Fibonacci Sequence

```
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
  {

       i=fib(n-1);


       j=fib(n-2);


       return i+j;
  }
}
```

```
int main()
{
  int n = 10;









    printf ("fib(%d) = %d\n", n, fib(n));

}
```

Fibonacci Sequence w/ Parallel Tasks

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

#pragma omp task shared(j) firstprivate(n)
       j=fib(n-2);

       #pragma omp taskwait
       return i+j;
  }
}
```

```
int main()
{
  int n = 10;

  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

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

OpenMP is an explicit programming model, offering the programmer full control over parallelization.

Parallelization can be as simple as taking a serial program and inserting compiler directives....

Or as complex as inserting subroutines to set multiple levels of parallelism, **locks and even nested locks**.

Locks - Defined

A **lock** is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. A lock is designed to enforce a mutual exclusion concurrency control policy; components that operate concurrently interact by messaging or by sharing accessed data, a certain component's consistency may be violated by another component.

Locks - In a Nutshell

In a nutshell, when running code in parallel, there's a lot of moving parts. Sometimes, our given OpenMP directives are not enough to ensure that certain data elements are protected from *other* parts.

Locks

- Locks and critical sections both give exclusive execution

- Subtle difference:
  - critical limits access to section of code
  - lock limits access to item of data

"a critical section is about code; a lock is about data. With a lock you make sure that some data elements can only be touched by one process at a time."

Example: writing to database

Locks

Create/destroy:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

`omp_lock_t` is a datatype that holds the status of a lock, whether the lock is available or if it's owned.

Locks

void omp_set_lock(omp_lock_t *lock);

forces the calling task region to wait until the specified lock is available before executing subsequent instructions. The calling task region is given ownership of the lock when it becomes available.

NOTE: If you call this routine with an uninitialized lock variable, the result of the call is undefined. If a task region that owns a lock tries to lock it again by issuing a call to omp_set_lock, the call produces a deadlock.

Locks

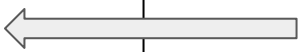
```
omp_lock_t mylock;

omp_init_lock(&mylock);

#pragma omp parallel for
for ( i = 0; i < x; i++ )
{
    // some stuff
   omp_set_lock(&mylock);
    // one task at a time stuff
    omp_unset_lock(&mylock);
    // some stuff
}

omp_destroy_lock(&mylock);
```

Wait here for your turn!



Release the lock so the next thread gets his turn

Locks

```
#include <stdio.h>
#include <omp.h>

omp_lock_t my_lock;

int main() {
   omp_init_lock(&my_lock);

   #pragma omp parallel num_threads(4)
   {
      int tid = omp_get_thread_num( );
      int i, j;
```

```
      for (i = 0; i < 5; ++i) {
         omp_set_lock(&my_lock);
         printf_s("Thread %d - starting locked
             region\n", tid);
         printf_s("Thread %d - ending locked
            region\n", tid);
         omp_unset_lock(&my_lock);
      }
   }

   omp_destroy_lock(&my_lock);
}
```

Fibonacci Sequence w/ Parallel Tasks

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

#pragma omp task shared(j) firstprivate(n)
       j=fib(n-2);

       #pragma omp taskwait
       return i+j;
  }
}
```

What can we do to prevent recomputation?

Fibonacci Sequence w/ Parallel Tasks

```
int fib(int n)
{
  int i, j;
  if (!done[n])
  {
    if (n<2)
      return n;
    else
    {
#pragma omp task shared(i) firstprivate(n)
         i=fib(n-1);

#pragma omp task shared(j) firstprivate(n)
         j=fib(n-2);

#pragma omp taskwait
         fibSeq[n] = i+j;
         done[n] = 1
    }
  }
  return value[n];
}
```

Would introducing a `done[]` and `fibSeq[]` be enough?

Fibonacci Sequence w/ Parallel Tasks

```
int fib(int n)
{
  int i, j;
  if (!done[n])
  {
    if (n<2)
      return n;
    else
    {
#pragma omp task shared(i) firstprivate(n)
         i=fib(n-1);

#pragma omp task shared(j) firstprivate(n)
         j=fib(n-2);

#pragma omp taskwait
         fibSeq[n] = i+j;
         done[n] = 1
    }
  }
  return value[n];
}
```

Would introducing a `done[]` and `fibSeq[]` be enough?

Nope. **`Data Race`**.

Fibonacci Sequence w/ Parallel Tasks

```
int fib(int n)
{
  int i, j;
  if (!done[n])
  {
    if (n<2)
      return n;
    else
    {
#pragma omp task shared(i) firstprivate(n)
         i=fib(n-1);

#pragma omp task shared(j) firstprivate(n)
         j=fib(n-2);

#pragma omp taskwait
         fibSeq[n] = i+j;
         done[n] = 1
    }
  }
  return value[n];
}
```

Would introducing a `done[]` and `fibSeq[]` be enough?

Nope. **`Data Race`**.

We need to add some locks.

Try:

add a lock for each task that sets `fibSeq[]` and `done[]`.

Final OpenMP Homework.

```
1111111010
0101110101
0111000111
1001111100
1101111110
1100110010
1111111111
1101111110
1111111011
0011101111
```

Traverse a Graph from the top left to the bottom right.

You can only move up, down, left, or right.

A 1 signifies the path is open.

A 0 signifies the path is closed.

Write a program that will read a 10x10 (for testing) and a 100x100 (final) matrix, *a*, from a file (this is in the repo)

Traverse **all** possible routes from $a_{1\ 1}$ to $a_{n\ n}$

Find the shortest route.

Final OpenMP Homework.

For example if ***n***=5, the matrix ***a*** may be:

```
11111
11011
11101
01111
11001
```

and the shortest path would be:

```
a[0,0]  to  a[0,1]  to  a[1,1]  to  a[2,1]  to  a[2,2]  to  a[3,2]  to  a[3,3]  to
a[3,4]  to  a[4,4]
```