并行计算

（Parallel Computing）

共享内存编程 - Pthreads

学习内容：

- 进程、线程、Pthreads
- Hello，World
- Pthreads中的矩阵 – 向量乘法
- 临界区（Critical Section）
- 忙 – 等（Busy-Waiting）

共享内存编程 - Pthreads

学习内容：

- 互斥量（Mutex）
- 生产者－消费者同步和信号量（Semaphore）
- 屏障（Barrier）和条件变量（Condition Variable）
- 读－写锁（Read-Write Lock）
- Cache，Cache一致性，伪共享（False Sharing）
- 线程安全性

4. 临界区（Critical Sections）

- 当多个线程试图更新同一块内存区域，结果如何？
  - 估算圆周率

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

4. 临界区（Critical Sections）

● 当多个线程试图更新同一块内存区域，结果如何？

- 估算圆周率

	n			
	10^5	10^6	10^7	10^8
π	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

有什么问题？

5.忙 – 等（Busy-Waiting）

- 进程在进入临界区前反复测试条件，直到条件满足
- 注意：编译器有可能对代码进行优化

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

flag initialized to 0 by main thread

5.忙 – 等（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 */
```

n = 10^8时串行算法优于并行算法（双核系统两个线程，19.5秒 vs. 2.8秒），why?

5.忙－等（Busy-Waiting）

```
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 */
```

减少执行临界区域代码的次数！ $n = 10^8$时，

（双核系统两个线程， 1.5秒 vs. 2.8秒）

6.互斥量（Mutexes）

- 忙－等方法会导致线程始终占用CPU，且不进行有效工作
- Mutex（mutual exclusion）互斥：一种特殊类型的变量，用来限制线程对临界区的访问
- 用来保证一个线程在执行临界区代码时，“排斥”其他线程

6.互斥量（Mutexes）

●Pthreads 标准包含互斥类型的变量：pthread_mutex_t，使用前需要进行初始化

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

```
pthread_mutex_t mutex;

……

pthread_mutex_init(&mutex, NULL);
```

6.互斥量（Mutexes）

- 当 Pthread 程序结束时要销毁互斥量

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

- 为了临界区的访问权，进程需调用：

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

- 当进程执行完临界区代码时，需调用：

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

6.互斥量（Mutexes）

```
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 */
```

6.互斥量（Mutexes）

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

Run-times (in seconds) of π programs using $n = 10^8$ terms on a system with two four-core processors.

6.互斥量（Mutexes）

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

two cores and five threads

6.互斥量（Mutexes）

- 典型的使用 Mutex 的顺序
  - 创建并初始化 mutex 变量
  - 线程试图 lock mutex
  - 只有一个线程成功，获得 mutex
  - 获得 mutex 的线程执行一系列操作
  - 获得 mutex 的线程 unlock mutex
  - 另一个线程获取 mutex 并重复操作
  - 最后 mutex 被销毁

6.互斥量（Mutexes）

Thread 1	Thread 2	Thread 3
Lock	Lock	
A = 2	A = A+1	A = A*B
Unlock	Unlock	

有问题？

6.互斥量（Mutexes）

- 当多个线程等待 locked mutex，哪个线程在 unlock mutex 后首先获得该 mutex？
  - 除非使用线程优先级调度，否则由系统调度程序决定，有随机性

7.生产者－消费者同步和信号量（Semaphore）

- 忙-等 强制线程按顺序访问临界区
- 使用互斥量，线程的访问顺序由系统决定
- 在一些应用中，需要控制线程访问临界区的顺序

7.生产者－消费者同步和信号量（Semaphore）

- 每个线程生成 n * n 矩阵，按照线程的 rank 顺序相乘

```
/* 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 */
```

7.生产者－消费者同步和信号量（Semaphore）

- 每个线程向其他线程发送消息：如 0 -> 1, 1 -> 2, …, t – 1 -> 0

```
/* 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 */
```

7.生产者－消费者同步和信号量（Semaphore）

- 每个线程向其他线程发送消息：如 0 -> 1, 1 -> 2, …, t – 1 -> 0

```
while (messages[my_rank] == NULL);
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
```

7.生产者－消费者同步和信号量（Semaphore）

- 每个线程向其他线程发送消息：如 0 -> 1, 1 -> 2, …, t – 1 -> 0

```
. . .
messages[dest] = my_msg;
Notify thread dest that it can proceed;

Await notification from thread source
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
. . .
```

mutex 初始化为 unlocked，何时调用 pthread_mutex_lock？

7.生产者－消费者同步和信号量（Semaphore）

- 每个线程向其他线程发送消息：如 0 -> 1, 1 -> 2, …, t – 1 -> 0

