Introduction to Parallel Computing

George Karypis

Programming Shared Address Space Platforms

Outline

- Shared Address-Space Programming Models
- Thread-based programming
  - POSIX API/Pthreads
- Directive-based programming
  - OpenMP API

Shared Memory Programming

- Communication is implicitly specified
- Focus on constructs for expressing concurrency and synchronization
  - Minimize data-sharing overheads

Commonly Used Models

- Process model
  - All memory is local unless explicitly specified/allocated as shared.
  - Unix processes.
- Light-weight process/thread model
  - All memory is global and can be accessed by all the threads.
    - Runtime stack is local but it can be shared.
  - POSIX thread API/Pthreads
    - Low-level & system-programming flavor to it.
- Directive model
  - Concurrency is specified in terms of high-level compiler directives.
    - High-level constructs that leave some of the error-prone details to the compiler.
  - OpenMP has emerged as a standard.

POSIX API/Pthreads

- Has emerged as the de-facto standard supported by most OS vendors.
  - Aids in the portability of threaded applications.
- Provides a good set of functions that allow for the creation, termination, and synchronization of threads.
  - However, these functions are low-level and the API is missing some high-level constructs for efficient data-sharing
    - There are no collective communication operation like those provided by MPI.

Pthreads Overview

- Thread creation & termination
- Synchronization primitives
  - Mutual exclusion locks
  - Conditional variables
- Object attributes

Thread Creation & Termination

```
#include <pthread.h>
int
pthread_create (
    pthread_t   *thread_handle,
    const pthread_attr_t   *attribute,
    void *   (*thread_function)(void *),
    void   *arg);
```

The pthread_create function creates a single thread that corresponds to the invocation of the function thread_function (and any other functions called by thread_function). On successful creation of a thread, a unique identifier is associated with the thread and assigned to the location pointed to by thread_handle. The thread has the attributes described by the attribute argument. When this argument is NULL, a thread with default attributes is created. We will discuss the attribute parameter in detail in Section 7.6. The arg field specifies a pointer to the argument to function thread_function. This argument is typically used to pass the workspace and other thread-specific data to a thread. In the compute_pi example, it is used to pass an integer id that is used as a seed for randomization. The thread_handle variable is written before the the function pthread_create returns; and the new thread is ready for execution as soon as it is created. If the thread is scheduled on the same processor, the new thread may, in fact, preempt its creator. This is important to note because all thread initialization procedures must be completed before creating the thread. Otherwise, errors may result based on thread scheduling. This is a very common class of errors caused by race conditions for data access that shows itself in some execution instances, but not in others. On successful creation of a thread, pthread_create returns 0; else it returns an error code. The reader is referred to the Pthreads specification for a detailed description of the error-codes.

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

A call to this function waits for the termination of the thread whose id is given by thread. On a successful call to pthread_join, the value passed to pthread_exit is returned in the location pointed to by ptr. On successful completion, pthread_join returns 0, else it returns an error-code.

Computing the value of π

```
#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);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)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 = local_hits;
    pthread_exit(0);
}
```

Synchronization Primitives

- Access to shared variable need to be controlled to remove race conditions and ensure serial semantics.

```
/* each thread tries to update variable best_cost as follows */
if (my_cost < best_cost)
    best_cost = my_cost;
```

To understand the problem with shared data access, let us examine one execution instance of the above code fragment. Assume that there are two threads, the initial value of `best_cost` is 100, and the values of `my_cost` are 50 and 75 at threads t1 and t2, respectively. If both threads execute the condition inside the `if` statement concurrently, then both threads enter the `then` part of the statement. Depending on which thread executes first, the value of `best_cost` at the end could be either 50 or 75. There are two problems here: the first is the non-deterministic nature of the result; second, and more importantly, the value 75 of `best_cost` is inconsistent in the sense that no serialization of the two threads can possibly yield this result. This is an undesirable situation, sometimes also referred to as a race condition (so called because the result of the computation depends on the race between competing threads).

Mutual Exclusion Locks

- Pthreads provide a special variable called a *mutex* lock that can be used to guard critical sections of the program.
  - The idea is for a thread to acquire the lock before entering the critical section and release on exit.
  - If the lock is already owned by another thread, the thread blocks until the lock is released.
- Lock represent serialization points, so too many locks can decrease the performance.

