Lecture 8:
Task Parallelism

Beakal Gizachew Assefa

Today

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

OpenMP Summary

- Work sharing
  - parallel, parallel for
  - scheduling directives: static(), dynamic(), guided()
- Data sharing and variable scopes
  - shared, private, reduction
- Environment variables
  - **OMP_NUM_THREADS**
- Library
  - E.g., omp_get_num_threads(), omp_get_thread_num()

Data Dependence

- A ***data dependence*** is an ordering on a pair of memory operations that must be preserved to maintain correctness.

- **Question:** When is parallelization guaranteed to be safe?
- **Answer:** If there are no data dependences across reordered computations.

- **Definition: Two memory accesses** are involved in a data dependence if they may refer to the same memory location and one of the accesses is a **write.**

Preserve Dependences

- Fundamental Theorem of Dependence:
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
- Parallelization
  - Computations that execute in parallel between synchronization points are potentially reordered.
  - Is that reordering safe? According to our definition, it is safe if it preserves the dependences in the code.

```
#pragma omp barrier

. . .
#pragma omp for nowait
for()
. . .
#pragma omp barrier
```

It’s programmer’s responsibility to ensure there is no violation of data dependences.

Parallel Regions

```
#pragma omp parallel
{

}
#pragma omp parallel for
for (i=0; i<size; i++) {

}
#pragma omp parallel {



}
```

```
#pragma omp parallel
{



}
```

- Avoid creating too many parallel regions if you can
  - Each time, threads are created and joined!!!
- Avoid also expanding a parallel region into multiple files
  - Find a compromise

Parallel Loops

- Guarantee that the same thread will compute the same code range if the same scheduling is used in the two loops

```
#pragma omp parallel
{
    #pragma omp for schedule(static) nowait
    for()
     a(i)=

    #pragma omp for schedule(static)
    for()
     . . = a(i)
}
```

I don’t recommend to use it because it can introduce a bug if you decide to change the scheduling type
Fuse loops if possible.

Can recommend because it can only

introduce a performance bug even if the loop schedulings are different
First loop performs the first-touch.

```
#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = 1.0;


. . .
#pragma omp parallel for
for (i=0; i<N; i++)
    //calculation on A
}
```

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)
{
   #pragma omp for
   for (i=0; i<n; i++)
      x[i] += y[i];

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

Single and Master Constructs

- Only one thread in team executes code enclosed
- Useful for things like I/O or initialization
- **Implicit barrier on exit**

```
#pragma omp single {
    <code-block>
}
```

- Similarly, only master executes code
- **No implicit barrier on exit**

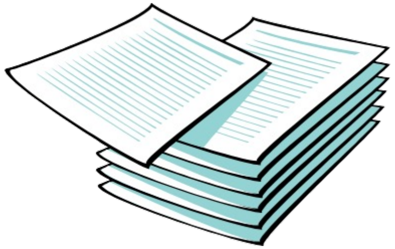
```
#pragma omp master {
    <code-block>
}
```

Types of Parallelism

- Task parallelism
  - Partition various distinct tasks carried out solving the problem in parallel.
- 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

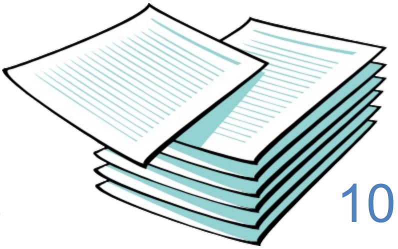
- Assume there are 30 students
  - Thus 30 exams and
  - 3 questions in the exam



Grading

- Ruth, Meba and Ikbal are grading the exams

10 exams



10 exams

10 exams

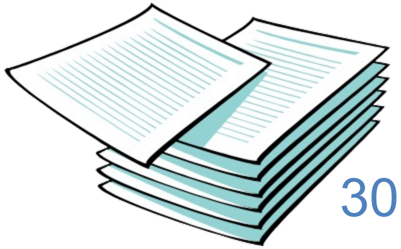
- Scenario #1: Each of us gets 10 students to grade

Grading

- Ruth, Meba and Ikbal are grading the exams

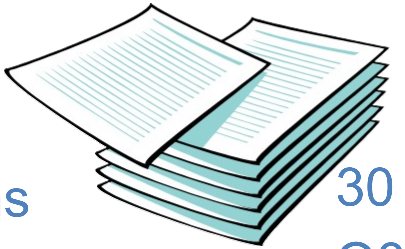
30 exams

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
- Scenario #2: Task parallelism
  - Tasks (questions) are divided, each TA gets a different question

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.