```
. . .
pthread_mutex_lock(mutex[dest]);
. . .
messages[dest] = my_msg;
pthread_mutex_unlock(mutex[dest]);
. . .
pthread_mutex_lock(mutex[my_rank]);
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
. . .
```

假设有两个线程，会出现什么问题？

7.生产者－消费者同步和信号量（Semaphore）

- 信号量（Semaphore）
  - 特殊的 unsigned int
  - 二进制信号量（0,1）
  - sem_wait：如果信号量为0，阻塞；如果信号量为1，信号量减1，运行
  - sem_post：信号量加1，使得 sem_wait 等待的线程可以运行

7.生产者－消费者同步和信号量（Semaphore）

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

```
#include <semaphore.h>

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 */);
```

7.生产者－消费者同步和信号量（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 */
```

7.生产者－消费者同步和信号量（Semaphore）

- 信号量与互斥量的区别
  - 没有与信号量相关联的所有者，主线程可以初始化所有信号量，任何线程可以在任意信号量上执行 sem_wait 或 sem_post
  - 消息发送问题并不包括临界区，一个线程等待另一个线程做出某种动作后，才能继续执行，这种同步称为“生产者 - 消费者同步”

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）
  - 同步线程以确保它们都到达程序中的同一点
  - 在所有线程都到达屏障之前，任何线程都不能穿越屏障

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）：为线程计时

```
/* 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;
```

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）：调试

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

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）
  - 许多 Pthreads 的实现不提供 barrier
  - 为了代码的可移植性，需要自己实现屏障
    - 忙－等和互斥量
    - 信号量
    - 条件变量

```
int MPI_Barrier(MPI_Comm   comm   /* in */);
```

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）： 忙－等和互斥量
  - 由互斥量保护的共享计数器
  - 当计数器指示每个进程都进入过临界区，进程可以离开“忙-等”循环

8.屏障（Barrier）和条件变量（Condition Variable）

●屏障（Barrier）： 忙－等和互斥量

```
/* 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.

8.屏障（Barrier）和条件变量（Condition Variable）

●屏障（Barrier）：信号量

Can

counter,

count_sem,

barrier_sem

be reused ?

```
/* 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);
   }
   . . .
```

8.屏障（Barrier）和条件变量（Condition Variable）

● 屏障（Barrier）：条件变量

- 条件变量是一个数据对象，它允许线程在某个事件或条件发生之前暂停执行
- 当事件或条件发生时，另一个线程可以向线程发出“唤醒”信号
- 条件变量与互斥量相关联

8.屏障（Barrier）和条件变量（Condition Variable）

- 典型的使用条件变量的顺序
  - 主线程
    - 声明并初始化需要同步的全局数据（如：count）
    - 声明并初始化条件变量对象
    - 声明并初始化相关联的 mutex
    - 创建线程 A 和线程 B 进行工作

8.屏障（Barrier）和条件变量（Condition Variable）

- 典型的使用条件变量的顺序
  - 线程 A
    - 进行工作，直到需要满足某个条件（如：count 必须为指定值）
    - lock 关联的 mutex 并检测全局数据的值
    - 调用 pthread_cond_wait 等待线程 B 发送的信号（pthread_cond_wait 自动 unlock 关联的 mutex）
    - 收到信号后被唤醒，mutex 自动被 lock
    - 显式的 unlock mutex
    - 继续工作

8.屏障（Barrier）和条件变量（Condition Variable）

- 典型的使用条件变量的顺序
  - 线程 B
    - 进行工作
    - lock 关联的 mutex
    - 修改线程 A 等待的全局数据
    - 检测该全局数据，如果符合线程 A 的条件，向其发送信号
    - unlock mutex
    - 继续工作

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）：条件变量

```
lock mutex;
if condition has occurred
    signal thread(s);
else {
    unlock the mutex and block;
    /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）：条件变量
  - Pthreads 中的条件变量：pthread_cond_t
  - 解锁一个阻塞的线程

```
int pthread_cond_signal(pthread_cond_t*  cond_var_p /* in/out */);
```

  - 解锁所有阻塞的线程

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p /* in/out */);
```

8.屏障（Barrier）和条件变量（Condition Variable）

● 屏障（Barrier）：条件变量

```
int pthread_cond_wait(
      pthread_cond_t*     cond_var_p    /* in/out */,
      pthread_mutex_t*    mutex_p       /* in/out */);
```

➢ unlock mutex_p 引用的互斥量，并导致执行的线程阻塞，直到其他线程调用 pthread_cond_signal 或 pthread_cond_broadcast，当线程重新运行，重新 lock 互斥量

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）：条件变量

```
int pthread_cond_wait(
      pthread_cond_t*     cond_var_p    /* in/out */,
      pthread_mutex_t*    mutex_p       /* in/out */);
```

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）：条件变量

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   . . .
}
```

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）：条件变量
  - 像互斥量和信号量一样，条件变量应该初始化和销毁

```
int pthread_cond_init(
      pthread_cond_t*             cond_p        /* out */,
      const pthread_condattr_t*   cond_attr_p   /* in  */);

int pthread_cond_destroy(pthread_cond_t*  cond_p  /* in/out */);
```