KOÇ UNIVERSITY

Lecture 7
Data vs Task Parallelism

Didem Unat

COMP 429/529 Parallel Programming

Today

- Types of Parallelism
  - Data Parallelism
  - Task Parallelism
- Task Parallelism in OpenMP

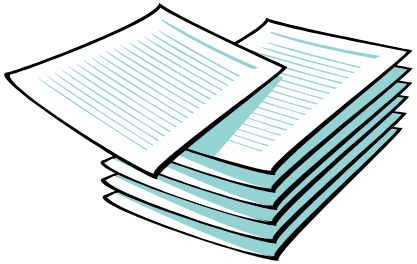
- Announcements
  - 1st Exam on Wednesday Nov 3rd (**in PS hour in-person**)
  - No Lecture on Oct 28th

Types of Parallelism

- Task parallelism
  - Partition various distinct tasks to solve the problem in parallel.
  - Each task performs a different operation/computation
- Data parallelism
  - Partition the data used in solving the problem in parallel
  - Each thread/process carries out similar operations on it's part of the data.

Task vs Data Parallelism

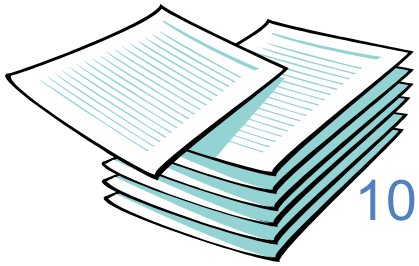
- Assume there are 30 students
  - Thus 30 exams and
  - 3 questions in the exam

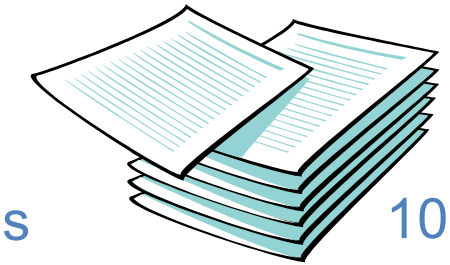


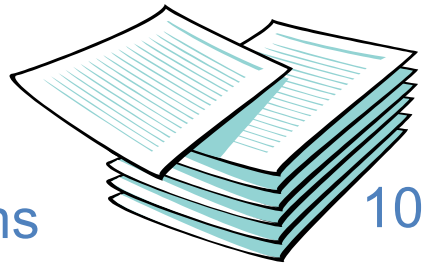
Grading

- An angry bird, me and Limon are grading the exams



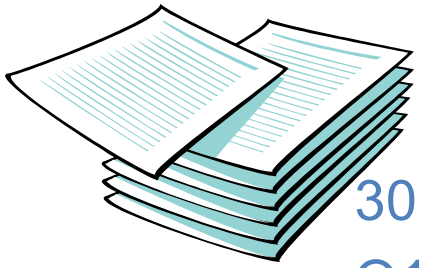




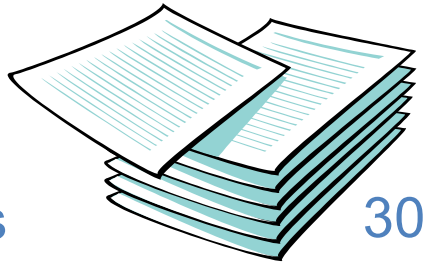
- Scenario #1: Each of us gets 10 students to grade

Grading

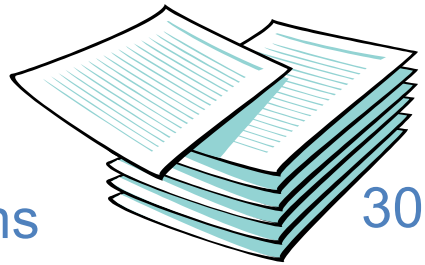
- An angry bird, me and Limon are grading the exams



30 exams
Q1



30 exams
Q2



30 exams
Q3

- Scenario #2: Each of us grades only 1 question

Task vs Data Parallelism

- Scenario #1: Data parallelism
  - Data (exam papers) is divided among all, the same work is performed on the data but different parts of the data
- Scenario #2: Task parallelism
  - Tasks (questions) are divided, each person gets a different question to grade

Task vs Data Parallelism- Discussion

- (a) Identify a portion of home building that can employ *data parallelism*, where "data" in this context is any object used as an input to the home-building process, as opposed to tools that can be thought of as processing resources.

