COMP/CS 605: Introduction to Parallel Computing
Lecture 25: Shared Memory Programming with Pthreads

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Presented: 03/21/16
Updated: 03/21/16
Table of Contents

1. Shared Memory Programming with PThreads
   - Shared Memory Systems
   - Threads and Processes
   - Basic Pthreads Program: Hello World
   - POSIX Threads API
   - Matrix-Vector Multiplication with Pthreads
Shared Memory System

Best candidates:

- can be organized into discrete, independent tasks which can execute concurrently
- routines can be interchanged, interleaved and/or overlapped in real time
Shared Memory System
What is a Process?

- A process is an instance of a running (or suspended) program.
- Can be "muti-threaded," created by OS, requires a fair amount of "overhead"
- Process ID, process group ID, user ID, and group ID, Environment
- Program instructions, registers, stack, heap, signals, libraries
- Working directory, file descriptors
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Ref: http://www.bottomupcs.com/elements_of_a_process.html
What is a Thread?

- Threads are analogous to a light-weight process.
- Shared memory program: single process may have multiple threads of control.
- Independent stream of instructions, run inside processes
- Programs/procedures: runs independently from main program (e.g. multiple functions running concurrently)
- Example: main program (a.out) that contains a number of procedures that can be scheduled to run simultaneously and/or independently

**Thread models:**

- **Manager/worker:** a single thread, manager assigns work to other threads (workers).
- **Pipeline:** task is broken into series of subops; each handled in series, but concurrently by another thread.
- **Peer:** After the main thread (manager) creates other threads, it participates in the work.
POSIX Threads

- Portable Operating System Interface
- IEEE’s POSIX Threads Model (Pthreads):
  - programming models for threads in a UNIX platform
  - Pthreads are included in the international standards ISO/IEC9945-1
- A standard for Unix-like operating systems.
- A library that can be linked with C programs.
- Specifies an application programming interface (API) for multi-threaded programming.

The Pthreads API is only available on POSIXR systems such as: Linux, MacOS X, Solaris, HPUX,
Phreads: Hello World

```c
/* File: pth_hello.c
   * Purpose:
   *   Illustrate basic use of pthreads: create some threads,
   *   each of which prints a message.
   * Input: none
   * Output: message from each thread
   * Compile: gcc -g -Wall -o pth_hello pth_hello.c -lpthread
   * Usage: ./pth_hello <thread_count>
   */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
const int MAX_THDS = 64;
/* Global variable: accessible to all threads */
int thread_count;
void Usage(char* prog_name);
void Hello(void* rank); /* Thread function */
/*------------------------------------------------*/
int main(int argc, char* argv[]) {
   /* Use long in case of a 64-bit system */
   long thread;
   pthread_t* thread_handles;
   /* Get number of threads from command line */
   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   if (thread_count <= 0 || thread_count > MAX_THDS)
      Usage(argv[0]);
   thread_handles = malloc (thread_count*sizeof(pthread_t));
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
                      Hello, (void*) thread);
   printf("Hello from the main thread\n");
   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);
   free(thread_handles);
   return 0;
} /* main */
/*------------------------------------------------*/
void Hello(void* rank) {
   /* Use long in case of a 64-bit system */
   long my_rank = (long) rank;
   printf("Hello from thread %ld of %d\n",
          my_rank, thread_count);
   return NULL;
} /* Hello */
/*------------------------------------------------*/
void Usage(char* prog_name) {
   /* Use long in case of a 64-bit system */
   long my_rank = (long) rank;
   printf("Hello from thread %ld of %d\n",
          my_rank, thread_count);
   return NULL;
} /* Usage */
#endif
```
Compiling and running a Pthreads program

- Pthreads is a standard C library
- Compile like standard C code:

```
[gidget] % gcc -g -Wall -o pth_hello pth_hello.c -lpthread
```

```
[gidget] % ./pth_hello 1
Hello from the main thread
Hello from thread 0 of 1

[gidget:dev/ipp.ch4/hello] mthomas% ./pth_hello 4
Hello from thread 0 of 4
Hello from thread 2 of 4
Hello from thread 1 of 4
Hello from the main thread
Hello from thread 3 of 4
```
Running a Pthreads program on tuckoo

