CSCI 5451: Introduction to Parallel Computing

Lecture 7: Threads to OpenMP

computer science
& engineering

University of Minnesota
Driven to Discover®

Announcements (9/24)

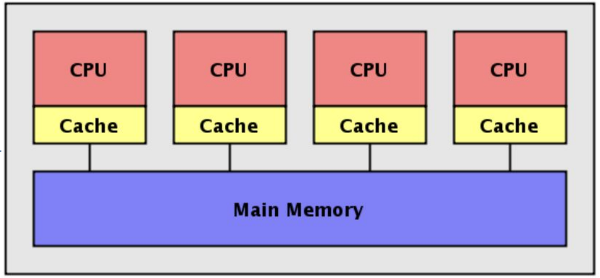
- HW1 released this Sunday
- HW Schedule for semester released later today
- Project Schedule released later today

Lecture Overview

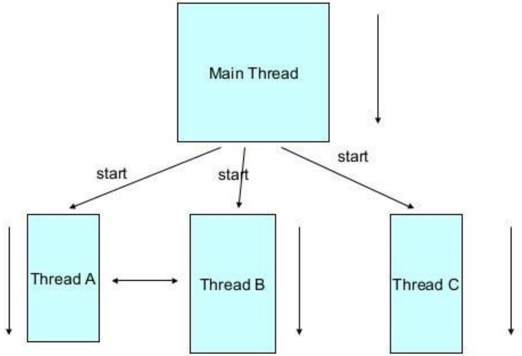
- **Recap**
- Threads
  - Background
  - Pi Computation Example
  - Threading in Detail
- OpenMP
  - Basics
  - Variable Scoping
  - Other Clauses & Functions
  - OpenMP Pi Program

Recap (Threads)

Threads as Shared Memory Processes



Source: Kaminsky/Parallel Java



Threads Enable Parallelism

Lecture Overview

- Recap
- **Threads**
  - **Background (cont'd)**
  - Pi Computation Example
  - Threading in Detail
- OpenMP
  - Basics
  - Variable Scoping
  - Other Clauses & Functions
  - OpenMP Pi Program

Creating a Thread (API)

- We create a thread using *pthread_create*
- The arguments for this function are
  - *thread_handle* → Pointer to thread being created
  - *attribute* → sets thread attributes such as stack size, priority, etc. Typically, you can just use NULL for default settings
  - *thread_function* → The function we want each thread to run. Its signature should be (void *) thread_function(void *)
  - arg → A pointer to the argument passed to *thread_function* (more complex arguments are typically wrapped in a struct)
- The function returns 0 if the thread completes successfully, otherwise an error code (nonzero)

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

Exiting a Thread (API)

- We exit from a thread with *pthread_exit*
- The arguments for this function are
  - *retval* → A pointer to the value we want to return to the callee thread
- No return value

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Waiting for Threads to Complete (API)

- We wait for threads on the main thread using *pthread_join*
- The arguments for this function are
  - *thread* → The thread we are waiting on to complete
  - *ptr* → The value returned by the thread upon exiting
- Returns 0 when there is no error code, a nonzero value otherwise

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

Simple Example

```
#include <stdio.h>
#include <pthread.h>
#include <stdint.h>   // for intptr_t

void* worker(void* arg) {
    int error_code = 42;  // nonzero value
    pthread_exit((void*)(intptr_t)error_code);
}

int main() {
    pthread_t thread;
    void* retval;

    pthread_create(&thread, NULL, worker, NULL);
    pthread_join(thread, &retval);

    int result = (int)(intptr_t)retval;  // retrieve the integer
    printf("Thread returned: %d\n", result);  // prints 42

    return 0;
}
```

