Shared Memory Programming with Pthreads

T. Yang. UCSB CS240A.
Outline

• Shared memory programming: Overview
• POSIX pthreads
• Critical section & thread synchronization.
  ▪ Mutexes.
  ▪ Producer-consumer synchronization and semaphores.
  ▪ Barriers and condition variables.
  ▪ Read-write locks.
• Thread safety.
Processes/Threads in Shared Memory Architecture

- A process is an instance of a running (or suspended) program.
- Threads are analogous to a “light-weight” process.
- In a shared memory program a single process may have multiple threads of control.
Execution Flow on one-core or multi-core systems

Concurrent execution on a single core system: Two threads run concurrently if their logical flows overlap in time

<table>
<thead>
<tr>
<th>single core</th>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
<th>T_4</th>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
<th>T_4</th>
<th>T_1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parallel execution on a multi-core system

<table>
<thead>
<tr>
<th>core 1</th>
<th>T_1</th>
<th>T_3</th>
<th>T_1</th>
<th>T_3</th>
<th>T_1</th>
<th>...</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>core 2</th>
<th>T_2</th>
<th>T_4</th>
<th>T_2</th>
<th>T_4</th>
<th>T_2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Benefits of multi-threading

• Responsiveness

• Resource Sharing
  - Shared memory

• Economy

• Scalability
  - Explore multi-core CPUs
Thread Programming with Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables**, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate **implicitly** by writing and reading shared variables.
  - Threads coordinate by **synchronizing** on shared variables

![Diagram of shared and private memory with threads P0, P1, Pn]
Shared Memory Programming

Several Thread Libraries/systems

- **Pthreads is the POSIX Standard**
  - Relatively low level
  - Portable but possibly slow; relatively heavyweight

- **OpenMP standard for application level programming**
  - Support for scientific programming on shared memory
  - [http://www.openMP.org](http://www.openMP.org)

- **Java Threads**

- **TBB: Thread Building Blocks**
  - Intel

- **CILK: Language of the C “ilk”**
  - Lightweight threads embedded into C
Overview of POSIX Threads

• **POSIX**: *Portable Operating System Interface for UNIX*
  - Interface to Operating System utilities

• **PThreads**: The POSIX threading interface
  - System calls to create and synchronize threads
  - In CSIL, compile a c program with gcc -lpthread

• **PThreads contain support for**
  - Creating parallelism and synchronization
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread
Creation of Unix processes vs. Pthreads
C function for starting a thread

`int pthread_create (pthread_t* thread_p /* out */, const pthread_attr_t* attr_p /* in */, void* (*start_routine) (void) /* in */, void* arg_p /* in */) ;`

One object for each thread.
int pthread_create (  
    pthread_t* thread_p /* out */ ,  
    const pthread_attr_t* attr_p /* in */ ,  
    void* (*start_routine) ( void ) /* in */ ,  
    void* arg_p /* in */ ) ;

We won’t be using, so we just pass NULL.

Allocate before calling.
int pthread_create ( 
    pthread_t* thread_p /* out */ ,
    const pthread_attr_t* attr_p /* in */ ,
    void* (*start_routine ) ( void ) /* in */ ,
    void* arg_p /* in */ ) ;

Pointer to the argument that should be passed to the function start_routine.

The function that the thread is to run.
Function started by pthread_create

• Prototype:
  
  void*  thread_function ( void*  args_p ) ;

• Void* can be cast to any pointer type in C.

• So args_p can point to a list containing one or more values needed by thread_function.

• Similarly, the return value of thread_function can point to a list of one or more values.
Wait for Completion of Threads

```c
pthread_join(pthread_t *thread, void **result);
```

- Wait for specified thread to finish. Place exit value into *result.

- We call the function `pthread_join` once for each thread.

- A single call to `pthread_join` will wait for the thread associated with the `pthread_t` object to complete.
Example of Pthreads

#include <pthread.h>
#include <stdio.h>

void *PrintHello(void * id){
    printf("Thread%d: Hello World!\n", id);
}

void main (){  
    pthread_t thread0, thread1;
    pthread_create(&thread0, NULL, PrintHello, (void *) 0);
    pthread_create(&thread1, NULL, PrintHello, (void *) 1);
}
Example of Pthreads with join

```c
#include <pthread.h>
#include <stdio.h>

void *PrintHello(void * id){
    printf("Hello from thread \%d\n", id);
}

void main (){ 
    pthread_t thread0, thread1;
    pthread_create(&thread0, NULL, PrintHello, (void *) 0);
    pthread_create(&thread1, NULL, PrintHello, (void *) 1);
    pthread_join(thread0, NULL);
    pthread_join(thread1, NULL);
} 
```
Some More Pthread Functions

