Synchronization Constructs

Bringing order to threads

So far, we've had little to no control of what order threads executed their tasks

- ~~Barriers~~
- ~~Critical Sections~~
  - ~~critical pragma~~
  - ~~atomic pragma~~
- Locks

But first… Understanding Tasks

-or- Who does what and when.

Tasks are a mechanism that OpenMP uses under the cover: if you specify something as being parallel, OpenMP will create a ‘block of work’: a section of code plus the data environment in which it occurred. This block is set aside for execution at some later point.

Tasks

Tasks are a mechanism that OpenMP uses under the cover: if you specify something as being parallel, OpenMP will create a ‘block of work’: a section of code plus the data environment in which it occurred. This block is set aside for execution at some later point.

Tasks

- Use a pragma to specify where the tasks are
- When a thread encounters a task construct, a new task is generated
- The moment of execution of a task is up to the scheduler
- Execution can either be immediate or delayed
- Completion of a task can be enforced through task synchronization

Tasks

Look at the following code snippet:

Code

```
x = f();

#pragma omp task
{
   y = g(x);
}

z = h();
```

Execution

```
x gets a value

task gets created
with the current
value of x


z gets a value
```

Tasks

Look at the following code snippet:

Code

```
x = f();

#pragma omp task
{
   y = g(x);
}

z = h();
```

Execution

```
x gets a value

task gets created
with the current
value of x


z gets a value
```

- The thread that executes this code creates the task

Tasks

Look at the following code snippet:

Code

```
x = f();

#pragma omp task
{
   y = g(x);
}

z = h();
```

Execution

```
x gets a value

task gets created
with the current
value of x


z gets a value
```

- The thread that executes this code creates the task
- This task will *probably* executed by a different thread

Tasks

Look at the following code snippet:

Code

```
x = f();

#pragma omp task
{
   y = g(x);
}

z = h();
```

Execution

```
x gets a value

task gets created
with the current
value of x


z gets a value
```

- The thread that executes this code creates the task
- This task will *probably* executed by a different thread
- Even though the snippet looks linear, it is *impossible* to say exactly when the task will be executed

Tasks

Look at the following code snippet:

Code

```
x = f();

#pragma omp task
{
   y = g(x);
}

z = h();
```

Execution

```
//x gets a value

//task gets created
//with the current
//value of x



//z gets a value
```

- The thread that executes this code creates the task
- This task will *probably* executed by a different thread
- Even though the snippet looks linear, it is *impossible* to say exactly when the task will be executed

Tasks

In contrast, the following does *not* do what you think it should

Code

```
x = f();

#pragma omp task
{
   y = g(x);
}

z = h(y);
```

Execution

```
//x gets a value

//task gets created with the current
//value of x



//z gets a value with the "current"
//value of y
```

when the statement computing z is executed, the task computing y has only been scheduled; it has not *necessarily* been executed yet.

Tasks

add a `taskwait` directive

Code

```
x = f();

#pragma omp task
{
   y = g(x);
}
#pragma omp taskwait
z = h(y);
```

Execution

```
//x gets a value

//task gets created with the current
//value of x



//wait for all tasks to execute
//z gets a value with the current
//value of y
```

NOTE: taskwait is a standalone directive; the code that follows is just code, it is not a structured block belonging to the directive.

Tasks

add a `taskwait` directive

Code

```
x = f();

#pragma omp task
{ y1 = g1(x);}

#pragma omp task
{ y2 = g2(x);}

#pragma omp taskwait
z = h(y1) + h(y2);
```

Execution

```
//x gets a value

//task gets created with the current
//value of x

//task gets created with the current
//value of y

//wait for all tasks to execute
//z gets a value with the current
//values of y1 and y2
```

Tasks

a silly example.

```
printf("this class ");
printf("rocks ");
printf("OpenMP ");
printf("\n");
```

Tasks

a silly example. Write some code that prints out 1, 2, 3

```
#pragma omp parallel
{
   printf("this class ");
   printf("rocks ");
   printf("with OpenMP ");
}
printf("\n");
```

Tasks

a silly example. Write some code that prints out 1, 2, 3

```
#pragma omp parallel
{
   #pragma omp single
   {
      printf("this class ");
      printf("rocks ");
      printf("with OpenMP ");
   }
}
printf("\n");
```

Tasks

a silly example. Write some code that prints out 1, 2, 3

```
#pragma omp parallel
{
   #pragma omp single
   {
      printf("this class ");
      #pragma omp task
      { printf("rocks "); }
      #pragma omp task
      { printf("with OpenMP "); }
   }
}
printf("\n");
```