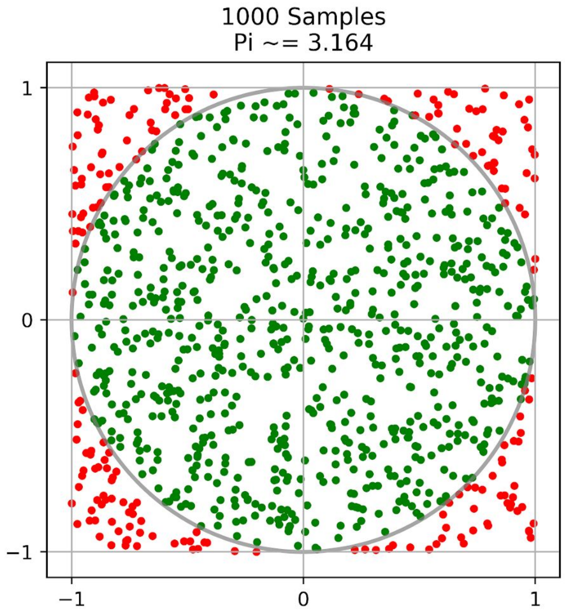
Lecture Overview

- Recap
- **Threads**
  - Background
  - **Pi Computation Example**
  - Threading in Detail
- OpenMP
  - Basics
  - Variable Scoping
  - Other Clauses & Functions
  - OpenMP Pi Program

Computing Pi with Monte Carlo Estimation

- We randomly sample x, y points in [-1,1]x[-1,1] (in the program on the next page, we use [0,1]x[0,1], but the principle is the same)
- If a point is less than a distance of 1 from [0,0], then we call it a hit (green),
- If it is greater than a distance of 1 from [0,0] we call it a miss (red)
- Pi is then approximately equal to

hits/(hits+misses)



Computing Pi with Monte Carlo Estimation

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

#define MAX_THREADS 512
void *compute_pi (void *);

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

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

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

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

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

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

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

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

Computing Pi with Monte Carlo Estimation

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

#define MAX_THREADS 512
void *compute_pi (void *);

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

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

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

In this example, an attributes object is created that enables threads to compete with all other threads in the system. Helpful for ensuring each CPU core is used. Is the default on most Linux/Unix systems.

Computing Pi with Monte Carlo Estimation

We create *num_threads* threads, pass in the attribute variable, a reference to the *compute_pi* function, and the location of the *hits* array we want this thread to update

```
    scanf("%d", &num_threads);

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

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
         pthread_create(&p_threads[i], &attr, compute_pi,
             (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
         ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

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

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

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

Computing Pi with Monte Carlo Estimation

Each thread samples *sample_points_per_thread*, determining the total number of *local_hits*.
Crucially, this is done in parallel.

```
    scanf("%d", &num_threads);

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

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
         pthread_create(&p_threads[i], &attr, compute_pi,
             (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
         pthread_join(p_threads[i], NULL);
         total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
         ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

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

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

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

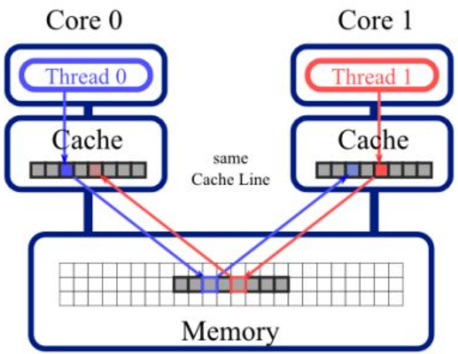
Computing Pi with Monte Carlo Estimation

Why are we not updating the *hit_pointer* directly (i.e. why do we create a local *local_hits* variable)?

```
    scanf("%d", &num_threads);

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

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

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

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

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

Computing Pi with Monte Carlo Estimation

False Sharing




```c
    scanf("%d", &num_threads);

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

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

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

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

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

Computing Pi with Monte Carlo Estimation

We make sure to set the *hit_pointer* to the *local_hits* computed on this thread.

```
    scanf("%d", &num_threads);

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

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

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

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

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

Computing Pi with Monte Carlo Estimation

The main thread waits for all of the threads to complete. Once they are complete, the *total_hits* variable accumulates the value in each thread.