- **pthread_yield();**
  - Informs the scheduler that the thread is willing to yield

- **pthread_exit(void *value);**
  - Exit thread and pass value to joining thread (if exists)

**Others:**

- **pthread_t me; me = pthread_self();**
  - Allows a pthread to obtain its own identifier pthread_t thread;

- **Synchronizing access to shared variables**
  - pthread_mutex_init, pthread_mutex_[un]lock
  - pthread_cond_init, pthread_cond_[timed]wait
Compiling a Pthread program

```
gcc -g -Wall -o pth_hello pth_hello.c -lpthread
```

link in the Pthreads library
Running a Pthreads program

. / pth_hello

Hello from thread 1
Hello from thread 0

. / pth_hello

Hello from thread 0
Hello from thread 1
Difference between Single and Multithreaded Processes

- Shared memory access for code/data
- Separate control flow -> separate stack/registers
CRITICAL SECTIONS
Data Race Example

```java
static int s = 0;

Thread 0

for i = 0, n/2-1
  s = s + f(A[i])

Thread 1

for i = n/2, n-1
  s = s + f(A[i])
```

- Also called critical section problem.
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously
Synchronization Solutions

1. Busy waiting
2. Mutex (lock)
3. Semaphore
4. Conditional Variables
Example of Busy Waiting

```plaintext
static int s = 0;
static int flag=0

Thread 0
int temp, my_rank
for i = 0, n/2-1
  temp0=f(A[i])
  while flag!=my_rank;
  s = s + temp0
  flag= (flag+1) %2

Thread 1
int temp, my_rank
for i = n/2, n-1
  temp=f(A[i])
  while flag!=my_rank;
  s = s + temp
  flag= (flag+1) %2
```

• A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.
• **Weakness:** Waste CPU resource. Sometime not safe with compiler optimization.
Mutexes (Locks)

- Code structure

- Mutex (mutual exclusion) is a special type of variable used to restrict access to a critical section to a single thread at a time.
  - guarantee that one thread “excludes” all other threads while it executes the critical section.
- When a thread waits on a mutex/lock, CPU resource can be used by others.
- Only thread that has acquired the lock can release this lock

Acquire mutex lock

Critical section

Unlock/Release mutex
Execution example with 2 threads

Thread 1
- Acquire mutex lock
- Critical section
- Unlock/Release mutex

Thread 2
- Acquire mutex lock
- Critical section
- Unlock/Release mutex
Mutexes in Pthreads

• A special type for mutexes: `pthread_mutex_t`.

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

• To gain access to a critical section, call

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

• To release

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

• When finishing use of a mutex, call

```c
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```
Global sum function that uses a mutex (1)

```c
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 */
Semaphore: Generalization from mutex locks

- Semaphore $S$ – integer variable
- Can only be accessed /modified via two (atomic) operations with the following semantics:
  - `wait (S)` 
    ```
    while S <= 0 wait in a queue;
    S--;
    ```
  - `post(S)` 
    ```
    S++;
    Wake up a thread that waits in the queue.
    ```
Why Semaphores?

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>Functionality/weakness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy waiting</td>
<td>Spinning for a condition. Waste resource. Not safe</td>
</tr>
<tr>
<td>Mutex lock</td>
<td>Support code with simple mutual exclusion</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Handle more complex signal-based synchronization</td>
</tr>
</tbody>
</table>

- **Examples of complex synchronization**
  - Allow a resource to be shared among multiple threads.
    - Mutex: no more than 1 thread for one protected region.
  - Allow a thread waiting for a condition after a signal
    - E.g. Control the access order of threads entering the critical section.
    - For mutexes, the order is left to chance and the system.
Semaphores are not part of Pthreads; you need to add this.
Producer-consumer
Synchronization and
Semaphores
Producer-Consumer Example

- Thread \( x \) produces a message for Thread \( x+1 \).
  - Last thread produces a message for thread 0.
- Each thread prints a message sent from its source.
- Will there be null messages printed?
  - A consumer thread prints its source message before this message is produced.
  - How to avoid that?
Flag-based Synchronization with 3 threads

To make sure a message is received/printed, use busy waiting.
First attempt at sending messages using pthreads

```c
/* 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);

    sprintf(my_msg, "Hello to %ld
ds, 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 */
```
Semaphore Synchronization with 3 threads

