Lecture 5: Parallel Programming with Thread (Part 1)
Outline

• Shared address-space programming models
• Thread-based programming
  ➢ The POSIX Thread API
• Directive-based programming
  ➢ OpenMP MPI
Principles of Shared Address Space Programming

- Some (or all) of the memory is accessible to all processes.
- Requires primitives to
  - Declare shared and private variables.
  - Spawn and combine processes.
- Communication among processes through reads or writes of shared variables.
  - Use mutexes, semaphores, locks, etc. to control access to shared variables.
- Synchronization of several processes using barriers.
Programming Paradigms

- Based on UNIX-like processes:
  - All data is private to processes, unless otherwise specified.
  - High overhead.

- Based on light-weight processes or threads:
  - All data are global except the thread stack or locally declared variables.
  - Managed by threads or light-weight process library in user space.
  - Low overhead.

- Based on directives:
  - Extends the thread model.
  - Creates and synchronize threads automatically.
Threads Basics

- A thread is a stream of instructions that can be executed independently.
- Each process has one or more threads.
Threads vs processes

• How threads and processes are similar
  ➢ Each has its own logical control flow.
  ➢ Each can run concurrently.
  ➢ Each is context switched.

• How threads and processes are different
  ➢ Threads share code and data, processes (typically) do not.
  ➢ Threads are somewhat less expensive than processes.
    ➢ process control (creating and termination) is more expensive than thread control.
    ➢ Linux/Pentium III numbers:
      • 20K cycles to create and terminate a process.
      • 10K cycles to create and terminate a thread.
# Thread Model

**Items shared by all threads of a process:**

- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

**Items private to each thread:**

- Program counter
- Registers
- Stack
- State
Threads Example

Matrix multiply:

\[
\begin{align*}
\text{for (row = 0; row < n; row++)} & \\
\text{for (column = 0; column < n; column++)} & \\
c[row][column] = & \\
\text{dot_product(get_row(a, row), get_col(b, col))}; & 
\end{align*}
\]

Each of \(n^2\) iterations can be executed independently using a thread per iteration:

\[
\begin{align*}
\text{for (row = 0; row < n; row++)} & \\
\text{for (column = 0; column < n; column++)} & \\
c[row][column] = & \\
\text{create_thread(dot_product(get_row(a, row), get_col(b, col))}); & 
\end{align*}
\]
Advantages of Threads

• Software portability:
  ➢ Parallel processing is easy: same program for single and multiprocessor machines.
  ➢ POSIX threads are commonly used.

• Latency hiding:
  ➢ Increased throughput in I/O bound applications.
  ➢ Server can respond to new requests while servicing the existing ones by spawning threads as needed.

• Scheduling and load balancing:
  ➢ Many threads can be spawned with small amount of work per thread.
  ➢ Easy to balance the load on processors in unstructured, dynamic applications.

• Easy of programming, widespread use.
POSIX Threads: Creation

Pthread creation:

```c
#include <pthread.h>

int pthread_create( pthread_t *thread_handle,
                    const pthread_attr_t *attribute,
                    void* (*thread_function)(void *),
                    void *arg);
```

Creates a single thread that corresponds to the invocation of the function `thread_function`. `thread_handle` contains the unique id of the newly created thread (if successful). `attribute` specifies the stack size, scheduling policy etc. `arg` is a pointer to the arguments of `thread_function`. On successful creation of a thread, 0 is returned.
POSIX Threads: Wait and Termination

Pthread exit:

```c
#include <pthread.h>

int pthread_exit( void *value_ptr );
```

Pthread wait:

```c
#include <pthread.h>

int pthread_join( pthread_t thread, void **ptr);
```

Waits for the termination of the thread whose id is given by `pthread_create()`. If successful the value passed to `pthread_exit` is returned in the location pointed by `ptr`. 
/ * hello.c – Pthreads "hello, world" program * /
#include <pthread.h>

void *thread(void *vargp);

int main() {
  pthread_t tid;

  pthread_create(&tid, NULL, thread, NULL);
  pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  pthread_exit(0);
}

Thread attributes (usually NULL)
Thread arguments (void *p)
return value (void **p)
Execution of “hello, world”

main thread

create peer thread

wait for peer thread to terminate

print output

terminate thread via pthread_exit()

exit() terminates main thread and any peer threads

peer thread
Example: Computing $\pi$

- **Algorithm**
  - Generate random numbers in a unit square.
  - The largest circle that can be inscribed will have radius $\frac{1}{2}$ and area $\pi/4$.
  - Compute the fraction of the points that fall within the circle.
  - Multiply that fraction by 4 to get the value of $\pi$.
  - Use multiple threads to speedup the computation of the fraction.