```
    scanf("%d", &num_threads);

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

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

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

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

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

Computing Pi with Monte Carlo Estimation

Once all other threads have completed execution, the main thread has the complete number of hits (*total_hits*) and can finish the estimation of pi.

```
    scanf("%d", &num_threads);

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

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

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

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

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

Lecture Overview

- Recap
- **Threads**
  - Background
  - Pi Computation Example
  - **Threading in Detail**
- OpenMP
  - Basics
  - Variable Scoping
  - Other Clauses & Functions
  - OpenMP Pi Program

Race Conditions in Threads

- Sometimes concurrent threads may issue updates to the same variable
- In general, if the outcome of a multithreaded program depends on the order of execution of the threads, then we say the program has a race condition

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

int counter = 0;  // shared global variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // <-- race condition occurs here
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // create two threads that increment the same counter
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

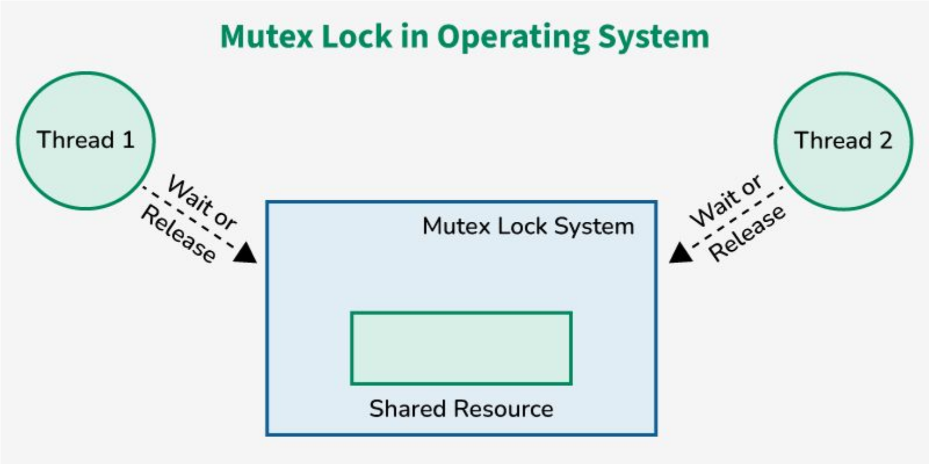
Race Conditions in Threads

How do we resolve this?

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

int counter = 0;  // shared global variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // <-- race condition occurs here
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // create two threads that increment the same counter
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Race Conditions in Threads

How do we resolve this?




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

int counter = 0;  // shared global variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // <-- race condition occurs here
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // create two threads that increment the same counter
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Mutex APIs

- For all four functions below:
  - Returns 0 if successful, nonzero for error code
  - Pointer to the lock (**mutex_lock*) for intended use as first argument
- *int pthread_mutex_lock(pthread_mutex_t *mutex_lock)*
  - Locks thread. If the lock is already held by another thread, then this call blocks.
- *Int pthread_mutex_unlock(pthread_mutex_t *mutex_lock)*
  - Unlocks thread. Allows other threads to lock the thread after waiting.
- *Int pthread_mutex_init(pthread_mutex_t *mutex_lock, const pthread_mutexattr_t *lock_attr)*
  - Initializes the lock to unlocked state
  - Allows attributes to alter the default state of the lock
- *Int pthread_mutex_destroy(pthread_mutex_t *mutex_lock)*
  - Destroys the given mutex lock and clears up memory usage by the lock

Mutex Example (Counter)

We alter the earlier example to use mutexes and resolve the race condition

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

int counter = 0;                  // shared global variable
pthread_mutex_t counter_mutex;    // mutex to protect counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&counter_mutex);  // lock before accessing counter
        counter++;
        pthread_mutex_unlock(&counter_mutex); // unlock after updating
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // initialize the mutex
    pthread_mutex_init(&counter_mutex, NULL);

    // create two threads
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);  // should always be 200000

    // destroy the mutex
    pthread_mutex_destroy(&counter_mutex);

    return 0;
}
```

Mutex Example (Counter)

Is this a good way to parallelize this program with mutexes?

