0.0.1 Condition Variables for Synchronization

- As it is noted in the previous section, indiscriminate use of locks can result in idling overhead from blocked threads. While the function *pthread_mutex_trylock* alleviates this overhead, it introduces the overhead of polling for availability of locks.

- For example, if the producer-consumer example is rewritten using *pthread_mutex_trylock* instead of *pthread_mutex_lock*, the producer and consumer threads would have to periodically poll for availability of lock (and subsequently availability of buffer space or tasks on queue).

- A natural solution to this problem is to suspend the execution of the producer until space becomes available (an interrupt driven mechanism as opposed to a polled mechanism).

- The availability of space is signaled by the consumer thread that consumes the task. The functionality to accomplish this is provided by a *condition variable.*

- A condition variable is a data object used for synchronizing threads. This variable allows a thread to block itself until specified data reaches a predefined state.

- In the producer-consumer case, the shared variable *task_available* must become 1 before the consumer threads can be signaled.

- The boolean condition *task_available == 1* is referred to as a predicate. A condition variable is associated with this predicate.

- When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.

- A single condition variable may be associated with more than one predicate. However, this is strongly discouraged since it makes the program difficult to debug.

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

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

- A call to this function blocks the execution of the thread until it receives a signal from another thread or is interrupted by an OS signal.
- In addition to blocking the thread, the **pthread_cond_wait** function releases the lock on mutex. This is important because otherwise no other thread will be able to work on the shared variable *task_available* and the predicate would never be satisfied.
- When the thread is released on a signal, it waits to reacquire the lock on mutex before resuming execution.
- It is convenient to think of each condition variable as being associated with a queue. Threads performing a condition wait on the variable relinquish their lock and enter the queue.
- When the condition is signaled (using **pthread_cond_signal**), one of these threads in the queue is unblocked, and when the mutex becomes available, it is handed to this thread (and the thread becomes runnable).
- In the context of our producer-consumer example, the producer thread produces the task and, since the lock on mutex has been relinquished (by waiting consumers), it can insert its task on the queue and set *task_available* to 1 after locking mutex. Since the predicate has now been satisfied, the producer must wake up one of the consumer threads by signaling it.

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

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

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

- The function *pthread_cond_init* initializes a condition variable (pointed to by *cond*) whose attributes are defined in the attribute object *attr*. Setting this pointer to *NULL* assigns default attributes for condition variables.

- If at some point in a program a condition variable is no longer required, it can be discarded using the function *pthread_cond_destroy*.

- These functions for manipulating condition variables enable us to rewrite our producer-consumer segment.

- **Producer-consumer using condition variables**

  - Condition variables can be used to block execution of the producer thread when the work queue is full and the consumer thread when the work queue is empty.
  - We use two condition variables *cond_queue_empty* and *cond_queue_full* for specifying empty and full queues respectively.
  - The predicate associated with *cond_queue_empty* is *task_available == 0*, and *cond_queue_full* is asserted when *task_available == 1*.
  - The producer queue locks the mutex *task_queue_cond_lock* associated with the shared variable *task_available*.
  - It checks to see if *task_available* is 0 (i.e., queue is empty). If this is the case, the producer inserts the task into the work queue and signals any waiting consumer threads to wake up by signaling the condition variable *cond_queue_full*. It subsequently proceeds to create additional tasks.
  - If *task_available* is 1 (i.e., queue is full), the producer performs a condition wait on the condition variable *cond_queue_empty* (i.e., it waits for the queue to become empty).
  - The reason for implicitly releasing the lock on *task_queue_cond_lock* becomes clear at this point. If the lock is not released, no consumer will be able to consume the task and the queue would never be empty. At this point, the producer thread is blocked.
  - Since the lock is available to the consumer, the thread can consume the task and signal the condition variable *cond_queue_empty* when the task has been taken off the work queue.
  - The consumer thread locks the mutex *task_queue_cond_lock* to check if the shared variable *task_available* is 1. If not, it performs a condition wait on *cond_queue_full*. (Note that this signal is generated from the producer when a task is inserted into the work queue.)

