Lecture 09: Programming with PThreads

Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Topics (Part 1)

• Introduction
• Principles of parallel algorithm design (Chapter 3)
• Programming on shared memory system (Chapter 7)
  – OpenMP
  – PThread, mutual exclusion, locks, synchronizations
  – Cilk/Cilkplus
• Analysis of parallel program executions (Chapter 5)
  – Performance Metrics for Parallel Systems
    • Execution Time, Overhead, Speedup, Efficiency, Cost
  – Scalability of Parallel Systems
  – Use of performance tools
Short Review

• Parallel algorithm design
  1. Tasks and Decomposition
     • Theory and practice (axpy, matvec and matmul)
  2. Processes and Mapping
  3. Minimizing Interaction Overheads

• Practice on task and decomposition
  – AXPY, Matrix vector multiplication, matrix matrix multiplication
OpenMP: Worksharing Constructs

Sequential code

```c
for(i=0;i<N;i++) { a[i] = a[i] + b[i]; }
```

OpenMP parallel region

```c
#pragma omp parallel shared (a, b)
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i]; }
}
```

OpenMP parallel region and a worksharing for construct

```c
#pragma omp parallel shared (a, b) private (i)
#pragma omp for schedule(static)
for(i=0;i<N;i++) { a[i] = a[i] + b[i]; }
```
Directives implemented via code modification and insertion of runtime library calls

- Basic step is outlining of code in parallel region

Runtime library responsible for managing threads

- Scheduling loops
- Scheduling tasks
- Implementing synchronization

Implementation effort is reasonable

Each compiler has custom run-time support. Quality of the runtime system has major impact on performance.

```
int main(void) {
    int a,b,c;
    #pragma omp parallel \ private(c)
    do_sth(a,b,c);
    return 0;
}
```

```
_INT32 main() {
    int a,b,c;
    /* microtask */
    void __ompregion_main1() {
        _INT32 __mplocal_c;
        /*shared variables are kept intact, substitute accesses to private variable*/
        do_sth(a, b, __mplocal_c);
    }
    ...
    /*OpenMP runtime calls */
    __ompc_fork(&__ompregion_main1);
    ...
```
main () {
    #pragma omp parallel
    printf("Hello,world.\n");
}

main () { ...
    __ompc_fork(0, &__ompregion_main1,
                reg_7);
    ...
}

void __ompregion_main1(__ompv_gtid_a__0,
                        __ompv_slink_a__0)
{...
    printf((const _INT8 *)(_INT8(*)[15LL]) "Hello,world.\n");
}

....
for (i=1; i< threads_to_create; i++)
{ return_value = pthread_create(
    &(__omp_level_1_pthread[i].uthread_id),
    &__omp_pthread_attr,
    (pthread_entry) __ompc_level_1_slave,
    (void *)((unsigned long int)i));
....
Execution Model

Start

Parallel region 1

Initialization

Execute Micro_task

Parallel region n

Execute Micro_task

Clean up

End

Level 1 slave thread Reused

Nested slave thread

Master thread

N-1 threads

Wait on Condition var.

Execute Micro_task

M-1 Nested threads

Execute nested Micro_task

Pthread_exit()
PThread

• Processing Element abstraction for software
  – PThreads
  – OpenMP/Cilk/others runtime use PThreads for their implementation

• The foundation of parallelism from computer system

• Topic Overview
  – Thread basics and the POSIX Thread API
  – Thread creation, termination and joining
  – Thread safety
  – Synchronization primitives in PThreads
What is a Thread

• OS view
  – An independent stream of instructions that can be scheduled to run by the OS.

• Software developer view
  – A “procedure” that runs independently from the main program
    • Imagine multiple such procedures of main run simultaneously and/or independently
  – Sequential program: a single stream of instructions in a program.
  – Multi-threaded program: a program with multiple streams
    • Multiple threads are needed to use multiple cores/CPUs

• A thread is a virtual representation of a hardware core
A thread is a single stream of control in the flow of a program:

\[
\text{for (i = 0; i < n; i++)}
\]
\[
y[i] = \text{dot_product}(\text{row}(A, i), b);
\]

\[
\text{for (i = 0; i < n; i++)}
\]
\[
y[i] = \text{create_thread}(\text{dot_product}(\text{row}(A, i), b));
\]

- think of the thread as an instance of a function that returns before the function has finished executing.
Processes

• **processes** contain information about program resources and program execution state, including:
  – Process ID, process group ID, user ID, and group ID
  – Environment, Working directory, Program instructions
  – Registers, Stack, Heap
  – File descriptors, Signal actions
  – Shared libraries, Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

• When we run a program, a process is created
  – E.g. ./a.out, ./axpy, etc
  – fork () system call
Threads

- Threads use, and exist within, the process resources
- Scheduled and run as independent entities
- Duplicate only the bare essential resources that enable them to exist as executable code
Threads

- A thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.