Thread 0

- Write a msg to #1
- Set msg[1]
- Post(semp[1])
- Wait(semp[0])
- Print msg[0]

Thread 1

- Write a msg to #2
- Set msg[2]
- Post(semp[2])
- Wait(semp[1])
- Print msg[1]

Thread 2

- Write a msg to #0
- Set msg[0]
- Post(semp[0])
- Wait(semp[2])
- Print msg[2]
Message sending with semaphores

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

sem_post(&semaphores[dest]);
    /* signal the dest thread*/
sem_wait(&semaphores[my_rank]);
    /* Wait until the source message is created */

printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
```
READERS-WRITERS PROBLEM
Synchronization Example for Readers-Writers Problem

- A data set is shared among a number of concurrent threads.
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write
- Requirement:
  - allow multiple readers to read at the same time.
  - Only one writer can access the shared data at the same time.

- Reader/writer access permission table:

<table>
<thead>
<tr>
<th></th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader</td>
<td>OK</td>
<td>No</td>
</tr>
<tr>
<td>Writer</td>
<td>NO</td>
<td>No</td>
</tr>
</tbody>
</table>
Readers-Writers (First try with 1 mutex lock)

- **writer**
  
  ```
  do {
      mutex_lock(w);
      // writing is performed
      mutex_unlock(w);
  } while (TRUE);
  ```

- **Reader**
  
  ```
  do {
      mutex_lock(w);
      // reading is performed
      mutex_unlock(w);
  } while (TRUE);
  ```
Readers-Writers (First try with 1 mutex lock)

- **writer**
  
  ```
  do {
    mutex_lock(w);
    // writing is performed
    mutex_unlock(w);
  } while (TRUE);
  ```

- **Reader**
  
  ```
  do {
    mutex_lock(w);
    // reading is performed
    mutex_unlock(w);
  } while (TRUE);
  ```

<table>
<thead>
<tr>
<th></th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Writer</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
2nd try using a lock + readcount

- **writer**
  ```c
  do {
    mutex_lock(w); // Use writer mutex lock
    //    writing is performed
    mutex_unlock(w);
  } while (TRUE);
  ```

- **Reader**
  ```c
  do {
    readcount++; // add a reader counter.
    if(readcount==1) mutex_lock(w);
    //    reading is performed
    readcount--;
    if(readcount==0) mutex_unlock(w);
  } while (TRUE);
  ```
Readers-Writers Problem with semaphore

• Shared Data
  ▪ Data set
  ▪ Lock `mutex` (to protect readcount)
  ▪ Semaphore `wrt` initialized to 1 (to synchronize between readers/writers)
  ▪ Integer `readcount` initialized to 0
Readers-Writers Problem

• A writer

  do {
    sem_wait(wrt);  // semaphore wrt
    // writing is performed
    sem_post(wrt);  //
  } while (TRUE);

Readers-Writers Problem (Cont.)

- **Reader**
  
  ```c
  do {
      mutex_lock(mutex);
      readcount ++ ;
      if (readcount == 1)
      
          sem_wait(wrt);   //check if anybody is writing
      mutex_unlock(mutex)
  
      // reading is performed

      mutex_lock(mutex);
      readcount - - ;
      if (readcount  == 0)
      
          sem_post(wrt) ;  //writing is allowed now
      nlock(mutex) ;
  } while (TRUE);
  ```
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.
- Availability:
  - No barrier provided by Pthreads library and needs a custom implementation
  - Barrier is implicit in OpenMP and available in MPI.
Condition Variables

- Why?
- More programming primitives to simplify code for synchronization of threads

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy waiting</td>
<td>Spinning for a condition. Waste resource. Not safe</td>
</tr>
<tr>
<td>Mutex lock</td>
<td>Support code with simple mutual exclusion</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Signal-based synchronization. Allow sharing (not wait unless semaphore=0)</td>
</tr>
<tr>
<td>Barrier</td>
<td>Rendezvous-based synchronization</td>
</tr>
<tr>
<td>Condition variables</td>
<td>More complex synchronization: Let threads wait until a user-defined condition becomes true</td>
</tr>
</tbody>
</table>
Synchronization Primitive: Condition Variables

- Used together with a lock
- One can specify more general waiting condition compared to semaphores.
- A thread is blocked when condition is not true:
  - placed in a waiting queue, yielding CPU resource to somebody else.
  - Wake up until receiving a signal
Pthread synchronization: Condition variables