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

int counter = 0;                    // shared global variable
pthread_mutex_t counter_mutex;    // mutex to protect counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&counter_mutex);  // lock before accessing counter
        counter++;
        pthread_mutex_unlock(&counter_mutex); // unlock after updating
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // initialize the mutex
    pthread_mutex_init(&counter_mutex, NULL);

    // create two threads
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);  // should always be 200000

    // destroy the mutex
    pthread_mutex_destroy(&counter_mutex);

    return 0;
}
```

Mutex Example (Counter)

Is this a good way to parallelize this program with mutexes?

No - the serial portion of the program is too large as every iteration of counter++ will be run serially.

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

int counter = 0;                    // shared global variable
pthread_mutex_t counter_mutex;    // mutex to protect counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&counter_mutex);  // lock before accessing counter
        counter++;
        pthread_mutex_unlock(&counter_mutex); // unlock after updating
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // initialize the mutex
    pthread_mutex_init(&counter_mutex, NULL);

    // create two threads
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);  // should always be 200000

    // destroy the mutex
    pthread_mutex_destroy(&counter_mutex);

    return 0;
}
```

Mutex Example (Counter)

Better Counter Program

Perform updates on a local counter first, then sum together into global counter

```
// We ignore headers for visualization purposes
int counter = 0;                    // shared global variable
pthread_mutex_t counter_mutex;    // mutex to protect counter
#define INCREMENTS 100000

void* increment(void* arg) {
    int local_counter = 0;
    for (int i = 0; i < INCREMENTS; i++) {
        local_counter++;
    }
    pthread_mutex_lock(&counter_mutex);
    counter += local_counter;
    pthread_mutex_unlock(&counter_mutex);
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&counter_mutex, NULL);

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&counter_mutex);

    return 0;
}
```

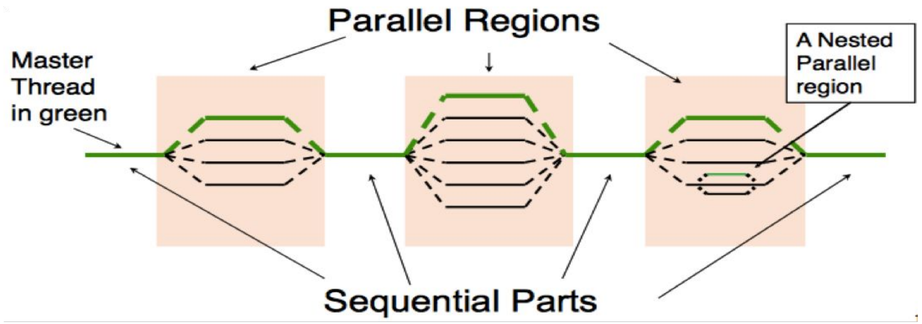
Lecture Overview

- Recap
- Threads
  - Background
  - Pi Computation Example
  - Threading in Detail
- **OpenMP**
  - **Basics**
  - Variable Scoping
  - Other Clauses & Functions
  - OpenMP Pi Program

OpenMP

- Builds on pthreads library
- Makes use of the fork-join model (image at right)
- Uses compiler directives to map from higher level *directives* + *clauses* to lower-level pthreads *function calls*
- Allows fine-grain control over *scheduling*, *variable scoping* & *synchronization* via simple clauses

Fork-Join



Directives, Clauses, Functions, Environment Variables

- The OpenMP library allows control via
  - Directives
  - Clauses
  - Functions
  - Environment Variables
- We will discuss each in turn

Directives, Clauses, Functions, Environment Variables

- The OpenMP library allows control via
  - Directives
  - Clauses
  - Functions
  - Environment Variables
- We will discuss each in turn

High level compiler directives used for determining parallel execution, defining work sharing & setting synchronization points

Directives, Clauses, Functions, Environment Variables

- The OpenMP library allows control via
  - Directives
  - Clauses
  - Functions
  - Environment Variables
- We will discuss each in turn

Defined using
#pragma omp directive
where ***directive*** is replaced with one of the below examples (to be covered in the following slides/lectures)

Example Directives
- parallel
- for
- sections
- critical
- atomic
- barrier
- …

Directives, Clauses, Functions, Environment Variables

- The OpenMP library allows control via
  - Directives
  - Clauses
  - Functions
  - Environment Variables
- We will discuss each in turn