[mthomas@tuckoo ch4] gcc -g -Wall -o pth_hello pth_hello.c -lpthread

[mthomas@tuckoo ch4] ./pth_hello 8

Hello from thread 0 of 8
Hello from thread 1 of 8
Hello from thread 2 of 8
Hello from thread 3 of 8
Hello from thread 4 of 8
Hello from thread 5 of 8
Hello from thread 6 of 8
Hello from the main thread
Hello from thread 7 of 8
Warning about global variables

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Limit use of global variables to situations where they are really needed:
  - Shared variables.
- Programmers are responsible for synchronizing access (protecting) globally shared data.
  - Can introduce subtle and confusing bugs
POSIX Threads API: Four Main Groups

- **Thread management**: Routines that work directly on threads - creating, detaching, joining, etc.
- **Mutexes**: Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion".
- **Condition variables**: Routines that address communications between threads that share a mutex. Includes functions to create, destroy, wait and signal based upon specified variable values.
- **Synchronization**: Routines that manage read/write locks and barriers.
Processes in MPI are usually started by a script.

In Pthreads the threads are started by the program executable.

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

- **Opaque**
- The actual data that they store is system-specific.
- Their data members aren’t directly accessible to user code.
- However, the Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it’s associated.
A closer look (1)

```c
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.
A closer look (2)

```c
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:
  ```c
  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.
Main thread forks and joins two threads
Stopping the Threads

- 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.
Shared Memory Programming with PThreads

POSIX Threads API

Source: https://computing.llnl.gov/tutorials/pthreads/
Matrix-Vector Multiplication with Pthreads

**Definition:** Let $A$ be an $[m \times n]$ matrix, and $x$ be a $[n \times 1]$, then $y$ will be a vector with the dimensions $[m \times 1]$.

Then

$$y_j = \sum_{t=1}^{m} a_{it} x_t = a_{i1} x_1 + a_{i2} x_2 + \cdots + a_{m-1,1} b_{m-1}$$

$$\begin{bmatrix}
a_{00} & \ldots & a_{0j} & \ldots & a_{0,n-1} \\
a_{10} & \ldots & a_{1j} & \ldots & a_{1,n-1} \\
\vdots & & \vdots & & \vdots \\
a_{i0} & \ldots & a_{ij} & \ldots & a_{i,n-1} \\
a_{m-1,0} & \ldots & a_{m-1,j} & \ldots & a_{m-1,n-1}
\end{bmatrix} \cdot \begin{bmatrix}
x_0 \\
x_1 \\
\vdots \\
x_i \\
x_n
\end{bmatrix} = \begin{bmatrix}
y_0 \\
y_1 \\
\vdots \\
y_j \\
y_{m-1}
\end{bmatrix}$$
Serial Pseudo-code

/* 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];
}

\[ y_i = \sum_{j=0}^{n-1} a_{ij}x_j \]
Using 3 Pthreads, 6 elements

<table>
<thead>
<tr>
<th>Thread</th>
<th>Components of y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y[0], y[1]</td>
</tr>
<tr>
<td>1</td>
<td>y[2], y[3]</td>
</tr>
<tr>
<td>2</td>
<td>y[4], y[5]</td>
</tr>
</tbody>
</table>

Thread 0

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

general case

```
y[i] = 0.0;
for (j = 0; j < n; j++)
    y[i] += A[i][j]*x[j];
```
Matrix-Vector Multiplication with Pthreads

Pthreads matrix-vector multiplication

/* File:
  *   pth_mat_vect.c
  *
  * Compile: gcc -g -Wall -o pth_mat_vect pth_mat_vect.c -lpthread
  * Usage:
  *   pth_mat_vect <thread_count>
  *
  * IPP: Section 4.3 (pp. 159 and ff.). Also Section 4.10 (pp. 191)
  */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variables */
int thread_count;
int m, n;
double* A;
double* x;
double* y;

/* Serial functions */
void Usage(char* prog_name);
void Read_matrix(char* prompt, double A[], int m, int n);
void Read_vector(char* prompt, double x[], int n);
void Print_matrix(char* title, double A[], int m, int n);
void Print_vector(char* title, double y[], double m);

/* Parallel function */
void *Pth_mat_vect(void* rank);
Pthreads matrix-vector multiplication

