CHAPTER 7

Programming Shared Address Space Platforms

Explicit parallel programming requires specification of parallel tasks along with their interactions. These interactions may be in the form of synchronization between concurrent tasks or communication of intermediate results. In shared address space architectures, communication is implicitly specified since some (or all) of the memory is accessible to all the processors. Consequently, programming paradigms for shared address space machines focus on constructs for expressing concurrency and synchronization along with techniques for minimizing associated overheads. In this chapter, we discuss shared-address-space programming paradigms along with their performance issues and related extensions to directive-based paradigms.

Shared address space programming paradigms can vary on mechanisms for data sharing, concurrency models, and support for synchronization. Process based models assume that all data associated with a process is private, by default, unless otherwise specified (using UNIX system calls such as `shmget` and `shmat`). While this is important for ensuring protection in multiuser systems, it is not necessary when multiple concurrent aggregates are cooperating to solve the same problem. The overheads associated with enforcing protection domains make processes less suitable for parallel programming. In contrast, lightweight processes and threads assume that all memory is global. By relaxing the protection domain, lightweight processes and threads support much faster manipulation. As a result, this is the preferred model for parallel programming and forms the focus of this chapter. Directive based programming models extend the threaded model by facilitating creation and synchronization of threads. In this chapter, we study various aspects of programming using threads and parallel directives.

7.1 Thread Basics

A ***t***hread is a single stream of control in the flow of a program. We initiate threads with a simple example:

Example 7.1 What are threads?

Consider the following code segment that computes the product of two dense matrices of size $n \times n$.

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

The `for` loop in this code fragment has n^2 iterations, each of which can be executed independently. Such an independent sequence of instructions is referred to as a thread. In the example presented above, there are n^2 threads, one for each iteration of the for-loop. Since each of these threads can be executed independently of the others, they can be scheduled concurrently on multiple processors. We can transform the above code segment as follows:

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(dot_product(get_row(a, row),
                                      get_col(b, col)));
```

Here, we use a function, `create_thread`, to provide a mechanism for specifying a C function as a thread. The underlying system can then schedule these threads on multiple processors. ■

Logical Memory Model of a Thread To execute the code fragment in Example 7.1 on multiple processors, each processor must have access to matrices a, b, and c. This is accomplished via a shared address space (described in Chapter **??**). All memory in the logical machine model of a thread is globally accessible to every thread as illustrated in Figure 7.1(a). However, since threads are invoked as function calls, the stack corresponding to the function call is generally treated as being local to the thread. This is due to the liveness considerations of the stack. Since threads are scheduled at runtime (and no *a priori* schedule of their execution can be safely assumed), it is not possible to determine which stacks are live. Therefore, it is considered poor programming practice to treat stacks (thread-local variables) as global data. This implies a logical machine model illustrated in Figure 7.1(b), where memory modules M hold thread-local (stack allocated) data.

While this logical machine model gives the view of an equally accessible address space,

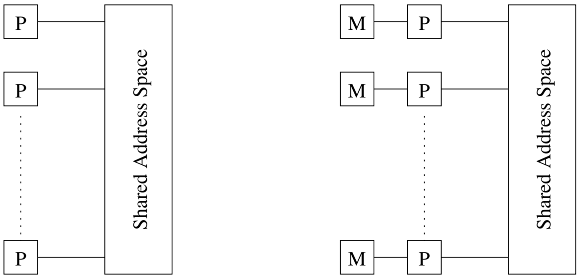

Figure 7.1 The logical machine model of a thread-based programming paradigm.

physical realizations of this model deviate from this assumption. In distributed shared address space machines such as the Origin 2000, the cost to access a physically local memory may be an order of magnitude less than that of accessing remote memory. Even in architectures where the memory is truly equally accessible to all processors (such as shared bus architectures with global shared memory), the presence of caches with processors skews memory access time. Issues of locality of memory reference become important for extracting performance from such architectures.

7.2 Why Threads?

Threaded programming models offer significant advantages over message-passing programming models along with some disadvantages as well. Before we discuss threading APIs, let us briefly look at some of these.

Software Portability Threaded applications can be developed on serial machines and run on parallel machines without any changes. This ability to migrate programs between diverse architectural platforms is a very significant advantage of threaded APIs. It has implications not just for software utilization but also for application development since supercomputer time is often scarce and expensive.

Latency Hiding One of the major overheads in programs (both serial and parallel) is the access latency for memory access, I/O, and communication. By allowing multiple threads to execute on the same processor, threaded APIs enable this latency to be hidden (as seen in Chapter **??**). In effect, while one thread is waiting for a communication operation, other threads can utilize the CPU, thus masking associated overhead.

Scheduling and Load Balancing While writing shared address space parallel programs, a programmer must express concurrency in a way that minimizes overheads of remote interaction and idling. While in many structured applications the task of allocating equal work to processors is easily accomplished, in unstructured and dynamic applications (such as game playing and discrete optimization) this task is more difficult. Threaded APIs allow the programmer to specify a large number of concurrent tasks and support system-level dynamic mapping of tasks to processors with a view to minimizing idling overheads. By providing this support at the system level, threaded APIs rid the programmer of the burden of explicit scheduling and load balancing.

Ease of Programming, Widespread Use Due to the aforementioned advantages, threaded programs are significantly easier to write than corresponding programs using message passing APIs. Achieving identical levels of performance for the two programs may require additional effort, however. With widespread acceptance of the POSIX thread API, development tools for POSIX threads are more widely available and stable. These issues are important from the program development and software engineering aspects.

7.3 The POSIX Thread API

A number of vendors provide vendor-specific thread APIs. The IEEE specifies a standard 1003.1c-1995, POSIX API. Also referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors. We will use the Pthreads API for introducing multithreading concepts. The concepts themselves are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well. All the illustrative programs presented in this chapter can be executed on workstations as well as parallel computers that support Pthreads.

7.4 Thread Basics: Creation and Termination

Let us start our discussion with a simple threaded program for computing the value of π.

Example 7.2 Threaded program for computing π

The method we use here is based on generating random numbers in a unit length square and counting the number of points that fall within the largest circle inscribed in the square. Since the area of the circle (πr^2) is equal to $\pi/4$, and the area of the square is 1×1, the fraction of random points that fall in the circle should approach $\pi/4$.

A simple threaded strategy for generating the value of π assigns a fixed number of points to each thread. Each thread generates these random points and keeps track of the number of points that land in the circle locally. After all threads finish execution, their counts are combined to compute the value of π (by calculating the fraction over all threads and multiplying by 4).

To implement this threaded program, we need a function for creating threads and waiting for all threads to finish execution (so we can accrue count). Threads can be created in the Pthreads API using the function pthread_create. The prototype of this function is:

```
#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.

In our program for computing the value of π, we first read in the desired number of threads, num_threads, and the desired number of sample points, sample_points. These points are divided equally among the threads. The program uses an array, hits, for assigning an integer id to each thread (this id is used as a seed for randomizing the random number generator). The same array is used to keep track of the number of hits (points inside the circle) encountered by each thread upon return. The program creates num_threads threads, each invoking the function compute_pi, using the pthread_create function.

Once the respective compute_pi threads have generated assigned number of random points and computed their hit ratios, the results must be combined to determine π. The main program must wait for the threads to run to completion. This is done using the function pthread_join which suspends execution of the calling thread until the specified thread terminates. The prototype of the `pthread_join` function is as follows:

```
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.

Once all threads have joined, the value of π is computed by multiplying the combined hit ratio by 4.0. The complete program is as follows:

```
#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);
}
```

Programming Notes The reader must note, in the above example, the use of the function `rand_r` (instead of superior random number generators such as `drand48`). The reason for this is that many functions (including `rand` and `drand48`) are not ***r***eentrant. Reentrant functions are those that can be safely called when another instance has been suspended in the middle of its invocation. It is easy to see why all thread functions must be reentrant because a thread can be preempted in the middle of its execution. If another thread starts executing the same function at this point, a non-reentrant function might not work as desired.

Performance Notes We execute this program on a four-processor SGI Origin 2000. The logarithm of the number of threads and execution time are illustrated in Figure 7.2 (the curve labeled "local"). We can see that at 32 threads, the runtime of the program is roughly 3.91 times less than the corresponding time for one thread. On a four-processor machine, this corresponds to a parallel efficiency of 0.98.

The other curves in Figure 7.2 illustrate an important performance overhead called ***f***alse sharing. Consider the following change to the program: instead of incrementing a local variable, `local_hits`, and assigning it to the array entry outside the loop, we now directly increment the corresponding entry in the `hits` array. This

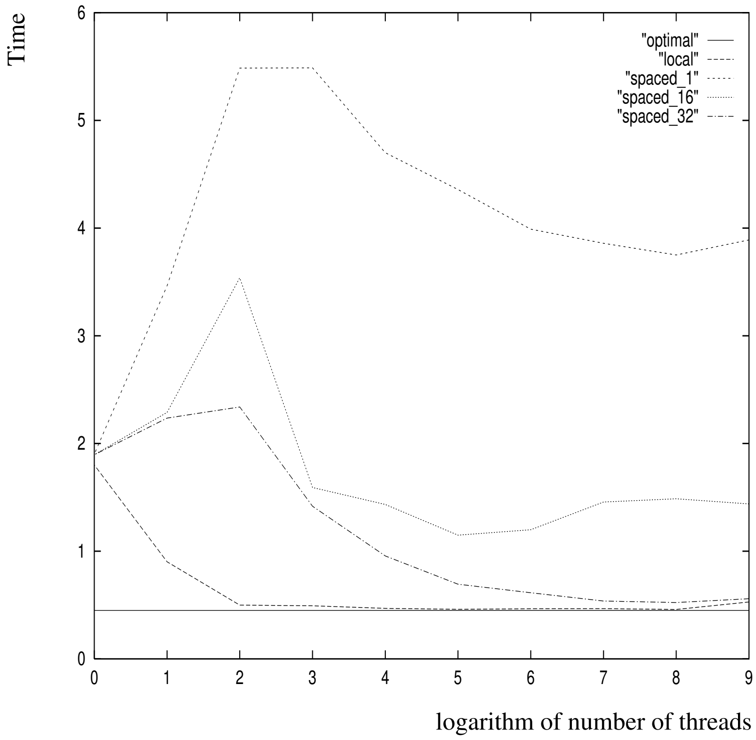

Figure 7.2 Execution time of the `compute_pi` program as a function of number of threads.

can be done by changing line 64 to `*(hit_pointer) ++;`, and deleting line 67. It is easy to verify that the program is semantically identical to the one before. However, on executing this modified program the observed performance is illustrated in the curve labeled "spaced_1" in Figure 7.2. This represents a significant slowdown instead of a speedup!

The drastic impact of this seemingly innocuous change is explained by a phenomenon called *f*alse sharing. In this example, two adjoining data items (which likely reside on the same cache line) are being continually written to by threads that might be scheduled on different processors. From our discussion in Chapter **??**, we know that a write to a shared cache line results in an invalidate and a subsequent read must fetch the cache line from the most recent write location. With this in mind, we can see that the cache lines corresponding to the `hits` array generate a large number of invalidates and reads because of repeated increment operations. This situation, in which two threads 'falsely' share data because it happens to be on the same cache line, is called false sharing.

It is in fact possible to use this simple example to estimate the cache line size of the system. We change `hits` to a two-dimensional array and use only the first column of the array to store counts. By changing the size of the second dimension, we can force entries in the first column of the `hits` array to lie on different cache lines (since arrays in C are stored row-major). The results of this experiment are illustrated in Figure 7.2 by curves labeled "spaced_16" and "spaced_32", in which the second dimension of the `hits` array is 16 and 32 integers, respectively. It is evident from the figure that as the entries are spaced apart, the performance improves. This is consistent with our understanding that spacing the entries out pushes them into different cache lines, thereby reducing the false sharing overhead. ■

Having understood how to create and join threads, let us now explore mechanisms in Pthreads for synchronizing threads.

7.5 Synchronization Primitives in Pthreads

While communication is implicit in shared-address-space programming, much of the effort associated with writing correct threaded programs is spent on synchronizing concurrent threads with respect to their data accesses or scheduling.

7.5.1 Mutual Exclusion for Shared Variables

Using `pthread_create` and `pthread_join` calls, we can create concurrent tasks. These tasks work together to manipulate data and accomplish a given task. When multiple threads attempt to manipulate the same data item, the results can often be incoherent if proper care is not taken to synchronize them. Consider the following code fragment being executed by multiple threads. The variable `my_cost` is thread-local and `best_cost` is a global variable shared by all threads.