```c
int status;     pthread_condition_t cond;
const pthread_condattr_t attr;
pthread_mutex mutex;
status = pthread_cond_init(&cond,&attr);
status = pthread_cond_destroy(&cond);
status = pthread_cond_wait(&cond,&mutex);
    - wait in a queue until somebody wakes up. Then the mutex is reacquired.
status = pthread_cond_signal(&cond);
    - wake up one waiting thread.
status = pthread_cond_broadcast(&cond);
    - wake up all waiting threads in that condition
```
How to Use Condition Variables: Typical Flow

- Thread 1: // try to get into critical section and wait for the condition

  Mutex_lock(mutex);
  While (condition is not satisfied)
    Cond_Wait(mutex, cond);
  Critical Section;
  Mutex_unlock(mutex)

- Thread 2: // Try to create the condition.

  Mutex_lock(mutex);
  When condition can satisfy, Signal(cond);
  Mutex_unlock(mutex);
Condition variables for in producer-consumer problem with unbounded buffer

Producer deposits data in a buffer for others to consume
First version for consumer-producer problem with unbounded buffer

- int avail=0; // # of data items available for consumption
- Consumer thread:

  ```
  while (avail <=0); //wait
  Consume next item;  avail = avail-1;
  ```

- Producer thread:

  ```
  Produce next item;  avail = avail+1;
  //notify an item is available
  ```
Condition Variables for consumer-producer problem with unbounded buffer

- int avail=0;  // # of data items available for consumption
- Pthread mutex m and condition cond;
- Consumer thread:

```c
mutex_lock(&m)
while (avail <=0) Cond_Wait(&cond, &m);
Consume next item; avail = avail-1;
mutex_unlock(&mutex)
```

- Producer thread:

```c
mutex_lock(&m);
Produce next item; avaiill = avail+1;
Cond_signal(&cond); //notify an item is available
mutex_unlock(&m);
```
When to use condition broadcast?

• When waking up one thread to run is not sufficient.
• Example: concurrent malloc()/free() for allocation and deallocation of objects with non-uniform sizes.
Running trace of malloc()/free()

- Initially 10 bytes are free.
- m() stands for malloc(). f() for free()

<table>
<thead>
<tr>
<th>Thread 1:</th>
<th>Thread 2:</th>
<th>Thread 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>m(10) – succ</td>
<td>m(5) – wait</td>
<td>m(5) – wait</td>
</tr>
<tr>
<td>f(10) – broadcast</td>
<td>Resume m(5)-succ</td>
<td>Resume m(5)-succ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m(3) –wait</td>
</tr>
<tr>
<td>m(7) – wait</td>
<td></td>
<td>Resume m(3)-succ</td>
</tr>
<tr>
<td>Resume m(7)-wait</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time
Issues with Threads: False Sharing, Deadlocks, Thread-safety
Problem: False Sharing

- Occurs when two or more processors/cores access different data in same cache line, and at least one of them writes.
  - Leads to ping-pong effect.
- Let’s assume we parallelize code with p=2:
  ```c
  for( i=0; i<n; i++ )
      a[i] = b[i];
  ```
  - Each array element takes 8 bytes
  - Cache line has 64 bytes (8 numbers)
Execute this program in two processors

```c
for (i=0; i<n; i++)
    a[i] = b[i];
```

Written by CPU 0

Written by CPU 1
False Sharing: Example

