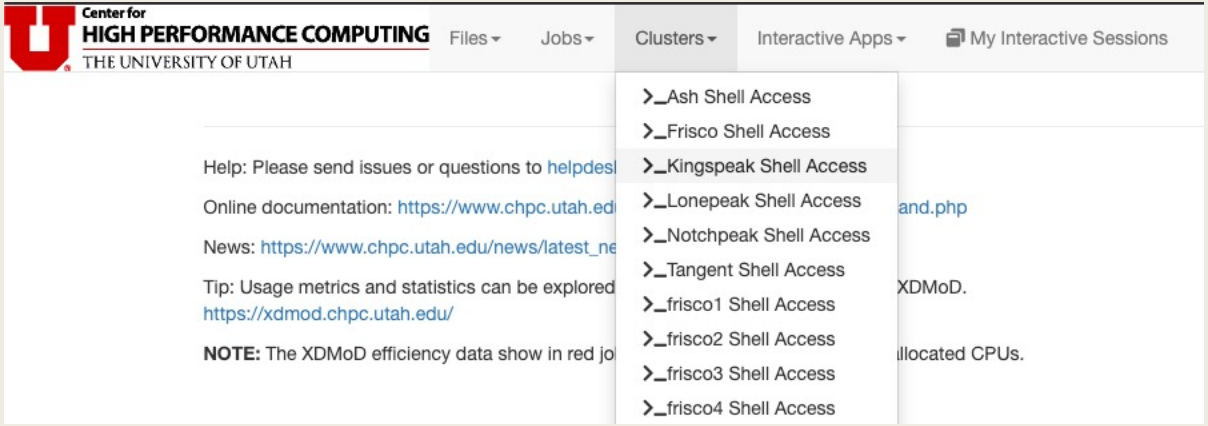
# Multithreaded programming

# Preliminary test for CHPC access

- Try to login to ondemand CHPC: https://ondemand.chpc.utah.edu/
- Try to start a terminal on one of the clusters (“Shell Access”)



# Setting up a Makefile to build your program

```
#target1 [target2 ...]: [pre-req-1 pre-req-2 ...]
#       [command1
#        command2
#        ......]
all: hello
hello: hello.o
        gcc hello.o -o hello -lpthread
hello.o: pth_hello.c
        gcc -c pth_hello.c -o hello.o
```

# Run Make

- Linux/MacOS
  - *make*
- Windows
  - *mingw32-make*

# CRITICAL SECTIONS

# Estimating π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

# How can we parallelize this code using multiple threads?

# A thread function for computing π

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

   return NULL;
}  /* Thread_sum */
```

# Using a dual core processor

| | $n$ | | | |
|---|---|---|---|---|
| | $10^5$ | $10^6$ | $10^7$ | $10^8$ |
| $\pi$ | 3.14159 | 3.141593 | 3.1415927 | 3.14159265 |
| 1 Thread | 3.14158 | 3.141592 | 3.1415926 | 3.14159264 |
| 2 Threads | 3.14158 | 3.141480 | 3.1413692 | 3.14164686 |

Note that as we increase n, the estimate with one thread gets better and better.

# Possible race condition

```
y = Compute(my_rank);

x = x + y;
```

| Time | Thread 0 | Thread 1 |
|---|---|---|
| 1 | Started by main thread | |
| 2 | Call `Compute()` | Started by main thread |
| 3 | Assign `y = 1` | Call `Compute()` |
| 4 | Put `x=0` and `y=1` into registers | Assign `y = 2` |
| 5 | Add 0 and 1 | Put `x=0` and `y=2` into registers |
| 6 | Store 1 in memory location `x` | Add 0 and 2 |
| 7 | | Store 2 in memory location `x` |

# Busy-Waiting

- A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.

- Beware of optimizing compilers, though!

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

flag initialized to 0 by main thread

# Pthreads global sum with busy-waiting

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

Busy wait every loop iteration!

Can we do a little better?

# Global sum function with critical section after loop

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;
   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

# Mutexes

- A thread that is busy-waiting may continually use the CPU accomplishing nothing.
- Mutex (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time.

# Mutexes


- Used to guarantee that one thread “excludes” all other threads while it executes the critical section.
- The Pthreads standard includes a special type for mutexes: pthread_mutex_t.