```
/* 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 *s*erialization 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).

The aforementioned situation occurred because the test-and-update operation illustrated above is an atomic operation; *i.e.*, the operation should not be broken into sub-operations. Furthermore, the code corresponds to a critical segment; *i.e.*, a segment that must be executed by only one thread at any time. Many statements that seem atomic in higher level languages such as C may in fact be non-atomic; for example, a statement of the form `global_count += 5` may comprise several assembler instructions and therefore must be handled carefully.

Threaded APIs provide support for implementing critical sections and atomic operations using ***m***utex-locks (mutual exclusion locks). Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation generally associated with a piece of code that manipulates shared data. To access the shared data, a thread must first try to acquire a mutex-lock. If the mutex-lock is already locked, the process trying to acquire the lock is blocked. This is because a locked mutex-lock implies that there is another thread currently in the critical section and that no other thread must be allowed in. When a thread leaves a critical section, it must unlock the mutex-lock so that other threads can enter the critical section. All mutex-locks must be initialized to the unlocked state at the beginning of the program.

The Pthreads API provides a number of functions for handling mutex-locks. The function `pthread_mutex_lock` can be used to attempt a lock on a mutex-lock. The prototype of the function is:

```
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.

On leaving a critical section, a thread must unlock the mutex-lock associated with the section. If it does not do so, no other thread will be able to enter this section, typically resulting in a deadlock. The Pthreads function `pthread_mutex_unlock` is used to unlock a mutex-lock. The prototype of this function is:

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

On calling this function, in the case of a ***n***ormal 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.

We need one more function before we can start using mutex-locks, namely, a function to initialize a mutex-lock to its unlocked state. The Pthreads function for this is pthread_mutex_init. The prototype of this function is as follows:

```
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.

Example 7.3 Computing the minimum entry in a list of integers

Armed with basic mutex-lock functions, let us write a simple threaded program to compute the minimum of a list of integers. The list is partitioned equally among the threads. The size of each thread's partition is stored in the variable partial_list_size and the pointer to the start of each thread's partial list is passed to it as the pointer list_ptr. The threaded program for accomplishing this is as follows:

```
#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);
}
```

Programming Notes In this example, the test-update operation for `minimum_value` is protected by the mutex-lock `minimum_value_lock`. Threads execute `pthread_mutex_lock` to gain exclusive access to the variable `minimum_value`. Once this access is gained, the value is updated as required, and the lock subsequently released. Since at any point of time, only one thread can hold a lock, only one thread can test-update the variable. ■

Example 7.4 Producer-consumer work queues

A common use of mutex-locks is in establishing a producer-consumer relationship between threads. The producer creates tasks and inserts them into a work-queue. The consumer threads pick up tasks from the task queue and execute them. Let us consider a simple instance of this paradigm in which the task queue can hold only one task (in a general case, the task queue may be longer but is typically of bounded size). Producer-consumer relations are ubiquitous. See Exercise 7.4 for an example application in multimedia processing. A simple (and incorrect) threaded program would associate a producer thread with creating a task and placing it in a shared data structure and the consumer threads with picking up tasks from this shared data structure and executing them. However, this simple version does not account for the following possibilities:

- The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
- The consumer threads must not pick up tasks until there is something present in the shared data structure.
- Individual consumer threads should pick up tasks one at a time.

To implement this, we can use a variable called `task_available`. If this variable is 0, consumer threads must wait, but the producer thread can insert tasks into the shared data structure `task_queue`. If `task_available` is equal to 1, the producer thread must wait to insert the task into the shared data structure but one of the consumer threads can pick up the task available. All of these operations on the variable `task_available` should be protected by mutex-locks to ensure that only one thread is executing test-update on it. The threaded version of this program is as follows:

```
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);
    }
}
```

Programming Notes In this example, the producer thread creates a task and waits for space on the queue. This is indicated by the variable `task_available` being 0. The test and update of this variable as well as insertion and extraction from the shared queue are protected by a mutex called `task_queue_lock`. Once space is available on the task queue, the recently created task is inserted into the task queue and the availability of the task is signaled by setting `task_available` to 1. Within the producer thread, the fact that the recently created task has been inserted into the queue is signaled by the variable `inserted` being set to 1, which allows the producer to produce the next task. Irrespective of whether a recently created task is successfully inserted into the queue or not, the lock is relinquished. This allows consumer threads to pick up work from the queue in case there is work on the queue to begin with. If the lock is not relinquished, threads would deadlock since a consumer would not be able to get the lock to pick up the task and the producer would not be able to insert its task into the task queue. The consumer thread waits for a task to become available and executes it when available. As was the case with the producer thread, the consumer relinquishes the lock in each iteration of the `while` loop to allow the producer to insert work into the queue if there was none. ■

Overheads of Locking

Locks represent serialization points since critical sections must be executed by threads one after the other. Encapsulating large segments of the program within locks can, therefore, lead to significant performance degradation. It is important to minimize the size of critical sections. For instance, in the above example, the `create_task` and `process_task` functions are left outside the critical region, but `insert_into_queue` and `extract_from_queue` functions are left inside the critical region. The former is left out in the interest of making the critical section as small as possible. The `insert_into_queue` and `extract_from_queue` functions are left inside because if the lock is relinquished after updating `task_available` but not inserting or extracting the task, other threads may gain access to the shared data structure while the insertion or extraction is in progress, resulting in errors. It is therefore important to handle critical sections and shared data structures with extreme care.

Alleviating Locking Overheads

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);
```

We illustrate the use of this function using the following example:

Example 7.5 Finding k matches in a list

We consider the example of finding k matches to a query item in a given list. The list is partitioned equally among the threads. Assuming that the list has n entries, each of the p threads is responsible for searching n/p entries of the list. The program segment for computing this using the pthread_mutex_lock function is as follows:

```
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);
}
```

This program segment finds an entry in its part of the database, updates the global count and then finds the next entry. If the time for a lock-update count-unlock cycle is t_1 and the time to find an entry is t_2, then the total time for satisfying the query is $(t_1 + t_2) \times n_{max}$, where n_{max} is the maximum number of entries found by any thread. If t_1 and t_2 are comparable, then locking leads to considerable overhead.

This locking overhead can be alleviated by using the function pthread_mutex_trylock. Each thread now finds the next entry and tries to acquire the lock and update count. If another thread already has the lock, the record is inserted into a local list and the thread proceeds to find other matches. When it finally gets the lock, it inserts all entries found locally thus far into the list (provided the number does not exceed the desired number of entries). The corresponding output_record function is as follows:

```
int output_record(struct database_record *record_ptr) {
    int count;
    int lock_status;
    lock_status = pthread_mutex_trylock(&output_count_lock);
    if (lock_status == EBUSY) {
        insert_into_local_list(record_ptr);
        return(0);
```

```
    }
    else {
        count = output_count;
        output_count += number_on_local_list + 1;
        pthread_mutex_unlock(&output_count_lock);
        print_records(record_ptr, local_list,
              requested_number_of_records - count);
        return(count + number_on_local_list + 1);
    }
}
```

Programming Notes Examining this function closely, we notice that if the lock for updating the global count is not available, the function inserts the current record into a local list and returns. If the lock is available, it increments the global count by the number of records on the local list, and then by one (for the current record). It then unlocks the associated lock and proceeds to print as many records as are required using the function `print_records`.

Performance Notes The time for execution of this version is less than the time for the first one on two counts: first, as mentioned, the time for executing a `pthread_mutex_trylock` is typically much smaller than that for a `pthread_mutex_lock`. Second, since multiple records may be inserted on each lock, the number of locking operations is also reduced. The number of records actually searched (across all threads) may be slightly larger than the number of records actually desired (since there may be entries in the local lists that may never be printed). However, since this time would otherwise have been spent idling for the lock anyway, this overhead does not cause a slowdown.

■

The above example illustrates the use of the function `pthread_mutex_trylock` instead of `pthread_mutex_lock`. The general use of the function is in reducing idling overheads associated with mutex-locks. If the computation is such that the critical section can be delayed and other computation can be performed in the interim, `pthread_mutex_trylock` is the function of choice. Another determining factor, as has been mentioned, is the fact that for most implementations `pthread_mutex_trylock` is a much cheaper function than `pthread_mutex_lock`. In fact, for highly optimized codes, even when a `pthread_mutex_lock` is required, a `pthread_mutex_trylock` inside a loop may often be desirable, since if the lock is acquired within the first few calls, it would be cheaper than a `pthread_mutex_lock`.

7.5.2 Condition Variables for Synchronization

As we noted in the previous section, indiscriminate use of locks can result in idling overhead from blocked threads. While the function `pthread_mutex_trylock` alleviates this overhead, it introduces the overhead of polling for availability of locks. For example, if the producer-consumer example is rewritten using `pthread_mutex_trylock` instead of `pthread_mutex_lock`, the producer and consumer threads would have to periodically poll for availability of lock (and subsequently availability of buffer space or tasks on queue). A natural solution to this problem is to suspend the execution of the producer until space becomes available (an interrupt driven mechanism as opposed to a polled mechanism). The availability of space is signaled by the consumer thread that consumes the task. The functionality to accomplish this is provided by a condition variable.

A condition variable is a data object used for synchronizing threads. This variable allows a thread to block itself until specified data reaches a predefined state. In the producer-consumer case, the shared variable `task_available` must become 1 before the consumer threads can be signaled. The boolean condition `task_available == 1` is referred to as a predicate. A condition variable is associated with this predicate. When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition. A single condition variable may be associated with more than one predicate. However, this is strongly discouraged since it makes the program difficult to debug.

A condition variable always has a mutex associated with it. A thread locks this mutex and tests the predicate defined on the shared variable (in this case `task_available`); if the predicate is not true, the thread waits on the condition variable associated with the predicate using the function `pthread_cond_wait`. The prototype of this function is:

```
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).

In the context of our producer-consumer example, the producer thread produces the task and, since the lock on `mutex` has been relinquished (by waiting consumers), it can insert its task on the queue and set `task_available` to 1 after locking `mutex`. Since the predicate has now been satisfied, the producer must wake up one of the consumer threads by signaling it. This is done using the function `pthread_cond_signal`, whose prototype is as follows:

```
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.

Before we rewrite our producer-consumer example using condition variables, we need to introduce two more function calls for initializing and destroying condition variables, `pthread_cond_init` and `pthread_cond_destroy` respectively. The prototypes of these calls are as follows:

```
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:

Example 7.6 Producer-consumer using condition variables

Condition variables can be used to block execution of the producer thread when the work queue is full and the consumer thread when the work queue is empty. We use two condition variables `cond_queue_empty` and `cond_queue_full` for specifying empty and full queues respectively. The predicate associated with `cond_queue_empty` is `task_available == 0`, and `cond_queue_full` is asserted when `task_available == 1`.

The producer queue locks the mutex `task_queue_cond_lock` associated with the shared variable `task_available`. It checks to see if `task_available` is 0 (*i.e.*, queue is empty). If this is the case, the producer inserts the task into the work queue and signals any waiting consumer threads to wake up by signaling the condition variable `cond_queue_full`. It subsequently proceeds to create additional tasks. If `task_available` is 1 (*i.e.*, queue is full), the producer performs a condition wait on the condition variable `cond_queue_empty` (*i.e.*, it waits for the queue to become empty). The reason for implicitly releasing the lock on `task_queue_cond_lock` becomes clear at this point. If the lock is not released, no consumer will be able to consume the task and the queue would never be empty. At this point, the producer thread is blocked. Since the lock is available to the consumer, the thread can consume the task and signal the condition variable `cond_queue_empty` when the task has been taken off the work queue.

The consumer thread locks the mutex `task_queue_cond_lock` to check if the shared variable `task_available` is 1. If not, it performs a condition wait on `cond_queue_full`. (Note that this signal is generated from the producer when a task is inserted into the work queue.) If there is a task available, the consumer takes it off the work queue and signals the producer. In this way, the producer and consumer threads operate by signaling each other. It is easy to see that this mode of operation is similar to an interrupt-based operation as opposed to a polling-based operation of `pthread_mutex_trylock`. The program segment for accomplishing this producer-consumer behavior is as follows:

```
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);
    }
}
```

Programming Notes An important point to note about this program segment is that the predicate associated with a condition variable is checked in a loop. One might expect that when cond_queue_full is asserted, the value of task_available must be 1. However, it is a good practice to check for the condition in a loop because the thread might be woken up due to other reasons (such as an OS signal). In other cases, when the condition variable is signaled using a condition broadcast (signaling all waiting threads instead of just one), one of the threads that got the lock earlier might invalidate the condition. In the example of multiple producers and multiple consumers, a task available on the work queue might be consumed by one of the other consumers.

