CSE502: Foundations of Parallel Programming

Lecture 19: Tasks-based Parallelism in OpenMP

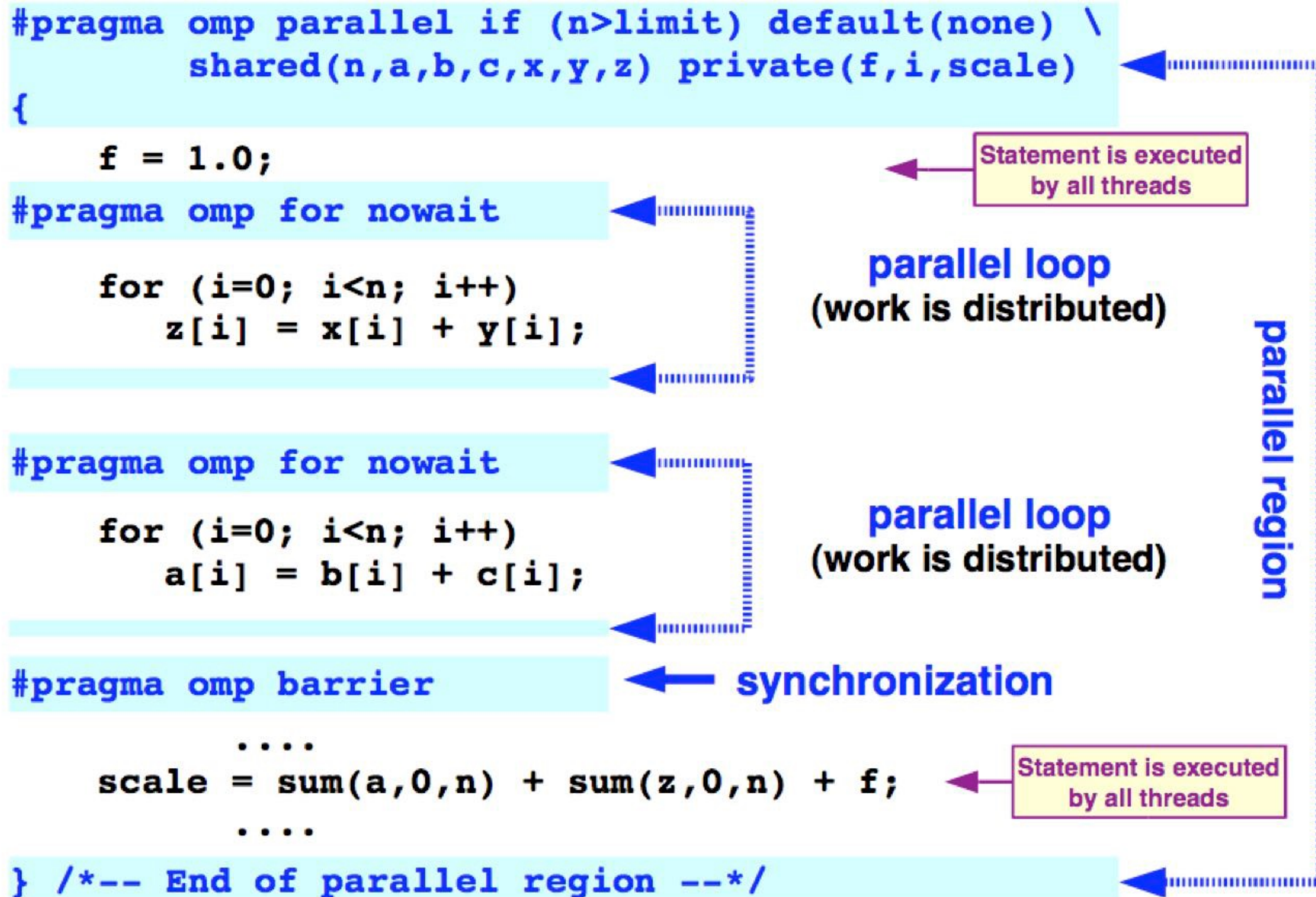
Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Class: work-sharing constructs in OpenMP



Today's Class

Programming Irregular Applications with OpenMP*

Tim Mattson
Intel Corp.
timothy.g.mattson@ intel.com

Alice Koniges
Berkeley Lab/NERSC
AEKoniges@lbl.gov

Clay Breshears
PAPPS
clay.breshears@gmail.com

Jeremy Kemp
University of Houston
jakemp@uh.edu

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

Acknowledgement: All the slides that appear in this lecture is adapted from the tutorial "Programming Irregular Applications with OpenMP" that was given at SC 2016 (Salt Lake City, Utah). Presenters of this tutorial were Dr. Tim Mattson, Dr. Alice Koniges, Dr. Clay Breshears, and Dr. Jeremy Kemp

Outline

- Explicit Tasks in OpenMP
- Data sharing across tasks

Not all programs have simple loops OpenMP can parallelize

- Consider a program to traverse a linked list:

```
p=head;
while (p) {
	processwork(p);
	p = p->next;
}
```

Can you use work-sharing pragmas to parallelize this program??