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

A call to this function attempts a lock on the mutex-lock `mutex_lock`. (The data type of a `mutex_lock` is predefined to be `pthread_mutex_t`.) If the mutex-lock is already locked, the calling thread blocks; otherwise the mutex-lock is locked and the calling thread returns. A successful return from the function returns a value 0. Other values indicate error conditions such as deadlocks.

```
int
pthread_mutex_unlock (
    pthread_mutex_t *mutex_lock);
```

On calling this function, in the case of a normal mutex-lock, the lock is relinquished and one of the blocked threads is scheduled to enter the critical section. The specific thread is determined by the scheduling policy. There are other types of locks (other than normal locks), which are discussed in Section 7.6 along with the associated semantics of the function `pthread_mutex_unlock`. If a programmer attempts a `pthread_mutex_unlock` on a previously unlocked mutex or one that is locked by another thread, the effect is undefined.

```
int
pthread_mutex_init (
    pthread_mutex_t    *mutex_lock,
    const pthread_mutexattr_t    *lock_attr);
```

This function initializes the mutex-lock `mutex_lock` to an unlocked state. The attributes of the mutex-lock are specified by `lock_attr`. If this argument is set to NULL, the default mutex-lock attributes are used (normal mutex-lock). Attributes objects for threads are discussed in greater detail in Section 7.6.

It is often possible to reduce the idling overhead associated with locks using an alternate function, `pthread_mutex_trylock`. This function attempts a lock on `mutex_lock`. If the lock is successful, the function returns a zero. If it is already locked by another thread, instead of blocking the thread execution, it returns a value `EBUSY`. This allows the thread to do other work and to poll the mutex for a lock. Furthermore, `pthread_mutex_trylock` is typically much faster than `pthread_mutex_lock` on typical systems since it does not have to deal with queues associated with locks for multiple threads waiting on the lock. The prototype of `pthread_mutex_trylock` is:

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

Computing the minimum element of an array.

```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);
}
```

Producer Consumer Queues

```
pthread_mutex_t task_queue_lock;
int task_available;

/* other shared data structures here */

main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    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) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}

void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0;
                extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);
    }
}
```

Conditional Variables

- Waiting-queue like synchronization principles.
  - Based on the outcome of a certain condition a thread may attach itself to a waiting queue.
  - At a later point in time, another thread that change the outcome of the condition, will wake up one/all of the threads so that they can see if they can proceed.
- Conditional variables are always associated with a mutex lock.

Conditional Variables API

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

A call to this function blocks the execution of the thread until it receives a signal from another thread or is interrupted by an OS signal. In addition to blocking the thread, the pthread_cond_wait function releases the lock on mutex. This is important because otherwise no other thread will be able to work on the shared variable task_available and the predicate would never be satisfied. When the thread is released on a signal, it waits to reacquire the lock on mutex before resuming execution. It is convenient to think of each condition variable as being associated with a queue. Threads performing a condition wait on the variable relinquish their lock and enter the queue. When the condition is signaled (using pthread_cond_signal), one of these threads in the queue is unblocked, and when the mutex becomes available, it is handed to this thread (and the thread becomes runnable).

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

The function unblocks at least one thread that is currently waiting on the condition variable cond. The producer then relinquishes its lock on mutex by explicitly calling pthread_mutex_unlock, allowing one of the blocked consumer threads to consume the task.

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

The function pthread_cond_init initializes a condition variable (pointed to by cond) whose attributes are defined in the attribute object attr. Setting this pointer to NULL assigns default attributes for condition variables. If at some point in a program a condition variable is no longer required, it can be discarded using the function pthread_cond_destroy. These functions for manipulating condition variables enable us to rewrite our producer-consumer segment as follows:

In the above example, each task could be consumed by only one consumer thread. Therefore, we choose to signal one blocked thread at a time. In some other computations, it may be beneficial to wake all threads that are waiting on the condition variable as opposed to a single thread. This can be done using the function pthread_cond_broadcast.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

It is often useful to build time-outs into condition waits. Using the function pthread_cond_timedwait, a thread can perform a wait on a condition variable until a specified time expires. At this point, the thread wakes up by itself if it does not receive a signal or a broadcast. The prototype for this function is:

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

If the absolute time abstime specified expires before a signal or broadcast is received, the function returns an error message. It also reacquires the lock on mutex when it becomes available.

Producer Consumer Example with Conditional Variables

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;

/* other data structures here */

main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* create and join producer and consumer threads */
}

void *producer(void *producer_thread_data) {
    int inserted;
    while (!done()) {
        create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 1)
            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);
    }
}

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);
    }
}
```

Attribute Objects

- Various attributes can be associated with threads, locks, and conditional variables.
  - Thread attributes:
    - scheduling parameters
    - stack size
    - detached state
  - Mutex attributes:
    - normal
      - only a single thread is allowed to lock it.
      - if a threads tries to lock it twice a deadlock occurs.
    - recursive
      - a thread can lock the mutex multiple time.
      - each successive lock increments a counter and each successive release decrements the counter.
      - a thread can lock a mutex only if its counter is zero.
    - errorcheck
      - like normal but an attempt to lock it again by the same thread leads to an error.
  - The book and the Posix thread API provide additional details.

OpenMP

- A standard directive-based shared memory programming API
  - C/C++/Fortran versions of the API exist
- API consists of a set of compiler directive along with a set of API functions.