Performance Notes When a thread performs a condition wait, it takes itself off the runnable list – consequently, it does not use any CPU cycles until it is woken up. This is in contrast to a mutex lock which consumes CPU cycles as it polls for the lock.

■

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);
```

An example of this is in the producer-consumer scenario with large work queues and multiple tasks being inserted into the work queue on each insertion cycle. This is left as an exercise for the reader (Exercise 7.2). Another example of the use of pthread_cond_broadcast is in the implementation of barriers illustrated in Section 7.8.2.

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.

7.6 Controlling Thread and Synchronization Attributes

In our discussion thus far, we have noted that entities such as threads and synchronization variables can have several attributes associated with them. For example, different threads may be scheduled differently (round-robin, prioritized, etc.), they may have different stack sizes, and so on. Similarly, a synchronization variable such as a mutex-lock may be of different types. The Pthreads API allows a programmer to change the default attributes of entities using ***a***ttributes objects.

An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties. When creating a thread or a synchronization variable, we can specify the attributes object that determines the properties of the entity. Once created, the thread or synchronization variable's properties are largely fixed (Pthreads allows the user to change the priority of the thread). Subsequent changes to attributes objects do not change the properties of entities created using the attributes object prior to the change. There are several advantages of using attributes objects. First, it separates the issues of program semantics and implementation. Thread properties are specified by the user. How these are implemented at the system level is transparent to the user. This allows for greater portability across operating systems. Second, using attributes objects improves modularity and readability of the programs. Third, it allows the user to modify the program easily. For instance, if the user wanted to change the scheduling from round robin to time-sliced for all threads, they would only need to change the specific attribute in the attributes object.

To create an attributes object with the desired properties, we must first create an object with default properties and then modify the object as required. We look at Pthreads functions for accomplishing this for threads and synchronization variables.

7.6.1 Attributes Objects for Threads

The function `pthread_attr_init` lets us create an attributes object for threads. The prototype of this function is

```
int
pthread_attr_init (
    pthread_attr_t   *attr);
```

This function initializes the attributes object `attr` to the default values. Upon successful completion, the function returns a 0, otherwise it returns an error code. The attributes object may be destroyed using the function `pthread_attr_destroy`. The prototype of this function is:

```
int
pthread_attr_destroy (
    pthread_attr_t   *attr);
```

The call returns a 0 on successful removal of the attributes object `attr`. Individual properties associated with the attributes object can be changed using the following functions: `pthread_attr_setdetachstate`, `pthread_attr_setguardsize_np`, `pthread_attr_setstacksize`, `pthread_attr_setinheritsched`, `pthread_attr_setschedpolicy`, and `pthread_attr_setschedparam`. These functions can be used to set the detach state in a thread attributes object, the stack guard size, the stack size, whether scheduling policy is inherited from the creating thread, the scheduling policy (in case it is not inherited), and scheduling parameters, respectively. We refer the reader to the Pthreads manuals for a detailed description of these functions. For most parallel programs, default thread properties are generally adequate.

7.6.2 Attributes Objects for Mutexes

The Pthreads API supports three different kinds of locks. All of these locks use the same functions for locking and unlocking; however, the type of lock is determined by the lock attribute. The mutex lock used in examples thus far is called a ***n***ormal mutex. This is the default type of a lock. Only a single thread is allowed to lock a ***n***ormal mutex once at any point in time. If a thread with a lock attempts to lock it again, the second locking call results in a deadlock.

Consider the following example of a thread searching for an element in a binary tree. To ensure that other threads are not changing the tree during the search process, the thread locks the tree with a single mutex `tree_lock`. The search function is as follows:

```
search_tree(void *tree_ptr)
{
    struct node *node_pointer;
    node_pointer = (struct node *) tree_ptr;
    pthread_mutex_lock(&tree_lock);
    if (is_search_node(node_pointer) == 1) {
        /* solution is found here */
        print_node(node_pointer);
        pthread_mutex_unlock(&tree_lock);
        return(1);
    }
    else {
        if (tree_ptr -> left != NULL)
            search_tree((void *) tree_ptr -> left);
        if (tree_ptr -> right != NULL)
            search_tree((void *) tree_ptr -> right);
    }
    printf("Search unsuccessful\n");
    pthread_mutex_unlock(&tree_lock);
}
```

If `tree_lock` is a normal mutex, the first recursive call to the function `search_tree` ends in a deadlock since a thread attempts to lock a mutex that it holds a lock on. For addressing such situations, the Pthreads API supports a ***r***ecursive mutex. A recursive mutex allows a single thread to lock a mutex multiple times. Each time a thread locks the mutex, a lock counter is incremented. Each unlock decrements the counter. For any other thread to be able to successfully lock a recursive mutex, the lock counter must be zero (*i.e.*, each lock by another thread must have a corresponding unlock). A recursive mutex is useful when a thread function needs to call itself recursively.

In addition to normal and recursive mutexes, a third kind of mutex called an ***e***rrorcheck mutex is also supported. The operation of an errorcheck mutex is similar to a normal mutex in that a thread can lock a mutex only once. However, unlike a normal mutex, when a thread attempts a lock on a mutex it has already locked, instead of deadlocking it returns an error. Therefore, an errorcheck mutex is more useful for debugging purposes.

The type of mutex can be specified using a mutex attribute object. To create and initialize a mutex attribute object to default values, Pthreads provides the function `pthread_mutexattr_init`. The prototype of the function is:

```
int
pthread_mutexattr_init (
    pthread_mutexattr_t   *attr);
```

This creates and initializes a mutex attributes object `attr`. The default type of a mutex is a normal mutex. Pthreads provides the function `pthread_mutexattr_settype_np` for setting the type of mutex specified by the mutex attributes object. The prototype for this function is:

```
int
pthread_mutexattr_settype_np (
    pthread_mutexattr_t   *attr,
    int   type);
```

Here, `type` specifies the type of the mutex and can take one of the following values:

- `PTHREAD_MUTEX_NORMAL_NP`
- `PTHREAD_MUTEX_RECURSIVE_NP`
- `PTHREAD_MUTEX_ERRORCHECK_NP`

A mutex-attributes object can be destroyed using the `pthread_attr_destroy` that takes the mutex attributes object `attr` as its only argument.

7.7 Thread Cancellation

Consider a simple program to evaluate a set of positions in a chess game. Assume that there are k moves, each being evaluated by an independent thread. If at any point of time, a position is established to be of a certain quality, the other positions that are known to be of worse quality must stop being evaluated. In other words, the threads evaluating the corresponding board positions must be canceled. Posix threads provide this cancellation feature in the function `pthread_cancel`. The prototype of this function is:

```
int
pthread_cancel (
    pthread_t   thread);
```

Here, `thread` is the handle to the thread to be canceled. A thread may cancel itself or cancel other threads. When a call to this function is made, a cancellation is sent to the specified thread. It is not guaranteed that the specified thread will receive or act on the cancellation. Threads can protect themselves against cancellation. When a cancellation is actually performed, cleanup functions are invoked for reclaiming the thread data structures. After this the thread is canceled. This process is similar to termination of a thread using the `pthread_exit` call. This is performed independently of the thread that made the original request for cancellation. The `pthread_cancel` function returns after a cancellation has been sent. The cancellation may itself be performed later. The function returns a 0 on successful completion. This does not imply that the requested thread has been canceled; it implies that the specified thread is a valid thread for cancellation.

7.8 Composite Synchronization Constructs

While the Pthreads API provides a basic set of synchronization constructs, often, there is a need for higher level constructs. These higher level constructs can be built using basic synchronization constructs. In this section, we look at some of these constructs along with their performance aspects and applications.

7.8.1 Read-Write Locks

In many applications, a data structure is read frequently but written infrequently. For such scenarios, it is useful to note that multiple reads can proceed without any coherence problems. However, writes must be serialized. This points to an alternate structure called a read-write lock. A thread reading a shared data item acquires a read lock on the variable. A read lock is granted when there are other threads that may already have read locks. If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait. Similarly, if there are multiple threads requesting a write lock, they must perform a condition wait. Using this principle, we design functions for read locks `mylib_rwlock_rlock`, write locks `mylib_rwlock_wlock`, and unlocking `mylib_rwlock_unlock`.

The read-write locks illustrated are based on a data structure called `mylib_rwlock_t`. This structure maintains a count of the number of readers, the writer (a 0/1 integer specifying whether a writer is present), a condition variable `readers_proceed` that is signaled when readers can proceed, a condition variable `writer_proceed` that is signaled when one of the writers can proceed, a count `pending_writers` of pending writers, and a mutex `read_write_lock` associated with the shared data structure. The function `mylib_rwlock_init` is used to initialize various components of this data structure.

The function `mylib_rwlock_rlock` attempts a read lock on the data structure. It checks to see if there is a write lock or pending writers. If so, it performs a condition wait on the condition variable `readers_proceed`, otherwise it increments the count of readers and proceeds to grant a read lock. The function `mylib_rwlock_wlock` attempts a write lock on the data structure. It checks to see if there are readers or writers; if so, it increments the count of pending writers and performs a condition wait on the condition variable `writer_proceed`. If there are no readers or writer, it grants a write lock and proceeds.

The function `mylib_rwlock_unlock` unlocks a read or write lock. It checks to see if there is a write lock, and if so, it unlocks the data structure by setting the `writer` field to 0. If there are readers, it decrements the number of readers `readers`. If there are no readers left and there are pending writers, it signals one of the writers to proceed (by signaling `writer_proceed`). If there are no pending writers but there are pending readers, it signals all the reader threads to proceed. The code for initializing and locking/unlocking is as follows:

```
typedef struct {
    int readers;
    int writer;
    pthread_cond_t readers_proceed;
    pthread_cond_t writer_proceed;
    int pending_writers;
    pthread_mutex_t read_write_lock;
} mylib_rwlock_t;


void mylib_rwlock_init (mylib_rwlock_t *l) {
   l -> readers = l -> writer = l -> pending_writers = 0;
   pthread_mutex_init(&(l -> read_write_lock), NULL);
   pthread_cond_init(&(l -> readers_proceed), NULL);
   pthread_cond_init(&(l -> writer_proceed), NULL);
}

void mylib_rwlock_rlock(mylib_rwlock_t *l) {
   /* if there is a write lock or pending writers, perform condition
   wait.. else increment count of readers and grant read lock */

   pthread_mutex_lock(&(l -> read_write_lock));
   while ((l -> pending_writers > 0) || (l -> writer > 0))
       pthread_cond_wait(&(l -> readers_proceed),
          &(l -> read_write_lock));
   l -> readers ++;
   pthread_mutex_unlock(&(l -> read_write_lock));
}


void mylib_rwlock_wlock(mylib_rwlock_t *l) {
   /* if there are readers or writers, increment pending writers
   count and wait. On being woken, decrement pending writers
   count and increment writer count */

   pthread_mutex_lock(&(l -> read_write_lock));
   while ((l -> writer > 0) || (l -> readers > 0))  {
       l -> pending_writers ++;
       pthread_cond_wait(&(l -> writer_proceed),
          &(l -> read_write_lock));
   }
   l -> pending_writers --;
   l -> writer ++
   pthread_mutex_unlock(&(l -> read_write_lock));
```

```
}


void mylib_rwlock_unlock(mylib_rwlock_t *l) {
   /* if there is a write lock then unlock, else if there are
   read locks, decrement count of read locks. If the count
   is 0 and there is a pending writer, let it through, else
   if there are pending readers, let them all go through */

   pthread_mutex_lock(&(l -> read_write_lock));
   if (l -> writer > 0)
      l -> writer = 0;
   else if (l -> readers > 0)
      l -> readers --;
   pthread_mutex_unlock(&(l -> read_write_lock));
   if ((l -> readers == 0) && (l -> pending_writers > 0))
      pthread_cond_signal(&(l -> writer_proceed));
   else if (l -> readers > 0)
      pthread_cond_broadcast(&(l -> readers_proceed));
}
```

We now illustrate the use of read-write locks with some examples.

Example 7.7 Using read-write locks for computing the minimum of a list of numbers

A simple use of read-write locks is in computing the minimum of a list of numbers. In our earlier implementation, we associated a lock with the minimum value. Each thread locked this object and updated the minimum value, if necessary. In general, the number of times the value is examined is greater than the number of times it is updated. Therefore, it is beneficial to allow multiple reads using a read lock and write after a write lock only if needed. The corresponding program segment is as follows:

```
void *find_min_rw(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 */
    mylib_rwlock_rlock(&read_write_lock);
    if (my_min < minimum_value) {
        mylib_rwlock_unlock(&read_write_lock);
        mylib_rwlock_wlock(&read_write_lock);
        minimum_value = my_min;
    }
    /* and unlock the mutex */
    mylib_rwlock_unlock(&read_write_lock);
    pthread_exit(0);
}
```

Programming Notes In this example, each thread computes the minimum element in its partial list. It then attempts a read lock on the lock associated with the global minimum value. If the global minimum value is greater than the locally minimum value (thus requiring an update), the read lock is relinquished and a write lock is sought. Once the write lock has been obtained, the global minimum can be updated. The performance gain obtained from read-write locks is influenced by the number of threads and the number of updates (write locks) required. In the extreme case when the first value of the global minimum is also the true minimum value, no write locks are subsequently sought. In this case, the version using read-write locks performs better. In contrast, if each thread must update the global minimum, the read locks are superfluous and add overhead to the program. ■

Example 7.8 Using read-write locks for implementing hash tables

A commonly used operation in applications ranging from database query to state space search is the search of a key in a database. The database is organized as a hash table. In our example, we assume that collisions are handled by chaining colliding entries into linked lists. Each list has a lock associated with it. This lock ensures that lists are not being updated and searched at the same time. We consider two versions of this program: one using mutex locks and one using read-write locks developed in this section.

The mutex lock version of the program hashes the key into the table, locks the mutex associated with the table index, and proceeds to search/update within the linked list. The thread function for doing this is as follows:

```
manipulate_hash_table(int entry) {
    int table_index, found;
    struct list_entry *node, *new_node;

    table_index = hash(entry);
    pthread_mutex_lock(&hash_table[table_index].list_lock);
    found = 0;
    node = hash_table[table_index].next;
    while ((node != NULL) && (!found)) {
        if (node -> value == entry)
            found = 1;
        else
            node = node -> next;
    }
    pthread_mutex_unlock(&hash_table[table_index].list_lock);
    if (found)
        return(1);
    else
        insert_into_hash_table(entry);
}
```

Here, the function `insert_into_hash_table` must lock `hash_table[table_index].list_lock` before performing the actual insertion. When a large fraction of the queries are found in the hash table (*i.e.*, they do not need to be inserted), these searches are serialized. It is easy to see that multiple threads can be safely allowed to search the hash table and only updates to the table must be serialized. This can be accomplished using read-write locks. We can rewrite the `manipulate_hash_table` function as follows:

```
manipulate_hash_table(int entry)
{
    int table_index, found;
    struct list_entry *node, *new_node;

    table_index = hash(entry);
    mylib_rwlock_rlock(&hash_table[table_index].list_lock);
    found = 0;
    node = hash_table[table_index].next;
    while ((node != NULL) && (!found)) {
        if (node -> value == entry)
            found = 1;
        else
            node = node -> next;
    }
    mylib_rwlock_rlock(&hash_table[table_index].list_lock);
    if (found)
        return(1);
    else
        insert_into_hash_table(entry);
}
```

Here, the function `insert_into_hash_table` must first get a write lock on `hash_table[table_index].list_lock` before performing actual insertion.

Programming Notes In this example, we assume that the `list_lock` field has been defined to be of type `mylib_rwlock_t` and all read-write locks associated with the hash tables have been initialized using the function `mylib_rwlock_init`. Using `mylib_rwlock_rlock` instead of a mutex lock allows multiple threads to search respective entries concurrently. Thus, if the number of successful searches outnumber insertions, this formulation is likely to yield better performance. Note that the `insert_into_hash_table` function must be suitably modified to use write locks (instead of mutex locks as before). ■

It is important to identify situations where read-write locks offer advantages over normal locks. Since read-write locks offer no advantage over normal mutexes for writes, they are beneficial only when there are a significant number of read operations. Furthermore, as the critical section becomes larger, read-write locks offer more advantages. This is because the serialization overhead paid by normal mutexes is higher. Finally, since read-write locks rely on condition variables, the underlying thread system must provide fast condition wait, signal, and broadcast functions. It is possible to do a simple analysis to understand the relative merits of read-write locks (Exercise 7.7).

7.8.2 Barriers

An important and often used construct in threaded (as well as other parallel) programs is a `barrier`. A barrier call is used to hold a thread until all other threads participating in the barrier have reached the barrier. Barriers can be implemented using a counter, a mutex and a condition variable. (They can also be implemented simply using mutexes; however, such implementations suffer from the overhead of busy-wait.) A single integer is used to keep track of the number of threads that have reached the barrier. If the count is less than the total number of threads, the threads execute a condition wait. The last thread entering (and setting the count to the number of threads) wakes up all the threads using a condition broadcast. The code for accomplishing this is as follows:

```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
    b -> count = 0;
    pthread_mutex_init(&(b -> count_lock), NULL);
    pthread_cond_init(&(b -> ok_to_proceed), NULL);
}

void mylib_barrier (mylib_barrier_t *b, int num_threads) {
    pthread_mutex_lock(&(b -> count_lock));
    b -> count ++;
    if (b -> count == num_threads) {
        b -> count = 0;
        pthread_cond_broadcast(&(b -> ok_to_proceed));
    }
    else
        while (pthread_cond_wait(&(b -> ok_to_proceed),
            &(b -> count_lock)) != 0);
    pthread_mutex_unlock(&(b -> count_lock));
}
```

In the above implementation of a barrier, threads enter the barrier and stay until the broadcast signal releases them. The threads are released one by one since the mutex `count_lock` is passed among them one after the other. The trivial lower bound on execution time of this function is therefore $O(n)$ for n threads. This implementation of a barrier can be speeded up using multiple barrier variables.

Let us consider an alternate barrier implementation in which there are $n/2$ condition variable-mutex pairs for implementing a barrier for n threads. The barrier works as follows: at the first level, threads are paired up and each pair of threads shares a single condition variable-mutex pair. A designated member of the pair waits for both threads to arrive at the pairwise barrier. Once this happens, all the designated members are organized into pairs, and this process continues until there is only one thread. At this point, we know that all threads have reached the barrier point. We must release all threads at this point. However, releasing them requires signaling all $n/2$ condition variables. We use the same hierarchical strategy for doing this. The designated thread in a pair signals the respective condition variables.

```
typedef struct barrier_node {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed_up;
    pthread_cond_t ok_to_proceed_down;
    int count;
} mylib_barrier_t_internal;

typedef struct barrier_node mylog_logbarrier_t[MAX_THREADS];
pthread_t p_threads[MAX_THREADS];
pthread_attr_t attr;

void mylib_init_barrier(mylog_logbarrier_t b) {
    int i;
    for (i = 0; i < MAX_THREADS; i++) {
        b[i].count = 0;
        pthread_mutex_init(&(b[i].count_lock), NULL);
        pthread_cond_init(&(b[i].ok_to_proceed_up), NULL);
        pthread_cond_init(&(b[i].ok_to_proceed_down), NULL);
    }
}

void mylib_logbarrier (mylog_logbarrier_t b, int num_threads,
           int thread_id) {
    int i, base, index;
    i = 2;
    base = 0;

    do {
        index = base + thread_id / i;
        if (thread_id % i == 0) {
            pthread_mutex_lock(&(b[index].count_lock));
            b[index].count ++;
            while (b[index].count < 2)
                pthread_cond_wait(&(b[index].ok_to_proceed_up),
                    &(b[index].count_lock));
                pthread_mutex_unlock(&(b[index].count_lock));
            }
            else {
                pthread_mutex_lock(&(b[index].count_lock));
                b[index].count ++;
                if (b[index].count == 2)
                    pthread_cond_signal(&(b[index].ok_to_proceed_up));
                while (pthread_cond_wait(&(b[index].ok_to_proceed_down),
                    &(b[index].count_lock)) != 0);
                pthread_mutex_unlock(&(b[index].count_lock));
                break;
            }
            base = base + num_threads/i;
            i = i * 2;
    } while (i <= num_threads);
    i = i / 2;
    for (; i > 1; i = i / 2) {
        base = base - num_threads/i;
        index = base + thread_id / i;
        pthread_mutex_lock(&(b[index].count_lock));
```