- OpenMP can only parallelize loops in a basic standard form with loop counts known at runtime

Linked lists with parallel loops

```
while (p != NULL) {
     p = p->next;
      count++;
}
p = head;
for(i=0; i<count; i++) {
      parr[i] = p;
      p = p->next;
   }
#pragma omp parallel
{
     #pragma omp for schedule(dynamic,1)
     for(i=0; i<count; i++)
        processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

Why “dynamic” schedule?

Linked lists with parallel loops

```
while (p != NULL) {
     p = p->next;
      count++;
}
p = head;
for(i=0; i<count; i++) {
      parr[i] = p;
      p = p->next;
   }
#pragma omp parallel
{
     #pragma omp for schedule(dynamic,1)
     for(i=0; i<count; i++)
        processwork(parr[i]);
}
```

Count number of items in the linked list

There has got to be a better way!!!

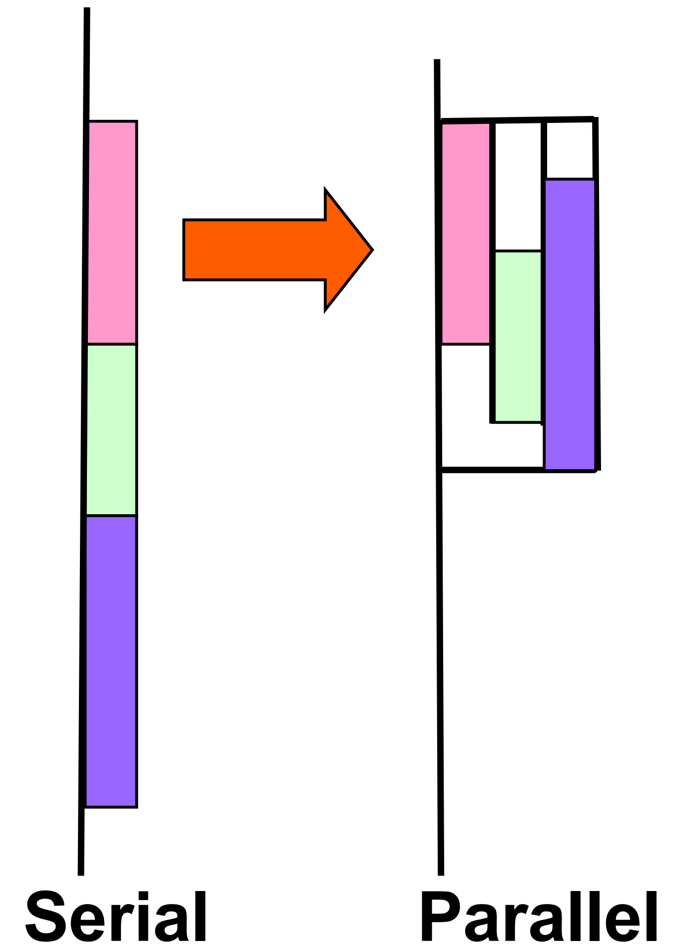
Process nodes in parallel with a for loop

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.
  - The thread that encounters the task construct may execute the task immediately.
  - The threads may defer execution until later

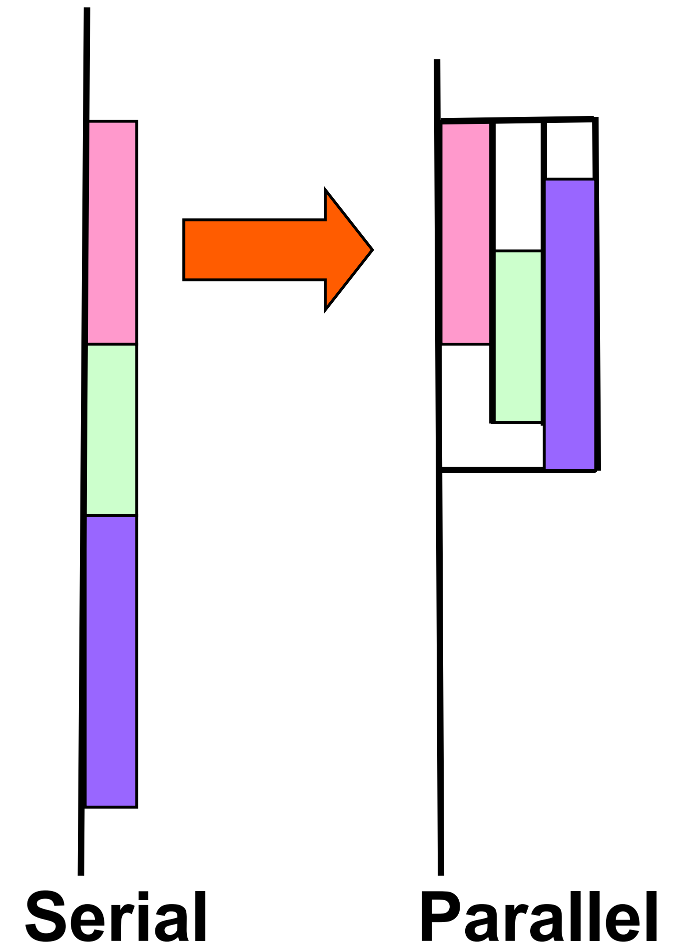
Lambda in async?

push_task_to_runtime() ?



What are 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
- Tasks can be nested: i.e. a task may itself generate tasks.
  - HClib async ?



Task Directive

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

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

Create some threads (→ #pragma omp parallel)

Thread 0 packages tasks (→ #pragma omp master)

Tasks executed by some thread in some order (→ each #pragma omp task)

All tasks complete before this barrier is released (→ closing })

Task Directive

Task Synchronization

- **Task Synchronization explained:**