```c
int main(int argc, char* argv[]) {
    long thread;
    pthread_t* thread_handles;

    if (argc != 2) Usage(argv[0]);
    thread_count = atoi(argv[1]);
    thread_handles = malloc(thread_count*sizeof(pthread_t));

    printf("Enter m and n\n"); scanf("%d%d", &m, &n);

    A = malloc(m*n*sizeof(double));
    x = malloc(n*sizeof(double));
    y = malloc(m*sizeof(double));

    Read_matrix("Enter the matrix", A, m, n); Print_matrix("We read", A, m, n);
    Read_vector("Enter the vector", x, n); Print_vector("We read", x, n);

    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Pth_mat_vect, (void*) thread);

    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);

    Print_vector("The product is", y, m);

    free(A); free(x); free(y);
    return 0;
```
Pthreads matrix-vector multiplication

```c
/*---------------------------------------------
* Function: Usage
* Purpose: print a message showing what the command line should
*         be, and terminate
* In arg : prog_name
*/
void Usage (char* prog_name) {
    fprintf(stderr, "usage: %s <thread_count>
", prog_name);
    exit(0);
} /* Usage */

/*---------------------------------------------
* Function: Read_matrix
* Purpose: Read in the matrix
* In args: prompt, m, n
* Out arg: A
*/
void Read_matrix(char* prompt, double A[], int m, int n) {
    int i, j;

    printf("%s\n", prompt);
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            scanf("%lf", &A[i*n+j]);
} /* Read_matrix */

/*---------------------------------------------
* Function: Read_vector
* Purpose: Read in the vector x
* In arg: prompt, n
* Out arg: x
*/
void Read_vector(char* prompt, double x[], int n) {
    int i;

    printf("%s\n", prompt);
    for (i = 0; i < n; i++)
        scanf("%lf", &x[i]);
} /* Read_vector */
```
Matrix-Vector Multiplication with Pthreads

```c
/*----------------------------------------------
 * Function: Pth_mat_vect
 * Purpose: Multiply an mxn matrix by an nx1 column vector
 * In arg: rank
 * Global in vars: A, x, m, n, thread_count
 * Global out var: y
 */
void *Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */

/*------------------------------------------------------------------
* Function: Print_matrix
* Purpose: Print the matrix
* In args: title, A[], m, n
*/
void Print_matrix( char* title, double A[], int m, int n) {
    int i, j;

    printf("%s\n", title);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            printf("%4.1f ", A[i*n+j]);
        printf("\n");
    }
}
```
Compiling and Running Pth_Mat_Vec on tuckoo

```
[mthomas@tuckoo pacheco/ch4] mthomas% gcc -g -Wall -o pth_mat_vect pth_mat_vect.c -lpthread
[mthomas@tuckoo pacheco/ch4] mthomas% ./pth_mat_vect 4
Enter m and n
4 4
Enter the matrix
1 2 3 4
5 6 7 8
9 10 11 12
1 2 3 4
We read
1.0 2.0 3.0 4.0
5.0 6.0 7.0 8.0
9.0 10.0 11.0 12.0
1.0 2.0 3.0 4.0
Enter the vector
9 7 6 3
We read
9.0 7.0 6.0 3.0
The product is
53.0 153.0 253.0 53.0
```
Matrix Mult Example

- More Straightforward because of shared memory
- Code only *reads* shared arrays \((A, x)\), so no contention associated with shared updates of same memory location
- No thread communication
- Small jobs, small memory

Next we’ll look at what happens when multiple threads need to update same memory location