```
            b[index].count = 0;
            pthread_cond_signal(&(b[index].ok_to_proceed_down));
            pthread_mutex_unlock(&(b[index].count_lock));
        }
  }
```

In this implementation of a barrier, we visualize the barrier as a binary tree. Threads arrive at the leaf nodes of this tree. Consider an instance of a barrier with eight threads. Threads 0 and 1 are paired up on a single leaf node. One of these threads is designated as the representative of the pair at the next level in the tree. In the above example, thread 0 is considered the representative and it waits on the condition variable `ok_to_proceed_up` for thread 1 to catch up. All even numbered threads proceed to the next level in the tree. Now thread 0 is paired up with thread 2 and thread 4 with thread 6. Finally thread 0 and 4 are paired. At this point, thread 0 realizes that all threads have reached the desired barrier point and releases threads by signaling the condition `ok_to_proceed_down`. When all threads are released, the barrier is complete.

It is easy to see that there are $n - 1$ nodes in the tree for an n thread barrier. Each node corresponds to two condition variables, one for releasing the thread up and one for releasing it down, one lock, and a count of number of threads reaching the node. The tree nodes are linearly laid out in the array `mylog_logbarrier_t` with the $n/2$ leaf nodes taking the first $n/2$ elements, the $n/4$ tree nodes at the next higher level taking the next $n/4$ nodes and so on.

It is interesting to study the performance of this program. Since threads in the linear barrier are released one after the other, it is reasonable to expect runtime to be linear in the number of threads even on multiple processors. In Figure 7.3, we plot the runtime of 1000 barriers in a sequence on a 32 processor SGI Origin 2000. The linear runtime of the sequential barrier is clearly reflected in the runtime. The logarithmic barrier executing on a single processor does just as much work asymptotically as a sequential barrier (albeit with a higher constant). However, on a parallel machine, in an ideal case when threads are assigned so that subtrees of the binary barrier tree are assigned to different processors, the time grows as $O(n/p + \log p)$. While this is difficult to achieve without being able to examine or assign blocks of threads corresponding to subtrees to individual processors, the logarithmic barrier displays significantly better performance than the serial barrier. Its performance tends to be linear in n as n becomes large for a given number of processors. This is because the n/p term starts to dominate the $\log p$ term in the execution time. This is observed both from observations as well as analytical intuition.

7.9 Tips for Designing Asynchronous Programs

When designing multithreaded applications, it is important to remember that one cannot assume any order of execution with respect to other threads. Any such order must be explicitly established using the synchronization mechanisms discussed above: mutexes,

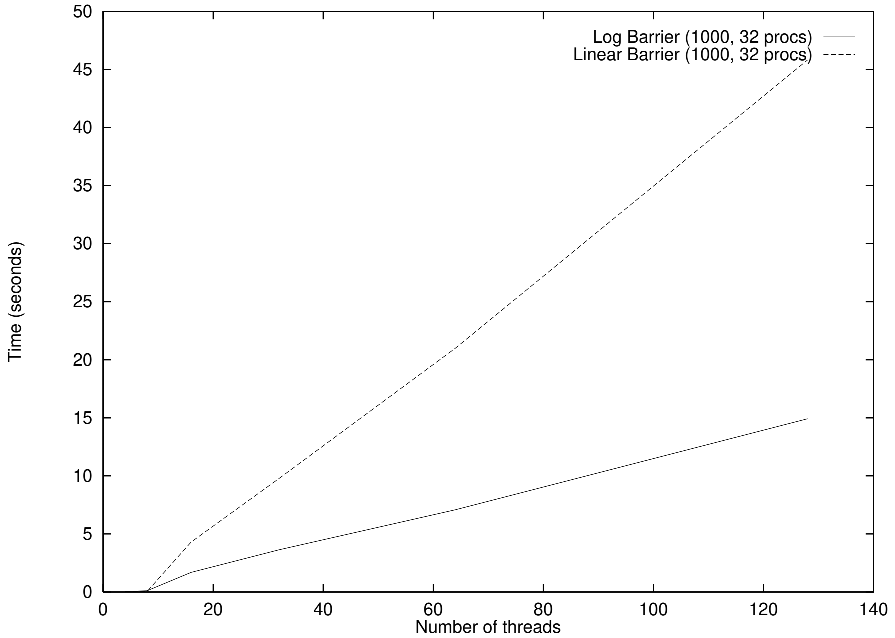

Figure 7.3 Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.

condition variables, and joins. In addition, the system may provide other means of synchronization. However, for portability reasons, we discourage the use of these mechanisms.

In many thread libraries, threads are switched at semi-deterministic intervals. Such libraries are more forgiving of synchronization errors in programs. These libraries are called *s*lightly asynchronous libraries. On the other hand, kernel threads (threads supported by the kernel) and threads scheduled on multiple processors are less forgiving. The programmer must therefore not make any assumptions regarding the level of asynchrony in the threads library.

Let us look at some common errors that arise from incorrect assumptions on relative execution times of threads:

- Say, a thread T1 creates another thread T2. T2 requires some data from thread T1. This data is transferred using a global memory location. However, thread T1 places the data in the location after creating thread T2. The implicit assumption here is that T1 will not be switched until it blocks; or that T2 will get to the point at which it uses the data only after T1 has stored it there. Such assumptions may lead to errors since it is possible that T1 gets switched as soon as it creates T2. In such a situation, T1 will receive uninitialized data.

- Assume, as before, that thread T1 creates T2 and that it needs to pass data to thread T2 which resides on its stack. It passes this data by passing a pointer to the stack location to thread T2. Consider the scenario in which T1 runs to completion before T2 gets scheduled. In this case, the stack frame is released and some other thread may overwrite the space pointed to formerly by the stack frame. In this case, what thread T2 reads from the location may be invalid data. Similar problems may exist with global variables.

- We strongly discourage the use of scheduling techniques as means of synchronization. It is especially difficult to keep track of scheduling decisions on parallel machines. Further, as the number of processors change, these issues may change depending on the thread scheduling policy. It may happen that higher priority threads are actually waiting while lower priority threads are running.

We recommend the following rules of thumb which help minimize the errors in threaded programs.

- Set up all the requirements for a thread before actually creating the thread. This includes initializing the data, setting thread attributes, thread priorities, mutex-attributes, etc. Once you create a thread, it is possible that the newly created thread actually runs to completion before the creating thread gets scheduled again.
- When there is a producer-consumer relation between two threads for certain data items, make sure the producer thread places the data before it is consumed and that intermediate buffers are guaranteed to not overflow.
- At the consumer end, make sure that the data lasts atleast until all potential consumers have consumed the data. This is particularly relevant for stack variables.
- Where possible, define and use group synchronizations and data replication. This can improve program performance significantly.

While these simple tips provide guidelines for writing error-free threaded programs, extreme caution must be taken to avoid race conditions and parallel overheads associated with synchronization.

7.10 OpenMP: a Standard for Directive Based Parallel Programming

In the first part of this chapter, we studied the use of threaded APIs for programming shared address space machines. While standardization and support for these APIs has come a long way, their use is still predominantly restricted to system programmers as opposed to application programmers. One of the reasons for this is that APIs such as Pthreads are considered to be low-level primitives. Conventional wisdom indicates that a large class of applications can be efficiently supported by higher level constructs (or directives) which rid the programmer of the mechanics of manipulating threads. Such directive-based languages have existed for a long time, but only recently have standardization efforts succeeded in the form of OpenMP. OpenMP is an API that can be used with FORTRAN, C, and C++ for programming shared address space machines. OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

7.10.1 The OpenMP Programming Model

We initiate the OpenMP programming model with the aid of a simple program. OpenMP directives in C and C++ are based on the `#pragma` compiler directives. The directive itself consists of a directive name followed by clauses.

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

OpenMP programs execute serially until they encounter the `parallel` directive. This directive is responsible for creating a group of threads. The exact number of threads can be specified in the directive, set using an environment variable, or at runtime using OpenMP functions. The main thread that encounters the `parallel` directive becomes the ***m***aster of this group of threads and is assigned the thread id 0 within the group. The `parallel` directive has the following prototype:

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

```

Each thread created by this directive executes the `structured block` specified by the parallel directive. The clause list is used to specify conditional parallelization, number of threads, and data handling.

- **Conditional Parallelization:** The clause `if (scalar expression)` determines whether the parallel construct results in creation of threads. Only one `if` clause can be used with a parallel directive.

- **Degree of Concurrency:** The clause `num_threads (integer expression)` specifies the number of threads that are created by the `parallel` directive.

- **Data Handling:** The clause `private (variable list)` indicates that the set of variables specified is local to each thread – *i.e.*, each thread has its own copy of each variable in the list. The clause `firstprivate (variable list)` is similar to the `private` clause, except the values of variables on entering the threads are initialized to corresponding values before the parallel directive. The clause `shared (variable list)` indicates that all variables in the list are shared across all the threads, *i.e.*, there is only one copy. Special care must be taken while handling these variables by threads to ensure serializability.

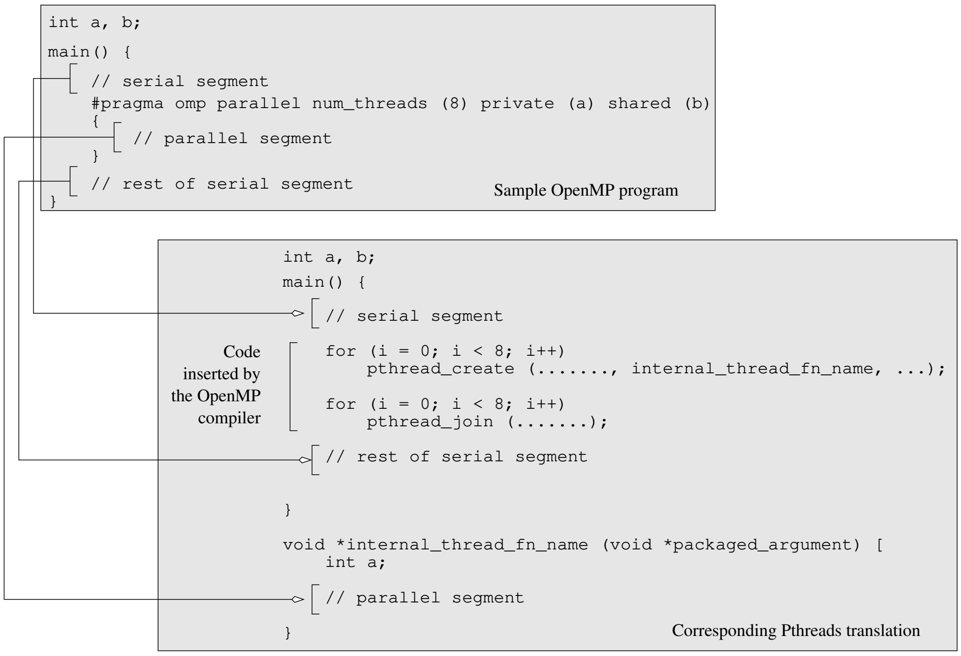

Figure 7.4 A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

It is easy to understand the concurrency model of OpenMP when viewed in the context of the corresponding Pthreads translation. In Figure 7.4, we show one possible translation of an OpenMP program to a Pthreads program. The interested reader may note that such a translation can easily be automated through a Yacc or CUP script.

Example 7.9 Using the `parallel` directive

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

Here, if the value of the variable `is_parallel` equals one, eight threads are created. Each of these threads gets private copies of variables `a` and `c`, and shares a single value of variable `b`. Furthermore, the value of each copy of `c` is initialized to the value of `c` before the parallel directive. ∎

The default state of a variable is specified by the clause `default (shared)` or `default (none)`. The clause `default (shared)` implies that, by default, a variable is shared by all the threads. The clause `default (none)` implies that the state of each variable used in a thread must be explicitly specified. This is generally recommended, to guard against errors arising from unintentional concurrent access to shared data.

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). ■

In addition to these data handling clauses, there is one other clause, `copyin`. We will describe this clause in Section 7.10.4 after we discuss data scope in greater detail.

We can now use the `parallel` directive along with the clauses to write our first OpenMP program. We introduce two functions to facilitate this. The `omp_get_num_threads()` function returns the number of threads in the parallel region and the `omp_get_thread_num()` function returns the integer i.d. of each thread (recall that the master thread has an i.d. 0).

Example 7.11 Computing PI using OpenMP directives

Our first OpenMP example follows from Example 7.2, which presented a Pthreads program for the same problem. The parallel directive specifies that all variables except `npoints`, the total number of random points in two dimensions across all threads, are local. Furthermore, the directive specifies that there are 8 threads, and the value of `sum` after all threads complete execution is the sum of local values at each thread. The function `omp_get_num_threads` is used to determine the total number of threads. As in Example 7.2, a `for` loop generates the required number of random points (in two dimensions) and determines how many of them are within the prescribed circle of unit diameter.

```
/* ******************************************************
   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 ++;
      }
    }
```

Note that this program is much easier to write in terms of specifying creation and termination of threads compared to the corresponding POSIX threaded program.

7.10.2 Specifying Concurrent Tasks in OpenMP

The `parallel` directive can be used in conjunction with other directives to specify concurrency across iterations and tasks. OpenMP provides two directives – `for` and `sections` - to specify concurrent iterations and 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.

Example 7.12 Using the `for` directive for computing π
Recall from Example 7.11 that each iteration of the `for` loop is independent, and can be executed concurrently. In such situations, we can simplify the program using the `for` directive. The modified code segment is as follows:

```
#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 `for` directive in this example specifies that the `for` loop immediately following the directive must be executed in parallel, *i.e.*, split across various threads. Notice that the loop index goes from 0 to `npoints` in this case, as opposed to `sample_points_per_thread` in Example 7.11. The loop index for the `for` directive is assumed to be private, by default. It is interesting to note that the only difference between this OpenMP segment and the corresponding serial code is the two directives. This example illustrates how simple it is to convert many serial programs into OpenMP-based threaded programs. ■

Assigning Iterations to Threads

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`.

Example 7.13 Scheduling classes in OpenMP – matrix multiplication.
We explore various scheduling classes in the context of dense matrix multiplication. The code for multiplying two matrices `a` and `b` to yield matrix `c` is as follows:

```
for (i = 0; i < dim; i++) {
    for (j = 0; j < dim; j++) {
        c(i,j) = 0;
        for (k = 0; k < dim; k++) {
```

```
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

The code segment above specifies a three-dimensional iteration space providing us with an ideal example for studying various scheduling classes in OpenMP. ■

Static The general form of the `static` scheduling class is `schedule(static[, chunk-size])`. This technique splits the iteration space into equal chunks of size `chunk-size` and assigns them to threads in a round-robin fashion. When no `chunk-size` is specified, the iteration space is split into as many chunks as there are threads and one chunk is assigned to each thread.