- (b) Identify *task parallelism* in home building by defining a set of tasks. Work out a schedule that shows when the various tasks can be performed.

- (c) Describe how task and data parallelism can be combined in building a home. What computations can be reassigned to different workers to balance the load?

Power of Data Parallelism

- Data parallelism: perform the same operation on multiple values (often array elements)
  - Also includes reductions, broadcast, scan..
- Many parallel programming models use some data parallelism
  - SIMD units (and previously SIMD supercomputers)
  - CUDA / GPUs
  - MapReduce
  - MPI collectives
  - OpenMP for loops

Data Parallelism in OpenMP

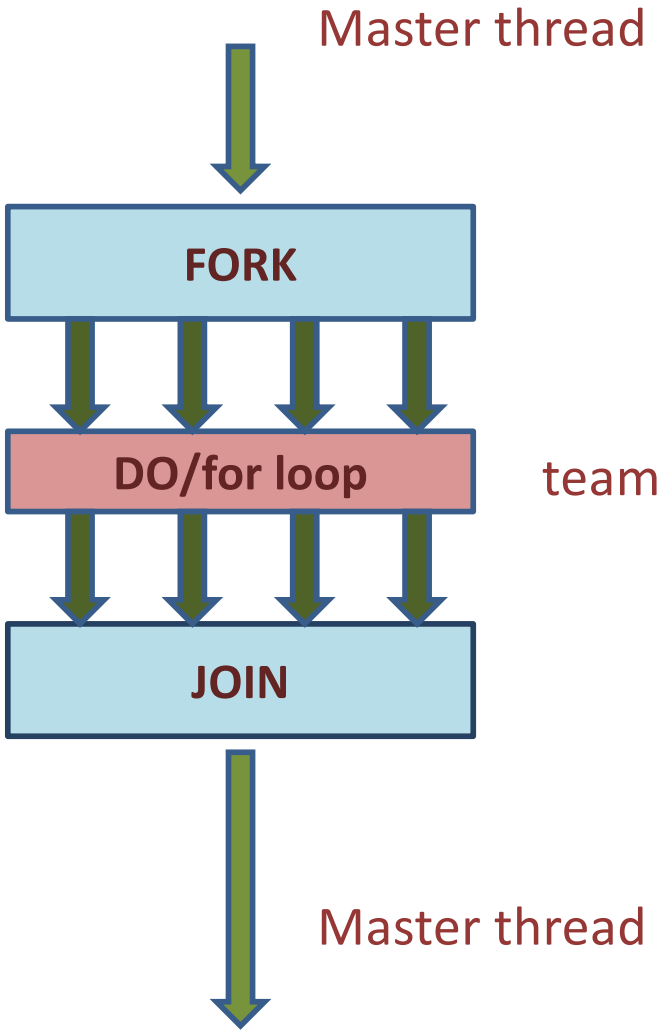
- Loop iteration space is divided between threads
- The same loop body is executed by different threads on different data points

