Name: Nitesh Sharma
Reg No: 23361
MTCS-103(P)
Parallel Processing Practicals
Assignment 9

==

Lock Routines:

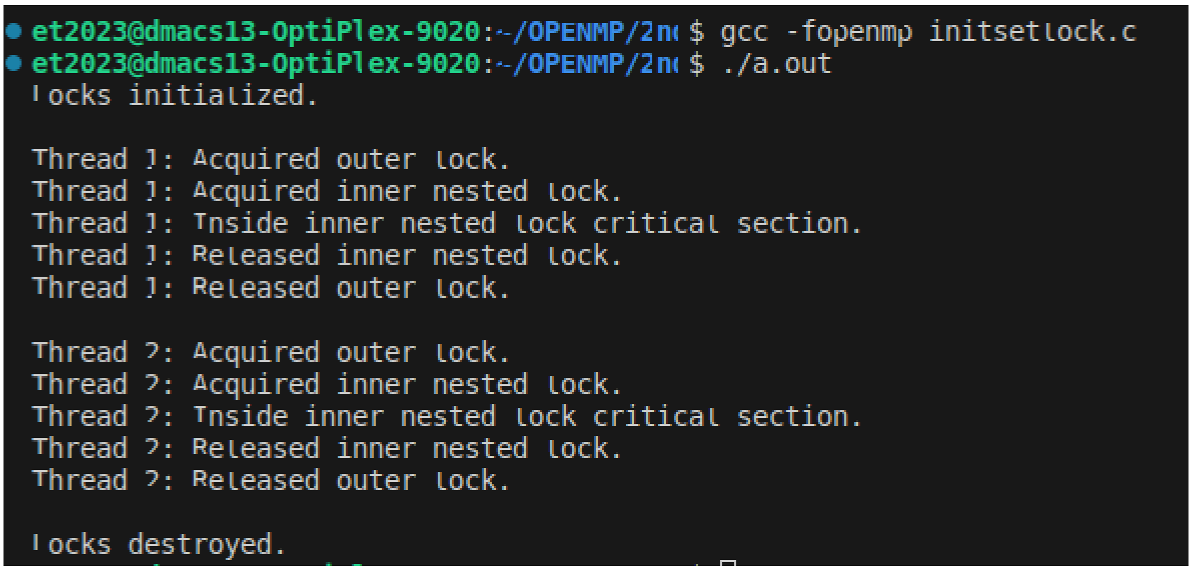
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);

The lock routines in OpenMP are designed to facilitate synchronization among threads using locks. The omp_init_lock function initializes a regular OpenMP lock, while the omp_init_nest_lock function initializes a nested OpenMP lock. Both of these routines are responsible for creating and initializing locks, enabling threads to use them for synchronization purposes. These locks help manage concurrent access to shared resources in a controlled and orderly manner, preventing race conditions and ensuring the correct behavior of parallel programs.

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h> // for usleep

int main() {
  omp_lock_t outer_lock;
  omp_nest_lock_t inner_lock;

  omp_init_lock(&outer_lock);
  omp_init_nest_lock(&inner_lock);

  printf("Locks initialized.\n\n");

  #pragma omp parallel sections
  {
     #pragma omp section
     {
        omp_set_lock(&outer_lock);
        printf("Thread %d: Acquired outer lock.\n", omp_get_thread_num());

        omp_set_nest_lock(&inner_lock);
        printf("Thread %d: Acquired inner nested lock.\n", omp_get_thread_num());
```

```
            printf("Thread %d: Inside inner nested lock critical section.\n", omp_get_thread_num());

            omp_unset_nest_lock(&inner_lock);
            printf("Thread %d: Released inner nested lock.\n", omp_get_thread_num());

            omp_unset_lock(&outer_lock);
            printf("Thread %d: Released outer lock.\n\n", omp_get_thread_num());
        }