Tasks

a silly example. … now add a finishing touch

```
#pragma omp parallel
{
   #pragma omp single
   {
      printf("this class ");
      #pragma omp task
      { printf("rocks "); }
      #pragma omp task
      { printf("with OpenMP "); }
      printf("#micdrop");
   }
}
printf("\n");
```

Tasks

a silly example. … now add a finishing touch

```
#pragma omp parallel
{
   #pragma omp single
   {
      printf("this class ");
      #pragma omp task
      { printf("rocks "); }
      #pragma omp task
      { printf("with OpenMP "); }
      #pragma omp taskwait
      printf("#micdrop");
   }
}
printf("\n");
```

Tasks

Look at the following:

```
#pragma omp parallel
#pragma omp single
{
     // code that generates tasks
}
```

Tasks

Now, we can do while loops in C

```
#pragma omp parallel
#pragma omp single
{
      while (true)
      {
         #pragma omp task
            //do something
      }
      #pragma omp taskwait
}
```

Tasks

A couple more things of note.

- Using the ‘taskwait’ directive you can explicitly indicate the join of the forked tasks. The instruction after the wait directive will therefore be dependent on the spawned tasks.

- The `taskgroup` directive, followed by a structured block, ensures completion of all tasks created in the block, even if recursively created.

- Each OpenMP task can have a `depend` clause, indicating what data dependency of the task. By indicating what data is produced or absorbed by the tasks, the scheduler can construct the dependency graph for you.

Tasks

A couple more things of note.

- Using the ‘taskwait’ directive you can explicitly indicate the join of the forked tasks. The instruction after the wait directive will therefore be dependent on the spawned tasks.

- The `taskgroup` directive, followed by a structured block, ensures completion of all tasks created in the block, even if recursively created.

- Each OpenMP task can have a `depend` clause, indicating what data dependency of the task. By indicating what data is produced or absorbed by the tasks, the scheduler can construct the dependency graph for you.

Tasks

`child tasks` `and` `taskgroup` usage.

```
// Thread N reaches here
#pragma omp taskgroup {
   #pragma omp task {
      //task1
   }
   #pragma omp task {
      //task2
      #pragma omp task {
         //task2 child task
      }
   }
}
// Thread N waits here until task1, task2
// and all the child tasks created inside them complete.
```

Tasks

`depend` clause usage

The dependence-type can be one or more of the following, the list items can be variables and array sections:

- `in(list)`: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an out or inout clause.

- `out(list)`: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an in, out, or inout clause.

- `inout(list)`: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an in, out, or inout clause.

Tasks

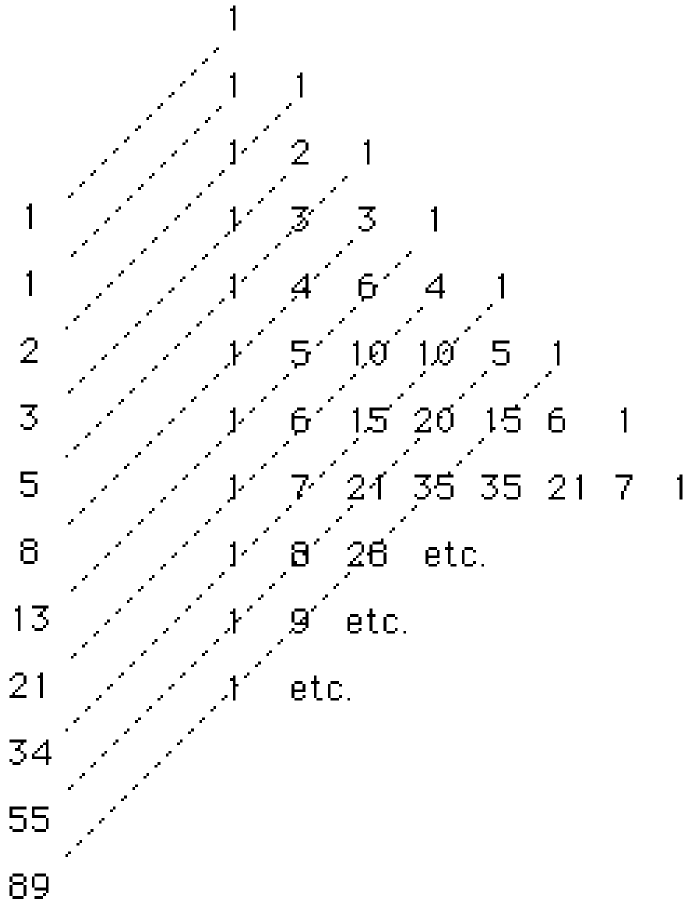
depend clause usage.