- **Two approaches:**
  - Each thread computes the fraction locally, and the results are combined.
  - All threads update global variables while computing the fraction.
Example: Computing $\pi$

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

#define MAX_THREADS 512

void *compute_pi (void *);

int total_hits, total_misses, hits[MAX_THREADS],
    sample_points, sample_points_per_thread, num_threads;

main() {
    int i;
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    double computed_pi;
    double time_start, time_end;
    struct timeval tv;
    struct timezone tz;

    pthread_attr_init (&attr);
    pthread_attr_setscope (&attr, PTHREAD_SCOPE_SYSTEM);
    printf("Enter number of sample points: ");
    scanf("%d", &sample_points);
    printf("Enter number of threads: ");
    scanf("%d", &num_threads);

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
        (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
    }
    computed_pi = (double)total_hits / (double)total_misses;
    printf("Computed pi = %f\n", computed_pi);
}```
Example: Computing $\pi$ (Cont’d)

```c
    total_hits += hits[i];
}
computed_pi = 4.0*(double) total_hits /
    ((double)(sample_points));
gettimeofday(&tv, &tz);
time_end = (double)tv.tv.tv_sec +
    (double)tv.tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x = (double)(rand_r(&seed)) / (double)((2<<14)-1);
        rand_no_y = (double)(rand_r(&seed)) / (double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *(hit_pointer)++;  // Highlighted line
    *hit_pointer = local_hits;
    pthread_exit(0);
}
A Performance Issue

Running on 4-processor SGI Origin 2000

Spaced_xx = time to compute $\pi$ when threads update global variables. (line 64:

*(hit_pointer)++;)

The spike with 4 threads is caused by false sharing of global data.
Synchronization Primitives: Mutex-Locks

Example: code executed by several threads simultaneously.
/*each thread tries to update variable best_cost*/
if( my_cost < best_cost )
    best_cost = my_cost;

Assume initially: best_cost = 100, my_cost = 50 at thread1 and 75 at thread2
Final value for best_cost = ?

• Non-deterministic execution (race condition).
• Test-and-Update must be an atomic operation.
• Mutexes can solve the problem!
Synchronization Primitives: Mutex-Locks

- Mutex-locks have two states: locked and unlocked.
- At any point in time only one thread can lock a mutex-lock.

```c
int pthread_mutex_lock(
    pthread_mutex_t *mutex_lock);

int pthread_mutex_unlock(
    pthread_mutex_t *mutex_lock);

int pthread_mutex_init (
    pthread_mutex_t *mutex_lock
    const pthread_mutexattr_t *lock_attr );
```
Example: Computing the minimum

```c
#include <pthread.h>
void *find_min(void *list_ptr);
pthread_mutex_t minimum_value_lock;
int minimum_value, partial_list_size;

main() {
    /* declare and initialize data structures and list */
    minimum_value = MIN_INT;
    pthread_init();
    pthread_mutex_init(&minimum_value_lock, NULL);

    /* initialize lists, list_ptr, and partial_list_size */
    /* create and join threads here */
}

void *find_min(void *list_ptr) {
    int *partial_list_pointer, my_min, i;
    my_min = MIN_INT;
    partial_list_pointer = (int *) list_ptr;
    for (i = 0; i < partial_list_size; i++)
        if (partial_list_pointer[i] < my_min)
            my_min = partial_list_pointer[i];
    /* lock the mutex associated with minimum_value and
    update the variable as required */
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
        minimum_value = my_min;
    /* and unlock the mutex */
    pthread_mutex_unlock(&minimum_value_lock);
    pthread_exit(0);
}
```
Example: Producer-Consumer

```c
pthread_mutex_t task_queue_lock;
int task_available;

main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    /* create and join producer and consumer threads */
}

void *producer(void *producer_thread_data) {
    int inserted;
    struct task my_task;
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {

            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1;
                inserted = 1;

            } else {
                /* continue producer processing */
            }

        }
    }
}

void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {

            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0;
                extracted = 1;

            } else {
                /* continue consumer processing */
            }

        }
        pthread_mutex_unlock(&task_queue_lock);
        process_task(my_task);
    }
}
```

Can we move this before lock()? Any benefit?

Is this lock() really needed? Multiple producers/consumers?
**Example: one-producer-one-consumer (Ver 2.0)**

```c
pthread_mutex_t task_queue_lock;
int task_available;
main() {
    /* declarations and initializations */
    task_available = 0;
pthread_mutex_init(&task_queue_lock, NULL);
    /* create and join producer and consumer threads */
}

void *producer(void *producer_thread_data) {
    struct task my_task;
    while (!done()) {
        create_task(&my_task);
        while (task_available == 1);
        insert_into_queue(my_task);
        task_available = 1;
    }
}

void *consumer(void *consumer_thread_data) {
    struct task my_task;
    while (!done()) {
        while (task_available == 0);
        extract_from_queue(my_task);
        task_available = 0;
    }
}
```
Alleviating Locking Overheads

- Critical sections must be executed by threads one after the other.
- If large segments of code are in the critical sections => significant performance degradation.
- Avoid the idling overhead by using:

```c
int pthread_mutex_trylock(
    pthread_mutex_t *mutex_lock);
```

- Attempts a lock on `mutex_lock`.
- If the lock is successful it returns zero.
- If not, instead of blocking the thread it returns a value `EBUSY`.
Example: k Matches in a List

```c
void *find_entries(void *start_pointer) {
    /* This is the thread function */

    struct database_record *next_record;
    int count;
    current_pointer = start_pointer;
    do {
        next_record = find_next_entry(current_pointer);
        count = output_record(next_record);
    } while (count < requested_number_of_records);
}

int output_record(struct database_record *record_ptr) {
    int count;
    pthread_mutex_lock(&output_count_lock);
    output_count ++;
    count = output_count;
    pthread_mutex_unlock(&output_count_lock);

    if (count <= requested_number_of_records)
        print_record(record_ptr);
    return (count);
}
```

```
t1 = lock-unlock time

T_{total} = (t1 + t2)n_{max}
```
**Synchronization using Condition Variables**

*Idea:* Instead of polling the lock, suspend the thread until a specified data reaches a predefined state.

*E.g. Producer-consumer:*

- Associate a condition variable with the predicate
  ```
  task_available == 1.
  ```
- When the predicate becomes true => signal the threads waiting for this condition variable.

A condition variable has always a mutex associated with it.

A thread locks this mutex and tests the predicate using:

```c
int pthread_cond_wait( pthread_cond_t *cond,
                      pthread_mutex_t *mutex );
```

- Blocks the thread until a signal is received from another thread or OS;
- Release the lock on mutex before blocking;
- When the thread is released on a signal, it waits to reacquire the lock on mutex before resuming execution.
Synchronization using Condition Variables

*Signaling* another thread:

```c
int pthread_cond_signal(pthread_cond_t *cond);
```

Unblocks at least one thread that is currently waiting on the condition variable `cond`.

Functions for *initializing* and *destroying* a condition variable:

```c
int pthread_cond_init( pthread_cond_t *cond
const pthread_condattr_t *attr );

int pthread_cond_destroy( pthread_cond_t *cond);
```
Example: Producer-Consumer using condition variable

```c
void *producer(void *producer_thread_data) {
    int inserted;
    while (!done()) {
        create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait(&cond_queue_empty,
                              &task_queue_cond_lock);
        insert_into_queue();
        task_available = 1;
        pthread_cond_signal(&cond_queue_full);
        pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

```c
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait(&cond_queue_full,
                              &task_queue_cond_lock);
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock
process_task(my_task);
    }
    return NULL;
}
```

How about using ‘if’?
Synchronization using Condition Variables

*Wakeup* all threads that are waiting on a condition variable:

```c
int pthread_cond_broadcast(
    pthread_cond_t *cond);
```
can be used to implement barrier synchronization.

*Time-out wait* on a condition variable:

```c
int pthread_cond_timedwait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime);
```