Contents

CHAPTER 6

Programming Shared Address Space Machines

Explicit parallel programming requires specification of parallel tasks along with their interactions. These interactions may be in the form of synchronization or communication. 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 support constructs for expressing concurrency and synchronization along with techniques for minimizing associated overheads. In this chapter, we will discuss these programming paradigms along with their performance aspects, and related extensions to directive-based languages.

As we briefly discussed in Chapter **??**, shared address space programming paradigms can vary widely on issues of 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. 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. In contrast, lightweight processes and threads assume that all memory is global. This is the preferred programming model for parallel programming and will be discussed extensively in this chapter. Directive based programming models extend the threaded model by facilitating creation and synchronization of threads.

is there a need for more on processes/LWPs/Threads here?

6.1 Thread Basics

Consider the following code segment that computes the product of two dense matrices.

```
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 performed independently. Such an independent sequence of instructions is referred to as a thread. A thread can be viewed as a single stream of control in the flow of a program. 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.

Logical Machine Model of a Thread To execute the above code fragment concurrently on multiple processors, each processor must have logically uniform access to matrices a, b, and c. This is accomplished through a shared address space. All memory in the logical machine model of a thread is globally accessable as illustrated in Figure 6.1(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 and will be discussed in greater detail subsequently. At this point, we will simply treat the stack as being local to the thread. This implies a logical machine model illustrated in Figure 6.1(b).

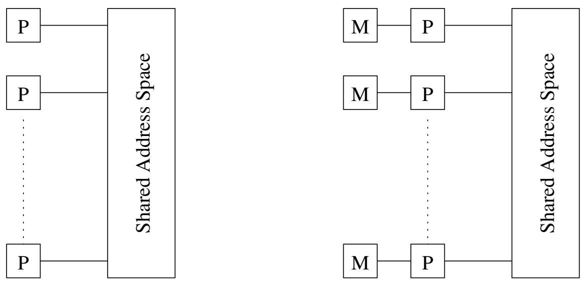

Figure 6.1 The logical machine model of a thread-based programming paradigm.

While this logical machine model gives the illusion of an equally accessible address space, 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, the presence of caches with processors skews memory access time. Issues of locality of memory reference become important for extracting performance from these machines. These issues form the focus of large parts of this chapter.

6.2 Why Threads?

Threaded programming models offer significant advantages over other programming models (along with some disadvantages as well). Let us briefly look at some of these:

6.2.1 Advantages of Threaded Models

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 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 and communication. By allowing multiple threads to execute on the same processor, threaded APIs allow this communication latency to be hidden. In effect, while one thread is waiting for a communication operation, other threads can utilize the CPU. Consequently, it is possible to use threads to minimize idling overheads.

Scheduling and Load Balancing In many applications, the degree of concurrency is higher than the available number of processors. In these cases, concurrent subtasks have to be scheduled on to processors to maximize locality. In other applications, tasks may be dynamically generated and the programmer must balance load periodically to ensure performance. Thread systems support scheduling and load balancing using a range of techniques. By providing this support at the system level, threaded APIs rid the programmer of the burden of explicit scheduling and load balancing.

Serial Performance By masking the latency of I/O and other operations that do not use the CPU, threaded programs are capable of yielding much better serial performance. In fact, most I/O intensive programs are now written in threaded APIs. In addition, serializations of good threaded programs are also capable of yielding better memory system performance. This is a result of the locality of memory access within threads; which in turn leads to better memory system performance.

Ease of Programming, Widespread Use Due to all of the advantages mentioned above, threaded programs are significantly easier to write than corresponding programs in message passing. (Achieving identical levels of performance for the two programs requires additional effort though.) Furthermore, threads have found extensive use in serial processing as well where they are used for user interfaces, device handlers, latency management etc. With widespread acceptance of the POSIX standard, it can be expected that tools for multithreaded programming will be easily available. These issues are very important from the program development and software engineering aspects.

6.3 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. Vendors such as Sun, DEC, HP, and SGI have recognised and support this standard. We will use POSIX standard for introducing multithreading concepts. The concepts themselves are largely independent of the API and can be used for programming with other thread libraries as well. All the illustrative programs presented in this chapter can be executed on workstations as well as parallel computers that support Pthreads.

6.4 Thread Basics: Creating and Terminating Threads

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

Example 6.1 Threaded program for computing PI.

The method we chose here is based on generating random numbers in a unit length square. We keep track of the number of points that fall within a 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 ratio of random points that fall in the circle to total points should approach $\pi/4$ for a large number of points.

A simple threaded strategy for generating the value of π assigns a fixed number of points to each thread. The thread generates these random points and keeps track of the ratio of hits locally. After all threads finish execution, their ratios are combined to compute the value of π.

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 using the `pthread_create` function. The prototype of this function is as follows:

```
#include <pthread.h>
int
pthread_create (
         pthread_t   *thread,
         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 invocation of a thread, a unique identifier is associated with the thread and assigned to the location pointed to by `thread`. The thread has the attributes described by the `attribute` artument. When this argument is NULL, a thread with default attributes is created. We will discuss the `attribute` parameter in detail in a later section. 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` 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 infact 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 in some execution instances, but not in others. On successful creation of a thread, `pthread_create` returns 0; else it returns the error code. The reader is referred to the Pthreads manual for a detailed description of the error-codes.

In our program for computing the value of PI, we first read in the desired number of threads `num_threads` and the desired number of sample points `sample_points`. The 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 encountered by each thread upon return. The program creates `num_threads` threads using the `pthread_create` function. This function calls a specified function `compute_pi` as a thread.

Once the respective `compute_pi` threads have generated assigned number of random points and computed their hit ratios, the results have to be combined to determine π. The main program must therefore 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 with the master thread, the value of π is computed by multiplying the combined hit ratio by 4.0. This value is then printed by the master thread. The program for doing this is as follows:

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

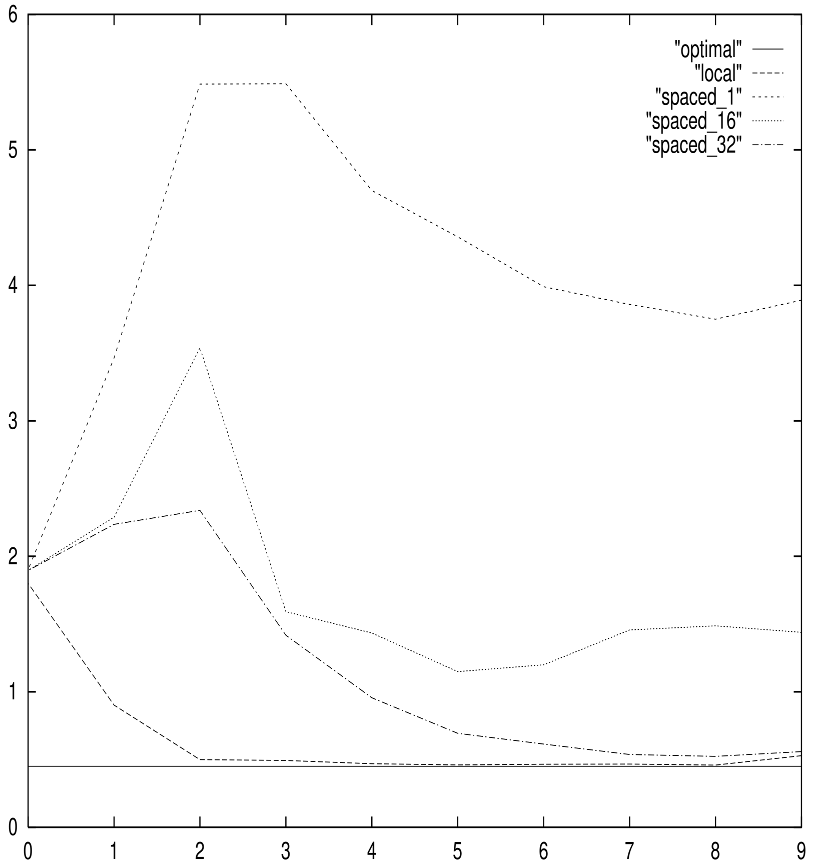

Figure 6.2 Execution time of the `compute_pi` program as a function of number of threads.

```

#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 start_time, end_time;
        struct timeval tz;
        struct timezone tx;
```

```
        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);

        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));

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

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

        hit_pointer = (int *) s;
        seed = *hit_pointer;

        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)
                        hits ++;
                seed *= i;
        }
        *hit_pointer = hits;
        pthread_exit(0);
}
```

The first thing the reader must note in the above example is 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 *reentrant*. All functions called by threads must be reentrant. Reentrant functions are those that can be safely called when another invocation has been suspended in the middle of its invocation. It is easy to see why thread functions must be reentrant because a thread can be pre-empted 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.

We execute this program on a four processor SGI Origin 2000. The log of number of threads and execution time are illustrated Figure 6.2 (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.

6.5 Synchronization Primitives in POSIX

6.5.1 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 `best_cost` are 50 and 75 at threads t1 and t2. 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, and more importantly, the value 75 of `best_cost` is inconsistent in the sense that no serialization of the two threads can possibly yield this result. This is an undesirable situation, sometimes also referred to as a race condition (so called because the result of the computation depends on the race between competing entities).