- Multiple threads share the process resources
- A thread dies if the process dies
- "lightweight" for creating and terminating threads that for processes
POSIX threads (Pthreads)

- Threads used to implement parallelism in shared memory multiprocessor systems, such as SMPs
- Historically, hardware vendors have implemented their own proprietary versions of threads
  - Portability a concern for software developers.
- For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard.
  - Implementations that adhere to this standard are referred to as POSIX threads
The POSIX Thread API

• Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
  – Implemented with a pthread.h header/include file and a thread library

• Functionalities
  – Thread management, e.g. creation and joining
  – Thread synchronization primitives
    • Mutex
    • Condition variables
    • Reader/writer locks
    • Pthread barrier
  – Thread-specific data

• The concepts discussed here are largely independent of the API
  – Applied to other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.
PThread API

• `#include <pthread.h>`

<table>
<thead>
<tr>
<th>Routine Prefix</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_</td>
<td>Threads themselves and miscellaneous subroutines</td>
</tr>
<tr>
<td>pthread_attr_</td>
<td>Thread attributes objects</td>
</tr>
<tr>
<td>pthread_mutex_</td>
<td>Mutexes</td>
</tr>
<tr>
<td>pthread_mutexattr_</td>
<td>Mutex attributes objects.</td>
</tr>
<tr>
<td>pthread_cond_</td>
<td>Condition variables</td>
</tr>
<tr>
<td>pthread_condattr_</td>
<td>Condition attributes objects</td>
</tr>
<tr>
<td>pthread_key_</td>
<td>Thread-specific data keys</td>
</tr>
</tbody>
</table>

• `gcc -lpthread`
Thread Creation

• Initially, main() program comprises a single, default thread
  – All other threads must be explicitly created

```c
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void * arg);
```

• **thread**: An *opaque*, unique identifier for the new thread returned by the subroutine
• **attr**: An *opaque* attribute object that may be used to set thread attributes
  You can specify a thread attributes object, or NULL for the default values
• **start_routine**: the C routine that the thread will execute once it is created
• **arg**: A single argument that may be passed to `start_routine`. It must be passed by
  reference as a pointer cast of type `void`. NULL may be used if no argument is to be
  passed.

Opaque object: A letter is an opaque object to the mailman, and sender and receiver
know the information.
Thread Creation

- **pthread_create** creates a new thread and makes it executable, i.e. run immediately in theory
  - can be called any number of times from anywhere within your code
- Once created, threads are peers, and may create other threads
- There is no implied hierarchy or dependency between threads
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *thread_id) {
    long tid = (long)thread_id;
    printf("Hello World! It's me, thread %#ld\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    long t;

    for(t=0;t<NUM_THREADS;t++) {
        printf("In main: creating thread %ld\n", t);
        int rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t );
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
Terminating Threads

- `pthread_exit` is used to explicitly exit a thread
  - Called after a thread has completed its work and is no longer required to exist
- If `main()` finishes before the threads it has created
  - If exits with `pthread_exit()`, the other threads will continue to execute
  - Otherwise, they will be automatically terminated when `main()` finishes
- The programmer may optionally specify a termination status, which is stored as a void pointer for any thread that may join the calling thread
- Cleanup: the `pthread_exit()` routine does not close files
  - Any files opened inside the thread will remain open after the thread is terminated
Thread Attribute

```c
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void * arg);
```

- Attribute contains details about
  - whether scheduling policy is inherited or explicit
  - scheduling policy, scheduling priority
  - stack size, stack guard region size

- `pthread_attr_init` and `pthread_attr_destroy` are used to initialize/destroy the thread attribute object
- Other routines are then used to query/set specific attributes in the thread attribute object
Passing Arguments to Threads

• The `pthread_create()` routine permits the programmer to pass one argument to the thread start routine

• For cases where multiple arguments must be passed:
  – Create a structure which contains all of the arguments
  – Then pass a pointer to the object of that structure in the `pthread_create()` routine.
  – All arguments must be passed by reference and cast to `(void *)`