```
#pragma omp parallel num_threads(np)
{
#pragma omp task
   function_A();
#pragma omp barrier
#pragma omp single
   {
#pragma omp task
      function_B();
   }
}
```

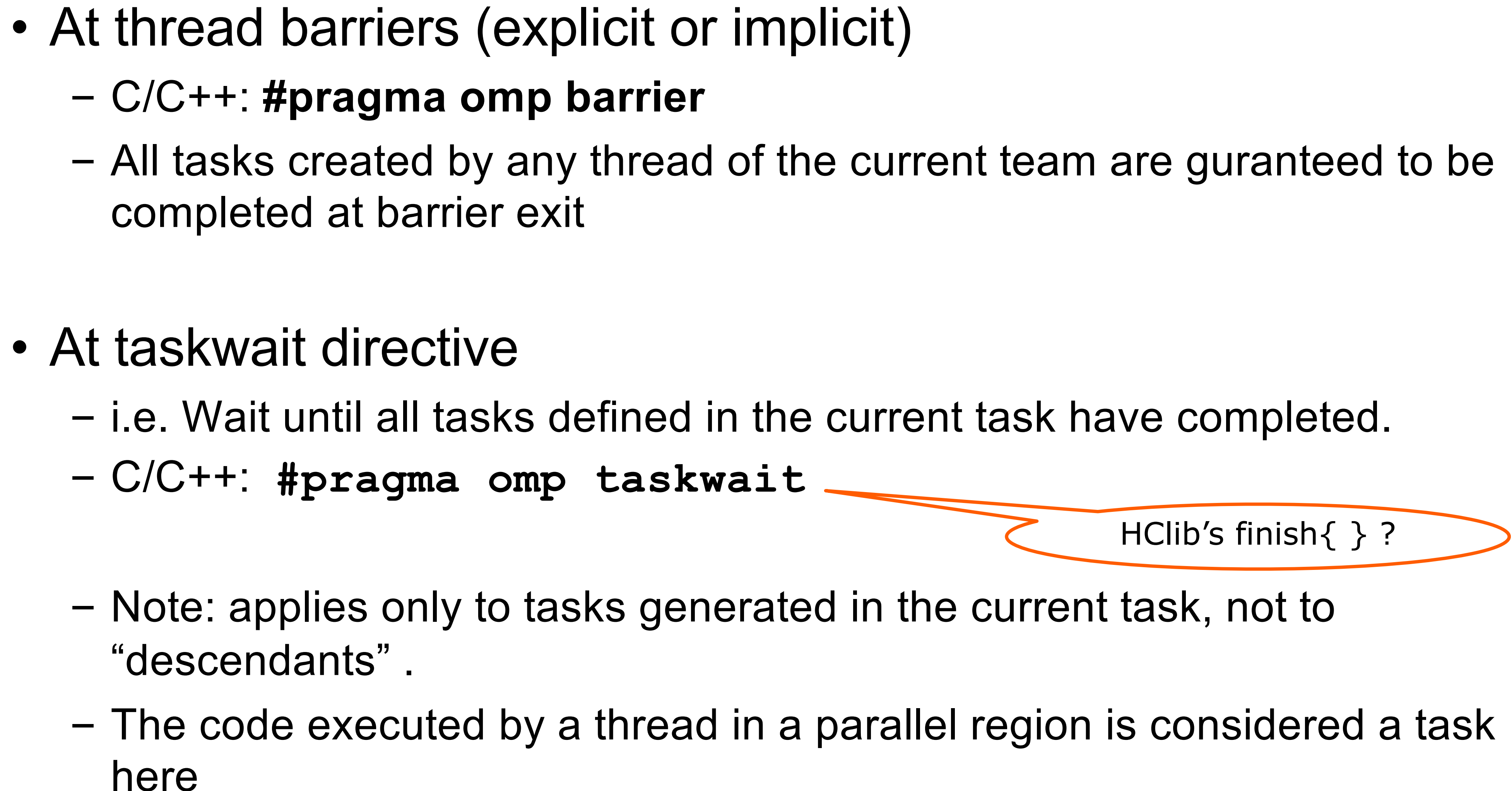
np Tasks created here, one for each thread

All Tasks guaranteed to be completed here

1 Task created here

B-Task guaranteed to be completed here

When/where are tasks complete?



- At thread barriers (explicit or implicit)
  - C/C++: **#pragma omp barrier**
  - All tasks created by any thread of the current team are guranteed to be completed at barrier exit

- At taskwait directive
  - i.e. Wait until all tasks defined in the current task have completed.
  - 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

Example

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

Can we have?
`#pragma omp single`

Yes, but "single" has an implicit barrier unlike "master"

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

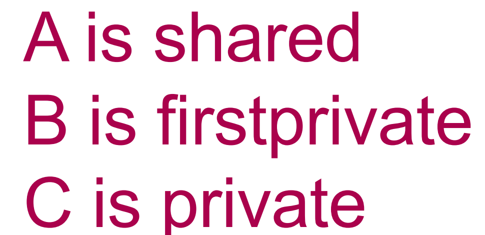
Data scoping with tasks

- 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);
   }
}
```