The above situation occured 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 are infact non-atomic. a statement of the form `global_count += 5` may in fact comprise of several assembler instructions, and therefore must be handled carefully.

All threaded APIs provide support for implementing critical sections and atomic operations using mutex-locks (mutually exclusive 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 associated with a piece of shared data. To access the shared data, a thread must first try to aquire a mutex-lock. If the mutex is already locked, the process trying to aquire the lock is blocked. This is because a locked mutex implies that there is another thread currently accessing the critical section and no other thread must be allowed in. When a thread leaves a critical section, it must unlock the mutex so that other threads can enter the critical section. All mutexes must be initialized to the unlocked state in the beginning of the program.

POSIX threads provide a number of functions for handling mutex-locks. The function `pthread_mutex_lock` can be used to attempt a lock on a mutex. 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 `mutex_lock`. (The data type of `mutex_lock` is predefined to be `pthread_mutex_t`.) If the mutex is already locked, the calling thread blocks; else the mutex 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 etc.

On leaving the critical section, a thread must unlock the mutex 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. The prototype of this function is:

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

In the case of a normal mutex, the mutex is actually unlocked and one of the blocked threads is scheduled to enter the critical section. (The specific thread is determined by the scheduling policy.) In the case of a recursive mutex, the lock count associated with the mutex is decremented and it is actually unlocked if the count reaches zero. The effect of this function is undefined if one tries to unlock a previously unlocked mutex or if the mutex was locked by another thread. The function returns 0 on successful completion; else it returns an error code.

We need one more function before we can start using mutex-locks: namely, a function to initialize a mutex-lock. 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 attributes are used (normal mutex). While using normal mutex, it is also possible to initialize it statically. The macro `PTHREAD_MUTEX_INITIALIZER` can be used to do this.

Example 6.2 Computing the minimum entry in a list of integers.

Armed with basic mutex-lock functions, let us write a threaded program to compute the minimum of a list of integers. The list is partitioned equally among the threads. The size of each threads 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;

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

        /* 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);
}
```

■

In the above example, the test-update operation for `minimum_value` was protected by the mutex `minimum_value_lock`. Threads executed a `pthread_mutex_lock` to gain exclusive access to the variable `minimum_value`. Once this access was 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 6.3 Producer-consumer work queues.

A common use of mutexes is in implementing 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 accomplish them. Let us consider a simple instance of this paradigm in which the task queue is only one task long (in a general case, the task queue may be longer but is typically of bounded size). A simplistic (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 accomplishing them. However, this simplistic version does not account for the following necessities:

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

To implement this, we can use a variable called `task_available`. If this variable is 0, consumer threads must wait, but 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 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 mutexes 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 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;
        while (!done()) {
                inserted = 0;
                create_task();
                while (inserted == 0) {
                        pthread_mutex_lock(&task_queue_lock);
                        if (task_available == 0) {
                                insert_into_queue();
                                inserted = 1;
                        }
                        pthread_mutex_unlock(&task_queue_lock);
                }
        }
}

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

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 is 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 signalled by setting `task_available` to 1. 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 was work on the queue to begin with. If the lock was not relinquished, the threads would deadlock since the 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 accomplishes 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. It is important to keep track of what goes into a critical section. For instance, 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 performance implications of this will be discussed later. 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.

While using locks, it is very important to keep two overheads in mind: locking overheads and idling overheads due to blocked threads. In the above program, the variable `current_index` is locked n times for initializing an array of length n. Clearly, a scenario as this will not yield any speedup. More specifically, it is important for the programmer to identify and minimize locking overheads. In many programs, this is possible by increasing thread granularity. In the above program, this corresponds to incrementing `current_index` by k where $k > 1$ on every lock. By adjusting the value of k, it is possible to reduce the locking overhead effectively.

It is often possible to reduce the idling overhead associated with locks as well. Pthreads provides an alternative function `pthread_mutex_trylock`. The prototype of this function is

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

This function attempts a lock on `mutex_lock`. If the lock is successful, the function returns a zero. If the 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 poll the mutex for a lock. This can be used to reduce idling overhead.

It is often possible to reduce the idling overhead associated with locks as well. Pthreads provides an alternative function `pthread_mutex_trylock`. The prototype of this function is

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

This function attempts a lock on `mutex_lock`. If the lock is successful, the function returns a zero. If the 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 poll the mutex for a lock. This can be used to reduce idling overhead.

Example 6.4 Finding k matches to a database query.

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

```
main() {
        /* initialize data structures here */

        /* initialize start_pointer to start of threads sub-list */

        /* create threads */
        for (i=0; i< NUM_THREADS; i++)
                pthread_create(&threads[i], &attr, find_entries,
                        (void *) start_pointer);
        for (i=0; i< NUM_THREADS; i++)
                pthread_join(threads[i], NULL);
}

void *find_entries(void *start_pointer) {
        struct database_record *next_record;
        int count;
        do {
                next_record = search_next_entry(start_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.

The locking overhead can be alleviated by using the pthread_mutex_trylock function. 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 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);
        }
}
```

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`. The time for execution of this version is less than the first one on two counts: the time for executing a `pthread_mutex_trylock` is typically much smaller than that for a `pthread_mutex_lock`. Also, since multiple records may be inserted on each lock, the number of locking operations is also reduced. The number of records actually searched may be slightly higher 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, the overhead does not cause a slowdown in this piece of code.

■

The use of `pthread_mutex_trylock` to reduce locking and idling overhead by utilizing asynchrony is common to many other applications as well. Consider the following example from discrete optimization.

Example 6.5 Branch and Bound tree search.

Branch-and-bound tree search is a commonly used techniques for solving discrete optimization problems such as 0/1 integer programming. The objective is to assign a set of variables $x_0 \dots x_{n-1}$ vales of 0 or 1 so that a set of inequalities $Ax < b$ is satisfied and a given objective function $c.x$ is minimized. A simple version of the branch-and-bound method instantiates the variables (gives them a value 0 or 1) one at a time so that they do not violate the inequalities. This operation is called branching. The algorithm stores the value of the best solution that has been found at any point. If a partial instantiation (i.e. some variables have been assigned values but others have not) is guaranteed to yield a solution worse than the current best solution, the partial instantiation is discarded (bound). The best possible cost of a partial instantiation can be evaluated by assigning all components of the x vectors x_i with positive coefficients c_i to 0 and all components x_j with negative coefficients c_j to 1.

A simple threaded version of this program partitions the tree among threads. Each partial instantiation is handed to a thread. The thread instantiates the remaining components of the vector while satisfying the inequalities and checking if the bound set by current best solution is exceeded. The function for expanding a node rooted at a particular node (partial instantiation) is as follows:

```
void *expand_partial_instantiation(int *partially_instantiated_vector, int n) {
/* partially_instantiated_vector contains a pointer to the partially
instantiated vector and n is the number of elements that have been
instantiated */
        int *x;
        x = (int *)partially_instantiated_vector;
        for (instantiation = 0; instantiation < 2; instantiation ++) {
        x[n] = instantiation;
        if (check_inequalities(x, n+1) == 1) {
                if (n == num_components) {
                        this_cost = compute_cost(x);
                        pthread_mutex_lock(&cost_lock);
                        if (this_cost < current_best_cost)
                                current_best_cost = this_cost;
                        pthread_mutex_unlock(&cost_lock);
                }
                else {
                        if (compute_best_possible_cost(x, n+1) <
                                current_best_cost)
                                expand_partial_instantiation(x, n+1);
                }
        }
}
```

Each thread calls the function `expand_partial_instantiation` with a partial instantiation. The function performs a depth-first search of the instantiation tree while ensuring that all inequalities are satisfied (using the function `check_inequalities`). If the inequalities are satisfied, the function checks to see if it is a complete instantiation; if so, it compares and updates the cost of current best solution if needed. To do so, it locks the mutex `cost_lock` associated with `current_best_cost`, checks if the current solution cost `this_cost` is less than `current_best_cost` and updates it if necessary. The lock on `cost_lock` is then released. If the instantiation is not a complete instantiation, the best possible cost of this partial instantiation is compared to the current best cost. If the former is less than the latter, further instantiations are initiated.

The drawback of the above threaded program is that a thread must suspend the exploration of alternate instantiations while waiting for a mutex to test and update current best cost. When the number of solutions and threads is large this can lead to considerable contention and idling. The branch-and-bound algorithm is interesting in that a delayed update on the current best solution does not invalidate the solution found; merely that it may potentially result in computation that is not performed by the single-threaded version of the program. This allows us to use `pthread_mutex_trylock` instead of `pthread_mutex_lock` to update current best cost. The operation of the threaded program is as follows: a thread tries to lock and update the current nest cost. If the lock is not available, it sets a flag indicating pending updation and proceeds with further expansions. If there is a pending updation, the thread periodically tries to update the current best cost. The function for accomplishing this is as follows:

```
void *expand_partial_instantiation(int *partially_instantiated_vector, int n) {
/* partially_instantiated_vector contains a pointer to the partially
instantiated vector and n is the number of elements that have been
instantiated */
        int *x, lock_status;
        x = (int *)partially_instantiated_vector;
        for (instantiation = 0; instantiation < 2; instantiation ++) {
        x[n] = instantiation;
        if (check_inequalities(x, n+1) == 1) {
                if (n == num_components) {
                        this_cost = compute_cost(x);
                        if ((pending_update == 1)&&(previous_cost < this_cost))
                                this_cost = previous_cost;
                        lock_status = pthread_mutex_trylock(&cost_lock);
                        if (lock_status == EBUSY) {
                                pending_update = 1;
                                previous_cost = this_cost;
                        }
                        else {
                                if (this_cost < current_best_cost)
                                        current_best_cost = this_cost;
                                pending_update = 0;
                                pthread_mutex_unlock(&cost_lock);
                        }
                }
                else {
                        if (compute_best_possible_cost(x, n+1) <
                                current_best_cost)
                                expand_partial_instantiation(x, n+1);
                }
        }
}
```

In this version of the function `expand_partial_instantiation`, on finding a complete instantiation, we check to see if there was a previously pending update on current best cost. If there was none, we attempt a lock on the mutex. If the lock is available, the current best cost is updated as required; else, the pending update flag is signaled and the cost of the current instantiation is stored as `previous_cost`. The exploration of alternate instantiations continues until the next solution is found. If the pending update has been signaled, then the better of the current and previous solutions is used to update the current best solution. In this way, there is no idling associated with the non-availability of the mutex for updating current best solution.

The above function only updates current best cost on finding a subsequent solution. This may not be the best strategy; indeed, one may periodically check and update the solution. This version is left as an exercise to the reader (Exercise **??**).

■

The above examples illustrate 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 mutexes. 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. This is also influenced by 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`.

Types of Locks The POSIX API supports three different kinds of locks. All of these locks use the same functions for locking and unlocking; however, the type od lock is determined by the lock attribute. The mutex lock used in the previous example is called a *normal mutex*. This is the default type of a lock. Only a single thread is allowed to lock a *normal mutex* once at any point of time. If the same thread attempts to lock the mutex, it 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 motex that it holds a lock on. For addressing such situations, the POSIX API supports a recursive 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). Recursive mutex is useful when a thread function needs to call itself recursively.

In addition to normal and recursive mutexes, a third kind of mutexes called an errorcheck 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 normal mutex, when a thread attempts a lock on a mutex it has already locked, instead of dead-locking, it returns an error. Therefore, an errorcheck mutex is more useful for debugging purposes.

A mutex can be set to one of the three types using the function `pthread_mutexattr_settype_np`. This function is discussed in greater length when discussing attribute objects for threads and synchronization objects.

6.5.2 Condition Variables for Synchronization

As we noted in the previous section, indiscriminate use of the `pthread_mutex_lock` function can result in idling overhead from blocked threads. While `pthread_mutex_trylock` alleviated this overhead, it introduced 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. 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 a 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 all 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, tests the predicate defined over the shared variable (in this case `task_available`) and 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);
```

The function takes the condition variable `cond` and the mutex `mutex` as arguments. A call to this function blocks the execution of the thread and releases the lock on `mutex`. The calling thread remains blocked until it receives a signal from another thread or is interrupted by a UNIX 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 to be 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 un-blocked and when the mutex becomes available, it is handed to this thread (and the thread becomes runnable).

The producer thread produces the task and since the lock on `mutex` has been relinquished, 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 signalling 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 atleast one thread that is currently waiting on the condition variable `cond`. It then relinquishes its lock on `mutex` 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`) and whose attributes are defined in the attribute object `attr`. Attribute objects for condition variables are discussed in Section **??**. For now, we will set this pointer to `NULL` and use 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 program.

Example 6.6 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 condition associated with `cond_queue_empty` is asserted when the variable `task_available` becomes 0 and `cond_queue_full` is asserted when `task_available` becomes 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 consimer 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 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);
        }
}
```

One important thing 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 UNIX signals and others). In other cases, when the condition variable is signaled using a condition broadcast (`pthread_cond_broadcast` is dicussed later), one of the threads that got the lock earlier might invalidate the condition. In the example of multiple producers, multiple consumers, a task available on the work queue might be consumed by one of the other consumers.

■

In the above example, each task could only be consumed by 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 **??**. Another example of the use of `pthread_cond_broadcast` is in the implementation of barriers illustrated in Section **??**.

It is often useful to build time-outs into condition waits. Using the `pthread_cond_timedwait` function, 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 recieve 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.

6.6 Controlling Thread and Synchronization Attributes

We have, until now discussed threads and synchronization objects, namely mutexes and condition variables. We have seen that there are different attributes (types) associated with threads, locks, and condition variables. For example, different threads may be scheduled differently (round-robin, prioritized, time-sliced etc), they may have different stack sizes and so on. Similarly, a synchronization variable such as a mutex-lock may be a normal mutex, a recursive mutex, etc. In our discussion thus far, we have not discussed how to create threads and variables with specific properties. Instead we have relied on the default properties of these objects. Posix threads allow us to change these default properties by using attribute objects.

An attributes object is a data-structure that describes thread properties. When creating a thread or a synchronization variable, we can specify the attributes object that determines the properties of the thread. 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 attribute objects do not change the thread properties created using the object.

There are several advantages of using attribute 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, he 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. Let us look at pthreads functions for doing these for threads and synchronization variables separately.

6.6.1 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. Individual properties of the attributes object may then be changed using the following functions:

- pthread_attr_setdetachstate
- pthread_attr_setguardsize_np
- pthread_attr_setinheritsched
- pthread_attr_setschedparam
- pthread_attr_setschedpolicy
- pthread_attr_setstacksize

Upon successful completion, the function returns a 0, else it returns an error condition. The attributes object may be destroyed using the function pthread_attr_destroy. The prototype of the function is:

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

The call returns a 0 on successful removal of the attributes object attr, else it returns an error value.

Scheduling Policy for Threads

The default scheduling policy for a thread is inherited from the creating thread. This is controlled by the inherit scheduling attribute. To change this, pthreads provides a function pthread_attr_setinheritsched. The prototype of this function is as follows:

```
int
pthread_attr_setinheritsched (
          pthread_attr_t   *attr,
          int   inheritsched);
```

The value of inheritsched determines the properties of the attributes variable attr. The variable inheritsched can take the following values:

- PTHREAD_INHERIT_SCHED: This is the default value in which the thread inherits its properties from the creating thread. Scheduling attributes in the attributes object are ignored.
- PTHREAD_EXPLICIT_SCHED: This indicates that the scheduling of the thread is specified in the scheduling attributes of the object.

Scheduling Policy Attribute If the inherit policy is set to `PTHREAD_EXPLICIT_SCHED`, this attribute determines how a thread is scheduled for execution. Pthreads supplies the function `pthread_attr_setschedpolicy` for setting the scheduling policy. The prototype of this function is:

```
int
pthread_attr_setschedpolicy (
        pthread_attr_t   *attr,
        int   policy);
```

The integer `policy` determines the scheduling policy and can be one of the following:

- **SCHED_FIFO** (first-in/first-out) In this scheduling mechanism, a ready thread with the highest priority executes until it is blocked or another thread with a higher priority becomes ready. In the latter case, the currently running thread is pre-empted and the new thread is scheduled. If multiple threads with highest identical priorities exist, then the first thread continues execution until it blocks.

- **SCHED_RR** (round-robin) This is identical to SCHED_FIFO, except, when multiple threads exist with identical highest priority, they are time-sliced.

- **SCHED_FG_NP** This is the default scheduling policy in which threads are time-sliced irrespective of their priority. Higher priority threads still get more execution time than lower priority threads. However, if SCHED_FIFO or SCHED_RR threads exist in the system, they may starve SCHED_FG_NP threads.

- **SCHED_BG_NP** (Background) This is similar to SCHED_FG_NP, however, threads scheduled as SCHED_BG_NP get less execution time in the presence of SCHED_FIFO or SCHED_RR threads than SCHED_FG_NP.

Pthreads allows us to change the scheduling policy associated with an attributes object. The prototype of the function is:

```
int
pthread_attr_setschedpolicy (
        pthread_attr_t   *attr,
        int   policy);
```

Here, `attr` specifies the attributes object to be modified and `policy` specifies the new value of the scheduling policy. The values for the `policy` are `SCHED_BG_NP`, `SCHED_FG_NP`, `SCHED_FIFO`, and `SCHED_RR`. These reflect the scheduling policy.

It is possible to change the scheduling policy of an existing thread (and, at the same time, the scheduling parameters) using the `pthread_setschedpar` function. The prototype of the function is:

```
int
pthread_attr_setschedparam (
        pthread_attr_t   *attr,
        const struct sched_param   *param);
```

Here, `param` is a structure that specifies new values for the scheduling parameters depending on the value of the scheduling policy specified in in `attr`. The `sched_priority` field of `*param` is used to specify the priority of the thread. To specify thread priority, the following steps must be performed. (i) the scheduling inheritance must be disabled using `pthread_attr_setinheritsched` function; (ii) set the scheduling policy using the function `pthread_attr_setschedpolicy`.

The variable `sched_priority` can take valid value ranges depending on the scheduling policy. The minimum and maximum ranges are as follows:

```
Policy            Low               High
SCHED_FIFO        PRI_FIFO_MIN      PRI_FIFO_MAX
SCHED_RR          PRI_RR_MIN        PRI_RR_MAX
SCHED_OTHER       PRI_OTHER_MIN     PRI_OTHER_MAX
SCHED_FG_NP       PRI_FG_MIN_NP     PRI_FG_MAX_NP
SCHED_BG_NP       PRI_BG_MIN_NP     PRI_BG_MAX_NP
```

The function `pthread_attr_setschedparam` returns 0 on successful completion.

Other Thread Parameters

POSIX threads also provides functions for controling other thread parameters such as stack size and guard size. Guard size refers to the size of the guard area following the stack that is used to prevent or detect stack overflow. For more details on the functions `pthread_attr_setstacksize` and `pthread_attr_setguardsize_np`, we refer the reader to the POSIX thread standard.

6.6.2 Attribute Objects for Synchronization Variables

In our discussion of mutex-locks, we had seen that there are different kinds of locks. The type of a 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`. Thr 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 prototypes of 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`

These mutex types correspond to the mutexes discussed in Section **??**. The function returns a 0 on successful completion.

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

6.7 Thread Cancellation

Let us consider a simple program to evaluate a set of positions in a chess game. Assume that there are k moves, each being evaulated 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 these board positions must be cancelled. Posix threads provides this cancellation feature in the function `pthread_cancel`. The prototype of this function is:

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

Here, `thread` is the thread to be cancelled. A thread may cancel itself or cancel other threads. When a call to this function is made, a cancellation is send to the specified routine. 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, clean-up routines are invoked for reclaiming the thread data structures. After this the thread is cancelled. This process is similar to termination of a thread using the `pthread_exit` call. This is performed independent of the thread that made the original request for cancellation. The `pthread_cancel` 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 cancelled; it implies that the specified thread is a valid thread for cancellation.

We will see examples of thread cancellation when we talk about the use of threads for programming task parallel applications in Section **??**.

6.8 Composite Synchronization Constructs

While POSIX provides a basic set of synchrinization constructs, often, there is a need for higher level constructs. These higher level constructs can be built using mutex locks and condition variables. In this section, we look at some of these constructs along with their performance aspects and applications.

6.8.1 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 structure aquires a read-lock on the variable. A read lock is allowed to go through when there are other threads that already have the read lock. 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 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 specified 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`, else, 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` inlocks a read or write lock. It checks to see if there is a write lock, 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 him 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));
}
```

Example 6.7 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 value and updated it if necessary. In general, the number of times the value is examined is far 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. 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);
}
```

In this example, each thread computes the minimum element in the partial list. It then attempts a read lock on the loac 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 gains 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 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 requires an updation of the global minimum, the read locks are superfluous and add overheads to the program.

Example 6.8 Using read-write locks for implementing hash tables.
A commonly used operation in applications ranging from database querying 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);
}
```

This formulation works well when there is a significant number of insertions being performed. However, when a large fraction of the queries are found in the hash table, the searches are serialized. It is easy to see that multiple threads can be safely allowed to search the hash table, but 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, 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 (region between lock and unlock) becomes larger, read-write locks offer more advantages. This is because the serialization overhead paid by normal mutexes is much higher. Finally, since read-write locks rely on condition variables, the underlying thread system must provide fast condition wait, signal, and broadcast. It is possible to do a simple analysis to understand the relative merits of read-write locks. This is explored in Exercise **??**.

6.8.2 Barriers

An important and often used construct in threaded (as well as other parallel) programs is the `barrier`. A barrier call is used to hold a thread until all other threads participating in the barrier have reached the barrier. A simple example of this is in the use of multiple threads to compute the variance and standard deviation of a list of numbers. Each thread is given a partial list. The threads compute sums of these partial lists and together compute the total sum. This sum is then used to compute the mean of the numbers in the list.

However, the mean cannot be computed until all threads have added their contributions to the overall sum. Therefore, we need a barrier before the mean can be computed.

Barriers can be implemented using a counter, a mutex and a condition variable. (They can also be implemented simply using mutexes, however, they barriers suffer from the overheads 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 between them one after the other. This implementation of a barrier can be speeded up by using multiple barrier variables `mylib_barrier_t`.

Let us consider an alternate barrier implementation in which there are $n/2$ condition variable-mutex pairs for implementing a barrier with 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 8 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 desired barrier point and releases threads by signaling the condition `ok_to_proceed_down`. When all threads are released, the barrier is completed.

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 take 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 linear in the number of threads even on multiple processors. In Figure 6.3, 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 in asymptotics as the 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 ideal case is difficult to achieve without being able to examine or assign threads to processors, the logarithmic barrier displays significantly better performance than the serial barrier. The performance of the logarighmic barrier tends to go linear in n as n becomes large for a given number of processors. This is borne out both by the observations and mathematical intuition.

6.8.3 Semaphores

6.9 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,

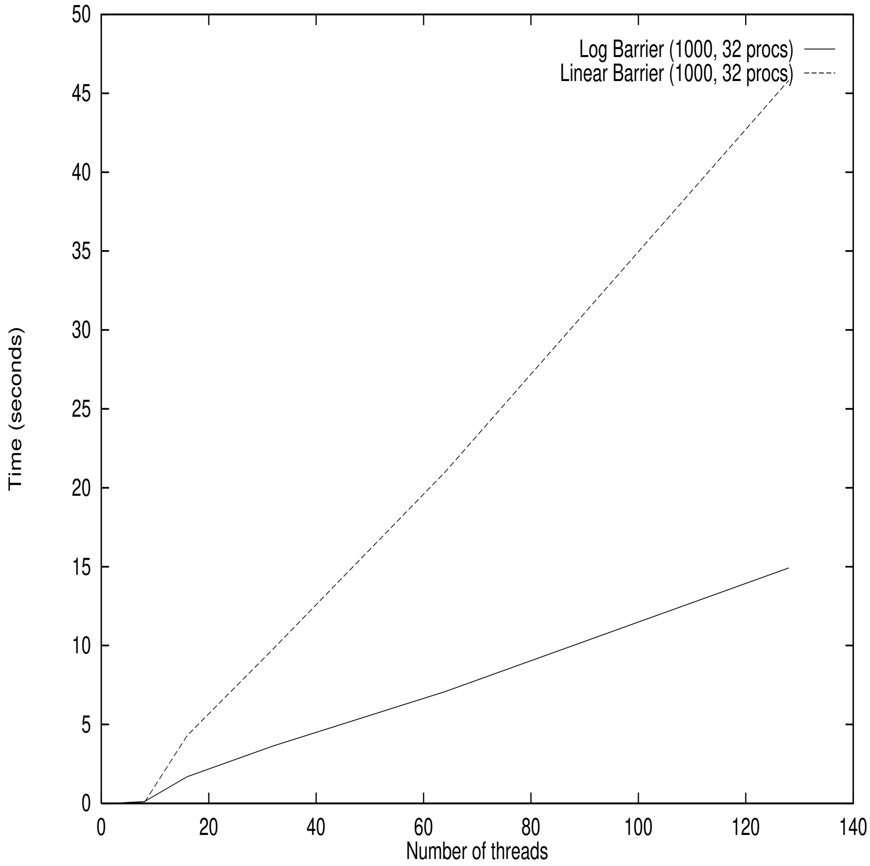

Figure 6.3 Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.

locks, 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 *slightly 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 with regards 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 created 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 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. 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 is released and some other thread may overwrite the space pointed to formerly by the stack. In this case, what thread T2 reads from the location may be incorrect data. Similar problems may exist with global variables.

- We strongly discourage the use of scheduling techniques as means of synchronization. It is very difficult to keep track of scheduling decisions on parallel machines. Further, as the number of processors change, these issues may change. 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 intializing the data, setting thread attributes (discussed in Section 6.6), 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. This is typically done using a flag: The producer sets the flag after it has stored the data. The consumer waits on the flag before accessing the data. In this way the producer-consumer integrity is maintained.

- At the consumer end, make sure that the data lasts at-least until all potential consumers have consumed the data. This is particularly relevant for stack variables.

6.10 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 threaded APIs such as POSIX are considered to be low-level primitives. Conventional wisdom says 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 borne fruit in the form of OpenMP. OpenMP is a standardized set of directives 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, etc.