Example 7.14 Static scheduling of loops in matrix multiplication

The following modification of the matrix-multiplication program causes the outermost iteration to be split statically across threads as illustrated in Figure 7.5(a).

```
#pragma omp parallel default(private) shared (a, b, c, dim) \
                    num_threads(4)
    #pragma omp for schedule(static)
    for (i = 0; i < dim; i++) {
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

Since there are four threads in all, if `dim = 128`, the size of each partition is 32 columns, since we have not specified the chunk size. Using `schedule(static, 16)` results in the partitioning of the iteration space illustrated in Figure 7.5(b). Another example of the split illustrated in Figure 7.5(c) results when each `for` loop in the program in Example 7.13 is parallelized across threads with a `schedule(static)` and nested parallelism is enabled (see Section 7.10.6).

■

Dynamic Often, because of a number of reasons, ranging from heterogeneous computing resources to non-uniform processor loads, equally partitioned workloads take widely varying execution times. For this reason, OpenMP has a `dynamic` scheduling class. The general form of this class is `schedule(dynamic[, chunk-size])`. The iteration space is partitioned into chunks given by `chunk-size`. However, these are assigned to threads as they become idle. This takes care of the temporal imbalances resulting from

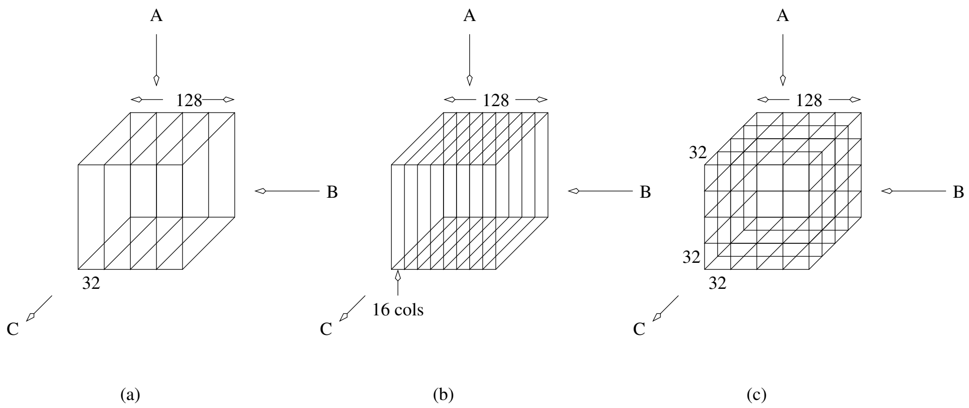

Figure 7.5 Three different schedules using the static scheduling class of OpenMP.

static scheduling. If no `chunk-size` is specified, it defaults to a single iteration per chunk.

Guided Consider the partitioning of an iteration space of 100 iterations with a chunk size of 5. This corresponds to 20 chunks. If there are 16 threads, in the best case, 12 threads get one chunk each and the remaining four threads get two chunks. Consequently, if there are as many processors as threads, this assignment results in considerable idling. The solution to this problem (also referred to as an *e*dge effect) is to reduce the chunk size as we proceed through the computation. This is the principle of the `guided` scheduling class. The general form of this class is `schedule(guided[, chunk-size])`. In this class, the chunk size is reduced exponentially as each chunk is dispatched to a thread. The `chunk-size` refers to the smallest chunk that should be dispatched. Therefore, when the number of iterations left is less than `chunk-size`, the entire set of iterations is dispatched at once. The value of `chunk-size` defaults to one if none is specified.

Runtime Often it is desirable to delay scheduling decisions until runtime. For example, if one would like to see the impact of various scheduling strategies to select the best one, the scheduling can be set to `runtime`. In this case the environment variable `OMP_SCHEDULE` determines the scheduling class and the chunk size.

When no scheduling class is specified with the `omp for` directive, the actual scheduling technique is not specified and is implementation dependent. The `for` directive places additional restrictions on the `for` loop that follows. For example, it must not have a break statement, the loop control variable must be an integer, the initialization expression of the `for` loop must be an integer assignment, the logical expression must be one of $<$, $\leq$, $>$, or $\geq$, and the increment expression must have integer increments or decrements only. For more details on these restrictions, we refer the reader to the OpenMP manuals.

Synchronization Across Multiple `for` Directives

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:

Example 7.15 Using the `nowait` clause

Consider the following example in which variable `name` needs to be looked up in two lists – `current_list` and `past_list`. If the name exists in a list, it must be processed accordingly. The name might exist in both lists. In this case, there is no need to wait for all threads to complete execution of the first loop before proceeding to the second loop. Consequently, we can use the `nowait` clause to save idling and synchronization overheads as follows:

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i = 0; i < nmax; i++)
            if (isEqual(name, current_list[i])
                processCurrentName(name);
    #pragma omp for
        for (i = 0; i < mmax; i++)
            if (isEqual(name, past_list[i])
                processPastName(name);
}
```

■

The `sections` Directive

The `for` directive is suited to partitioning iteration spaces across threads. Consider now a scenario in which there are three tasks (`taskA`, `taskB`, and `taskC`) that need to be executed. Assume that these tasks are independent of each other and therefore can be assigned to different threads. OpenMP supports such non-iterative parallel task assignment using the `sections` directive. The general form of the `sections` directive is as follows:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

This `sections` directive assigns the structured block corresponding to each section to one thread (indeed more than one section can be assigned to a single thread). The `clause list` may include the following clauses – `private`, `firstprivate`, `lastprivate`, `reduction`, and `nowait`. The syntax and semantics of these clauses are identical to those in the case of the `for` directive. The `lastprivate` clause, in this case, specifies that the last section (lexically) of the `sections` directive updates the value of the variable. The `nowait` clause specifies that there is no implicit synchronization among all threads at the end of the `sections` directive.

For executing the three concurrent tasks `taskA`, `taskB`, and `taskC`, the corresponding `sections` directive is as follows:

```
#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.

Merging Directives

In our discussion thus far, we use the directive `parallel` to create concurrent threads, and `for` and `sections` to farm out work to threads. If there was no `parallel` directive specified, the `for` and `sections` directives would execute serially (all work is farmed to a single thread, the master thread). Consequently, `for` and `sections` directives are generally preceded by the `parallel` directive. OpenMP allows the programmer to merge the `parallel` directives to `parallel for` and `parallel sections`, respectively. The clause list for the merged directive can be from the clause lists of either the `parallel` or `for / sections` directives.

For example:

```
#pragma omp parallel default (private) shared (n)
{
    #pragma omp for
```

```
        for (i = 0 < i < n; i++) {
        /* body of parallel for loop */
        }
    }
```

is identical to:

```
    #pragma omp parallel for default (private) shared (n)
    {
        for (i = 0 < i < n; i++) {
        /* body of parallel for loop */
        }
    }
```

and:

```
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
               taskA();
            }
            #pragma omp section
            {
               taskB();
            }
            /* other sections here */
        }
    }
```

is identical to:

```
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        /* other sections here */
    }
```

Nesting `parallel` Directives

Let us revisit Program 7.13. To split each of the `for` loops across various threads, we would modify the program as follows:

```
#pragma omp parallel for default(private) shared (a, b, c, dim) \
                    num_threads(2)
```

```
    for (i = 0; i < dim; i++) {
    #pragma omp parallel for default(private) shared (a, b, c, dim) \
                    num_threads(2)
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            #pragma omp parallel for default(private) \
                    shared (a, b, c, dim) num_threads(2)
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

We start by making a few observations about how this segment is written. Instead of nesting three `for` directives inside a single `parallel` directive, we have used three `parallel for` directives. This is because OpenMP does not allow `for`, `sections`, and `single` directives that bind to the same `parallel` directive to be nested. Furthermore, the code as written only generates a ***l***ogical team of threads on encountering a nested `parallel` directive. The newly generated logical team is still executed by the same thread corresponding to the outer `parallel` directive. To generate a new set of threads, nested parallelism must be enabled using the `OMP_NESTED` environment variable. If the `OMP_NESTED` environment variable is set to `FALSE`, then the inner `parallel` region is serialized and executed by a single thread. If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled. The default state of this environment variable is `FALSE`, *i.e.*, nested parallelism is disabled. OpenMP environment variables are discussed in greater detail in Section 7.10.6.

There are a number of other restrictions associated with the use of synchronization constructs in nested parallelism. We refer the reader to the OpenMP manual for a discussion of these restrictions.

7.10.3 Synchronization Constructs in OpenMP

In Section 7.5, we described the need for coordinating the execution of multiple threads. This may be the result of a desired execution order, the atomicity of a set of instructions, or the need for serial execution of code segments. The Pthreads API supports mutexes and condition variables. Using these we implemented a range of higher level functionality in the form of read-write locks, barriers, monitors etc. The OpenMP standard provides this high-level functionality in an easy-to-use API. In this section, we will explore these directives and their use.

Synchronization Point: The `barrier` Directive

A barrier is one of the most frequently used synchronization primitives. OpenMP provides a `barrier` directive, whose syntax is as follows:

```
    #pragma omp barrier
```

On encountering this directive, all threads in a team wait until others have caught up, and then release. When used with nested `parallel` directives, the `barrier` directive binds to the closest `parallel` directive. For executing barriers conditionally, it is important to note that a `barrier` directive must be enclosed in a compound statement that is conditionally executed. This is because pragmas are compiler directives and not a part of the language. Barriers can also be effected by ending and restarting `parallel` regions. However, there is usually a higher overhead associated with this. Consequently, it is not the method of choice for implementing barriers.

Single Thread Executions: The `single` and `master` Directives

Often, a computation within a parallel section needs to be performed by just one thread. A simple example of this is the computation of the mean of a list of numbers. Each thread can compute a local sum of partial lists, add these local sums to a shared global sum, and have one thread compute the mean by dividing this global sum by the number of entries in the list. The last step can be accomplished using a `single` directive.

A `single` directive specifies a structured block that is executed by a single (arbitrary) thread. The syntax of the `single` directive is as follows:

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

The clause list can take clauses `private`, `firstprivate`, and `nowait`. These clauses have the same semantics as before. On encountering the `single` block, the first thread enters the block. All the other threads proceed to the end of the block. If the `nowait` clause has been specified at the end of the block, then the other threads proceed; otherwise they wait at the end of the `single` block for the thread to finish executing the block. This directive is useful for computing global data as well as performing I/O.

The `master` directive is a specialization of the `single` directive in which only the master thread executes the structured block. The syntax of the `master` directive is as follows:

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

In contrast to the `single` directive, there is no implicit barrier associated with the `master` directive.

Critical Sections: The `critical` and `atomic` Directives

In our discussion of Pthreads, we had examined the use of locks to protect critical regions – regions that must be executed serially, one thread at a time. In addition to explicit lock management (Section 7.10.5), OpenMP provides a `critical` directive for implementing critical regions. The syntax of a `critical` directive is:

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

Here, the optional identifier `name` can be used to identify a critical region. The use of `name` allows different threads to execute different code while being protected from each other.

Example 7.16 Using the `critical` directive for producer-consumer threads

Consider a producer-consumer scenario in which a producer thread generates a task and inserts it into a task-queue. The consumer thread extracts tasks from the queue and executes them one at a time. Since there is concurrent access to the task-queue, these accesses must be serialized using critical blocks. Specifically, the tasks of inserting and extracting from the task-queue must be serialized. This can be implemented as follows:

```
#pragma omp parallel sections
{
    #pragma parallel section
    {
        /* producer thread */
        task = produce_task();
        #pragma omp critical ( task_queue)
        {
            insert_into_queue(task);
        }
    }
    #pragma parallel section
    {
        /* consumer thread */
        #pragma omp critical ( task_queue)
        {
            task = extract_from_queue(task);
        }
        consume_task(task);
    }
}
```

Note that queue full and queue empty conditions must be explicitly handled here in functions `insert_into_queue` and `extract_from_queue`.

■

The `critical` directive ensures that at any point in the execution of the program, only one thread is within a critical section specified by a certain name. If a thread is already inside a critical section (with a name), all others must wait until it is done before entering the named critical section. The name field is optional. If no name is specified, the critical section maps to a default name that is the same for all unnamed critical sections. The names of critical sections are global across the program.

It is easy to see that the `critical` directive is a direct application of the corresponding `mutex` function in Pthreads. The name field maps to the name of the mutex on which the lock is performed. As is the case with Pthreads, it is important to remember that `critical` sections represent serialization points in the program and therefore we must reduce the size of the critical sections as much as possible (in terms of execution time) to get good performance.

There are some obvious safeguards that must be noted while using the `critical` directive. The `block` of instructions must represent a structured block, *i.e.*, no jumps are permitted into or out of the block. It is easy to see that the former would result in non-critical access and the latter in an unreleased lock, which could cause the threads to wait indefinitely.

Often, a critical section consists simply of an update to a single memory location, for example, incrementing or adding to an integer. OpenMP provides another directive, `atomic` for such atomic updates to memory locations. The `atomic` directive specifies that the memory location update in the following instruction should be performed as an atomic operation. The update instruction can be one of the following forms:

```
x binary_operation = expr
x++
++x
x--
--x
```

Here, `expr` is a scalar expression that does not include a reference to `x`, `x` itself is an lvalue of scalar type, and `binary_operation` is one of $\{+, *, -, /, \&, \hat{}\ \|, \ll, \gg\}$. It is important to note that the `atomic` directive only atomizes the load and store of the scalar variable. The evaluation of the expression is not atomic. Care must be taken to ensure that there are no race conditions hidden therein. This also explains why the `expr` term in the `atomic` directive cannot contain the updated variable itself. All `atomic` directives can be replaced by `critical` directives provided they have the same name. However, the availability of atomic hardware instructions may optimize the performance of the program, compared to translation to `critical` directives.