```
Two CPUs execute:
for( i=0; i<n; i++ )
a[i] = b[i];
```

- **a[0]**  Written by CPU 0
- **a[1]**  Written by CPU 1
- **a[2]**
- **a[3]**
- **a[4]**
- **a[5]**
- **a[6]**
- **a[7]**

**cache line**
Matrix-Vector Multiplication with Pthreads

Parallel programming book by Pacheco book P.159-162
Sequential code

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix} \times \begin{pmatrix}
1 \\
2 \\
3
\end{pmatrix} = \begin{pmatrix}
1 \times 1 + 2 \times 2 + 3 \times 3 \\
4 \times 1 + 5 \times 2 + 6 \times 3 \\
7 \times 1 + 8 \times 2 + 9 \times 3
\end{pmatrix} = \begin{pmatrix}
14 \\
32 \\
50
\end{pmatrix}
\]

/\* For each row of A */\*
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /\* For each element of the row and each element of x */\*
    for (j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}\*

<table>
<thead>
<tr>
<th>$a_{00}$</th>
<th>$a_{01}$</th>
<th>$\cdots$</th>
<th>$a_{0,n-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{10}$</td>
<td>$a_{11}$</td>
<td>$\cdots$</td>
<td>$a_{1,n-1}$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$a_{i0}$</td>
<td>$a_{i1}$</td>
<td>$\cdots$</td>
<td>$a_{i,n-1}$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$a_{m-1,0}$</td>
<td>$a_{m-1,1}$</td>
<td>$\cdots$</td>
<td>$a_{m-1,n-1}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_0$</th>
<th>$y_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$y_1$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$x_{n-1}$</td>
<td>$y_{m-1}$</td>
</tr>
</tbody>
</table>

$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$
Task partitioning

For \( i=0; \ i<m; \ i=i+1 \)

\[
\text{Task } S_i \text{ for Row } i\\
y[i]=0; \\
\text{For } (j=0; \ j<n; \ j=j+1) \\
y[i]=y[i]+a[i][j]*x[j]
\]

Task graph

\[
\begin{align*}
S0 & \quad S1 & \quad \ldots & \quad Sm \\
S0 & \quad S1 & \quad \ldots & \quad S2 & \quad S3 & \quad \ldots
\end{align*}
\]

Mapping to threads

\[
\begin{align*}
\text{Thread 0} & \quad \text{Thread 1} \\
S0 & \quad S1 & \quad S2 & \quad S3 & \quad \ldots
\end{align*}
\]
Using 3 Pthreads for 6 Rows: 2 row per thread

<table>
<thead>
<tr>
<th>Thread</th>
<th>Components of y</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y[0], y[1]</td>
<td>S0, S1</td>
</tr>
<tr>
<td>1</td>
<td>y[2], y[3]</td>
<td>S2, S3</td>
</tr>
<tr>
<td>2</td>
<td>y[4], y[5]</td>
<td>S4, S5</td>
</tr>
</tbody>
</table>

Code for S0

```c
y[0] = 0.0;
for (j = 0; j < n; j++)
    y[0] += A[0][j]*x[j];
```

Code for Si

```c
y[i] = 0.0;
for (j = 0; j < n; j++)
    y[i] += A[i][j]*x[j];
```
i-th thread calls `Pth_mat_vect( &i)`

- m is # of rows in this matrix A.
- n is # of columns in this matrix A.
- local_m is # of rows handled by this thread.
Impact of false sharing on performance of matrix-vector multiplication

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$8,000,000 \times 8$</td>
<td>0.393</td>
<td>1.000</td>
<td>$8000 \times 8000$</td>
<td>0.345</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$8 \times 8,000,000$</td>
<td>0.441</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0.217</td>
<td>0.906</td>
<td></td>
<td>0.188</td>
<td>0.918</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.300</td>
<td>0.735</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0.139</td>
<td>0.707</td>
<td></td>
<td>0.115</td>
<td>0.750</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.388</td>
<td>0.290</td>
</tr>
</tbody>
</table>

(times are in seconds)

Why is performance of $8 \times 8,000,000$ matrix bad?

How to fix that?
Deadlock and Starvation

- **Deadlock** – two or more threads are waiting indefinitely for an event that can be only caused by one of these waiting threads.

- **Starvation** – indefinite blocking (in a waiting queue forever).

Let $S$ and $Q$ be two mutex locks:

```
P_0
  Lock(S);
  Lock(Q);
  ...
  Unlock(Q);
Unlock(S);

P_1
  Lock(Q);
  Lock(S);
  ...
  Unlock(S);
Unlock(Q);
```
Deadlock Avoidance

• Order the locks and always acquire the locks in that order.
• Eliminate circular waiting

\[
P_0
\]

- \text{Lock(S)};
- \text{Lock(Q)};
- \text{Unlock(Q)};
- \text{Unlock(S)};

\[
P_1
\]

- \text{Lock(S)};
- \text{Lock(Q)};
- \text{Unlock(Q)};
- \text{Unlock(S)};
Thread-Safety

• A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

• When you program your own functions, you know if they are safe to be called by multiple threads or not.

• You may forget to check if system library functions used are thread-safe.
  - Unsafe function: `strtok()` from C `string.h` library
  - Other example.
    - The random number generator `random` in `stdlib.h`.
    - The time conversion function `localtime` in `time.h`. 
Concluding Remarks

• A thread in shared-memory programming is analogous to a process in distributed memory programming.
  ▪ However, a thread is often lighter-weight than a full-fledged process.
• When multiple threads access a shared resource without controlling, it may result in an error: we have a race condition.
  ▪ A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time
  ▪ Mutex, semaphore, condition variables
• Issues: false sharing, deadlock, thread safety