– If there is a task available, the consumer takes it off the work queue and signals the producer. In this way, the producer and consumer threads operate by signaling each other. It is easy to see that this mode of operation is similar to an interrupt-based operation as opposed to a polling-based operation of *pthread_mutex_trylock*.

```
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);
  }
}
```

- An important point to note about this program segment is that the predicate associated with a condition variable is checked in a loop.
- One might expect that when *cond_queue_full* is asserted, the value of *task_available* must be 1. However, it is a good practice to check for the condition in a loop because the thread might be woken up due to other reasons (such as an OS signal).
- In other cases, when the condition variable is signaled using a condition broadcast (signaling all waiting threads instead of just one), one of the threads that got the lock earlier might invalidate the condition.
- In the example of multiple producers and multiple consumers, a task available on the work queue might be consumed by one of the other consumers.
- When a thread performs a condition wait, it takes itself off the runnable list consequently, it does not use any CPU cycles until it is woken up. This is in contrast to a mutex lock which consumes CPU cycles as it polls for the lock.
- In the above example, each task could be consumed by only one consumer thread. Therefore, we choose to signal one blocked thread at a time.
- In some other computations, it may be beneficial to wake all threads that are waiting on the condition variable as opposed to a single thread. This can be done using the function *pthread_cond_broadcast*.

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

- An example of this is in the producer-consumer scenario with large work queues and multiple tasks being inserted into the work queue on each insertion cycle. Another example of the use of *pthread_cond_broadcast* is in the implementation of barriers.
- It is often useful to build time-outs into condition waits. Using the function **pthread_cond_timedwait**, a thread can perform a wait on a condition variable until a specified time expires. At this

point, the thread wakes up by itself if it does not receive a signal or a broadcast.

```
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.

0.1 Controlling Thread and Synchronization Attributes

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

- To create an attributes object with the desired properties, we must first create an object with default properties and then modify the object as required.

0.1.1 Attributes Objects for Threads

- The function **pthread_attr_init** lets us create an attributes object for threads.

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

- This function initializes the attributes object *attr* to the default values. Upon successful completion, the function returns a 0, otherwise it returns an error code.

- The attributes object may be destroyed using the function **pthread_attr_destroy**.

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

- The call returns a 0 on successful removal of the attributes object *attr*.

- Individual properties associated with the attributes object can be changed using the following functions: **pthread_attr_setdetachstate, pthread_attr_setguardsize_np, pthread_attr_setstacksize, pthread_attr_setinheritsched, pthread_attr_setschedpolicy, and pthread_attr_setschedparam**.

- These functions can be used to set the detach state in a thread attributes object, the stack guard size, the stack size, whether scheduling policy is inherited from the creating thread, the scheduling policy (in case it is not inherited), and scheduling parameters, respectively.

- For most parallel programs, default thread properties are generally adequate.

0.2 Thread Cancellation

- Consider a simple program to evaluate a set of positions in a chess game. Assume that there are k moves, each being evaluated by an independent thread.

- If at any point of time, a position is established to be of a certain quality, the other positions that are known to be of worse quality must stop being evaluated. In other words, the threads evaluating the corresponding board positions must be canceled. Posix threads provide this cancellation feature in the function **pthread_cancel**.

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

- Here, *thread* is the handle to the thread to be canceled. A thread may cancel itself or cancel other threads.

- When a call to this function is made, a cancellation is sent to the specified thread. It is not guaranteed that the specified thread will receive or act on the cancellation. Threads can protect themselves against cancellation.

- When a cancellation is actually performed, cleanup functions are invoked for reclaiming the thread data structures. After this the thread is canceled. This process is similar to termination of a thread using the **pthread_exit** call.

- This is performed independently of the thread that made the original request for cancellation.

- The **pthread_cancel** function returns after a cancellation has been sent. The cancellation may itself be performed later. The function returns a 0 on successful completion. This does not imply that the requested thread has been canceled; it implies that the specified thread is a valid thread for cancellation.

0.3 Composite Synchronization Constructs

While the Pthreads API provides a basic set of synchronization constructs, often, there is a need for higher level constructs. These higher level constructs can be built using basic synchronization constructs.

0.3.1 Barriers

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

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

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

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

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

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

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

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

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

```

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

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

- In this implementation of a barrier, we visualize the barrier as a binary tree. Threads arrive at the leaf nodes of this tree.