6.10.1 The OpenMP Programming Model

Let us initiate the OpenMP programming model with the aid of a simple program. Each OpemMP directive starts with the preamble (or directive sentinel) `!$OMP`, `C$OMP`, or `*$OMP`. The directive itself consists of a directive name followed by clauses. 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 set using environment variables or at runtime using OpenMP supported subroutines. The main thread that encounters the `parallel` directive becomes the *master* of this group of threads and is assigned the thread id 0 within the group. The `parallel` directive looks as follows:

```
!$OMP parallel [clause list]

body

!$OMP end parallel
```

Each thread created by this directive executes the `body` specified by the parallel directive. The clause list is used to specify data handling across threads. For example, the clause `private(a, b, c)` specifies that variables `a`, `b`, and `c` are private to each thread, i.e., each thread should make its own copy of these variable and operate on them. Similarly, `shared(d)` implies that all threads share a single copy of the variable `d`. The default state of a variable can be assigned using a clause of the form `default(private)`. This clause specifies that all variables in the lexical extent of parallel region are by default private. Other clauses that can be used with the `parallel` directive are `firstprivate`, `reduction`, `if`, and `copyin`. We will discuss these clauses in greater detail subsequently in Section **??**. For now, this allows us to write our first OpenMP program.

The program uses the `parallel` directive to compute the sum of two vectors `x` and `y` to yield result vector `z`. The routines `omp_get_thread_num` and `omp_get_num_threads` are used to determine the thread id and the total number of threads. Once these are known, the vectors are assigned in a block fashion to various threads. A `do` loop is then used to compute the sums of corresponding subvectors. Our first OpenMP program therefore looks as follows:

```
C *****************************************************
C     This program illustrates the use of the parallel
C     directive in OpenMP in adding two vectors x and y
C     to obtain result vector z
C *****************************************************


      subroutine vectorsum(x, y, z, npoints)
      integer x(npoints), y(npoints), z(npoints)
      integer omp_get_thread_num, omp_get_num_threads
!$OMP parallel default(private) shared (x, y, z, npoints)
      my_thread_id = omp_get_thread_num()
      num_threads = omp_get_num_threads()
      num_elements = npoints/num_threads
      i_start = my_thread_id * num_elements + 1
      i_end = i+start + num_elements - 1
      do i = i_start, i_end
        z(i) = x(i) + y(i)
      enddo
!$OMP end parallel
      end

      program example_one
      integer i
      parameter (n = 1024)
      integer x(n), y(n), z(n)
c     initialize the vectors
      do i = 1, n
        x(i) = i
        y(i) = 2.0 * i
      enddo
      call vectorsum(x, y, z, n)
      end
```

6.10.2 Compiling and Executing the Program

The program can be compiled using an f77 compiler with the appropriate flags for OpenMP directives. For instance, the above program can be compiled on an Origin2000 using the following command:

```
ananth@smp1 % f77 -mp e1.f
```

Before we can execute the file, we have to make sure that the number of threads is set appropriately. This can be done by setting the environment variable OMP_NUM_THREADS as follows:

```
ananth@smp1 % setenv OMP_NUM_THREADS 16
```

The program can now be executed by invoking the executable:

```
ananth@smp1 % a.out
```

This will create 16 threads to accomplish the vector addition. The number of physical processors used is a function of the hardware available.

6.10.3 Specifying Concurrent Tasks in OpenMP

The `parallel` directive can be used in conjuction with other directives to specify concurrency across iterations, tasks, and sections. Here, we discuss these directives in greater detail. The example of adding two vectors was simply a do-loop running through the vector. Each iteration of the loop is entirely independent of the others. In such situations, we can simplify the program using the `DO` directive. The subroutine vectorsum can be re-written as follows:

```
      subroutine vectorsum(x, y, z, npoints)
      integer x(npoints), y(npoints), z(npoints)
!$OMP parallel do default(private) shared (x, y, z, npoints)
      do i = 1, npoints
        z(i) = x(i) + y(i)
      enddo
!$OMP end parallel do
      end
```

6.10.4 The `parallel do` and `do` Directives

The `parallel do` directive used in this example specifies that the `do` loop immediately following the directive must be executed in parallel, i.e. split across various threads. The mechanism of splitting the iterations across threads will be discussed subseuently. The `do` loop cannot be a `do while` or a `do` loop without loop control. The reason for this will become obvious when we discuss scheduling strategies. The reader will note in the above example that we do not declare the loop index `i` as private. This is because the loop index is assumed to be private by default. The general form of a `parallel do` directive is as follows:

```
!$OMP parallel do [clause list]

      do loop

[!$OMP end parallel do]
```

The ending `end parallel do` directive is optional since the end of the do loop specifies the end of the `parallel` directive as well. The clauses that can be used in this context are: `private`, `firstprivate`, `lastprivate`, `reduction`, `schedule`, and `ordered`. The first four clauses deal with data handling, and `schedule` and `ordered` deal with controling concurrent execution of the loop.

The `schedule` clause 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`. Let us explore these scheduling classes in the context of a dense matrix multiplication.

```
      subroutine dense_matrix_multiplication(a, b, c, dim)
      integer a(dim, dim), b(dim, dim), c(dim, dim), dim
```

```
!$OMP parallel default(private) shared (a, b, c, dim)
!$OMP do
      do i = 1, dim
!$OMP do
        do j = 1, dim
          c(i,j) = 0
!$OMP do
          do k = 1, dim
            c(i,j) = c(i,j) + a(i, k) * b(k, j)
          enddo
        enddo
      enddo
      end
```

The subroutine `dense_matrix_multiplication` splits the three-dimensional iteration space into threads along each dimension. It provides 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. For example, the following modification of the matrix-multiplication program causes the iteration space to be split along the columns of the `c` matrix (as illustrated in Figure 6.4(a)).

```
!$OMP parallel default(private) shared (a, b, c, dim)
!$OMP do schedule(static)
      do i = 1, dim
        do j = 1, dim
          c(i,j) = 0
          do k = 1, dim
            c(i,j) = c(i,j) + a(i, k) * b(k, j)
          enddo
        enddo
      enddo
      end
```

If there are four threads in all and `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 6.4(b). Another example of the split illustrated in Figure 6.4(c) results when each `parallel do` has a `schedule(static)` with it.

Dynamic. Often, because of a number of reasons, ranging from heterogeneous computing resources to non-uniform processor loads, equally partitioned workloads often 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

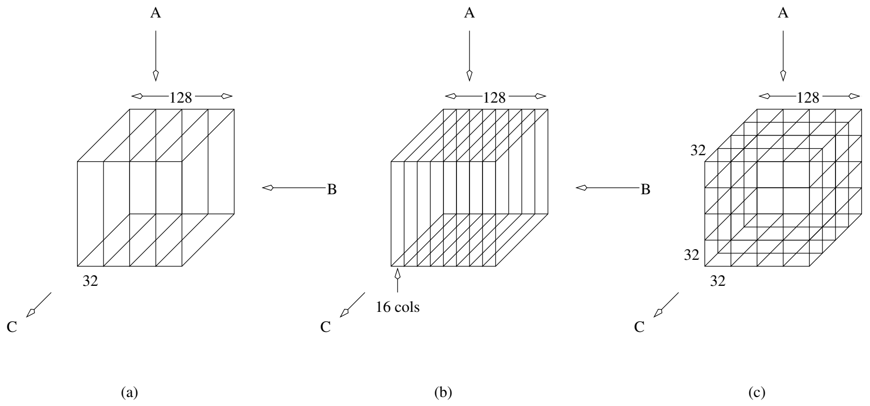

Figure 6.4 Three different schedules using the static scheduling class of OpenMP.

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 static scheduling. If no `chunk-size` is specified, it default 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 4 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 refered to as *edge effects*) is to reduce the chunk size as we proceed through the computation - *By giving smaller pieces when the pie is smaller, we can distribute the pie better among the eaters*. 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 of `runtime`. In this case the environment variable `OMP_SCHEDULE` determines the scheduling class and the chunk size. The use of this environment variable is discussed in Section **??**.

When no scheduling class is specified with the `parallel do` directive, the actual scheduling technique is implementation dependent.

One of the major overheads in thread management is synchronization between threads. Often, it is desirable to execute a sequence of do-loops within a parallel constructs. Furthermore, the threads may not need to synchronize between these loops. For instance, consider the problem of computing $E = A \times B + C \times D$; where A, B, C, D, and E are all dense $n \times n$ matrices. We cah use the dense matrix multiplication program to compute $A \times B$ and $C \times D$. The structure of this program would be as follows:

```
!$OMP parallel default(private) shared(a, b, c, d, e, dim)
!$OMP do
c     compute E = A x B here
!$OMP end do
!$OMP do
c     compute E =E + C x D here
!$OMP end do
!$OMP end parallel
```

This pseudocode creates threads to compute $E = A \times B$. Once this has been computed, the threads synchronize, and then $E = E + C \times D$ is computed. It is easy to see that the synchronization between these two steps is not necessary. The overhead associated with this synchronization can be avoided by specifying that the threads do not need to wait for other threads before proceeding to the next phase. To accomplish this, OpenMP supports the `nowait` option for parallel `do` loops. Using the `nowait` option in the above example results in the following code:

```
!$OMP parallel default(private) shared(a, b, c, d, e, dim)
!$OMP do
c     compute E = A x B here
!$OMP end do nowait
!$OMP do
c     compute E =E + C x D here
!$OMP end do
!$OMP end parallel
```

By specifying that the first set of threads do not have to wait for synchronization using the `nowait` primitive, we can compute the desired sum of products more effectively.