Clauses modify and define the function of the directive. More than one can be used at a time

#pragma omp directive [clause list]

Example Clauses

- if
- num_threads
- private
- firstprivate
- shared
- default
- …

Directives, Clauses, Functions, Environment Variables

- The OpenMP library allows control via
  - Directives
  - Clauses
  - Functions
  - Environment Variables
- We will discuss each in turn

OpenMP provides a number of functions which can be used as any other normal C functions

Example functions

- omp_get_num_threads()
- omp_get_thread_num()
- …

Directives, Clauses, Functions, Environment Variables

- The OpenMP library allows control via
  - Directives
  - Clauses
  - Functions
  - Environment Variables
- We will discuss each in turn

OpenMP uses a set of environment variables that can be set outside of the program and influence program execution

Example Environment Variables

- OMP_NUM_THREADS
- OMP_DYNAMIC
- OMP_NESTED
- OMP_SCHEDULE

Parallel directive

- The first directive we discuss, and the most important, is the ***parallel*** directive
- All OpenMP programs are serial outside of ***parallel*** blocks
- On entering the block, a number of threads is created
- Each thread executed ***structured block***
- The thread that first encounters the parallel directive is called the ‘main’ thread and is assigned thread id 0 in that thread group

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

Mapping From OpenMP to pthreads

What we program

```
int main() {
    // serial segment

    #pragma omp parallel
    {
        // parallel segment
    }
}
```

Mapping From OpenMP to pthreads

What we program

```
int main() {
    // serial segment

    #pragma omp parallel
    {
        // parallel segment
    }
}
```

What the program is translated to

```
#define NUM_THREADS 16

void* parallel_segment(void* arg) {
    // parallel segment
}

int main() {
    // serial segment

    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, parallel_segment, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
}
```

Mapping From OpenMP to pthreads

What the program is translated to

Where does this definition come from?
We discuss in later slides/lectures.

```
#define NUM_THREADS 16

void* parallel_segment(void* arg) {
    // parallel segment
}

int main() {
    // serial segment

    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, parallel_segment, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
}
```

Lecture Overview

- Recap
- Threads
  - Background
  - Pi Computation Example
  - Threading in Detail
- **OpenMP**
  - Basics
  - **Variable Scoping**
  - Other Clauses & Functions
  - OpenMP Pi Program

Scoping Options

- Allows control over Data - which variables should be shared, private, equivalent, etc.
- Multiple variables can be scoped at the same time with

#pragma omp directive scope(a, b, c, …)

- Clauses to Control Scope
    - private
    - shared
    - firstprivate
    - lastprivate [Covered in later lecture]
    - threadprivate [Covered in later lecture]

Scoping Options

- Allows control over Data - which variables should be shared, private, equivalent, etc.
- Multiple variables can be scoped at the same time with

#pragma omp directive scope(a, b, c, ...)

- Clauses to Control Scope
  - private
  - shared
  - firstprivate
  - lastprivate [Covered in later lecture]
  - threadprivate [Covered in later lecture]

For now we assume this is ***parallel***, but we will see other examples of different directives in the next lectures

scope in this context is one of the options below (private, shared, etc.)

Scoping Options

- Allows control over Data - which variables should be shared, private, equivalent, etc.
- Multiple variables can be scoped at the same time with

#pragma omp directive scope(a, b, c, ...)

- Clauses to Control Scope
  - private
  - shared
  - firstprivate
  - lastprivate [Covered in later lecture]
  - threadprivate [Covered in later lecture]

- Each thread gets its own **uninitialized copy** of the variable.
- The original value (from before entering the parallel region) is **not copied in**.
- The private copy is created when the thread enters the parallel region and destroyed when it exits.
- When the region ends, the private value is lost; the original variable outside the region is unchanged.

Scoping Options

- Allows control over Data - which variables should be shared, private, equivalent, etc.
- Multiple variables can be scoped at the same time with

#pragma omp directive scope(a, b, c, ...)