        #pragma omp section
        {
            usleep(1000);

            omp_set_lock(&outer_lock);
            printf("Thread %d: Acquired outer lock.\n", omp_get_thread_num());

            omp_set_nest_lock(&inner_lock);
            printf("Thread %d: Acquired inner nested lock.\n", omp_get_thread_num());

            printf("Thread %d: Inside inner nested lock critical section.\n", omp_get_thread_num());

            omp_unset_nest_lock(&inner_lock);
            printf("Thread %d: Released inner nested lock.\n", omp_get_thread_num());

            omp_unset_lock(&outer_lock);
            printf("Thread %d: Released outer lock.\n\n", omp_get_thread_num());
        }
    }

    omp_destroy_lock(&outer_lock);
    omp_destroy_nest_lock(&inner_lock);

    printf("Locks destroyed.\n");

    return 0;
}
```

Output:



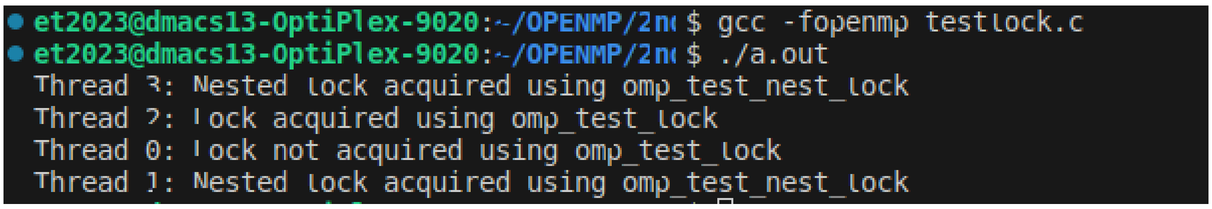
```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp initsetlock.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
 Locks initialized.

 Thread 1: Acquired outer lock.
 Thread 1: Acquired inner nested lock.
 Thread 1: Inside inner nested lock critical section.
 Thread 1: Released inner nested lock.
 Thread 1: Released outer lock.

 Thread 2: Acquired outer lock.
 Thread 2: Acquired inner nested lock.
 Thread 2: Inside inner nested lock critical section.
 Thread 2: Released inner nested lock.
 Thread 2: Released outer lock.

 Locks destroyed.
```

Explanation:
The above code demonstrates the usage of locks in OpenMP for synchronization. It initializes an outer lock and a nested lock. Two sections of parallel execution are defined, each representing a thread. Within each section, a thread acquires the outer lock and then the nested lock, entering a critical section. A small delay is introduced after releasing the inner nested lock to allow other threads to access the critical section. The output shows the threads acquiring and releasing locks, along with entering and exiting the critical section. The interleaved execution of threads highlights the synchronization achieved through the locks, as different threads take turns accessing the critical section while respecting the locks' mutual exclusion. Finally, the locks are destroyed at the end of the program.

int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);

The functions omp_test_lock and omp_test_nest_lock are used in OpenMP for attempting to acquire a lock without suspending the execution of the calling task. These routines check whether a lock can be acquired and return a value indicating success or failure. If the lock is available, the function acquires it and returns a non-zero value; otherwise, it returns 0 without acquiring the lock. These functions are useful when you want to avoid waiting for a lock and instead continue executing the task if the lock is not available at the moment. This can help improve task throughput and efficiency by reducing unnecessary waiting times.

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

int main() {
  omp_lock_t lock;
  omp_init_lock(&lock);

  int num_threads = 4;
  omp_set_num_threads(num_threads);
  #pragma omp parallel
  {
      int thread_id = omp_get_thread_num();

      if (thread_id % 2 == 0) {
          int lock_acquired = omp_test_lock(&lock);
          if (lock_acquired) {
              printf("Thread %d: Lock acquired using omp_test_lock\n", thread_id);
              omp_unset_lock(&lock);
          } else {
              printf("Thread %d: Lock not acquired using omp_test_lock\n", thread_id);
          }
      } else {
          omp_nest_lock_t nest_lock;
          omp_init_nest_lock(&nest_lock);

          int lock_acquired = omp_test_nest_lock(&nest_lock);
          if (lock_acquired) {
              printf("Thread %d: Nested lock acquired using omp_test_nest_lock\n", thread_id);
              omp_unset_nest_lock(&nest_lock);
          } else {
              printf("Thread %d: Nested lock not acquired using omp_test_nest_lock\n", thread_id);
          }
      }
  }

  omp_destroy_lock(&lock);

  return 0;
}
```
Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp testlock.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
  Thread 3: Nested lock acquired using omp_test_nest_lock
  Thread 2: Lock acquired using omp_test_lock
  Thread 0: Lock not acquired using omp_test_lock
  Thread 1: Nested lock acquired using omp_test_nest_lock
```

Explanation:

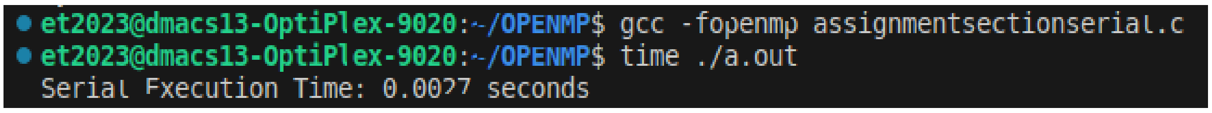
In this code, multiple threads attempt to acquire locks using omp_test_lock and omp_test_nest_lock. Threads with even thread IDs attempt to acquire a regular lock, while threads with odd thread IDs attempt to acquire a nested lock.

double omp_get_wtime(void):

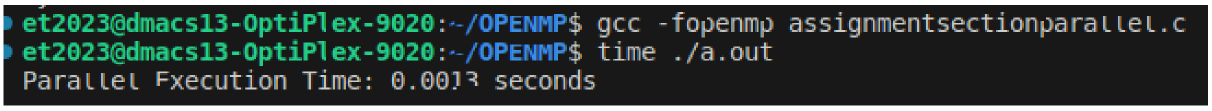
The omp_get_wtime() function is a useful OpenMP routine that provides a way to measure the elapsed wall clock time in seconds. It returns a double value representing the current time, which can be used to measure the execution time of a section of code. This function is particularly valuable when dealing with parallelized code, as it allows you to accurately measure the time taken by parallel threads or tasks. By taking timestamps before and after a specific code segment, you can calculate the time taken for its execution, aiding in performance analysis and optimization of parallel programs.