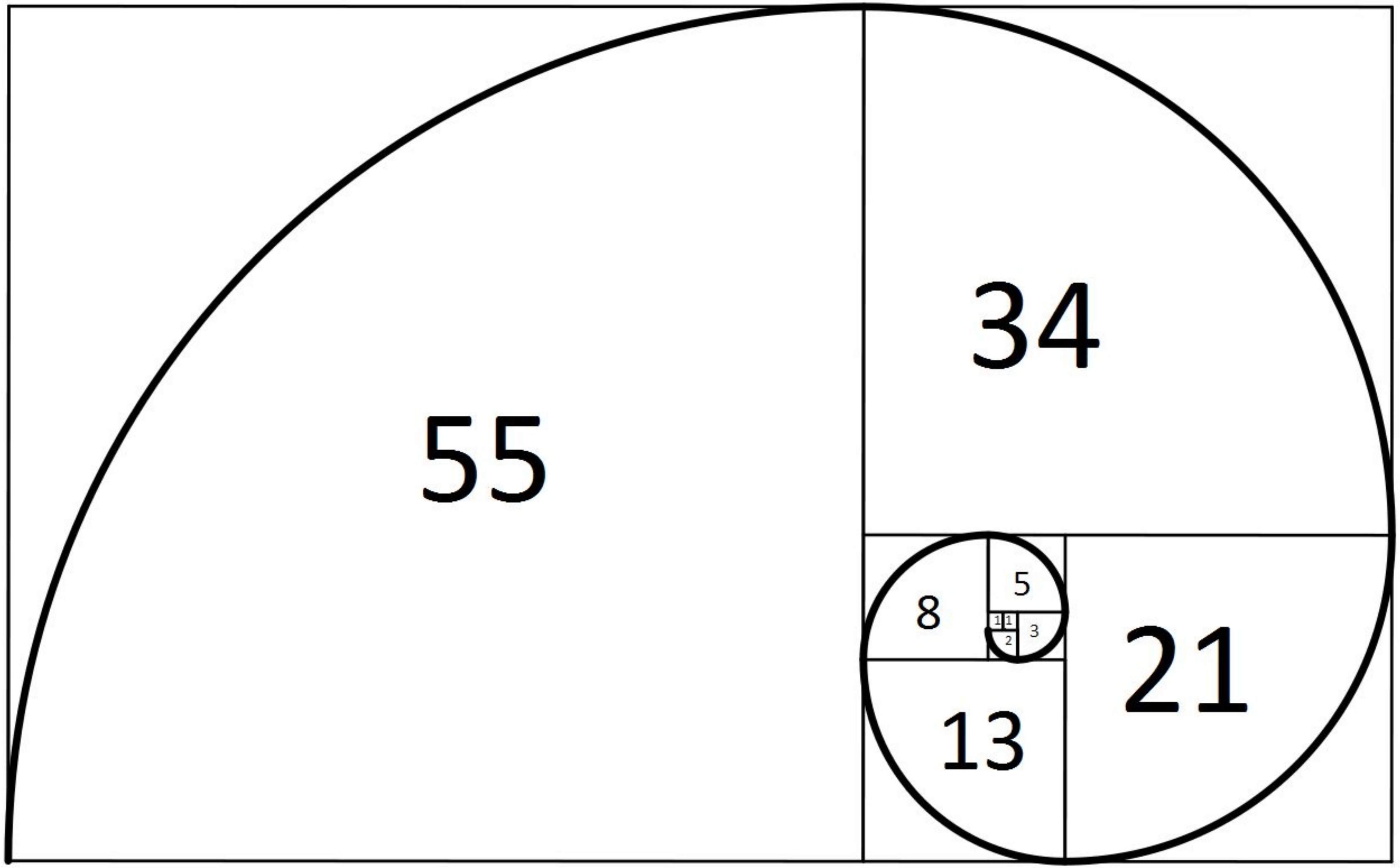
The first task does not depend on any previous one. The second and third tasks depend on the first one, but not on each other. The last task depends on the second and third.

```
#pragma omp task depend(out:a) {}
#pragma omp task depend(in:a) {}
#pragma omp task depend(in:a) {}
#pragma omp task depend(out:a) {}
```

The first task does not depend on any previous one.
The second and third tasks depend on the first one, but not on each other.
The last task depends on the second and third.

Diagonal Sums




55
34
21
13
8
5
3
2
1
1

Exercise 6, In Class Lab, Fibonacci Sequence

The Fibonacci sequence is recursively defined as

$$F(0) = 1,\ F(1) = 1,\ F(n) = F(n - 1) + F(n - 2) \text{ for } n \geq 2.$$

Using tasks, generate the fibonacci sequence.

Hints:

- Use one task for F(n - 1) and one task for F(n - 2)
- You may wish to set a threshold for n, only calling the separate tasks with n is greater that say, 20.

For Thursday, even though this is in parallel, this isn't optimal, be prepared to discuss what to do with any "redundancy" we see here.