- Consider an instance of a barrier with eight threads. Threads 0 and 1 are paired up on a single leaf node. One of these threads is designated as the representative of the pair at the next level in the tree.

- In the above example, thread 0 is considered the representative and it waits on the condition variable *ok_to_proceed_up* for thread 1 to catch up. All even numbered threads proceed to the next level in the tree. Now thread 0 is paired up with thread 2 and thread 4 with thread 6. Finally thread 0 and 4 are paired. At this point, thread 0 realizes that all threads have reached the desired barrier point and releases threads by signaling the condition *ok_to_proceed_down*. When all threads are released, the barrier is complete.

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

- In many thread libraries, threads are switched at *semi-deterministic* intervals. Such libraries are more forgiving of synchronization errors in programs. These libraries are called *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 regarding the level of asynchrony in the threads library.

- The following rules of thumb which help minimize the errors in threaded programs are recommended.

  - Set up all the requirements for a thread before actually creating the thread. This includes initializing the data, setting thread attributes, thread priorities, mutex-attributes, etc. Once you create a thread, it is possible that the newly created thread actually runs to completion before the creating thread gets scheduled again.

  - When there is a producer-consumer relation between two threads for certain data items, make sure the producer thread places the

  data before it is consumed and that intermediate buffers are guaranteed to not overflow.
  - At the consumer end, make sure that the data lasts at least until all potential consumers have consumed the data. This is particularly relevant for stack variables.
  - Where possible, define and use group synchronizations and data replication. This can improve program performance significantly.

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

0.5 OpenMP: a Standard for Directive Based Parallel Programming

- While standardization and support for these threaded APIs has come a long way, their use is still predominantly restricted to system programmers as opposed to application programmers. One of the reasons for this is that APIs such as Pthreads are considered to be low-level primitives.

- Conventional wisdom indicates that a large class of applications can be efficiently supported by higher level constructs (or directives) which rid the programmer of the mechanics of manipulating threads.

- Such directive-based languages have existed for a long time, but only recently have standardization efforts succeeded in the form of OpenMP. OpenMP is an API that can be used with FORTRAN, C, and C++ for programming shared address space machines.

- OpenMP directives provide support for concurrency, synchronization, and data handling while avoiding the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

0.5.1 The OpenMP Programming Model

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

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

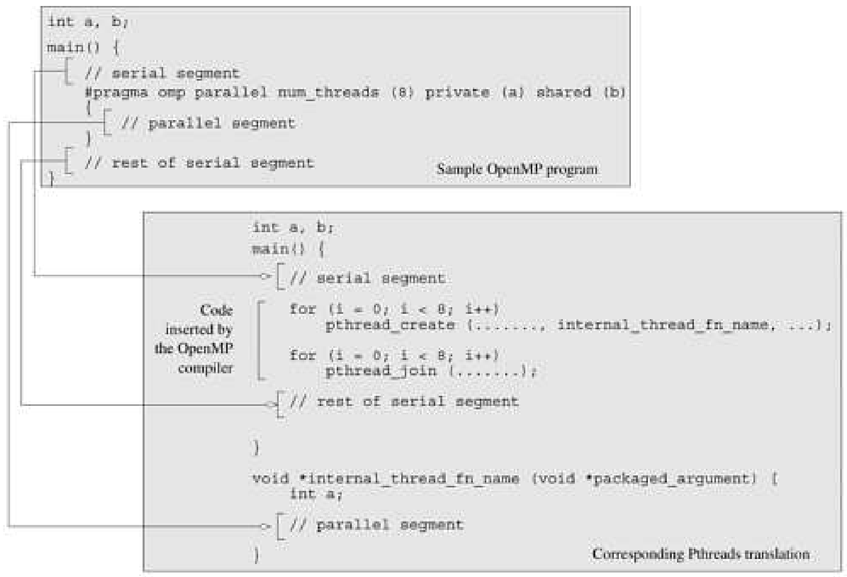
- OpenMP programs execute serially until they encounter the *parallel* directive. This directive is responsible for creating a group of threads.
- The exact number of threads can be specified in the directive, set using an environment variable, or at runtime using OpenMP functions.
- The main thread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