Often, it is desired that the iterations execute in the order they would have executed in a serial execution model. This is accomplished using the `ordered` directive. When an `ordered` clause is associated with a `do` directive, the serial execution order of the iterations is maintained. We will discuss this directive to a greater extent in Section **??** when dealing with synchronization mechanisms for threads.

Restrictions on the Use of do directives. The following restrictions apply on the use of the `do` directive. These restrictions follow fairly obviously from the threaded execution model.

- It is not legal to branch out of a DO loop associated with a `do` directive.
- The loop control variable must be an integer (for scheduling purposes).
- All threads in the group associated with a `do` directive must use the same loop control variable.

- The `enddo` directive must immediately follow the end of the do loop enclosed in the `do` directive.
- Only a single `schedule` clause is allowed for a `do` directive (otherwise iteration assignment to threads is non-deterministic).
- Only a single `ordered` clause is allowed for a `do` directive.

6.10.5 The `sections` Directive

Consider a scenario in which there are three tasks (TASK_A, TASK_B, and TASK_C) that need to be accomplished. Furthermore, 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:

```
!$OMP sections [clause-list]
[!OMP section]
c     body of first section
[!OMP section
c     body of other sections
!$OMP end sections [nowait]
```

This construct assigns the body of each section to one thread (indeed more than one sections can be assigned to a single thread). All sections are contained in the lexical scope defined by the `sections` and `end sections` directives. Since the start of the first section is implicitly defined as following the `sections` directive, the first `section` directive is optional. The clause list for the `sections` directive can be `private`, `firstprivate`, `lastprivate`, and `reduction`. These are discussed in greater detail in Section **??**.

Assuming that `TASK_A`, `TASK_B`, and `TASK_C` are three independent tasks coded in subroutines by their respective names, the `sections` directive can be used to parallelize them as follows:

```
!$OMP parallel
!$OMP sections
!$OMP section
      call task_a
!$OMP section
      call task_b
!$OMP section
      call task_c
!$OMP end sections
!$OMP end parallel
```

If there are three threads, each section (in this case subroutine) is assigned to one thread. At the end of execution of the assigned section, the threads synchronize (unless the `nowait` option is used with the `end sections` directive).

Restrictions on the Use of `sections` directives. The following restrictions apply to the use of the `sections` directives.

- It is illegal to branch in and out of `section` blocks.
- All `section` directives must be enclosed in the lexical scope of the `sections` directive.

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

For example:

```
 !$OMP parallel default(private) shared (a, b, c, dim)
 !$OMP do schedule(static)
       do i = 1, dim
 C       body of do loop
       enddo
       end parallel do
```

is identical to

```
 !$OMP parallel do default(private) shared (a, b, c, dim) do
 !$OMP*schedule(static)
       do i = 1, dim
 C       body of do loop
       enddo
       end parallel do
```

and

```
 !$OMP parallel
 !$OMP sections
 !$OMP section
       call task_a
 !$OMP section
       call task_b
 !$OMP section
       call task_c
 !$OMP end sections
 !$OMP end parallel
```

is identical to

```
 !$OMP parallel sections
 !$OMP section
       call task_a
```

```
!$OMP section
      call task_b
!$OMP section
      call task_c
!$OMP end parallel sections
```

Nesting `parallel` Directives Let us rewrite our dense matrix multiplication subroutine `dense_matrix_multiplication` as follows:

```
      subroutine dense_matrix_multiplication(a, b, c, dim)
      integer a(dim, dim), b(dim, dim), c(dim, dim), dim

!$OMP parallel do default(private) shared (a, b, c, dim)
      do i = 1, dim
!$OMP parallel do default(private) shared (a, b, c, dim)
        do j = 1, dim
          c(i,j) = 0
!$OMP parallel do default(private) shared (a, b, c, dim)
          do k = 1, dim
            c(i,j) = c(i,j) + a(i, k) * b(k, j)
          enddo
        enddo
      enddo
      end
```

In this example, we have `parallel` directives nested, one inside the other. If we execute this subroutine, and examine the execution, we realize that the inner parallel directives are simply ignored and only the outermost parallel directive results in multiple threads. The number of threads created in such a situation is determined by a variety of factors. Nesting of `parallel` directives can be controlled using environment variables as well as runtime functions supported by OpenMP. The `OMP_NESTED` environment variable determines if nesting parallel directives should result in the creation of additional threads or not. 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. However, the number of threads created on nested parallel regions is implementation dependent. The default state of this environment variable is `FALSE`, i.e. nested parallelism is disabled. The `OMP_NESTED` environment variable is discussed in greater detail in Section **??**. Parallel directive nesting can also be controlled at runtime using the function `OMP_SET_NESTED`. This has the same effect as the environment variable and is discussed in section **??**.

6.10.6 Synchronization Constructs in OpenMP

In Section **??**, we have seen that there is often a need to coordinate 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. POSIX threads supported mutexes and condition variables. Using these we implemented a range of higher level functionality in the form of non-exclusive 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

Consider the example of using multiple threads to compute the standard deviation of a list of numbers (refer Section **??**). The list is partitioned equally among the threads. Each thread first sums its numbers and together, the threads compute the mean. Once the mean has been computed, each thread computes the sum of mean squared differences, and subsequently the variance and standard deviation. As we had seen in Section **??**, all threads must wait for the computation of the mean before proceeding. What is required is a `barrier` that enables a thread to wait for all the others to catch up, then compute the average and proceed. The `barrier` directive in OpenMP provides this functionality. On encountering this directive, all threads in the team wait for others to catch up and then proceed. The standard-deviation example can be coded in OpenMP as follows:

```
c *****************************************************************
c Assume that the array a contains the list we need to compute mean
c and standard deviation of. Also assume that all variable type
c declarations have been made appropriately
c *****************************************************************

!$OMP parallel shared(a, local_sums), default(private)

c initialize variables

      my_thread_id = omp_get_thread_num()
      num_threads = omp_get_num_threads()
      local_sums(my_thread_id + 1) = 0

c compute local sums of partial arrays

!$OMP do schedule(static)
      do i = 1, n
        local_sums(my_thread_id + 1) = local_sums(my_thread_id + 1)
     *                                 + a(i)
      enddo
!$OMP barrier

c all threads compute their own means after barrier

      array_mean = 0
      do i = 1, num_threads
        array_mean = array_mean + local_sums(i)
      enddo
      array_mean = array_mean / n

c now compute sum of mean squared differences

      local_sums(my_thread_id + 1) = 0
!$OMP do schedule(static)
      do i = 1, n
        local_sums(my_thread_id + 1) = local_sums(my_thread_id + 1) +
```

```
     *       (a(i) - array_mean) ** 2
      enddo
!$OMP barrier

c and each thread computes variance and standard deviation independently

      array_square_sum = 0
      do i = 1, num_threads
        array_square_sum = array_square_sum + local_sums(i)
      enddo
      variance = array_square_sum / n
      std_dev = sqrt(variance)
!$OMP end parallel
```

The above example illustrates the use of the `barrier` directive to compute the mean and standard deviation of array `a`. The `do` loop for computing the sum is split statically across the threads. Each thread computes the sum of its partial list and stores it in the corresponding entry of `local_sums`. Once all `local_sums` have been computed, these sums are accrued by all threads to compute the global sum. This is accomplished using a `barrier`. Indeed, it is possible to use a `reduction` clause to compute global sum; this will be discussed later in data handling (Section **??**.

After computing the mean, a similar process is repeated for computing the After computing the mean, a similar process is repeated for computing the sum of mean-squared differences, followed by a `barrier` and computation of variance and standard deviation. As before, the variance and standard deviation are computed by all the threads. Note that the variables corresponding to each sum, standard deviation, and variance are all private (i.e. each thread has its own copy of these variables).

`Barrier` directives are useful when there is a strict execution order between two phases of computation. Often, barriers can also be effected by ending and re-starting `parallel` regions. However, this corresponds to joining and re-creating threads. 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` Directive

In the previous example, we declared the variables `variance` and `std_dev` private (by default) and had each thread compute these. Indeed, we could have specified these variables to be shared and have only on thread compute them. Often in situation such as these and others (such as I/O), we want only one thread to execute a particular segment of code. OpenMP provides the `single` directive for accomplishing this. The general form of the single directive is as follows:

```
!$OMP single [clause-list]
c    statements to be executed by only one thread in the group
!$OMP end single [nowait]
```

The clause list can take clauses `private` and `firstprivate`. These clauses are discussed in greater detail in Section **??**. 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; else they wait at the end of the `single` block for the thread to finish executing the block. This directive is very useful for computing global data as well as I/O.

Let us use the `single` directive to modify the above program so that only one thread computes and prints the variance and standard deviation.

```
!$OMP parallel shared(a, local_sums, variance, std_dev),
!$OMP* default(private)
c
c compute mean, and mean-square sums as before
c
!$OMP single
      array_square_sum = 0
      do i = 1, num_threads
        array_square_sum = array_square_sum + local_sums(i)
      enddo
      variance = array_square_sum / n
      std_dev = sqrt(variance)
      print *, variance, std_dev
!$OMP end single nowait
!$OMP end parallel
```

By specifying the computation of `variance` and `std_dev` in a `single` block, we ensure that only one thread computes these. Note that the `parallel` directive has been modified to make these variables visible to all threads. While this is not necessary for this particular example, in general, we may want the `variance` and `std_dev` variables to be visible to all threads for future computations. Furthermore, note that by putting the `print` statement within the `single` block, we ensure that only one thread (processor) executes the `print` statement. Similarly, the `single` directive is also extremely useful in input of program data.