- Clauses to Control Scope
  - private
  - shared
  - firstprivate
  - lastprivate [Covered in later lecture]
  - threadprivate [Covered in later lecture]

What we program

```
#pragma omp parallel private(a)
```

What the program is translated to

```
#define N 8
void* f(void* arg){
        int a;
        //parallel block
}

int main(){
   int a=9;
   pthread_t t[N];
   for(int i=0;i<N;i++) pthread_create(&t[i],0,f,0);
   for(int i=0;i<N;i++) pthread_join(t[i],0);
}
```

Scoping Options

- Allows control over Data - which variables should be shared, private, equivalent, etc.
- Multiple variables can be scoped at the same time with

#pragma omp directive scope(a, b, c, ...)

- Clauses to Control Scope
  - private
  - shared
  - firstprivate
  - lastprivate [Covered in later lecture]
  - threadprivate [Covered in later lecture]

- All threads share the same instance of the variable.
- Modifications by one thread are visible to all others.
- Requires synchronization (locks, atomics, reductions, etc.) if multiple threads write to it.

Scoping Options

- Allows control over Data - which variables should be shared, private, equivalent, etc.
- Multiple variables can be scoped at the same time with

#pragma omp directive scope(a, b, c, ...)

- Clauses to Control Scope
  - private
  - shared
  - firstprivate
  - lastprivate [Covered in later lecture]
  - threadprivate [Covered in later lecture]

What we program

```
#pragma omp parallel shared(a)
```

What the program is translated to

```
#define N 8
void* f(void* arg){
        int* a = (int*)arg;
        //parallel block - must be synchronized
}

int main(){
    int a=9;
    pthread_t t[N];
    for(int i=0;i<N;i++) pthread_create(&t[i],0,f,&a);
    for(int i=0;i<N;i++) pthread_join(t[i],0);
}
```

Scoping Options

- Allows control over Data - which variables should be shared, private, equivalent, etc.
- Multiple variables can be scoped at the same time with

#pragma omp directive scope(a, b, c, ...)

- Clauses to Control Scope
  - private
  - shared
  - firstprivate
  - lastprivate [Covered in later lecture]
  - threadprivate [Covered in later lecture]

- Like `private`, each thread gets its own copy.
- But **the value of the variable before the region is copied in**.
- Each thread's private copy starts with the same initial value (copied from the "master" thread's variable at entry).

Scoping Options

- Allows control over Data - which variables should be shared, private, equivalent, etc.
- Multiple variables can be scoped at the same time with

#pragma omp directive scope(a, b, c, ...)

- Clauses to Control Scope
  - private
  - shared
  - firstprivate
  - lastprivate [Covered in later lecture]
  - threadprivate [Covered in later lecture]

What we program

```
#pragma omp parallel firstprivate(a)
```

What the program is translated to

```
#define N 8
void* f(void* arg){
        int a = *(int*)arg;
        //parallel block
}

int main(){
   int a=9;
   pthread_t t[N];
   for(int i=0;i<N;i++) pthread_create(&t[i],0,f,&a);
   for(int i=0;i<N;i++) pthread_join(t[i],0);
}
```

Scoping Defaults

- We can set the default behavior for variables
- Options
  - shared, private, firstprivate
  - none
- External Variables
- Locally Declared Variables
- Referenced Data Structures

Scoping Defaults

- We can set the default behavior for variables
- Options
  - shared, private, firstprivate
  - none
- External Variables
- Locally Declared Variables
- Referenced Data Structures

All variables used within the ***parallel*** block, but declared outside of the block, use the chosen option

Example:
The following sets integers ***a,b*** to be private

```
int a, b;
#pragma omp parallel default (private)
{
  // a, b used somewhere here
}
```

Scoping Defaults

- We can set the default behavior for variables
- Options
  - shared, private, firstprivate
  - none
- External Variables
- Locally Declared Variables
- Referenced Data Structures

All variables used within the ***parallel*** block, but declared outside of the block, must be explicitly specified, or the program will throw an error

Example:
The following sets integers ***a,b*** to be private

```
int a, b;
#pragma omp parallel default (none) private (a, b)
{
  // a, b used somewhere here
}
```