• Make sure that all passed data is thread safe: data racing
  – it can not be changed by other threads
  – It can be changed in a determinant way
    • Thread coordination
Example 2: Argument Passing

```c
#include <pthread.h>
#define NUM_THREADS 8

struct thread_data {
    int thread_id;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg) {
    int taskid;
    char *hello_msg;

    sleep(1);
    struct thread_data *my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    hello_msg = my_data->message;
    printf("Thread %d: %s\n", taskid, hello_msg);
    pthread_exit(NULL);
}
```
Example 2: Argument Passing

```c
int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int t;
    char *messages[NUM_THREADS];
    messages[0] = "English: Hello World!";
    messages[1] = "French: Bonjour, le monde!";
    messages[3] = "Klingon: Nuq neH!";
    messages[4] = "German: Guten Tag, Welt!";
    messages[5] = "Russian: Zdravstvytye, mir!";
    messages[6] = "Japan: Sekai e konnichiwa!";
    messages[7] = "Latin: Orbis, te saluto!";

    for(t=0; t<NUM_THREADS; t++) {
        struct thread_data * thread_arg = &thread_data_array[t];
        thread_arg->thread_id = t;
        thread_arg->message = messages[t];
        pthread_create(&threads[t], NULL, PrintHello, (void *) thread_arg);
    }
    pthread_exit(NULL);
}
```

Thread 3: Klingon: Nuq neH!
Thread 0: English: Hello World!
Thread 1: French: Bonjour, le monde!
Thread 2: Spanish: Hola al mundo
Thread 5: Russian: Zdravstvytye, mir!
Thread 4: German: Guten Tag, Welt!
Thread 6: Japan: Sekai e konnichiwa!
Thread 7: Latin: Orbis, te saluto!
Wait for Thread Termination

Suspend execution of calling thread until thread terminates

```c
#include <pthread.h>
int pthread_join(
    pthread_t thread,
    void **value_ptr);
```

- **thread**: the joining thread
- **value_ptr**: ptr to location for return code a terminating thread passes to `pthread_exit`

It is a logical error to attempt simultaneous multiple joins on the same thread
#include <pthread.h>
define NUM_THREADS 4

void *BusyWork(void *t) {
    int i;
    long tid = (long)t;
    double result=0.0;
    printf("Thread %ld starting...
", tid);
    for (i=0; i<1000000; i++) {
        result = result + sin(i) * tan(i);
    }
    printf("Thread %ld done. Result = %e\n", tid, result);
    pthread_exit((void*) t);
}

Example 3: Pthread Joining
Example 3: Pthread joining

```c
int main (int argc, char *argv[])
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    long t;
    void *status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(t=0; t<NUM_THREADS; t++) {
        printf("Main: creating thread %ld\n", t);
        pthread_create(&thread[t], &attr, BusyWork, (void*)t);
    }

    /* Free attribute and wait for the other threads */
    pthread_attr_destroy(&attr);
    for(t=0; t<NUM_THREADS; t++) {
        pthread_join(thread[t], &status);
        printf("Main: joined with thread %ld, status: %ld\n", t, (long)status);
    }

    printf("Main: program completed. Exiting.\n");
    pthread_exit(NULL);
}
```
Shared Memory and Threads

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.
Thread Consequences

• Shared State!
  – Accidental changes to global variables can be fatal.
  – Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads
  – Two pointers having the same value point to the same data
  – Reading and writing to the same memory locations is possible
  – Therefore requires explicit synchronization by the programmer

• Many library functions are not thread-safe
  – Library Functions that return pointers to static internal memory. E.g. gethostbyname()

• Lack of robustness
  – Crash in one thread will crash the entire process
Thread-safeness

- Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions

- Example: an application creates several threads, each of which makes a call to the same library routine:
  - This library routine accesses/modifies a global structure or location in memory.
  - As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
  - If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.
Thread-safeness
Thread-safeness

The implication to users of external library routines:

• If you aren't 100% certain the routine is thread-safe, then you take your chances with problems that could arise.

