Introduction to OpenMP

Lecture 7: Tasks

- The task construct defines a section of code
- Inside a parallel region, a thread encountering a task construct will package up the task for execution
- Some thread in the parallel region will execute the task at some point in the future

task directive

|epcc|

Syntax:

Fortran:

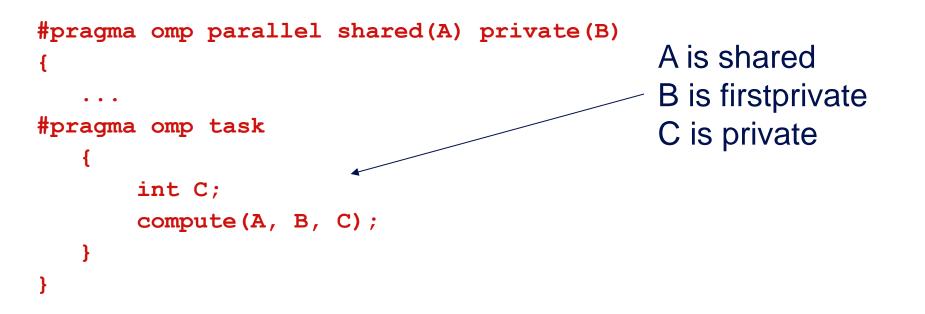
C/C++:

!\$OMP TASK [clauses]
 structured block
!\$OMP END TASK
#pragma omp task [clauses]

structured-block

Data Sharing

- The default for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope).
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared.



- At thread barriers (explicit or implicit)
 - applies to all tasks generated in the current parallel region up to the barrier
- At taskwait directive
 - i.e. Wait until all tasks defined in the current task have completed.
 - Fortran: **!\$OMP TASKWAIT**
 - C/C++: #pragma omp taskwait
 - Note: applies only to tasks generated in the current task, not to "descendants".

Example



```
p = listhead ;
while (p) {
    process (p);
    p=next(p) ;
}
```

- Classic linked list traversal
- Do some work on each item in the list
- Assume that items can be processed independently
- Cannot use an OpenMP loop directive

Parallel pointer chasing



Only one thread

```
packages tasks
#pragma omp parallel
  #pragma omp single private(p)
    p = listhead ;
    while (p) {
        #pragma omp task
                 process (p);
        p=next (p) ;
                                     p is firstprivate by
                                     default inside this
                                     task
```

Parallel pointer chasing on multiple lists

```
All threads package
#pragma omp parallel
                                tasks
   #pragma omp for private(p)
   for ( int i =0; i <numlists ; i++) {</pre>
       p = listheads [ i ] ;
       while (p) {
       #pragma omp task
            process (p);
       p=next (p ) ;
        }
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete

- |epcc|
- Certain constructs have task scheduling points at defined locations within them
- When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (called *task switching*)
- It can then return to the original task and resume

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
    process(item[i]);
}</pre>
```

- Risk of generating too many tasks
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
 - execute an already generated task (draining the "task pool")
 - execute the encountered task

Using tasks

- Getting the data attribute scoping right can be quite tricky
 - default scoping rules different from other constructs
 - as ever, using **default(none)** is a good idea

- Don't use tasks for things already well supported by OpenMP
 - e.g. standard do/for loops
 - the overhead of using tasks is greater

- Don't expect miracles from the runtime
 - best results usually obtained where the user controls the number and granularity of tasks



• Mandelbrot example using tasks.