Data scoping defaults (1/6)

```
int a=1, b=2;
#pragma omp parallel default(none)
{
    int c=3;
    #pragma omp task
    {
        printf(“IN: a=%d, b=%d, c=%d\n”,a++,b++,c++);
    }
    #pragma omp taskwait
    printf(“OUT: c=%d\n”,c);
}
printf(“OUT: a=%d, b=%d\n”,a,b);
```

What will be output ? (OMP_NUM_THREADS=1)
>>compilation error !!

Note: This is not from SC16 tutorial

Data scoping defaults (2/6)

```
int a=1, b=2;
#pragma omp parallel default(none) shared(a,b)
{
   int c=3;
   #pragma omp task
   {
      printf("IN: a=%d, b=%d, c=%d\n",a++,b++,c++);
   }
   #pragma omp taskwait
   printf("OUT: c=%d\n",c);
}
printf("OUT: a=%d, b=%d\n",a,b);
```

What will be output ? (OMP_NUM_THREADS=1)
>> IN: a=1,b=2,c=3
>> OUT: c=3
>> OUT: a=2,b=3

Note: This is not from SC16 tutorial

Data scoping defaults (3/6)

```
int a=1, b=2;
#pragma omp parallel default(none) private(a,b)
{
    int c=3;
    #pragma omp task
    {
        printf("IN: a=%d, b=%d, c=%d\n",a++,b++,c++);
    }
    #pragma omp taskwait
    printf("OUT: c=%d\n",c);
}
printf("OUT: a=%d, b=%d\n",a,b);
```

What will be output ? (OMP_NUM_THREADS=1)
>> IN: a=0,b=0,c=3
>> OUT: c=3
>> OUT: a=1,b=2