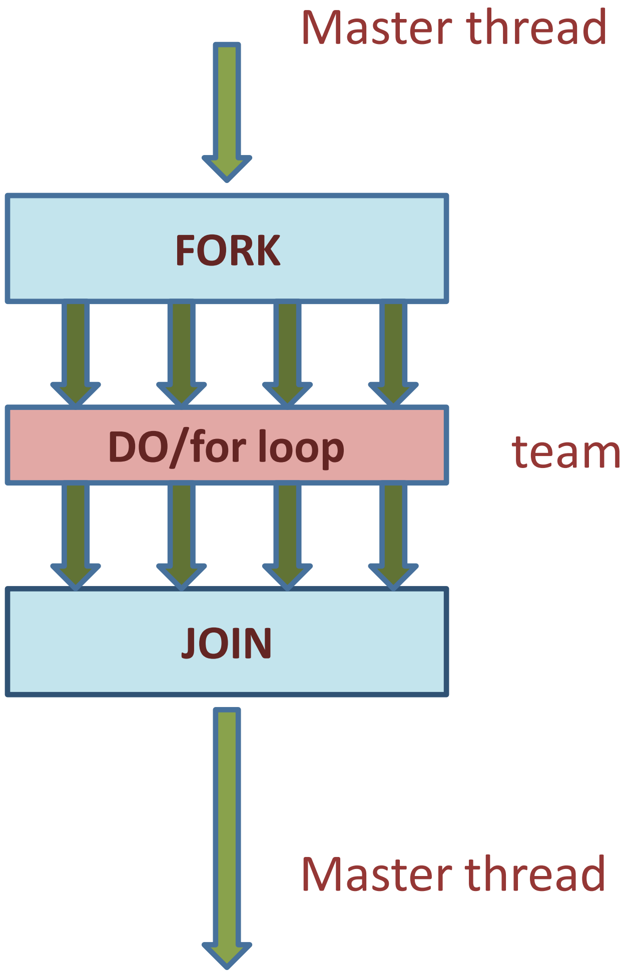
- (b) Identify *task parallelism* in home building by defining a set of tasks. Work out a schedule that shows when the various tasks can be performed.

- (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?

Task Parallelism

- OpenMP supports for task parallelism
  - Parallel sections: different threads execute different code
    - OpenMP Pre–3.0
  - Tasks (NEW): tasks are created and executed at separate times
    - OpenMP 3.0

