NTNU – Trondheim
Norwegian University of
Science and Technology

Pthread operations, synchronization

Jan.Christian.Meyer@ntnu.no

Today's topic

- Last time, we looked at starting and stopping pthreads
- I have said that they can only really do 3 more things
  - lock/unlock
  - wait for a signal
  - wait at a barrier
- This time, we'll cover those operations

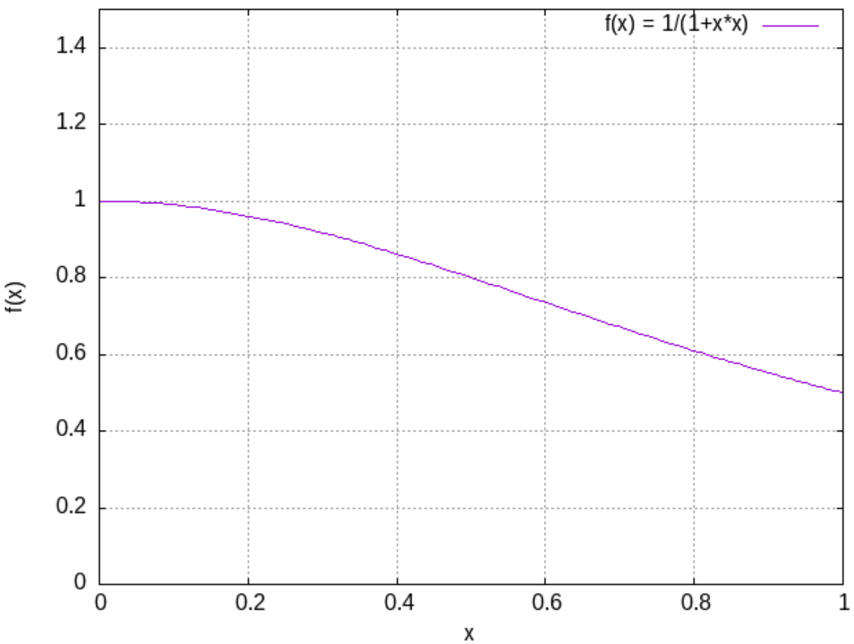
We need a computation

- These operations all have to do with synchronization
  - All communication is implicit with threads, so we just have to organize who gets to work where and when
- A simple example is just to require some shared value
  - A global sum, for instance
- We can recycle the example problem we used with reductions
  - Estimate the value of Pi by adding up a lot of rectangle areas

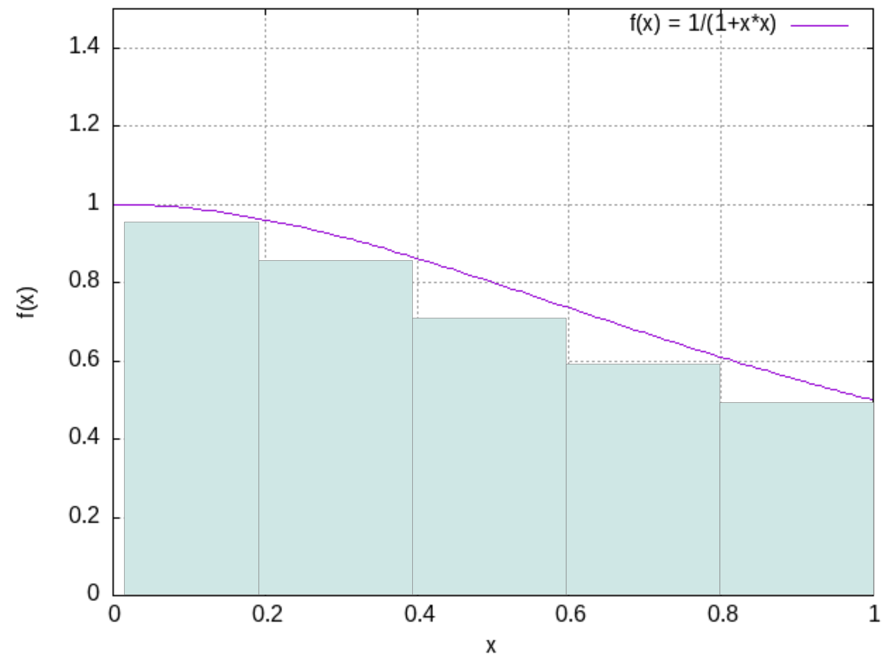
Quick recap

- In case you forgot, here's the problem again:

Area under the curve is pi/4



Approximate it with rectangles



Example code directory

- The example code archive contains a directory ‘02_pi_estimate’
- There are 8 different versions of the program inside, numbered in the sequence we’ll go through them
- Some of them don’t actually work, that’s intentional
  - We’ll go through why in this lecture

01_pi_seq

- This is our sequential baseline
- Its kernel fits on a slide:

```
#define STEPS (1e8)
#define H (1.0/STEPS)

int
main ( )
{
    double pi = 0.0, x = 0.0;
    for ( size_t i=0; i<STEPS; i++ )
    {
        x += H;
        pi += H / (1.0 + x*x);
    }
    pi *= 4.0;
    printf ( "Estimated %e, missed by %e\n", pi, fabs(pi-M_PI) );
    exit ( EXIT_SUCCESS );
}
```

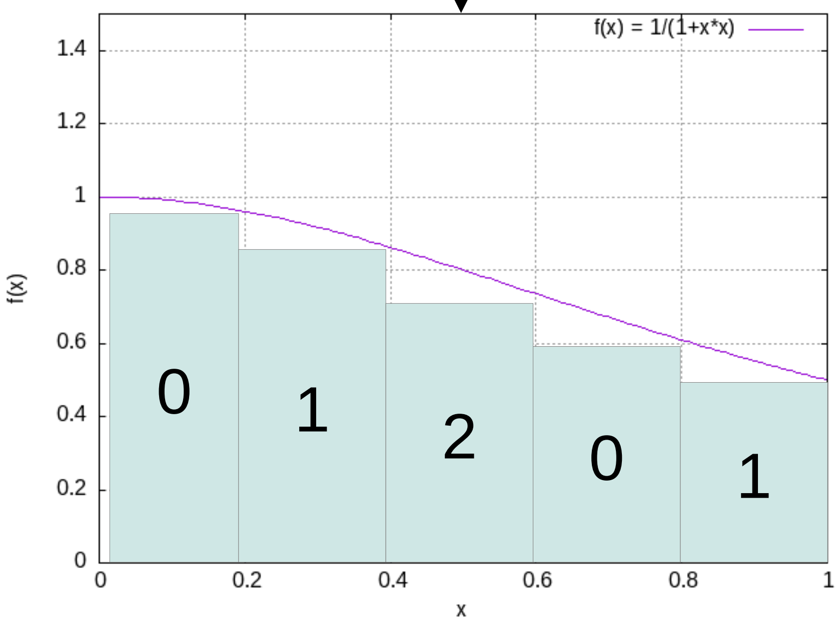
NTNU – Trondheim
Norwegian University of
Science and Technology

Parallelizing it badly

(02_pi_nolock.c)

- Make pi global, everyone contributes to it,
- hand out rectangles round-robin (*e.g.* for 3 threads), and
- get Wrong Answer™ because everyone tries to update pi willy-nilly (a real-life race condition)

```
void *
integrate ( void *in )
{
    int64_t tid = (int64_t)in;
    double x = tid*H;
    for ( size_t i=0; i<STEPS; i+=n_threads )
    {
        x += n_threads*H;
        pi += H / (1.0 + x*x);
    }
    return NULL;
}
```



<u>What are we **doing**?!?</u>

- We're writing to a shared value in every iteration of a tight loop
- Performance-wise, this is an <u>unconditionally bad idea</u>

  (and not just because it gets a wrong answer)
- Much better would be to add up a thread-local sum and combine them all at the end

HOWEVER

- That would only *show* the race condition once in a blue moon *(try it at home)*
- <u>*It would still be there*</u>
  - Beware, Wrong Programs can give Right Answers
  - We're justifying the need for mutual exclusion
  - I promise to fix the program afterwards

Do-It-Yourself mutual exclusion

(03_pi_lock.c)

```
for ( size_t i=0; i<STEPS; i+=n_threads )
{
    x += n_threads*H;
    while ( flag != tid );
    pi += H / (1.0 + x*x);
    flag = (flag + 1) % n_threads;
}
```

- We can add a shared integer ('flag') which says whose turn it is to update the shared value
- Each thread *busy-waits* for its turn (eagerly doing nothing useful)
- Pass the turn round-robin (0,1,2,0,1,2,0,1,2…)

- We have effective serialized *this* program (and added contention for the flag variable), but this scheme kind of works…

  ...and it would get better with more work *vs.* a smaller critical section

...but it only <u>kind of</u> works

- The effect of the waiting loop (it's called a "spin-lock") depends very strong on a strict order of program statements
- Notice that the Makefile goes out of its way to build 03_pi_diy_lock without any optimization flags
- Compiler optimizations can take liberties with instructions that don't produce visible results
- Make 03_pi_diy_deadlock to see what might happen with exactly the same source code + optimizations