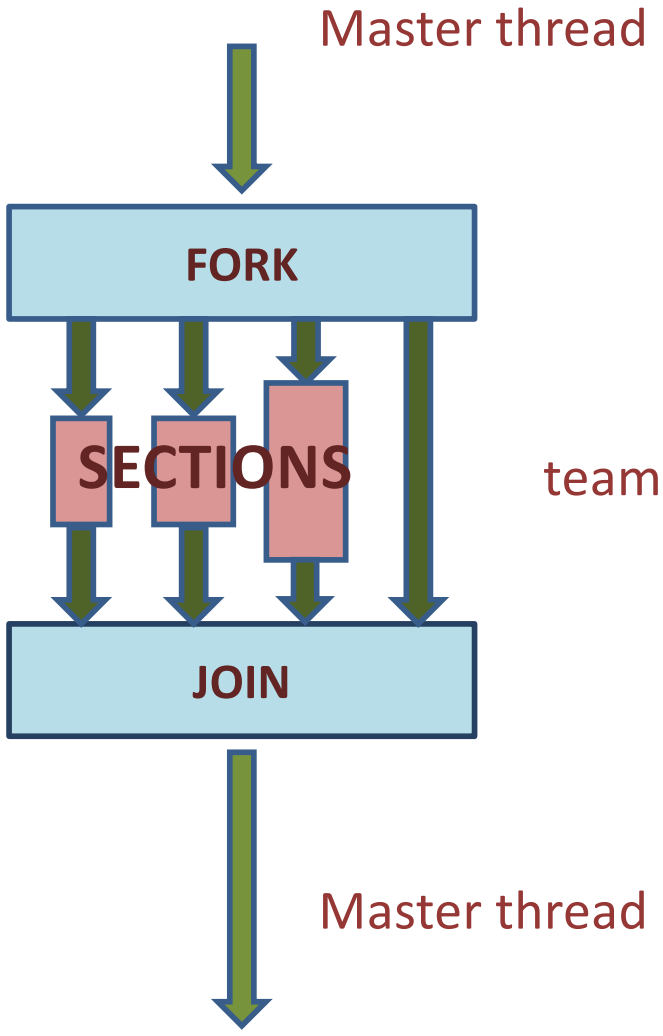
```
Parallel Program


double res[N];


#pragma omp parallel for
for(int i=0; i < N; i++)
      do_huge_comp(res[i]);
```

Task Parallelism in OpenMP

- OpenMP supports for task parallelism
  - Parallel sections: different threads execute different code
    - OpenMP Pre-3.0
  - Tasks: tasks are created and executed at separate times
    - Starting OpenMP 3.0, tasks are supported in OpenMP

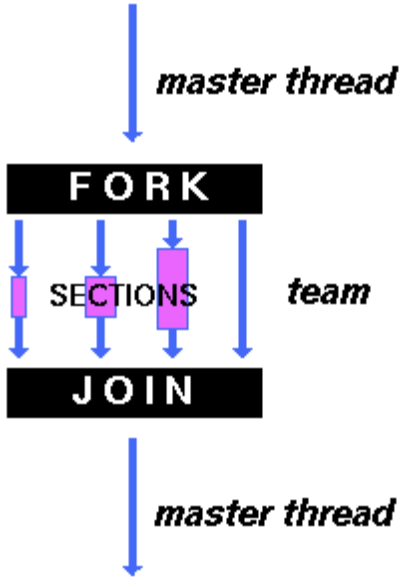
OpenMP Sections





Parallel Sections in OpenMP

```
#pragma omp parallel shared(n,a,b,c,d) private(i)
{
   #pragma omp sections
   {
      #pragma omp section
         for (i=0; i<n; i++)
            d[i] = 1.0/c[i];
      #pragma omp section
         for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;
   } /*-- End of sections --*/

} /*-- End of parallel region
```