```
#include <stdio.h>
#include <omp.h>
#define ARRAY_SIZE 1000000
void calculateSquaresParallel(long long int array[]) {
  #pragma omp parallel sections
  {
      #pragma omp section
      for (long long int i = 0; i < ARRAY_SIZE / 5; ++i) {
          array[i] = array[i] * array[i];
      }
      #pragma omp section
      for (long long int i = ARRAY_SIZE / 5; i < 2 * ARRAY_SIZE / 5; ++i) {
          array[i] = array[i] * array[i];
      }
      #pragma omp section
      for (long long int i = 2 * ARRAY_SIZE / 5; i < 3 * ARRAY_SIZE / 5; ++i) {
          array[i] = array[i] * array[i];
      }
      #pragma omp section
      for (long long int i = 3 * ARRAY_SIZE / 5; i < 4 * ARRAY_SIZE / 5; ++i) {
          array[i] = array[i] * array[i];
      }
      #pragma omp section
      for (long long int i = 4 * ARRAY_SIZE / 5; i < ARRAY_SIZE; ++i) {
          array[i] = array[i] * array[i];
      }
  }
}
int main() {
  long long int array[ARRAY_SIZE];

  for (long long int i = 0; i < ARRAY_SIZE; ++i) {
      array[i] = i + 1;
  }
  double start_time = omp_get_wtime();
  calculateSquaresParallel(array);
  double end_time = omp_get_wtime();
  printf("Parallel Execution Time: %.4f seconds\n", end_time - start_time);

  return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ gcc -fopenmp assignmentsectionserial.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ time ./a.out
  Serial Execution Time: 0.0027 seconds
```



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ gcc -fopenmp assignmentsectionparallel.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ time ./a.out
 Parallel Execution Time: 0.0013 seconds
```

Explanation:

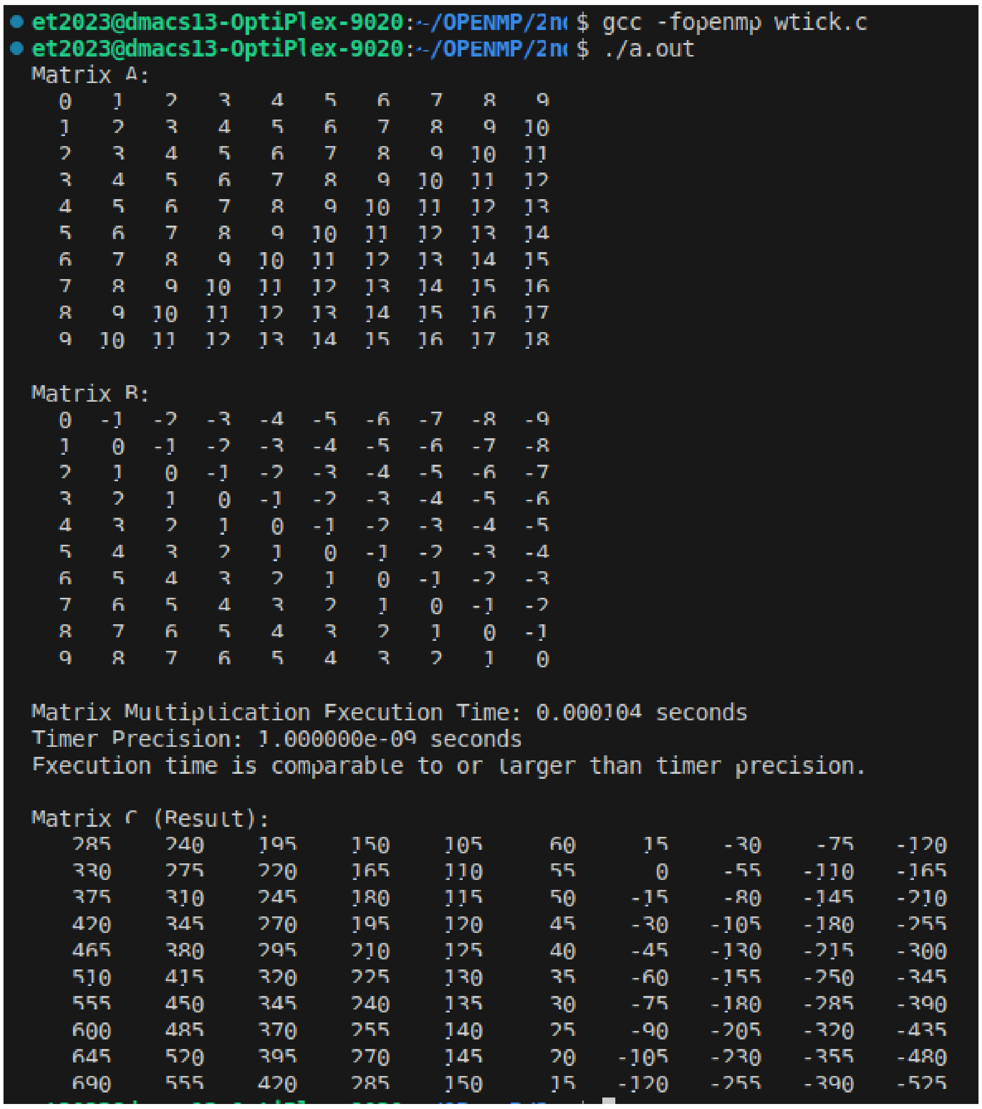
The provided code demonstrates parallel computation using OpenMP to calculate the squares of elements in an array of a significant size. The calculateSquaresParallel function divides the array into five sections and assigns each section to a separate thread for parallel processing. The omp_get_wtime() function is used to measure the execution time of the parallel computation. By recording the start and end times around the calculateSquaresParallel call, the code calculates and prints the elapsed time taken by the parallel section. Since the parallel computation leverages multiple threads to process the array sections concurrently, the execution time is significantly reduced compared to a sequential approach. This is evident when comparing the execution times of the parallel and serial versions, with the parallel execution being notably faster due to efficient parallelization.

double omp_get_wtick(void);