  (...or maybe you can guess it from the name)

Aside:
The compiler doesn't know about threads

- We create and join them using function calls to a system library
  - The source code doesn't explicitly say that these calls multiply the control flows
  - We could technically replace them with implementations that didn't
  - It's an invisible side effect, like I/O functions have
- The `volatile` keyword is <u>not a memory fence</u>
  - I only point this out because many people mistake it for one
  - Declaring a variable as volatile means the compiler is forbidden from moving read and write instructions that access it around in the code
  - If two threads simultaneously access a volatile variable, we still get a race condition

Proper spin-locking

- In plain C, we must account for the fact that memory updates aren’t strictly ordered
- In order to do that *efficiently*, we must abandon C and reach into computer architecture, to look for *atomic operations*
  - Special instructions that have been wired into the CPU and interconnection fabric so that they are impossible to interrupt
- Let’s not do that here, it’s a whole separate lecture

Pthreads to the rescue!

(04_pi_mutex.c)

- Add a shared variable `pthread_mutex_t lock;`
- Initialize it with `pthread_mutex_init ( &lock, NULL );`
- Destroy it with `pthread_mutex_destroy ( &lock );`
- Now we can do this:

```
for ( size_t i=0; i<STEPS; i+=n_threads )
  {
      x += n_threads*H;
      pthread_mutex_lock ( &lock );
      pi += H / (1.0 + x*x);
      pthread_mutex_unlock ( &lock );
  }
```

Also better because mutex doesn't spin while the lock is held.
Try 03_pi_diy_lock with n_threads>cores if you want,
But reduce STEPS and prepare to wait a while...

Finally, as promised

(05_pi_mutex_fast.c)

```
for ( size_t i=0; i<STEPS; i+=n_threads )
 {
     x += n_threads*H;
     pi_local += H / (1.0 + x*x);
 }
 pthread_mutex_lock ( &lock );
 pi += pi_local;
 pthread_mutex_unlock ( &lock );
```

- Make local partial sums and add total at the end
- Doing most of the work on thread-local values actually obtains a speedup
- We have also shown that the lock isn't just for decoration

Synchronized iterations

- Many, many scientific parallel applications work in data-parallel steps separated by synchronization
  - Like our advection solver
  - In 1996, this pattern accounted for an estimated 90% of parallel computations altogether*
  - Such estimates are harder to make now that everyone has a parallel computer, the numbers have surely changed since
  - The point is that this is something lots and lost of parallel programs do
- Using our example problem, we can mimic this behavior by running the computation many times over
  - No thread must start the next pi-estimate before the previous one is complete
  - Resetting pi to 0 happens at the synchronization point

** G. C. Fox: An application perspective on high-performance computing and communications, (1996)*

Condition variables

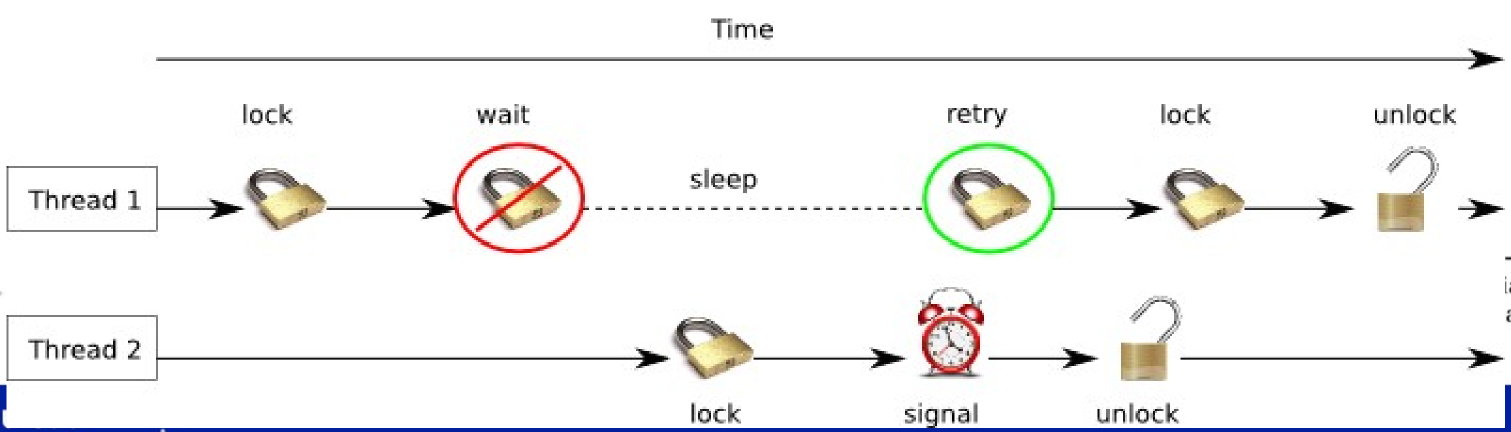
- `pthread_cond_t` is a type of variable that attaches a simple sleep/wake signaling mechanism to a mutex