Scoping Defaults

- We can set the default behavior for variables
- Options
  - shared, private, firstprivate
  - none
- External Variables
- Locally Declared Variables
- Referenced Data Structures

If variables are declared outside of the scope of the ***parallel*** block, they do not need to have their scope declared before entering

Example:
The following ignores ***a, b*** as they are not used in the block.

```
int a, b;
#pragma omp parallel
{
  // a, b are not used here
}
```

Scoping Defaults

- We can set the default behavior for variables
- Options
  - shared, private, firstprivate
  - none
- External Variables
- Locally Declared Variables
- Referenced Data Structures

If variables are declared inside the ***parallel*** block, then we do not need to declare their scopes with a clause

Example:
The following ignores ***a, b*** for scoping clauses as they are declared inside of the ***parallel block***

```
#pragma omp parallel
{
   int a, b;
  // a, b are used here
}
```

Scoping Defaults

- We can set the default behavior for variables
- Options
  - shared, private, firstprivate
  - none
- External Variables
- Locally Declared Variables
- Referenced Data Structures

Pointers to structs & arrays will not scope recursively. In other words, the elements in a struct, or the items in an array will not inherit the scope of the pointer when used in an OpenMP clause

Example:
The following example shows how declaring a pointer to an int array will not make private each entry in ***a***.

```
int *a;
// prepare 'a' as a vector of ints
#pragma omp parallel private (a)
{
  // The pointer may be updated in each thread
 // but each entry in a* is shared
}
```

Scoping Defaults

- We can set the default behavior for variables
- Options
  - shared, private, firstprivate
  - none
- External Variables
- Locally Declared Variables
- Referenced Data Structures

Caveat: Statically declared arrays will be copied.

Pointers to structs & arrays will not scope recursively. In other words, the elements in a struct, or the items in an array will not inherit the scope of the pointer when used in an OpenMP clause

Example:
The following example shows how declaring a pointer to an int array will not make private each entry in ***a***.

```
int *a;
// prepare 'a' as a vector of ints
#pragma omp parallel private (a)
{
  // The pointer may be updated in each thread
 // but each entry in a* is shared
}
```

Lecture Overview

- Recap
- Threads
  - Background
  - Pi Computation Example
  - Threading in Detail
- **OpenMP**
  - Basics
  - Variable Scoping
  - **Other Clauses & Functions**
  - OpenMP Pi Program

Example Clauses

In addition to variable scoping clauses,
there are a number of other clauses

- ***if***
- ***num_threads***
- ***reduction***
- [More discussed in later lectures]

Example Clauses

In addition to variable scoping clauses, there are a number of other clauses

- ❑ ***if***
- ❑ ***num_threads***
- ❑ ***reduction***
- ❑ [More discussed in later lectures]

- The `if` clause determines at **runtime** whether a parallel region executes in parallel or serial.
- Syntax: `#pragma omp parallel if(condition)` → if `condition` is true, parallel threads are spawned; if false, the region runs with **only the master thread**.
- Useful for **small workloads** where parallel overhead would outweigh benefits.

Example Clauses

In addition to variable scoping clauses, there are a number of other clauses

- ***if***
- ***num_threads***
- ***reduction***
- [More discussed in later lectures]

What we program

```
#pragma omp parallel if (is_parallel == 1)
```

What the program is translated to

```
#define N 8
void* f(void* arg){
    // parallel block
    return NULL;
}

int main(){
    int is_parallel;
    // is_parallel is set dynamically during execution
    pthread_t t[N];
    if(is_parallel){
        for(int i=0;i<N;i++) pthread_create(&t[i],0,f,0);
        for(int i=0;i<N;i++) pthread_join(t[i],0);
    } else {
        f(0); // serial execution
    }
}
```

Example Clauses

In addition to variable scoping clauses, there are a number of other clauses

- ❑ ***if***
- ❑ ***num_threads***
- ❑ ***reduction***
- ❑ [More discussed in later lectures]