The function omp_get_wtick() returns the precision of the timer used by the omp_get_wtime() function in OpenMP. This precision value indicates the smallest time interval that can be reliably measured by the timer. In other words, it provides information about the granularity of the timer in terms of time units. A smaller value returned by omp_get_wtick() indicates a higher precision timer, which can accurately measure smaller time intervals. This information is helpful for understanding the level of detail and accuracy that can be achieved when using the omp_get_wtime() function to measure execution times in OpenMP programs.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 10
void matrix_multiply(int A[N][N], int B[N][N], int C[N][N]) {
  #pragma omp parallel for
  for (int i = 0; i < N; ++i) {
     for (int j = 0; j < N; ++j) {
        C[i][j] = 0;
        for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
int main() {
    int A[N][N], B[N][N], C[N][N];

    // Initialize matrices A and B
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            A[i][j] = i + j;
            B[i][j] = i - j;
        }
    }
    printf("Matrix A:\n");
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            printf("%3d ", A[i][j]);
        }
        printf("\n");
    }
    printf("\nMatrix B:\n");
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            printf("%3d ", B[i][j]);
        }
        printf("\n");
    }
    double start_time = omp_get_wtime();
    matrix_multiply(A, B, C);
    double end_time = omp_get_wtime();
    double execution_time = end_time - start_time;
    printf("\nMatrix Multiplication Execution Time: %.6f seconds\n", execution_time);
    double precision = omp_get_wtick();
    printf("Timer Precision: %.6e seconds\n", precision);

    if (execution_time < precision) {
        printf("Execution time is significantly smaller than timer precision.\n");
    } else {
        printf("Execution time is comparable to or larger than timer precision.\n");
    }
```

```c
    // Print resulting matrix C
    printf("\nMatrix C (Result):\n");
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            printf("%6d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```
Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp wtick.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
  Matrix A:
    0   1   2   3   4   5   6   7   8   9
    1   2   3   4   5   6   7   8   9  10
    2   3   4   5   6   7   8   9  10  11
    3   4   5   6   7   8   9  10  11  12
    4   5   6   7   8   9  10  11  12  13
    5   6   7   8   9  10  11  12  13  14
    6   7   8   9  10  11  12  13  14  15
    7   8   9  10  11  12  13  14  15  16
    8   9  10  11  12  13  14  15  16  17
    9  10  11  12  13  14  15  16  17  18

  Matrix B:
    0  -1  -2  -3  -4  -5  -6  -7  -8  -9
    1   0  -1  -2  -3  -4  -5  -6  -7  -8
    2   1   0  -1  -2  -3  -4  -5  -6  -7
    3   2   1   0  -1  -2  -3  -4  -5  -6
    4   3   2   1   0  -1  -2  -3  -4  -5
    5   4   3   2   1   0  -1  -2  -3  -4
    6   5   4   3   2   1   0  -1  -2  -3
    7   6   5   4   3   2   1   0  -1  -2
    8   7   6   5   4   3   2   1   0  -1
    9   8   7   6   5   4   3   2   1   0

  Matrix Multiplication Execution Time: 0.000104 seconds
  Timer Precision: 1.000000e-09 seconds
  Execution time is comparable to or larger than timer precision.

  Matrix C (Result):
     285     240     195     150     105      60      15     -30     -75    -120
     330     275     220     165     110      55       0     -55    -110    -165
     375     310     245     180     115      50     -15     -80    -145    -210
     420     345     270     195     120      45     -30    -105    -180    -255
     465     380     295     210     125      40     -45    -130    -215    -300
     510     415     320     225     130      35     -60    -155    -250    -345
     555     450     345     240     135      30     -75    -180    -285    -390
     600     485     370     255     140      25     -90    -205    -320    -435
     645     520     395     270     145      20    -105    -230    -355    -480
     690     555     420     285     150      15    -120    -255    -390    -525
```
Explanation:

The above code performs matrix multiplication for square matrices of size 10x10 using OpenMP parallelization. The matrices A and B are initialized with values based on their indices, and the resulting matrix C is calculated by distributing the computation across multiple threads using OpenMP's parallel for construct. The execution time of the matrix multiplication is measured using omp_get_wtime(), which returns the elapsed wall clock time in seconds. The code also calculates the timer precision using omp_get_wtick(), which provides the precision of the timer. It then compares the execution time with the timer precision to determine whether the execution time is significantly smaller, comparable, or larger than the timer precision. Finally, the resulting matrix C is printed.