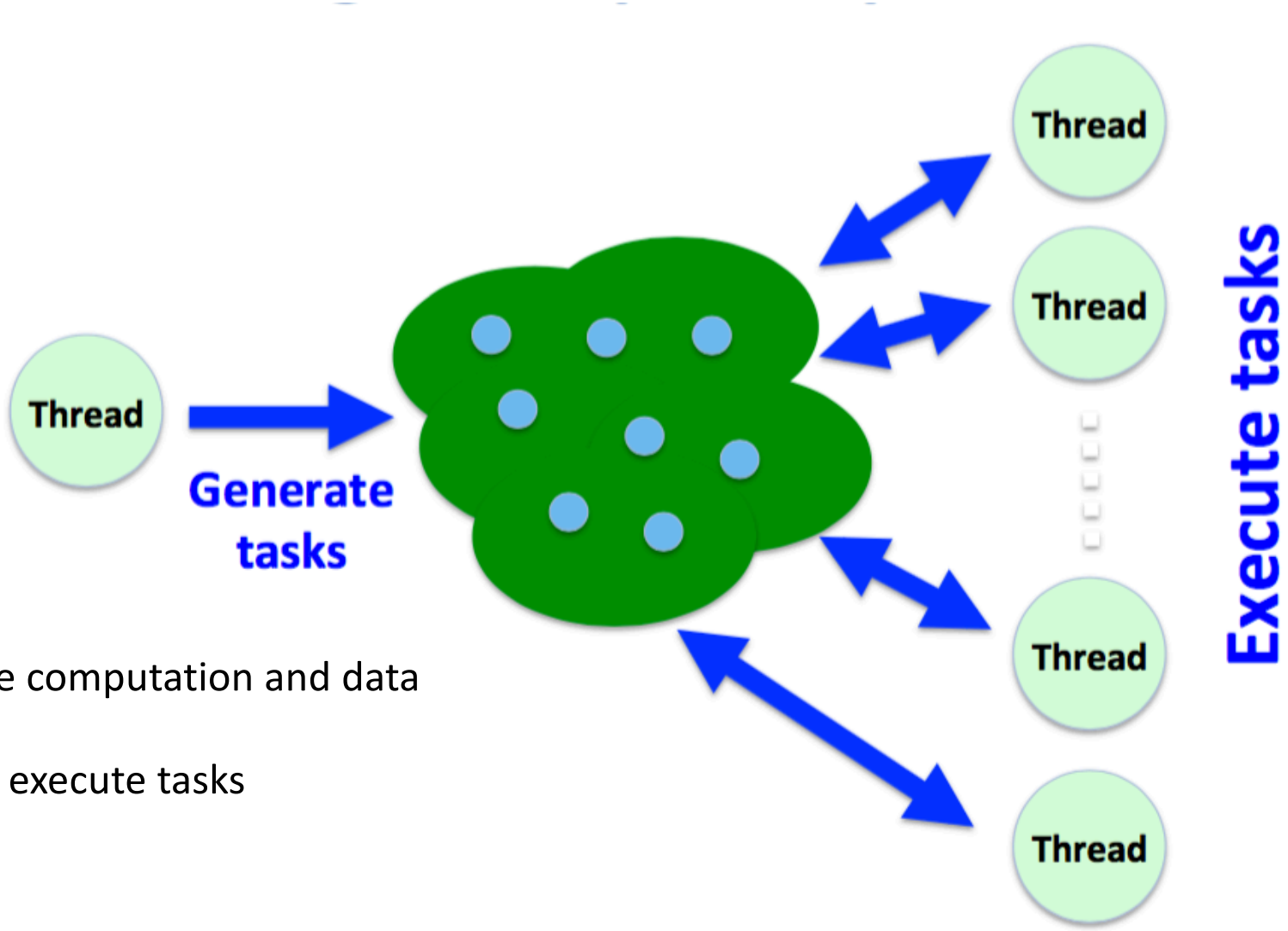
OpenMP Sections

- The SECTIONS directive is a work-sharing construct
  - It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Independent SECTION directives are nested within a SECTIONS directive.
- Each SECTION is executed once by a thread in the team.
- Different sections may be executed by different threads.
  - It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.

Tasks in OpenMP 3.0

- A more flexible and general approach
- A task has
  - Code to execute
  - A data environment (shared, private, reduction)
  - An assigned thread executes the code and uses the data
- Thread vs Task
  - When a thread encounters a task construct, it may choose to execute the task immediately or
  - Defer its execution until a later time.
    - If task execution is deferred, then the task is placed in a pool of tasks.
    - A thread that executes a task may be different from the thread that originally encountered it.

Tasks vs Threads



Tasks are computation and data

Threads execute tasks

Definitions

- Task construct – task directive plus structured block

```
#pragma omp task
{
    //structured block
}
```

- Task – the package of code and instructions for allocating data created when a thread encounters a task construct
- Task region – the dynamic sequence of instructions produced by the execution of task by a thread
- Tasks can be nested: i.e. a task may itself generate tasks.