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

Let us revisit the previous example again. We introduced an additional array `list_sums` to hold partial sums. Each element of this array was updated by one thread. Finally, each thread computed the total sum by adding the elements of `list_sums` into a private variable, `array_mean`. While this seems contrived, there was a reason for setting up the example this way. The question to ask is why we did not have a shared variable, say `global_sum` and have each thread add its elements to this global sum. The resulting code would look as follows:

```
c incorrect way of simplifying the above example.
c
      global_sum = 0
!$OMP parallel shared(a, global_sum), default(private)
!$OMP do schedule(static)
      do i = 1, n
        global_sum = global_sum + a(i)
```

```
      enddo
c
c and so on
!$OMP end parallel
```

The above example is incorrect because the statement `global_sum = global_sum + a(i)` is not an atomic statement. This statement translates to several assembler instructions. If a thread was in the middle of executing this set of instructions and another thread was concurrently trying to execute the same set of instructions, we would end up with incorrect results. These race conditions were discussed while studying mutexes in Section **??**. We had seen that such critical code segments must be protected using mutexes so that only one thread is allowed to execute the set of instructions at any point of time. OpenMP provides the `critical` and `atomic` directives to enforce critical sections.

The general form of a `critical` section is as follows:

```
!$OMP critical [name]
block
!$OMP end critical [name]
```

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 `mutex` function discussed in POSIX threads. The name field maps to the name of the mutex on which the lock is performed. Let us see how we can use the `critical` directive for fixing the previous program.

```
c correct way of simplifying the above example.
c
      global_sum = 0
!$OMP parallel shared(a, global_sum), default(private)
!$OMP do schedule(static)
      do i = 1, n
!$OMP critical
        global_sum = global_sum + a(i)
!$OMP end critical
      enddo
c
c and so on
!$OMP end parallel
```

By making the summation critical, we have ensured the atomicity of the operation and thus fixed the concurrent write access error on the variable `global_sum`. While this fixes the program, the reader will also note that this effectively serializes the parallel do loop. Consequently, the program will infact result in a slow-down instead of a speedup, since a lock-unlock operations is typically much slower than a single addition.

As was the case with POSIX threads, it is important to remember that `critical` sections represent serialization points in the program. It is important to reduce the size of the critical sections as much as possible (in terms of execution time) to get good performance. To this effect, we can re-write the above example as follows:

```
!$OMP parallel shared(a, local_sums), default(private)

c initialize variables

      my_thread_id = omp_get_thread_num()
      num_threads = omp_get_num_threads()
      local_sums(my_thread_id + 1) = 0

c compute local sums of partial arrays

!$OMP do schedule(static)
      do i = 1, n
        local_sums(my_thread_id + 1) = local_sums(my_thread_id + 1)
     *                                 + a(i)
      enddo
!$OMP critical
        global_sum = global_sum + local_sums(my_thread_id + 1)
!$OMP end critical

c and so on
!$OMP end parallel
```

In this case, we have reduced the serial component from the number of entries in the lize (n) to the number of threads. Assuming the latter is much smaller, we can expect speedups from this program for `n >> num_threads`. It is infact possible to reduce the serial component to log(`num_threads`) using multiple critical sections. This is left as an exercise for the reader (Exercise **??**).

One of the most common uses of the `critical` directive is in producer-consumer scenarios. The producer generates a chunk of work and a set of consumers take work pieces of the work-queue and accomplish associated tasks. The queuing and de-queuing of work are non-atomic operations and therefore must be protected using the `critical` directive. An example of this is motivated in Exercise **??** for the readers.

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 later in an un-released lock; which could cause the threads to wait indefinitely. Similarly, the names associated with a pair of `critical` and `end critical` directives must be identical. Since there is a mapping from the name of the section to the underlying mutex, this is necessary.

In the above example, the critical section consisted simply of an update of a memory location. OpenMP provides another directive, `atomic` for such atomic updates to memory locations. The `atomic` directive specifies that the memory location updation in the next instruction should be performed as an atomic operation. The update instruction can be one of the following forms:

```
x = x oper expr
x = expr oper x
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

Here, `expr` is a scalar expression that does not include `x`, `x` itself is a scalar variable, `intrinsic` can be `MAX, MIN, IAND, IOR,` or `IEOR`, and `oper` can be $+$, $*$, $-$, $/$, `.AND., .OR., .EQV.,` or `.NEQV.`.

In our previous example, we could replace the `critical` section by the following:

```
c
c
!$OMP atomic
        global_sum = global_sum + local_sums(my_thread_id + 1)
c
c and so on
```

This would have the effect of making updates to `global_sum` atomic in nature. 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. It is easy to see that all `atomic` directives can be replaced by `critical` directives provided they have the same name.

In-Order Execution: The `ordered` Directive

Consider the following problem: given a list of automobile owners, ages, and their cars, we are trying to compute the age profile of Caddillac owners. Specifically, we are trying to compute the number of Caddillac owners below various ages. The serial program for accomplishing this has the following structure:

```
c
c setting up data structures, reading lists etc.

      do i = min_age, max_age
        count = 0
c determine how many people of age i drive Caddillacs
c and store it in variable count
        cumulative_count[i] = cumulative_count[i - 1] + count
      end do
```

The segment consists of a single loop that runs through the ages, keeps count of the number of people of a certain age driving Caddillacs, and uses it to maintain cumulative count in the table `cumulative_count` of all people of ages less than i who own Caddillacs.

The loop can be simply threaded as follows:

```
c incorrect threading of the above loop
!$OMP parallel shared(min_age, max_age, cumulative_count, table),
!$OMP* default(private)
!$OMP do schedule(static)
      do i = min_age, max_age
        count = 0
c determine how many people of age i drive Caddillacs
c and store it in variable count
        cumulative_count[i] = cumulative_count[i - 1] + count
      end do
!$OMP end parallel
```

However, this threaded version is unlikely to give correct results. Assume that the iteration for age 50 was scheduled before the iteration for age 49. The cumulative count for 50 (and all subsequent ages) will be wrong because the statement `cumulative_count[i] = cumulative_count[i - 1] + count` relies on the previous cumulative count being available.

What is needed in this example is a mechanism to specify that the cumulative summation instruction must be performed in the same order as would be executed by the serial loop. This is accomplished using the `ordered` directive. The general form of the ordered directive is as follows:

```
!$OMP ordered
block
!$OMP end ordered
```

This directive appearing in a `do` or `sections` directive specifies that the enclosed structured block must be executed in the same order as would be executed by the serial version. In the case of the `sections` directive, it implies that the block must be executed in the lexical order of the `sections` directive. When used inside `do` directives, it is necessary that the `ordered` block binds to the `do` directive with `ordered` specified in the clause list. Let us rewrite the above do loop to fix the problem associated with the cumulative table.

```
c correct threading of the loop
!$OMP parallel shared(min_age, max_age, cumulative_count, table),
!$OMP* default(private)
!$OMP do schedule(static) ordered
      do i = min_age, max_age
        count = 0
c determine how many people of age i drive Caddillacs
c and store it in variable count
!$OMP ordered
        cumulative_count[i] = cumulative_count[i - 1] + count
!$OMP end ordered
      enddo
!$OMP end parallel
```

It is important to note that the `ordered` directive represents a serialization point in the program. Only a single thread can enter an ordered block. Therefore, if large portions of a loop are enclosed in ordered directives, it will result in poor speedups. In the above example, if the time taken to compute the number of people of a certain age who drive caddillacs is much higher than the time to update the table entry, we can expect speedups from the loop. (The reader will note that the cumulative table represents a sum scan across threads and can be implemented in logarithmic time instead of linear time in number of threads as in this case.)

There are some restrictions to the use of `ordered` directives that follow naturally from its semantics. The block of instructions enclosed in the directive must be a structured block, i.e. no jumps in or out of the block. The `do` loop to which the `ordered` directive binds must have `ordered` specified in the clause-list. Finally, a single iteration of the `do` loop must enter an `ordered` section only once, i.e. no goto statements back to the head of the section. Furthermore, it must execute only one `ordered` block in a single iteration.

Memory Consistency: The `flush` Directive

6.10.7 Data Handling in OpenMP

One of the critical parameters influencing program performance is the manipulation of data by threads. There are some basic groundrules that must be followed for good performance:

- 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.

- 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 on the same processor and access will not result in interprocessor communication.

- 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 subsequently accrue it using a single summation at thread join time.

- 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 must be shared among various threads.

These thumb-rules guide data manipulation in threaded APIs and OpenMP supports these using a range of primitives. In this section, we will study these directives.

Thread-Local Data: `private`, `firstprivate`, and, `lastprivate` Directives

Often, the data manipulated by threads is local to a thread and is not accessed by other threads. Such data can be conveniently allocated on the stack and moves with the thread. This helps in minimizing the communication overhead. Examples of such data include loop indices and variables for storing intermediate results. We can identify three different categories that fall into this class:

- Temporary storage whole active range is a subset of the life of a thread; i.e. the data is initialized within the thread and the value is not used anywhere outside the thread.
- Temporary storage that is live-in on the thread but not live-out; i.e. the thread inherits the variable value at creation but its value is not used after execution of the thread.
- Temporary storage that is live-out but not live in; i.e. data that is initialized in the thread but is used outside the execution scope of the thread.

The first case in which data is neither live-in nor live-out of the thread is supported by the OpenMP directive `private`. By specifying that a variable is private, a new copy of the variable is instantiated at the creation of the thread. The value of this variable at creation is undefined. The instance of the variable is undefined when the thread exits. The general form of a `private` clause is:

```
private (variable-list)
```

This clause indicates that all the variables in the list are private to each thread created by the parallel construct. In our standard-deviation program, we use the `private` directive to create multiple copies of variables such as `my_thread_id`, `num_threads`, `array_mean`, and `array_square_sums`. Since the variable `my_thread_id` is different for each thread and is initialized independently, we need this to be private. Similarly, since we use each thread to keep track of partial sums of sub-lists, each thread maintains its own copy. Note that these partial sums are copied into a global (shared) array before the thread exits. One is tempted to ask why we need this private variable as opposed to updating the global array itself. The reason for this is to minimize operations on shared variables for optimizing performance. We had seen the effect of this on false sharing in Section **??**.

In each of the above examples of variables, the variable was not live-in entering the thread. Often, the variable needs to be initialized to its value before entering the construct. As an example of this, consider the problem of using a set of threads to search for an integer in an array of integers `i_array`. Assume that the starting location of the search region is `i_start` and the ending location is `i_end`. If the search region is equally distributed among the threads, then thread i's local region is delimited by indices `i_start + (i_end - i_start) * (my_thread_id / num_threads)` and `i_start + (i_end - i_start) * ((my_thread_id+1) / num_threads)`. To compute this within a parallel region, it would (i) need a local copy of i_start and i_end, and (ii) need the values before entering the parallel region. This is accomplished using the firstprivate directive. The directive specifies a specialization of the private directive in which each thread is given a private copy of the variable; furthermore, the value of the variable entering the construct is copied from its value just before the construct. The general form of this construct is as follows:

```
       firstprivate (variable-list)
```

The construct can be used to code the list-searching problem as follows:

```
c searching array i_array between indices i_start and i_end
c example assumes that (i_end - i_start) is a perfect multiple of
c number of threads num_threads.
c
c declarations, initializations
c
!$OMP parallel firstprivate (i_start, i_end) shared (i_array)
     * default (private)
      my_thread_id = omp_get_thread_num()
      num_threads = omp_get_num_threads()
      num_entries_per_thread = (i_end - i_start) / num_threads
      i_start = i_start + num_entries_per_thread * my_thread_id
      i_end = i_start + num_entries_per_thread

c and search performed here over local region

!$OMP end parallel
```

The above example illustrates the use of variables that are live-in to a thread but not live out of a thread. Often when using do or sections directive for farming work to threads, it is desired that the last iteration (as defined by serial execution) of the do loop or lexically last section of the sections directive update the value of a variable. This is accomplished using the lastprivate directive.

If the last iteration or last section does not update the value of a variable that has been specified as lastprivate, the value of the variable outside the construct is not defined.

Global data: The shared Clause

As we have used in many of the examples in the discussion of OpenMP, threads often share data items. These data items must be specified as shared. The general form of the shared directive is:

```
        shared(variable-list)
```

Since shared data is typically a source of interprocessor communication, care must be taken in handling these. Overheads such as false-sharing are often associated with shared data. Manipulations on shared data must be minimized by using intermediate private variables and aggregating multiple updates into single operations.

Specifying Data Scope Defaults: The `default` Clause

The default data scope can be set using the `default` clause. The general form of the clause is:

```
default (private | shared | none)
```

By specifying `default (private)`, we indicate that all variables in the lexical scope of the thread be treated as `private` unless otherwise specified. This includes the `common` block but excludes those that have been explicitly specified `threadprivate`. When `default (shared)` is used with a construct, all variables in the lexical scope of the parallel regions are treated as though they were specified as `shared` unless otherwise specified. If no data scope is declared, the default is assumed to be `shared`. Concurrent operations on shared data will often result in warnings from the compiler and must be examined carefully since they may result in race conditions. The defaults for variables can also be turned off by specifying `default (none)`. In this case, the data scope of each variable must be explicitly specified.

Defaults can be combined with explicit specifications of data scope. Where explicit specifications exist, they override the default. Furthermore, a single parallel region can only have one default associated with it.

Aggregate Operations: The `reduction` Clause

While computing the sum of a list of numbers for determining their mean, we stored partial sums in a list and had a single thread compute the sum of partial sums. There is infact a simpler way of doing this using the `reduction` clause. To compute the sum of a list of numbers, we can use the following:

```
!$OMP parallel do default(private) firstprivate(n) reduction(+: sum)
      do i = 1, n
        sum = sum + a(i)
      enddo
!$OMP end parallel
```

The `reduction(+: sum)` clause creates multiple copies of the variable `sum`. At the end of the loop, the '+' operation is applied to the copies ot `sum` and the value of `sum` ourside the parallel region is updated to reflect this aggregated value.

The general form of the `reduction` clause is:

```
reduction (operator | intrinsic : variable-list)
```

The operators supported are: $+$, $-$, $*$, `.AND., .OR., .EQV., .NEQV` and the intrinsics are `MAX, MIN, IAND, IOR, IEOR`. On encountering a `reduction` clause, local copies of each variable in the list are made. These copies are initialized to the identities for the respective operators (for example, 1 for '*', 0 for '+', `TRUE` for `AND` and so on). The parallel region operates on the local copies. At the end of the parallel region, these copies are accrued by applying the specified operator/intrinsic on them. The value following the parallel region is updated to reflect this accrued value. The moment the first thread reaches a `reduction` clause, the value of the shared variable outside the scope of the construct becomes undefined. It remains thus until all threads finish the reduction operation. If a `nowait` has been specified in the construct in addition to the reduction, then the value of the shared variable continues to be undefined until a `barrier` is reached.

6.10.8 Environment Variables for OpenMP

6.10.9 Auxillary Function Calls

Controlling Number of Threads and Processors

Explicit Lock Management in OpenMP

While OpenMP provides support for critical sections and atomic updates, there are situations where it is more convenient to use an explicit lock. **need to think of an example here**. For such programs, OpenMP provides runtime routines for initializing, locking, unlocking, and discarding locks. The lock data structure in OpenMP is an 8 byte integer and can be defined as follows:

```
integer(kind = 8) my_lock
```

Before a lock can be used, it must be initialized. This is done using the `omp_init_lock` routine. The prototype of this routine is as follows:

```
omp_init_lock ( lock_variable_name )
```

Furthermore, when a lock is no longer needed, it must be discarded using the function `omp_destroy_lock`. The function has the following prototype:

```
omp_destroy_lock ( lock_variable_name )
```

It is illegal to initialize a previously initialized lock and destroy an uninitialized lock.

Once the lock has been initialized, it can be locked and unlocked using the functions `omp_set_lock` and `omp_unset_lock`. The prototypes of these functions are:

```
omp_set_lock( lock_variable_name )
omp_unset_lock( lock_variable_name )
```

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.

Let us use these routines to implement a simple parallel loop for computing the global sum of a list of numbers:

```
c using locks to protect critical sections
c
      integer (kind = 8) global_sum_lock
      global_sum = 0
      call omp_init_lock (global_sum_lock)
!$OMP parallel shared(a, global_sum), default(private)
      local_sum = 0
!$OMP do schedule(static)
      do i = 1, n
        local_sum = local_sum + a(i)
      enddo
      call omp_set_lock (global_sum_lock)
      global_sum = global_sum + local_sum
      call omp_unset_lock (global_sum_lock)
!$OMP end parallel
      call omp_destroy_lock (global_sum_lock)
```

In this example, each thread computes the sum of its partial list in `local_sum`. The local sums are then added to the global sum. However, as we have seen, the updation of `global_sum` is not atomic and must therefore be protected as a critical region. We do this by having each thread lock on `global_sum_lock` before updating the global sum. Since only one thread can hold the lock at any time, the updation is serialized. Finally, after all threads have added their contributions to `global_sum`, the lock is destroyed. It is easy to see that the OpenMP lock API is very similar to the POSIX thread mutex API.

OpenMP provides another function to test if a lock is available. This is similar to the `pthread_mutex_trylock` function of POSIX API. The function `omp_test_lock` is used to check is a lock is available. If it is, the thread gets the lock, and the function returns `.TRUE.` else, the function returns the value `.FALSE.`. The prototype of this function is:

```
        logical function omp_test_lock (lock_variable)
```

The applicability of this function is similar to that of the `pthread_mutex_trylock` function discussed in Section **??**.