• Recommendation
  – Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness.
  – When in doubt, assume that they are not thread-safe until proven otherwise
  – This can be done by "serializing" the calls to the uncertain routine, etc.
Example 4: Data Racing

```c
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *thread_id) { /* thread func */
    long tid = *((long*)thread_id);
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    long t;
    for(t=0;t<NUM_THREADS;t++) {
        printf("In main: creating thread %ld\n", t);
        int rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #3!
Hello World! It's me, thread #3!
Hello World! It's me, thread #3!
In main: creating thread 4
Hello World! It's me, thread #4!
Hello World! It's me, thread #5!
Why Pthreads (not processes)?

• The primary motivation
  – To realize potential program performance gains
• Compared to the cost of creating and managing a process
  – A thread can be created with much less OS overhead
• Managing threads requires fewer system resources than managing processes
• All threads within a process share the same address space
• Inter-thread communication is more efficient and, in many cases, easier to use than inter-process communication
pthread_create vs fork

- Timing results for the `fork()` subroutine and the `pthreads_create()` subroutine
  - Timings reflect 50,000 process/thread creations
  - units are in seconds
  - no optimization flags

<table>
<thead>
<tr>
<th>Platform</th>
<th>fork()</th>
<th>pthread_create()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>user</td>
</tr>
<tr>
<td>AMD 2.4 GHz Opteron (8cpus/node)</td>
<td>41.07</td>
<td>60.08</td>
</tr>
<tr>
<td>IBM 1.9 GHz POWER5 p5-575 (8cpus/node)</td>
<td>64.24</td>
<td>30.78</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4 (8cpus/node)</td>
<td>104.05</td>
<td>48.64</td>
</tr>
<tr>
<td>INTEL 2.4 GHz Xeon (2 cpus/node)</td>
<td>54.95</td>
<td>1.54</td>
</tr>
<tr>
<td>INTEL 1.4 GHz Itanium2 (4 cpus/node)</td>
<td>54.54</td>
<td>1.07</td>
</tr>
</tbody>
</table>
Why pthreads

- Potential performance gains and practical advantages over non-threaded applications:
  - Overlapping CPU work with I/O
    - For example, a program may have sections where it is performing a long I/O operation
    - While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.

- Priority/real-time scheduling
  - Tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.

- Asynchronous event handling
  - Tasks which service events of indeterminate frequency and duration can be interleaved
  - For example, a web server can both transfer data from previous requests and manage the arrival of new requests.
AXPY with PThreads

- \( y = \alpha \cdot x + y \)
  - \( x \) and \( y \) are vectors of size \( N \)
  - In C, \( x[N], y[N] \)
  - \( \alpha \) is scalar

- Decomposition and mapping to pthreads

```c
void dist(int tid, int N, int num_tasks, int *Nt, int *start) {
    int remain = N % num_tasks;
    int esize = N / num_tasks;
    if (tid < remain) {
        *Nt = esize + 1;
        *start = *Nt * tid;
    } else {
        *Nt = esize;
        *start = esize * tid + remain;
    }
}

void axpy_dist(int N, REAL Y[], REAL X[], REAL a, int num_tasks) {
    int tid;
    for (tid = 0; tid < num_tasks; tid++) {
        int Nt, start;
        dist(tid, N, num_tasks, &Nt, &start);
        axpy_base_sub(start, Nt, N, Y, X, a);
    }
}
```

A task will be mapped to a pthread
struct axpy_dist_pthread_data {
    int Nt;
    int start;
    int N;
    REAL *Y;
    REAL *X;
    REAL a;
};

void * axpy_thread_func(void * axpy_thread_arg) {
    struct axpy_dist_pthread_data * arg = (struct axpy_dist_pthread_data *) axpy_thread_arg;
    axpy_base_sub(arg->start, arg->Nt, arg->N, arg->Y, arg->X, arg->a);
    pthread_exit(NULL);
}

void axpy_dist_pthread(int N, REAL Y[], REAL X[], REAL a, int num_tasks) {
    struct axpy_dist_pthread_data pthread_data_array[num_tasks];
    pthread_t task_threads[num_tasks];
    int tid;
    for (tid = 0; tid < num_tasks; tid++) {
        int Nt, start;
        dist(tid, N, num_tasks, &Nt, &start);
        struct axpy_dist_pthread_data *task_data = &pthread_data_array[tid];
        task_data->start = start;
        task_data->Nt = Nt;
        task_data->a = a;
        task_data->X = X;
        task_data->Y = Y;
        task_data->N = N;
        pthread_create(&task_threads[tid], NULL, axpy_thread_func, (void*)task_data);
    }
    for (tid = 0; tid < num_tasks; tid++) {
        pthread_join(task_threads[tid], NULL);
    }
}