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

```

- Each thread created by this directive executes the *structured block* specified by the parallel directive.
- The clause list is used to specify conditional parallelization, number of threads, and data handling.
  - *Conditional Parallelization:* The clause *if (scalar expression)* determines whether the parallel construct results in creation of threads. Only one *if* clause can be used with a parallel directive.
  - *Degree of Concurrency:* The clause *num_threads (integer expression)* specifies the number of threads that are created by the *parallel* directive.
  - *Data Handling:* The clause *private (variable list)* indicates that the set of variables specified is local to each thread - i.e., each thread has its own copy of each variable in the list. The clause *firstprivate (variable list)* is similar to the private clause, except the values of variables on entering the threads are initialized to corresponding values before the parallel directive. The clause *shared (variable list)* indicates that all variables in the list are shared across all the threads, i.e., there is only one copy. Special care must be taken while handling these variables by threads to ensure serializability.

- It is easy to understand the concurrency model of OpenMP when viewed in the context of the corresponding Pthreads translation. In Figure 1, one possible translation of an OpenMP program to a Pthreads program is shown (such a translation can easily be automated through a Yacc or CUP script).


Sample OpenMP program
Code inserted by the OpenMP compiler
Corresponding Pthreads translation

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

```
int a, b;
main() {
    // serial segment

    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);

    // rest of serial segment

}
void *internal_thread_fn_name (void *packaged_argument) {
    int a;
    // parallel segment
}
```

Figure 1: A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

```
#include <omp.h>
main ()  {
int var1, var2, var3;
Serial code
      .
      .
Beginning of parallel section. Fork a team of threads.
Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
  {
  Parallel section executed by all threads
        .
        .
  All threads join master thread and disband
  }
Resume serial code
      .
      .
}
************
```

```
#include <omp.h>
int a,b,num_threads;
int main()
{
  printf("I am in sequential part.\n");
#pragma omp parallel num_threads (8) private (a) shared (b)
  {
    num_threads=omp_get_num_threads();
    printf("I am openMP parellized part and thread %d \n",omp_get_thread_num());
  }
}
```

Using the parallel directive

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

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

- The default state of a variable is specified by the clause *default (shared)* or *default (none)*.
  - The clause *default (shared)* implies that, by default, a variable is shared by all the threads.
  - The clause *default (none)* implies that the state of each variable used in a thread must be explicitly specified. This is generally recommended, to guard against errors arising from unintentional concurrent access to shared data.
- Just as *firstprivate* specifies how multiple local copies of a variable are initialized inside a thread, the *reduction* clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
  - The usage of the *reduction* clause is *reduction (operator: variable list)*.

  - This clause performs a reduction on the scalar variables specified in the list using the *operator*. The variables in the list are implicitly specified as being private to threads. The *operator* can be one of

    `+, *, -, &, |, ^, &&, and ||.`

  Using the reduction clause

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

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

- Computing PI using OpenMP directives (presented a Pthreads program for the same problem). The *omp_get_num_threads()* function returns the number of threads in the parallel region and the *omp_get_thread_num()* function returns the integer i.d. of each thread (recall that the master thread has an i.d. 0).

- The parallel directive specifies that all variables except *npoints*, the total number of random points in two dimensions across all threads, are local.

- Furthermore, the directive specifies that there are eight threads, and the value of sum after all threads complete execution is the sum of local values at each thread.

- The function *omp_get_num_threads* is used to determine the total number of threads. A for loop generates the required number of random points (in two dimensions) and determines how many of them are within the prescribed circle of unit diameter.

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

    #pragma omp parallel default(private) shared (npoints) \
                         reduction(+: sum) num_threads(8)
    {
      num_threads = omp_get_num_threads();
      sample_points_per_thread = npoints / num_threads;
      sum = 0;
      for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum ++;
      }
    }
```

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

0.6 Assignment:

Write a program to illustrate that errors may arise from incorrect assumptions on relative execution times of threads. (Do only 1 question)

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

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