```
#pragma omp directive [clause list]
```

Parallel Region

- Parallel regions are specified by the `parallel` directive:

```
#pragma omp parallel [clause list]
/* structured block */

```

- The *clause list* contains information about:
  - conditional parallelization
    - `if (scalar expression)`
  - degree of concurrency
    - `num_threads (integer expression)`
  - data handling
    - `private (var list), firstprivate (var list), shared (var list)`
    - `default(shared|private|none)`

```
#pragma omp parallel if (is_parallel == 1) num_threads(8) \
                     private (a) shared (b) firstprivate(c)
{
    /* structured block */
}
```

Reduction clause

Just as `firstprivate` specifies how multiple local copies of a variable are initialized inside a thread, the `reduction` clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit. The usage of the `reduction` clause is `reduction (operator: variable list)`. This clause performs a reduction on the scalar variables specified in the list using the `operator`. The variables in the list are implicitly specified as being private to threads. The `operator` can be one of `+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`.

Example 7.10 Using the `reduction` clause

```
#pragma omp parallel reduction(+: sum) num_threads(8)
{
    /* compute local sums here */
}
/* sum here contains sum of all local instances of sums */
```

In this example, each of the eight threads gets a copy of the variable `sum`. When the threads exit, the sum of all of these local copies is stored in the single copy of the variable (at the master thread). ■

Computing the value of π

```
/* ******************************************************
   An OpenMP version of a threaded program to compute PI.
   ****************************************************** */

    #pragma omp parallel default(private) shared (npoints) \
                         reduction(+: sum) num_threads(8)
    {
      num_threads = omp_get_num_threads();
      sample_points_per_thread = npoints / num_threads;
      sum = 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)
            sum ++;
      }
    }
```

Specifying concurrency

- Concurrent tasks are specified using the `for` and `sections` directives.
  - The `for` directive splits the iterations of a loop across the different threads.
  - The `sections` directive assigns each thread to explicitly identified tasks.

The for directive

The `for` directive is used to split parallel iteration spaces across threads. The general form of a `for` directive is as follows:

```
#pragma omp for [clause list]
    /* for loop */
```

The clauses that can be used in this context are: `private`, `firstprivate`, `lastprivate`, `reduction`, `schedule`, `nowait`, and `ordered`. The first four clauses deal with data handling and have identical semantics as in the case of the `parallel` directive. The `lastprivate` clause deals with how multiple local copies of a variable are written back into a single copy at the end of the parallel `for` loop. When using a `for` loop (or `sections` directive as we shall see) for farming work to threads, it is sometimes desired that the last iteration (as defined by serial execution) of the `for` loop update the value of a variable. This is accomplished using the `lastprivate` directive.

The `schedule` clause of the `for` directive deals with the assignment of iterations to threads. The general form of the `schedule` directive is `schedule(scheduling_class[, parameter])`. OpenMP supports four scheduling classes: `static`, `dynamic`, `guided`, and `runtime`.

Often, it is desirable to have a sequence of `for`-directives within a parallel construct that do not execute an implicit barrier at the end of each `for` directive. OpenMP provides a clause – `nowait`, which can be used with a `for` directive to indicate that the threads can proceed to the next statement without waiting for all other threads to complete the `for` loop execution. This is illustrated in the following example:

An example

```
#pragma omp parallel default(private) shared (npoints) \
                     reduction(+: sum) num_threads(8)
{
  sum = 0;
  #pragma omp for
  for (i = 0; i < npoints; 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)
        sum ++;
  }
}
```

The loop index for the `for` directive is assumed to be private.

More one `for` directive

- Loop scheduling schemes
  - `schedule(static[, chunk-size])`
    - splits the iterations into consecutive chucks of size chunk-size and assigns them in round-robin fashion.
  - `schedule(dynamic [, chunk-size])`
    - splits the iterations into consecutive chunks of size chunk-size and gives to each thread a chunk as soon as it finishes processing its previous chunk.
  - `schedule(guided [, chunk-size])`
    - like dynamic but the chunk-size is reduced exponentially as each chunk is dispatched to a thread.
  - `schedule(runtime)`
    - is determined by reading an environmental variable.

Restrictions on the `for` directive

- For loops must not have break statements.
- Loop control variables must be integers.
- The initialization expression of the control variable must be an integer.
- The logical expression must be one of < <=, >, >=.
- The increment expression must have integer increments and decrements.

The `sections` directive

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

If there are three threads, each section (in this case, the associated task) is assigned to one thread. At the end of execution of the assigned section, the threads synchronize (unless the `nowait` clause is used). Note that it is illegal to branch in and out of `section` blocks.

Synchronization Directives

- barrier directive

```
#pragma omp barrier
```

- single/master directives

```
#pragma omp single [clause list]
        structured block
```

```
#pragma omp master
        structured block
```

- critical/atomic directives

```
#pragma omp critical [(name)]
        structured block
```

- ordered directive

```
#pragma omp ordered
    structured block
```