Developer's View vs Runtime

- **Developer**
  - Uses pragmas to specify what the tasks do
  - Responsible for identifying what can execute independently and what can't
- **OpenMP Runtime System**
  - When a thread encounters a task construct, a new task is generated
  - The moment of execution of the task is up to the runtime system
  - Execution can either be immediate or delayed
  - Completion of a task can be enforced through **task synchronization**

Example

- Parallel pointer chasing using tasks

```
#pragma omp parallel
{
    #pragma omp single
    {
        node* p = listhead;
        while (p){
          #pragma omp task firstprivate(p)
           {
                process(p);
           }
           p = p->next;
        }
    }
}
```

One thread generates tasks

Task gets its own copy of p initialized to the value of p when the task is defined

for each p in the list, create a new task to process it

Implicit task wait

Example

- Parallel pointer chasing using tasks

```
#pragma omp parallel
{
    #pragma omp single
    {
        node* p = listhead;
        while (p){
          #pragma omp task
           {
                process(p);
           }
           p = p->next;
         }
    }
}
```

`firstprivate` clause is used to initialize the private copy of a variable for each thread.

p is `firstprivate` by default

Why? because the task may not be executed until later (and variables may have gone out of scope)

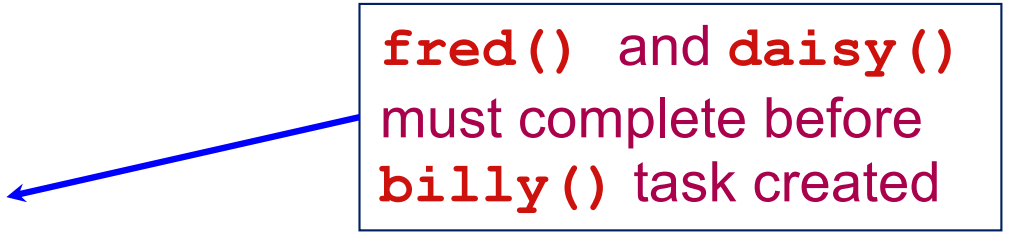
Variables that are private when the task construct is encountered are firstprivate by default [OpenMP grandmothering you here]

When/Where are tasks complete?

- At barriers, explicit or implicit
  - Applies to all tasks generated in the current parallel region
- At task barriers
  - Wait until **all tasks defined in the current task** have completed
  - #pragma omp taskwait
  - Note: applies only to tasks generated in the current task

Taskwait

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



Lab 4: Task and Data Scoping

- ssh username@login.kuacc.ku.edu.tr
- Copy the lab from here to your home directory

  /kuacc/users/dunat/COMP429/OpenMP/Labs/Lab4-task-data-scope.c

- Request time in an interactive queue
  - srun -N 1 -n 4 -p short --time=00:30:00 --pty bash

Lab 4 Todos

- TODO 1:
  - Add single clause to create one task
  - #pragma omp single
- TODO 2
  - Comment in the printf within single clause after the task block
- TODO 3
  - Add omp taskwait
  - #pragma omp taskwait
- What do you observe?

Data Scoping

```
int a = 1;
void foo()
{
    int b=2,c=3;
    #pragma omp parallel shared(b)
    {
        ...
        #pragma omp parallel private(b)
        {
            int d=4;
            #pragma omp task
            {
                int e = 5;
                // a = //shared
                // b = //firstprivate
                // c = //shared
                // d = //firstprivate
                // e = //private
            }
        }
    }
}
```

- To make sure about scopes, clearly state them in the pragma construct
- Use default(none)
- Firstprivate inherits the value assigned to it (initialized) but it is private to a thread `

Data Scoping

```
int a = 1;
void foo()
{
    int b=2,c=3;
    #pragma omp parallel shared(b)
    {
        ...
        #pragma omp parallel private(b) //Line X
        {
            int d=4;
            #pragma omp task
            {
                int e = 5;
                a = //shared
                b = //firstprivate
                c = //shared
                d = //firstprivate
                e = //private
            }
        }
    }
}
```

1
0 or undefined
3
4
5

The value of b is zero or undefined
because it is not specified as firstprivate in line X

Another Example

```
int fib(long n){
    long i, j;
    if (n < 2) return n;
    else {
        #pragma omp task shared (i)
            i = fib(n-1);
        #pragma omp task shared (j)
            j = fib(n-2);
        #pragma omp taskwait
        return i+j;
     }
}
```

Need to create parallel region to execute tasks

```
int main(){
...
#pragma omp parallel\
                shared(n,v)
{
    #pragma omp single
        v=fib(n);
}
...
}
```

One thread executes the initial fib(n) call

Parallel Fibonacci Example

- 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)
- By default, only 2 threads will be active in most implementations. Set OMP_MAX_ACTIVE_LEVELS with n>1 to get n-levels of **nested parallelism**
- **x,y** are local, and so by default they are private to current task
  - must be shared on child tasks so they don't create their own firstprivate copies at this level!