In-Order Execution: The `ordered` Directive

In many circumstances, it is necessary to execute a segment of a parallel loop in the order in which the serial version would execute it. For example, consider a `for` loop in which, at some point, we compute the cumulative sum in array `cumul_sum` of a list stored in array `list`. The array `cumul_sum` can be computed using a `for` loop over index `i` serially by executing `cumul_sum[i] = cumul_sum[i-1] + list[i]`. When executing this `for` loop across threads, it is important to note that `cumul_sum[i]` can be computed only after `cumul_sum[i-1]` has been computed. Therefore, the statement would have to executed within an `ordered` block.

The syntax of the `ordered` directive is as follows:

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

Since the `ordered` directive refers to the in-order execution of a `for` loop, it must be within the scope of a `for` or `parallel for` directive. Furthermore, the `for` or `parallel for` directive must have the `ordered` clause specified to indicate that the loop contains an `ordered` block.

Example 7.17 Computing the cumulative sum of a list using the `ordered` directive

As we have just seen, to compute the cumulative sum of `i` numbers of a list, we can add the current number to the cumulative sum of `i-1` numbers of the list. This loop must, however, be executed in order. Furthermore, the cumulative sum of the first element is simply the element itself. We can therefore write the following code segment using the `ordered` directive.

```
cumul_sum[0] = list[0];
#pragma omp parallel for private (i) \
            shared (cumul_sum, list, n) ordered
for (i = 1; i < n; i++)
{
    /* other processing on list[i] if needed */

    #pragma omp ordered
    {
        cumul_sum[i] = cumul_sum[i-1] + list[i];
    }
}
```

■

It is important to note that the `ordered` directive represents an ordered serialization point in the program. Only a single thread can enter an ordered block when all prior threads (as determined by loop indices) have exited. Therefore, if large portions of a loop are enclosed in `ordered` directives, corresponding speedups suffer. In the above example, the parallel formulation is expected to be no faster than the serial formulation unless there is significant processing associated with `list[i]` outside the `ordered` directive. A single `for` directive is constrained to have only one `ordered` block in it.

Memory Consistency: The `flush` Directive

The `flush` directive provides a mechanism for making memory consistent across threads. While it would appear that such a directive is superfluous for shared address space machines, it is important to note that variables may often be assigned to registers and register-allocated variables may be inconsistent. In such cases, the `flush` directive provides a memory fence by forcing a variable to be written to or read from the memory system. All write operations to shared variables must be committed to memory at a flush and all references to shared variables after a fence must be satisfied from the memory. Since private variables are relevant only to a single thread, the `flush` directive applies only to shared variables.

The syntax of the `flush` directive is as follows:

```
#pragma omp flush[(list)]
```

The optional list specifies the variables that need to be flushed. The default is that all shared variables are flushed.

Several OpenMP directives have an implicit `flush`. Specifically, a `flush` is implied at a `barrier`, at the entry and exit of `critical`, `ordered`, `parallel`, `parallel for`, and `parallel sections` blocks and at the exit of `for`, `sections`, and `single` blocks. A `flush` is ***n***ot implied if a `nowait` clause is present. It is also not implied at the entry of `for`, `sections`, and `single` blocks and at entry or exit of a `master` block.

7.10.4 Data Handling in OpenMP

One of the critical factors influencing program performance is the manipulation of data by threads. We have briefly discussed OpenMP support for various data classes such as `private`, `shared`, `firstprivate`, and `lastprivate`. We now examine these in greater detail, with a view to understanding how these classes should be used. We identify the following heuristics to guide the process:

- If a thread initializes and uses a variable (such as loop indices) and no other thread accesses the data, then a local copy of the variable should be made for the thread. Such data should be specified as `private`.

- If a thread repeatedly reads a variable that has been initialized earlier in the program, it is beneficial to make a copy of the variable and inherit the value at the time of thread creation. This way, when a thread is scheduled on the processor, the data can reside at the same processor (in its cache if possible) and accesses will not result in interprocessor communication. Such data should be specified as `firstprivate`.

- If multiple threads manipulate a single piece of data, one must explore ways of breaking these manipulations into local operations followed by a single global operation. For example, if multiple threads keep a count of a certain event, it is beneficial to keep local counts and to subsequently accrue it using a single summation at the end of the parallel block. Such operations are supported by the `reduction` clause.

- If multiple threads manipulate different parts of a large data structure, the programmer should explore ways of breaking it into smaller data structures and making them private to the thread manipulating them.

- After all the above techniques have been explored and exhausted, remaining data items may be shared among various threads using the clause `shared`.

In addition to `private`, `shared`, `firstprivate`, and `lastprivate`, OpenMP supports one additional data class called `threadprivate`.

The `threadprivate` and `copyin` Directives Often, it is useful to make a set of objects locally available to a thread in such a way that these objects persist through parallel and serial blocks provided the number of threads remains the same. In contrast to `private` variables, these variables are useful for maintaining persistent objects across parallel regions, which would otherwise have to be copied into the master thread's data space and reinitialized at the next parallel block. This class of variables is supported in OpenMP using the `threadprivate` directive. The syntax of the directive is as follows:

```
#pragma omp threadprivate(variable_list)
```

This directive implies that all variables in `variable_list` are local to each thread and are initialized once before they are accessed in a parallel region. Furthermore, these variables persist across different parallel regions provided dynamic adjustment of the number of threads is disabled and the number of threads is the same.

Similar to `firstprivate`, OpenMP provides a mechanism for assigning the same value to `threadprivate` variables across all threads in a parallel region. The syntax of the clause, which can be used with `parallel` directives, is `copyin(variable_list)`.

7.10.5 OpenMP Library Functions

In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs. As we shall notice, these functions are similar to corresponding Pthreads functions; however, they are generally at a higher level of abstraction, making them easier to use.

Controlling Number of Threads and Processors

The following OpenMP functions relate to the concurrency and number of processors used by a threaded program:

```
#include <omp.h>

void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```

The function `omp_set_num_threads` sets the default number of threads that will be created on encountering the next `parallel` directive provided the `num_threads` clause is not used in the `parallel` directive. This function must be called outside the scope of a parallel region and dynamic adjustment of threads must be enabled (using either the `OMP_DYNAMIC` environment variable discussed in Section 7.10.6 or the `omp_set_dynamic` library function).

The `omp_get_num_threads` function returns the number of threads participating in a team. It binds to the closest parallel directive and in the absence of a parallel directive, returns 1 (for master thread). The `omp_get_max_threads` function returns the maximum number of threads that could possibly be created by a `parallel` directive encountered, which does not have a `num_threads` clause. The `omp_get_thread_num` returns a unique thread i.d. for each thread in a team. This integer lies between 0 (for the master thread) and `omp_get_num_threads() -1`. The `omp_get_num_procs` function returns the number of processors that are available to execute the threaded program at that point. Finally, the function `omp_in_parallel` returns a non-zero value if called from within the scope of a parallel region, and zero otherwise.

Controlling and Monitoring Thread Creation

The following OpenMP functions allow a programmer to set and monitor thread creation:

```
#include <omp.h>

void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
```

The `omp_set_dynamic` function allows the programmer to dynamically alter the number of threads created on encountering a parallel region. If the value `dynamic_threads` evaluates to zero, dynamic adjustment is disabled, otherwise it is enabled. The function must be called outside the scope of a parallel region. The corresponding state, *i.e.*, whether dynamic adjustment is enabled or disabled, can be queried using the function `omp_get_dynamic`, which returns a non-zero value if dynamic adjustment is enabled, and zero otherwise.

The `omp_set_nested` enables nested parallelism if the value of its argument, `nested`, is non-zero, and disables it otherwise. When nested parallelism is disabled, any nested parallel regions subsequently encountered are serialized. The state of nested parallelism can be queried using the `omp_get_nested` function, which returns a non-zero value if nested parallelism is enabled, and zero otherwise.

Mutual Exclusion

While OpenMP provides support for critical sections and atomic updates, there are situations where it is more convenient to use an explicit lock. For such programs, OpenMP provides functions for initializing, locking, unlocking, and discarding locks. The lock data structure in OpenMP is of type `omp_lock_t`. The following functions are defined:

```
#include <omp.h>

void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

Before a lock can be used, it must be initialized. This is done using the `omp_init_lock` function. When a lock is no longer needed, it must be discarded using the function `omp_destroy_lock`. It is illegal to initialize a previously initialized lock and destroy an uninitialized lock. Once a lock has been initialized, it can be locked and unlocked using the functions `omp_set_lock` and `omp_unset_lock`. On locking a previously unlocked lock, a thread gets exclusive access to the lock. All other threads must wait on this lock when they attempt an `omp_set_lock`. Only a thread owning a lock can unlock it. The result of a thread attempting to unlock a lock owned by another thread is undefined. Both of these operations are illegal prior to initialization or after the destruction of a lock. The function `omp_test_lock` can be used to attempt to set a lock. If the function returns a non-zero value, the lock has been successfully set, otherwise the lock is currently owned by another thread.

Similar to recursive mutexes in Pthreads, OpenMP also supports nestable locks that can be locked multiple times by the same thread. The lock object in this case is `omp_nest_lock_t` and the corresponding functions for handling a nested lock are:

```
#include <omp.h>

void omp_init_nest_lock (omp_nest_lock_t *lock);
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
void omp_set_nest_lock (omp_nest_lock_t *lock);
void omp_unset_nest_lock (omp_nest_lock_t *lock);
int omp_test_nest_lock (omp_nest_lock_t *lock);
```

The semantics of these functions are similar to corresponding functions for simple locks. Notice that all of these functions have directly corresponding mutex calls in Pthreads.

7.10.6 Environment Variables in OpenMP

OpenMP provides additional environment variables that help control execution of parallel programs. These environment variables include the following.

`OMP_NUM_THREADS` This environment variable specifies the default number of threads created upon entering a `parallel` region. The number of threads can be changed using either the `omp_set_num_threads` function or the `num_threads` clause in the `parallel` directive. Note that the number of threads can be changed dynamically only if the variable OMP_SET_DYNAMIC is set to TRUE or if the function omp_set_dynamic has been called with a non-zero argument. For example, the following command, when typed into csh prior to execution of the program, sets the default number of threads to 8.

```
setenv OMP_NUM_THREADS 8
```

OMP_DYNAMIC This variable, when set to TRUE, allows the number of threads to be controlled at runtime using the omp_set_num_threads function or the num_threads clause. Dynamic control of number of threads can be disabled by calling the omp_set_dynamic function with a zero argument.

OMP_NESTED This variable, when set to TRUE, enables nested parallelism, unless it is disabled by calling the omp_set_nested function with a zero argument.

OMP_SCHEDULE This environment variable controls the assignment of iteration spaces associated with for directives that use the runtime scheduling class. The variable can take values static, dynamic, and guided along with optional chunk size. For example, the following assignment:

```
setenv OMP_SCHEDULE "static,4"
```

specifies that by default, all for directives use static scheduling with a chunk size of 4. Other examples of assignments include:

```
setenv OMP_SCHEDULE "dynamic"
setenv OMP_SCHEDULE "guided"
```

In each of these cases, a default chunk size of 1 is used.

7.10.7 Explicit Threads versus OpenMP Based Programming

OpenMP provides a layer on top of native threads to facilitate a variety of thread-related tasks. Using directives provided by OpenMP, a programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc. This convenience is especially useful when the underlying problem has a static and/or regular task graph. The overheads associated with automated generation of threaded code from directives have been shown to be minimal in the context of a variety of applications.

However, there are some drawbacks to using directives as well. An artifact of explicit threading is that data exchange is more apparent. This helps in alleviating some of the overheads from data movement, false sharing, and contention. Explicit threading also provides a richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations as illustrated in Section 7.8. Finally, since explicit threading is used more widely than OpenMP, tools and support for Pthreads programs is easier to find.

A programmer must weigh all these considerations before deciding on an API for programming.