Note: This is not from SC16 tutorial

Data scoping defaults (4/6)

```
int a=1, b=2;
#pragma omp parallel default(none) firstprivate(a,b)
{
    int c=3;
    #pragma omp task
    {
        printf(“IN: a=%d, b=%d, c=%d\n”,a++,b++,c++);
    }
    #pragma omp taskwait
    printf(“OUT: c=%d\n”,c);
}
printf(“OUT: a=%d, b=%d\n”,a,b);
```

What will be output ? (OMP_NUM_THREADS=1)
>> IN: a=1,b=2,c=3
>> OUT: c=3
>> OUT: a=1,b=2

Note: This is not from SC16 tutorial

Data scoping defaults (5/6)

```
int a=1, b=2;
#pragma omp parallel default(none) shared(a) private(b)
{
    int c=3;
    #pragma omp task
    {
        printf(“IN: a=%d, b=%d, c=%d\n”,a++,b++,c++);
    }
    #pragma omp taskwait
    printf(“OUT: c=%d\n”,c);
}
printf(“OUT: a=%d, b=%d\n”,a,b);
```

What will be output ? (OMP_NUM_THREADS=1)
>> IN: a=1,b=0,c=3
>> OUT: c=3
>> OUT: a=2,b=2

Note: This is not from SC16 tutorial

Data scoping defaults (6/6)

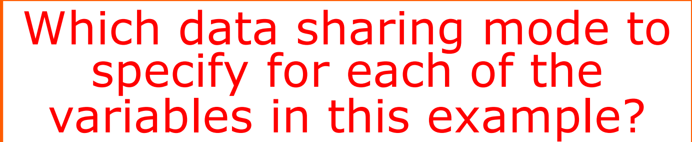
```
int a=1, b=2;
#pragma omp parallel default(none) shared(a) private(b)
{
   int c=3;
   b = 1;
   #pragma omp task
   {
      printf("IN: a=%d, b=%d, c=%d\n",a++,b++,c++);
   }
   #pragma omp taskwait
   printf("OUT: c=%d,b=%d\n",c,b);
}
printf("OUT: a=%d, b=%d\n",a,b);
```

What will be output ? (OMP_NUM_THREADS=1)
>> IN: a=1,b=1,c=3
>> OUT: c=3,b=1
>> OUT: a=2,b=2

Note: This is not from SC16 tutorial

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 NW = 40;
   fib(NW);
}
```



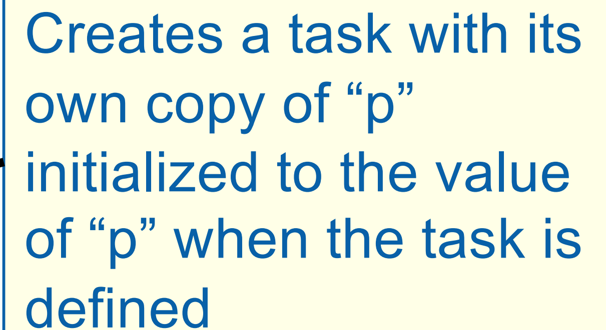
Parallel Fibonacci

```
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 NW = 40;
   #pragma omp parallel
   {
      #pragma omp master
           fib(NW);
   }
}
```

You must specify “shared” for “x” and “y”, as otherwise they will become “fristprivate” to tasks

Linked lists with tasks

```
#pragma omp parallel
{
  #pragma omp single
  {
     p=head;
    while (p) {
      #pragma omp task firstprivate(p)
            processwork(p);
        p = p->next;
    }
  }
}
```



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 >
```

Task switching

- Consider the following example ... Where the program may generate so many tasks that the internal data structures managing tasks overflow.

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

Task switching

- Consider the following example ... Where the program may generate so many tasks that the internal data structures managing tasks overflow.

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

- Solution … Task switching; Threads can switch to other tasks at certain points called ***thread scheduling*** points.
- With Task switching, a thread can
  - Execute an already generated task … to “drain the task pool”
  - Execute the encountered task immediately (instead of deferring task execution for later)

if Clause

```
#pragma omp task if(expr)
```

- If the expression of an if clause on a task evaluates to false
  - The encountering task is suspended
  - The new task is executed immediately
  - The parent task resumes when new tasks finishes
  - Used for optimization, e.g. avoid creation of small tasks

Next Class

- Introduction to distributed memory parallel programming
- Lab-6 on Saturday (Tuesday-TT)
  - Syllabus: Today's lecture
- Quiz-4 (Last remaining quiz)
  - **Syllabus: Lectures 17-19**