Nfib example

- Creates way to many tasks
- Tasks do nothing but create tasks
- Parent task who creates the task also does nothing
- Can severely degrade the performance
- While generating these tasks, the implementation may reach its limit on unassigned tasks and it may suspend its task generation.

- We can prune the task tree by adding a condition

Conditional Parallelization

`if (scalar expression)`

Only execute in parallel if expression evaluates to true
Otherwise, execute serially

```
#pragma omp parallel if (n > threshold) \
shared(n,x,y) private(i)
{

} /*-- End of parallel region --*/
```

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 task finishes

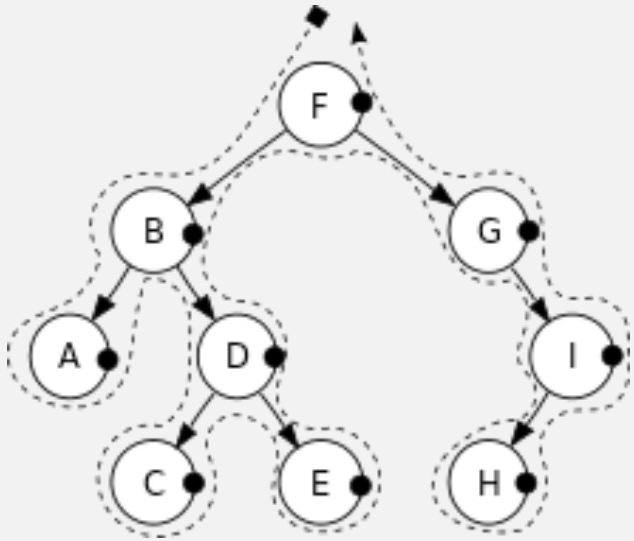
```
#pragma omp task if(expr)
```

Used for optimization, e.g. avoid creation of too many small tasks

Another Example

- Post-order tree traversal
- Parent task is suspended until children tasks complete (A, C, E, D, B, H, I, G, F)

```
void postorder (node* p){

   if (p-> left)
     #pragma omp task
         postorder(p-> left);
   if (p-> right)
     #pragma omp task
         postorder (p->right);

   //wait for descendants
   #pragma omp taskwait
     process (p);
}
```



D only starts when C and E are completed!

Nested Parallelism

- Nested parallelism enables the programmer to create parallelism within a parallel execution
  - For example, it can combine task parallelism and data parallelism
  - Not every multithreading library allows this
- OpenMP has support for nested parallelism:
  - Create a parallel region inside of a parallel region
  - Or create a data parallel region (with a for-loop) inside of a task.

Nested Parallel Regions

```
int main() {


   #pragma omp parallel num_threads(2) {

         #pragma omp parallel num_threads(2)
         {
               #pragma omp parallel num_threads(2)
               {

               }
         }
   }
   return(0);
}
```

How many threads are there?

Combining Task and Data Parallelism

```
#pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task
      {

      }
      #pragma omp task
      {
        #pragma omp parallel num_threads(4)
        {
        }
      }
    }
  }
  return 0;
}
```

In a parallel region, only one thread enters into single which creates two tasks, one of them creates another parallel region with 4 threads.

Nested Parallelism Example

- Parallel pointer chasing on multiple lists using tasks

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

All spawn tasks in the for-loop are guaranteed to be finished here!
Implicit barrier at the end of the omp for

Acknowledgments

- These slides are inspired and partly adapted from
  - Mary Hall (Univ. of Utah)
  - The course book (Pacheco)
  - Vivek Sarkar (Rice Univ.)
  - Ruud van der Pas (Oracle)
  - https://computing.llnl.gov/tutorials/openMP/