OpenMP Sections



Master thread

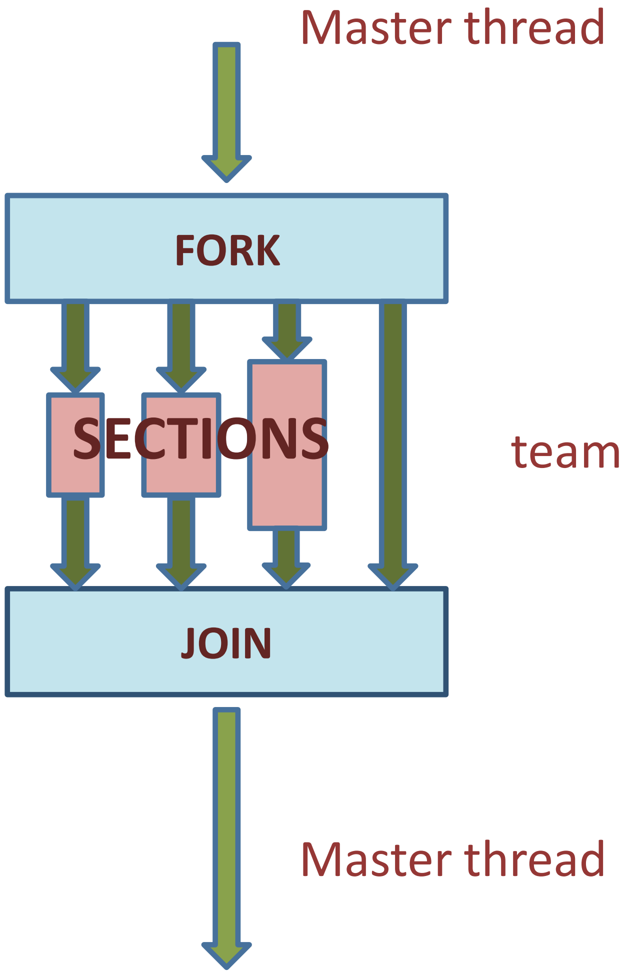
FORK

DO/for loop

team

JOIN

Master thread



Master thread

FORK

SECTIONS

team

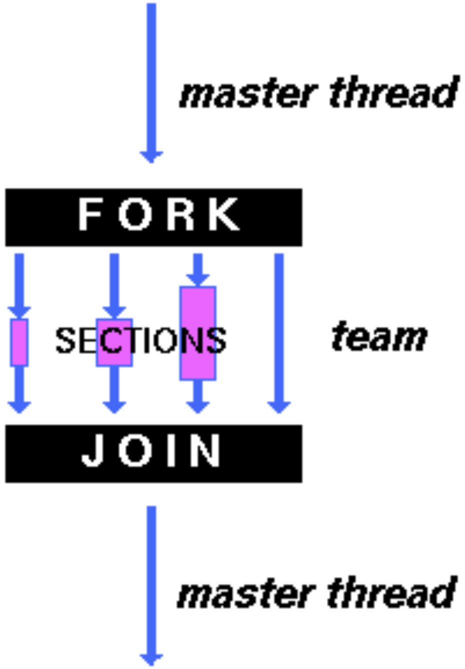
JOIN

Master thread

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])
    } /*-- 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 task has
  - Code to execute
  - A data environment (shared, private, reduction)
  - An assigned thread that executes the code and uses the data
- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task
  - Some thread in the team executes the thread at a later time

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

When/Where are tasks complete?

- At thread 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

Example

- Parallel pointer chasing using tasks

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

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

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 = next (p)
       }
   }
}
```

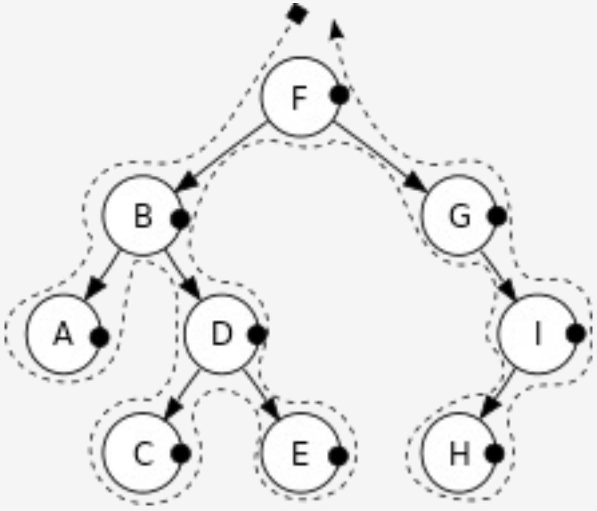
Example

- Post-order tree traversal
- Parent task 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-> data);
}
```



Final Remarks on Tasks

- Integrated into OpenMP 3.0
- Flexible model for irregular parallelism
  - Graph traversal
- Nested parallelism where tasks creates new tasks

```
#pragma omp parallel
{
   if(tid == 0)
        //do this
   else
        //do that
   #pragma omp barrier
}
```

If you have a work sharing construct inside else, then tid=0 will not reach that construct! Result is incorrect

```
else {
#pragma omp for
 for … {
   …
 }
}
```

Acknowledgments

- These slides are inspired and partly adapted from
  - Mary Hall (Univ. of Utah)
  - The course book (Pacheco)
  - https://computing.llnl.gov/tutorials/openMP/