- The `num_threads` clause **explicitly sets** the number of threads for a parallel region.
- Syntax: `#pragma omp parallel num_threads(n)` → spawns exactly `n` threads for that region (if system resources allow).
- There are multiple ways the number of threads can be set (we reserve other methods to future lectures)

Example Clauses

In addition to variable scoping clauses, there are a number of other clauses

- ***if***
- ***num_threads***
- ***reduction***
- [More discussed in later lectures]

What we program

```
#pragma omp parallel num_threads (42)
```

What the program is translated to

```
#define N 42
void* f(void* arg){
    // parallel block
    return NULL;
}

int main(){
    pthread_t t[N];
    for(int i=0;i<N;i++) pthread_create(&t[i],0,f,0);
    for(int i=0;i<N;i++) pthread_join(t[i],0);
}
```

Example Clauses

In addition to variable scoping clauses, there are a number of other clauses

- ❑ ***if***
- ❑ ***num_threads***
- ❑ ***reduction***
- ❑ [More discussed in later lectures]

- The `reduction` clause **combines a private copy of a variable from each thread** into a single result at the end of a parallel region. All local copies are combined using some operator (`+`, `-`, etc.).
- Syntax: `#pragma omp parallel for reduction(+:sum)` → each thread has its own `sum`, which are combined using the `+` operator after the loop.
- Supports common operators like `+`, `*`, `-`, `&`, `|`, and user-defined functions. User defined functions should be symmetric functions (i.e. f(x, y) = f(y, x)).
- Local copies are combined in a tree-like, recursive pattern when there are a large number of threads in order to increase parallelism.
- All copies of the variable to be reduced will be private on each thread

Example Clauses

In addition to variable scoping clauses, there are a number of other clauses

- ***if***
- ***num_threads***
- ***reduction***
- [More discussed in later lectures]

Example in the final slide

- The `reduction` clause **combines a private copy of a variable from each thread** into a single result at the end of a parallel region. All local copies are combined using some operator (+, -, etc.).
- Syntax: `#pragma omp parallel for reduction(+:sum)` → each thread has its own `sum`, which are combined using the `+` operator after the loop.
- Supports common operators like `+`, `*`, `-`, `&`, `|`, and user-defined functions. User defined functions should be symmetric functions (i.e. f(x, y) = f(y, x)).
- Local copies are combined in a tree-like, recursive pattern when there are a large number of threads in order to increase parallelism.

Functions

- omp_get_num_threads()
  - Returns the **number of threads currently in the team** executing a parallel region
  - Only valid **inside a parallel region**; outside it, it returns 1.
- omp_get_thread_num()
  - Returns the **ID of the calling thread** within the current team
  - Thread IDs range from 0 to num_threads - 1
- [More discussed in later lectures]

```
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        // Only the master thread (id 0) prints the number of threads
        if (omp_get_thread_num() == 0) {
            int nthreads = omp_get_num_threads();
            printf("Number of threads: %d\n", nthreads);
        }
    }
    return 0;
}
```

Lecture Overview

- Recap
- Threads
    - Background
    - Pi Computation Example
    - Threading in Detail
- **OpenMP**
    - Basics
    - Variable Scoping
    - Other Clauses & Functions
    - **OpenMP Pi Program**

Pi Example

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

Pi Example

Let's walk through each clause/directive in detail

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

Pi Example

Is this a directive or clause? Why is it needed?

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

Pi Example

This *directive* is needed to enable parallel execution

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

Pi Example

These scoping clauses will determine how variables are copied/shared between threads. Which variables in the below program are impacted by these two statements? What impact does this statement have on those variables?

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

Pi Example

i, num_threads, sample_points_per_thread, sum, rand_no_x, rand_no y are all private (although technically, the ***reduction(+: sum)*** clause overrides.

n_points is shared

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

Pi Example

What does this clause do in the program?

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

Pi Example

Declares sum as private to each variable during block execution. Sums ('+') each local copy of ***sum*** into the final summation. Results in a full ***sum*** stored on the master thread at the end of program execution.

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

Pi Example

And this clause?

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

Pi Example

Parallelizes the block with a number of threads equal to 8.

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