  Create and destroy with

  pthread_cond_init (&var, NULL);

  pthread_cond_destroy (&var);

- Its semantics are a little counterintuitive, but manageable
  - Use of its *wait* and *signal* operations can be illustrated by this sequencing diagram:



DIY barrier using signals

(06_pi_cond_signal.c)

- The 1st through (n_threads-1)th arriving thread will:
  - Lock and add local partial sum
  - Increment global count of waiting threads
  - Sleep, waiting for condition variable
  - …
  - Wake and regain the lock
  - Decrement global count of waiting threads
  - Signal another sleeping thread
  - Release lock
- The last arriving thread recognizes that the barrier is complete, and skips the sleeping step
- The last departing thread skips the signaling step

In code

```
void
signal_barrier ( pthread_mutex_t *lock, pthread_cond_t *cond, int64_t *count )
{
    pthread_mutex_lock ( lock );
    (*count)++;
    if ( (*count) < n_threads )
        while ( pthread_cond_wait ( cond, lock ) != 0 );
    (*count)--;
    if ( (*count) > 0 )
        pthread_cond_signal ( cond );
    pthread_mutex_unlock ( lock );
    return;
}
```

Last arrival skips this

Last departure skips this

- This is a function because we need to do it twice:
  - Once to make sure the global sum is complete
  - Once to make sure nobody adds to the global sum before it is reset
- Hence, there are
  - 3 locks (for ‘pi’, ‘arrive’ and ‘depart’)
  - 2 conds (for ‘arrive’ and ‘depart’)
  - 2 countrs (also for ‘arrive’ and ‘depart’)

DIY barrier with broadcast

(07_pi_cond_broadcast.c)

- `pthread_cond_signal` wakes one waiting thread
- `pthread_cond_broadcast` wakes all waiting threads in turn
- We can use this to simplify our synchronization:
  - 1st through (n_threads-1)th arriving threads
    - Lock
    - Add local part to global sum
    - Increment arrival count
    - Sleep
    - Wake, and unlock
  - Last arriving thread
    - Prints global sum
    - Resets arrivals and global sum
    - Wakes everyone else up
    - Unlocks

In code

```
pthread_mutex_lock ( &lock );
  pi += pi_local;
  arrived++;
  if ( arrived < n_threads )
      while ( pthread_cond_wait ( &cond, &lock ) != 0 );
  else
  {
      arrived = 0;
      pi *= 4.0;
      printf ( "Estimated %e, missed by %e (thread %ld)\n", pi, fabs(pi-M_PI), tid );
      pi = 0.0;
      pthread_cond_broadcast ( &cond );
  }
  pthread_mutex_unlock ( &lock );
```

- Only 1 lock and cond pair is necessary
- We've delegated the "master only" work to the last arriving thread, thus removing the need for a 2nd barrier
  - That's OK because the rest are sleeping at the time

Barrier using… a barrier
(08_pi_barrier.c)

- `pthread_barrier_t` is an object that behaves like our broadcast barrier, initialize and destroy with

```
pthread_barrier_init ( &var, NULL, count );
pthread_barrier_destroy ( &var );
```

- `pthread_barrier_wait ( &var );`
  - Suspends threads until #count of them have called it,
  - Resets var and resumes all threads
- This is an optional feature of pthreads, so the program contains

  #define _GNU_SOURCE

  before

  #include <pthread.h>

  in order to enable it.
- We can't put the master computation into it, so it's called twice for the same reason as our home-made signal based barrier

NTNU – Trondheim
Norwegian University of Science and Technology

Summary

- We have looked at
  - Where pthreads come from
  - Creating and joining threads
  - Race conditions and the trouble with manual locking
  - Mutex variables
  - Condition variables
  - Barriers
- We haven't looked at semaphores
  - Like barriers, semaphores are not a mandatory feature of pthreads implementations
  - Chapter 4.7 in the book is a high-level overview, it's more relevant to concurrent programs than our parallel number-crunching applications
  - You can read about semaphores, we won't spend a lecture on them
- What remains is to say something about how cache memory acts when we write in it