
        // Do some actual work.
    else
    {
        // Split up workitem *p further.
        Node *q = <Something to halve p>

        #pragma omp task
        doSomething (q);

        #pragma omp task
        doSomething (p);

        // Wait for child tasks to finish.
        #pragma omp taskwait
    }
    return;
}
```

# OpenMP Recursive Tasks

```
void doSomething (Node *p, int depth)
{
    if (p->size > thresholdSize)
        // Do some actual work.
    else
    {
        // Split up workitem *p further.
        Node *q = <Something to halve p>

        #pragma omp task
        doSomething (q, depth+1);

        #pragma omp task if (depth < thresholdDepth)
        doSomething (p, depth+1);

        // Wait for child tasks to finish.
        #pragma omp taskwait
    }
    return;
}
```

# OpenMP Dependent Tasks

```
ListNode *head = InitializeListItems( allItems ); // Serially
#pragma omp parallel shared(default)
#pragma omp single nowait
{
    ListNode *p = head;
    for (ListNode *p = head; p != NULL; p = p->next)
    {
        ListItem *item = p->item;

        #pragma omp task shared(item) depend(out: item)
        doSomethingA (item);

        #pragma omp task shared(item) depend(in: item)
        doSomethingB (item);

        #pragma omp task shared(item) depend(in: item)
        doSomethingC (item);
    }
} // All threads block (or sync) here.
```