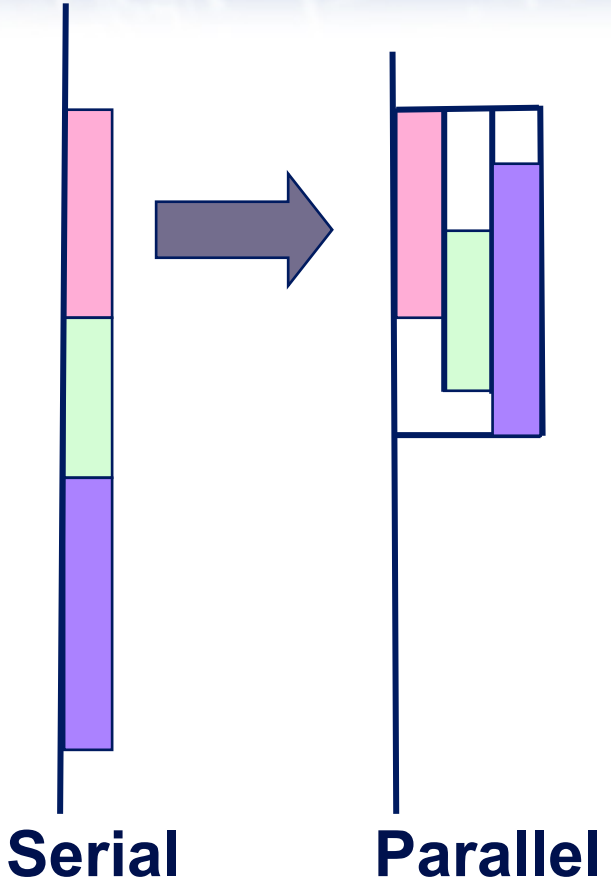
Threaded
Programming
Lecture 6: Tasks

What are tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task.



OpenMP tasks

- The task construct includes a structured block of code
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- Some thread in the parallel region will execute the task at some point in the future
  - note: could be encountering thread, right now
- Tasks can be nested: i.e. a task may itself generate tasks.

`task` directive

Syntax:

Fortran:

```
!$OMP TASK [clauses]
    structured block
!$OMP END TASK
```

C/C++:

```
#pragma omp task [clauses]
    structured-block
```

Example

```
#pragma omp parallel
{
  #pragma omp master
   {
     #pragma omp task
        fred();
     #pragma omp task
        daisy();
     #pragma omp task
        billy();
   }
}
```

Create some threads

Thread 0 packages tasks

Tasks executed by some thread in some order

When/where are tasks complete?

- 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" .
  - The code executed by a thread in a parallel region is considered a task here

When/where are tasks complete?

- At the end of a taskgroup region
  - Fortran:

```
!$OMP TASKGROUP
    structured block
!$OMP END TASKGROUP
```

  - C/C++:

```
#pragma omp taskgroup
    structured-block
```

  - wait until all tasks created within the taskgroup have completed
  - applies to all “descendants”

Example

```
#pragma omp parallel
{
  #pragma omp master
   {
     #pragma omp task
        fred();
     #pragma omp task
        daisy();
     #pragma taskwait
     #pragma omp task
        billy();
   }
}
```

`fred()` and `daisy()` must complete before `billy()` starts

Linked list traversal

```
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 linked list traversal

```
#pragma omp parallel
{
  #pragma omp master
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
               {
                 process (p);
               }
       p=next (p) ;
     }
   }
}
```

Only one thread packages tasks

makes a copy of `p` when the task is packaged

Parallel linked list traversal

Thread 0:

```
p = listhead ;
while (p) {
< package up task >
   p=next (p) ;
}

while (tasks_to_do){
  < execute task >
}

< barrier >
```

Other threads:

```
while (tasks_to_do) {
< execute task >
}

< barrier >
```

Parallel pointer chasing on multiple lists

```
#pragma omp parallel
{
    #pragma omp for private(p)
    for ( int i =0; i <numlists; i++) {
        p = listheads[i] ;
        while (p ) {
        #pragma omp task firstprivate(p)
            {
              process(p);
            }
        p=next(p);
        }
    }
}
```

All threads package tasks

Data scoping with tasks

- Variables can be shared, private or firstprivate with respect to task
- These concepts are a little bit different compared with threads:
  - If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered
  - If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  - If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered

Data scoping defaults

- The behavior you want 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 private when the task construct is encountered are firstprivate by default
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
   ...
#pragma omp task
   {
      int C;
      compute(A, B, C);
   }
}
```

A is shared
B is firstprivate
C is private

Example: Fibonacci numbers

```
int fib (int n)
{
  int x,y;
  if ( n < 2 ) return n;
  x = fib(n-1);
  y = fib(n-2);
  return x+y
}

int main()
{
  int NN = 5000;
  fib(NN);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Inefficient $O(n^2)$ recursive implementation!

Parallel Fibonacci

```
int fib ( int n )
{
int x,y;
   if ( n < 2 ) return n;
#pragma omp task shared(x)
   x = fib(n-1);
#pragma omp task shared(y)
   y = fib(n-2);
#pragma omp taskwait
   return x+y
}
int main()
{  int NN = 5000;
   #pragma omp parallel
   {
       #pragma omp master
          fib(NN);
   }
}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- `x,y` are local, and so private to current task
  - must be shared on child tasks so they don't create their own firstprivate copies at this level!

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

Parallel pointer chasing again

```
#pragma omp parallel
{
  #pragma omp single private(p)
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
              {
                process (p,nitems);
              }
         for (i=0; i<nitems &&p; i++){
            p=next (p) ;
         }
     }
   }
}
```

process **nitems** at a time

skip **nitems** ahead in the list

Parallel Fibonacci again

```
int fib ( int n )
{
int x,y;
   if ( n < 2 ) return n;
#pragma omp task shared(x) if(n>30)
   x = fib(n-1);
#pragma omp task shared(y) if(n>30)
   y = fib(n-2);
#pragma omp taskwait
   return x+y
}
int main()
{  int NN = 5000;
   #pragma omp parallel
   {
       #pragma omp master
          fib(NN);
   }
}
```

- Stop creating tasks at some level in the tree.

Exercise

- Mandelbrot example using tasks.