```
int  pthread_mutex_init(
         pthread_mutex_t*              mutex_p    /* out */
         const  pthread_mutexattr_t*   attr_p     /* in  */);
```

# Mutexes

- When a Pthreads program finishes using a mutex, it should call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```

- In order to gain access to a critical section a thread calls

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p /* in/out */);
```

# Mutexes

- When a thread is finished executing the code in a critical section, it should call

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p  /* in/out */);
```

# C++ mutex

- std::mutex my_mutex;
- my_mutex.lock() //lock the mutex
- my_mutex.unlock() //unlock the mutex

# How can we use mutex in our “PI” program?

# Global sum function that uses a mutex (1)

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;
   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

| Threads | Busy-Wait | Mutex |
|---|---|---|
| 1 | 2.90 | 2.90 |
| 2 | 1.45 | 1.45 |
| 4 | 0.73 | 0.73 |
| 8 | 0.38 | 0.38 |
| 16 | 0.50 | 0.38 |
| 32 | 0.80 | 0.40 |
| 64 | 3.56 | 0.38 |

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \texttt{thread\_count}$$

Run-times (in seconds) of π programs using n = 108 terms on a system with two four-core processors.

| Time | flag | Thread | | | | |
|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | crit sect | busy wait | susp | susp | susp |
| 1 | 1 | terminate | crit sect | susp | busy wait | susp |
| 2 | 2 | — | terminate | susp | busy wait | busy wait |
| ⋮ | ⋮ | | | ⋮ | ⋮ | ⋮ |
| ? | 2 | — | — | crit sect | susp | busy wait |

Possible sequence of events with busy-waiting and more threads than cores.

# PRODUCER-CONSUMER SYNCHRONIZATION AND SEMAPHORES

# Issues

- Busy-waiting enforces the order threads access a critical section.
- Using mutexes, the order is left to chance and the system.
- There are applications where we need to control the order threads access the critical section.

# Problems with a mutex solution

Example: product of matrices is not commutative

```
/* n and product_matrix are shared and initialized by the main thread */
/* product_matrix is initialized to be the identity matrix            */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);
   Free_matrix(&my_mat);
   return NULL;
}  /* Thread_work */
```

# Another example: a first attempt at sending messages using pthreads

```
/* messages has type char**. It's allocated in main. */
/* Each entry is set to NULL in main.                */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);

   return NULL;
}  /* Send_msg */
```

# What can we do to ensure that every thread receives its message?

# Use mutex??

```
1  . . .
2  pthread_mutex_lock(mutex[dest]);
3  . . .
4  messages[dest] = my_msg;
5  pthread_mutex_unlock(mutex[dest]);
6  . . .
7  pthread_mutex_lock(mutex[my_rank]);
8  printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
9  . . .
```

**What can happen here?**

# Syntax of the various semaphore functions

```
#include <semaphore.h>
```

Semaphores are not part of Pthreads; you need to add this.

```
int  sem_init(
        sem_t*      semaphore_p   /* out */,
        int         shared        /* in  */,
        unsigned    initial_val   /* in  */);
```

```
int  sem_destroy(sem_t*   semaphore_p   /* in/out */);
int  sem_post(sem_t*      semaphore_p   /* in/out */);
int  sem_wait(sem_t*      semaphore_p   /* in/out */);
```

Post: increments the semaphore value

Wait, will block if the semaphore is 0, otherwise it will decrement the semaphore value and continue… and… decrement when unblock!

# Use of semaphore

```
/* messages is allocated and initialized to NULL in main   */
/* semaphores is allocated and initialized to 0 (locked) in
      main */
void* Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;
   sem_post(&semaphores[dest])
         /* ''Unlock'' the semaphore of dest */

   /* Wait for our semaphore to be unlocked */
   sem_wait(&semaphores[my_rank]);
   printf("Thread %ld > %s\n", my_rank, messages[my_rank]);

   return NULL;
}  /* Send_msg */
```

# BARRIERS AND CONDITION VARIABLES

# Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.

# Using barriers to time the slowest thread

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
```

```
my_elapsed = my_finish - my_start;

elapsed = Maximum of my_elapsed values;
```

# Using barriers for debugging

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

# Busy-waiting and a Mutex

- Implementing a barrier using busy-waiting and a mutex is straightforward.
- We use a shared counter protected by the mutex.
- When the counter indicates that every thread has entered the critical section, threads can leave the critical section.

# Busy-waiting and a Mutex

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < thread_count);
    . . .
}
```

We need one counter variable for each instance of the barrier, otherwise problems are likely to occur.

# Implementing a barrier with semaphores

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
```