7.11 Bibliographic Remarks

A number of excellent references exist for both explicit thread-based and OpenMP-based programming. Lewis and Berg [26, 25] provide a detailed guide to programming with Pthreads. Kleiman, Shah, and Smaalders [20] provide an excellent description of thread systems as well as programming using threads. Several other books have also addressed programming and system software issues related to multithreaded programming [30, 7, 15, 24, 34, 31].

Many other thread APIs and systems have also been developed and are commonly used in a variety of applications. These include Java threads [13, 29, 19, 22], Microsoft thread APIs [32, 10, 42, 4], and the Solaris threads API [20, 36]. Thread systems have a long and rich history of research dating back to the days of the HEP Denelcor [17] from both the software as well as the hardware viewpoints. More recently, software systems such as Cilk [5, 23], OxfordBSP [16], Active Threads [41], and Earth Manna [18] have been developed. Hardware support for multithreading has been explored in the Tera computer system [33], MIT Alewife [3], Horizon [21], simultaneous multithreading [39, 40], multiscalar architecture [14], and superthreaded architecture [38], among others.

The performance aspects of threads have also been explored. Early work on the performance tradeoffs of multithreaded processors was reported in [1, 35, 2, 11, 27]. Consistency models for shared memory have been extensively studied. Other areas of active research include runtime systems, compiler support, object-based extensions, performance evaluation, and software development tools. There have also been efforts aimed at supporting software shared memory across networks of workstations. All of these are only tangentially related to the issue of programming using threads.

Due to its relative youth, relatively few texts exist for programming in OpenMP [8]. The OpenMP standard and an extensive set of resources is available at `http://www.openmp.org`. A number of other articles (and special issues) have addressed issues relating to OpenMP performance, compilation, and interoperability [6, 9, 12, 28, 37].

Problems

7.1 Estimate the time taken for each of the following in Pthreads:

- Thread creation.
- Thread join.
- Successful lock.
- Successful unlock.
- Successful trylock.
- Unsuccessful trylock.

- Condition wait.
- Condition signal.
- Condition broadcast.

In each case, carefully document the method used to compute the time for each of these function calls. Also document the machine on which these observations were made.

7.2 Implement a multi-access threaded queue with multiple threads inserting and multiple threads extracting from the queue. Use mutex-locks to synchronize access to the queue. Document the time for 1000 insertions and 1000 extractions each by 64 insertion threads (producers) and 64 extraction threads (consumers).

7.3 Repeat Problem 7.2 using condition variables (in addition to mutex locks). Document the time for the same test case as above. Comment on the difference in the times.

7.4 A simple streaming media player consists of a thread monitoring a network port for arriving data, a decompressor thread for decompressing packets and generating frames in a video sequence, and a rendering thread that displays frames at programmed intervals. The three threads must communicate via shared buffers – an in-buffer between the network and decompressor, and an out-buffer between the decompressor and renderer. Implement this simple threaded framework. The network thread calls a dummy function `listen_to_port` to gather data from the network. For the sake of this program, this function generates a random string of bytes of desired length. The decompressor thread calls function `decompress`, which takes in data from the in-buffer and returns a frame of predetermined size. For this exercise, generate a frame with random bytes. Finally the render thread picks frames from the out buffer and calls the display function. This function takes a frame as an argument, and for this exercise, it does nothing. Implement this threaded framework using condition variables. Note that you can easily change the three dummy functions to make a meaningful streaming media decompressor.

7.5 Illustrate the use of recursive locks using a binary tree search algorithm. The program takes in a large list of numbers. The list is divided across multiple threads. Each thread tries to insert its elements into the tree by using a single lock associated with the tree. Show that the single lock becomes a bottleneck even for a moderate number of threads.

7.6 Improve the binary tree search program by associating a lock with each node in the tree (as opposed to a single lock with the entire tree). A thread locks a node when it reads or writes it. Examine the performance properties of this implementation.

7.7 Improve the binary tree search program further by using read-write locks. A thread read-locks a node before reading. It write-locks a node only when it needs to write into the tree node. Implement the program and document the range of program parameters where read-write locks actually yield performance improvements over regular locks.

7.8 Implement a threaded hash table in which collisions are resolved by chaining. Implement the hash table so that there is a single lock associated with a block of k hash-table entries. Threads attempting to read/write an element in a block must first lock the corresponding block. Examine the performance of your implementation as a function of k.

7.9 Change the locks to read-write locks in the hash table and use write locks only when inserting an entry into the linked list. Examine the performance of this program as a function of k. Compare the performance to that obtained using regular locks.

7.10 Write a threaded program for computing the Sieve of Eratosthenes. Think through the threading strategy carefully before implementing it. It is important to realize, for instance, that you cannot eliminate multiples of 6 from the sieve until you have eliminated multiples of 3 (at which point you would realize that you did not need to eliminate multiples of 6 in the first place). A pipelined (assembly line) strategy with the current smallest element forming the next station in the assembly line is one way to think about the problem.

7.11 Write a threaded program for solving a 15-puzzle. The program takes an initial position and keeps an open list of outstanding positions. This list is sorted on the "goodness" measure of the boards. A simple goodness measure is the Manhattan distance (*i.e.*, the sum of x-displacement and y-displacement of every tile from where it needs to be). This open list is a work queue implemented as a heap. Each thread extracts work (a board) from the work queue, expands it to all possible successors, and inserts the successors into the work queue if it has not already been encountered. Use a hash table (from Problem 7.9) to keep track of entries that have been previously encountered. Plot the speedup of your program with the number of threads. You can compute the speedups for some reference board that is the same for various thread counts.

7.12 Modify the above program so that you now have multiple open lists (say k). Now each thread picks a random open list and tries to pick a board from the random list and expands and inserts it back into another, randomly selected list. Plot the speedup of your program with the number of threads. Compare your performance with the previous case. Make sure you use your locks and trylocks carefully to minimize serialization overheads.

7.13 Implement and test the OpenMP program for computing a matrix-matrix product in Example 7.14. Use the OMP_NUM_THREADS environment variable to control the number of threads and plot the performance with varying numbers of threads. Consider three cases in which (i) only the outermost loop is parallelized; (ii) the outer two loops are parallelized; and (iii) all three loops are parallelized. What is the observed result from these three cases?

7.14 Consider a simple loop that calls a function `dummy` containing a programmable delay. All invocations of the function are independent of the others. Partition this loop across four threads using `static`, `dynamic`, and `guided` scheduling. Use different parameters for static and guided scheduling. Document the result of this experiment as the delay within the `dummy` function becomes large.

7.15 Consider a sparse matrix stored in the compressed row format (you may find a description of this format on the web or any suitable text on sparse linear algebra). Write an OpenMP program for computing the product of this matrix with a vector. Download sample matrices from the Matrix Market (`http://math.nist.gov/MatrixMarket/`) and test the performance of your implementation as a function of matrix size and number of threads.

7.16 Implement a producer-consumer framework in OpenMP using `sections` to create a single `producer` task and a single `consumer` task. Ensure appropriate synchronization using locks. Test your program for a varying number of producers and consumers.

Bibliography

[1] Anant Agarwal. Performance tradeoffs in multithreaded processors. Technical Report 89-566, Massachusetts Institute of Technology, Microsystems Program Office, Cambridge, MA, 1989.

[2] Anant Agarwal. Performance tradeoffs in multithreaded processors. Technical report MIT/LCS/TR 501; VLSI memo no. 89-566, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1991.

[3] Anant Agarwal, Geoffrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Daniel Nussbaum, Mike Parkin, and Donald Yeung. The MIT alewife machine : A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic, 1991.

[4] Jim Beveridge and Robert Wiener. *Multithreading Applications in Win32: the Complete Guide to Threads*. Addison-Wesley Developers Press, Reading, MA, 1997.

[5] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[6] Randall Bramley. Technology news & reviews: Chemkin software; OpenMP Fortran Standard; ODE toolbox for Matlab; Java products; Scientific WorkPlace 3.0. *IEEE Computational Science and Engineering*, 4(4):75–78, October/December 1997.

[7] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, MA, 1997.

[8] Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon (editors). *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[9] B. Chapman and P. Mehrotra. OpenMP and HPF: Integrating two paradigms. *Lecture Notes in Computer Science*, 1470, 1998.

[10] Aaron Cohen, Mike Woodring, and Ronald Petrusha. *Win32 Multithreaded Programming*. O'Reilly & Associates, 1998.

[11] David E. Culler, Michial Gunter, and James C. Lee. Analysis of multithreaded microprocessors under multiprogramming. Report UCB/CSD 92/687, University of California, Berkeley, Computer Science Division, Berkeley, CA, May 1992.

[12] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January/March 1998.

[13] Donald G. Drake. Introduction to Java threads. *JavaWorld: IDG's magazine for the Java community*, 1(2), April 1996.

[14] Manoj Franklin. The multiscalar architecture. Technical Report CS-TR-1993-1196, University of Wisconsin, 1993.

[15] Bill O. Gallmeister. *Posix. 4 : Programming for the Real World*. O'Reilly & Associates, 1995.

[16] J. Hill, S. Donaldson, and A. McEwan. Installation and user guide for the oxford bsp toolset: User guide for the oxford bsp toolset (v1.3) implementation of bsplib. Technical report, Oxford University Computing Laboratory, 1997.

[17] R. E. Hiromoto, O. M. Lubeck, and J. Moore. Experiences with the Denelcor HEP. *Parallel Computing*, 1(3–4):197–206, 1984.

[18] H. Hum, O. Maquelin, K. Theobald, X. Tian, and G. Gao. A study of the earth-manna multithreaded system. *Intl. J. of Par. Prog.*, 24:319–347, 1996.

[19] Paul Hyde. *Java Thread Programming*. Sams, 1999.

[20] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads*. SunSoft Press, Mountainview, CA, 1995.

[21] J. Kuehn and B. Smith. The horizon supercomputing system: architecture and software. In *Proceedings of Supercomputing Conference*, pages 28–34, 1988.

[22] Doug Lea. *Concurrent Programming in Java, Second Edition: Design Principles and Patterns*. Addison-Wesley, 1999.

[23] C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, CA, 1995.

[24] Donald A. Lewine. *Posix Programmer's Guide : Writing Portable Unix Programs with the Posix. 1 Standard*. O'Reilly & Associates, 1991.

[25] Bil Lewis and Daniel J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. Prentice Hall PTR/Sun Microsystems Press, 1995.

[26] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall PTR/Sun Microsystems Press, 1997.

[27] Beng-Hong Lim and Ricardo Bianchini. Limits on the performance benefits of multithreading and prefetching. Research report RC 20238 (89547), IBM T. J. Watson Research Center, Yorktown Heights, NY, October 1995.

[28] Honghui Lu, Charlie Hu, and Willy Zwaenepoel. OpenMP on networks of workstations. In *SC '98, High Performance Networking and Computing Conference*, Orlando, Florida, 1998.

[29] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 1999.

[30] Bradford Nichols, Bick Buttlar, and Jackie Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Newton, MA 02164, 1996.

[31] Scott J. Norton and Mark D. DiPasquale. *Thread time: the multithreaded programming guide*. Hewlett-Packard professional books. Prentice-Hall, Englewood Cliffs, NJ 07632, 1996.

[32] Thuan Q. Pham and Pankaj K. Garg. *Multithreaded Programming with Win32*. Prentice Hall, 1998.

[33] A. Porterfield R. Alverson, D. Callahan, D. Cummings, B. Koblenz and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, 1990.

[34] Kay A. Robbins, Steven Robbins, Kay Ret Robbins, and Steve Robbins. *Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading*. Prentice Hall, 1996.

[35] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten Von Eiken. Analysis of multithreaded architectures for parallel computing. Report UCB/CSD 90/569, University of California, Berkeley, Computer Science Division, Berkeley, CA, April 1990.

[36] SunSoft. *Solaris multithreaded programming guide*. SunSoft Press, Mountainview, CA, 1995.

[37] Joe Throop. Standards: OpenMP: Shared-memory parallelism from the ashes. *Computer*, 32(5):108–109, May 1999.

[38] Jean-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, 1996.

[39] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.

[40] Dean Michael Tullsen. *Simultaneous multithreading*. PhD thesis, University of Washington, Seattle, WA, 1996.

[41] B. Weissman. Active threads manual. Technical Report TR-97-037, International Computer Science Institute, Berkeley, CA 94704, 1997.

[42] Al Williams. *Windows 2000 Systems Programming Black Book*. The Coriolis Group, 2000.

Index