Name: Amber Khurshid
Section: BAI-6A
Roll No : 22P-9295

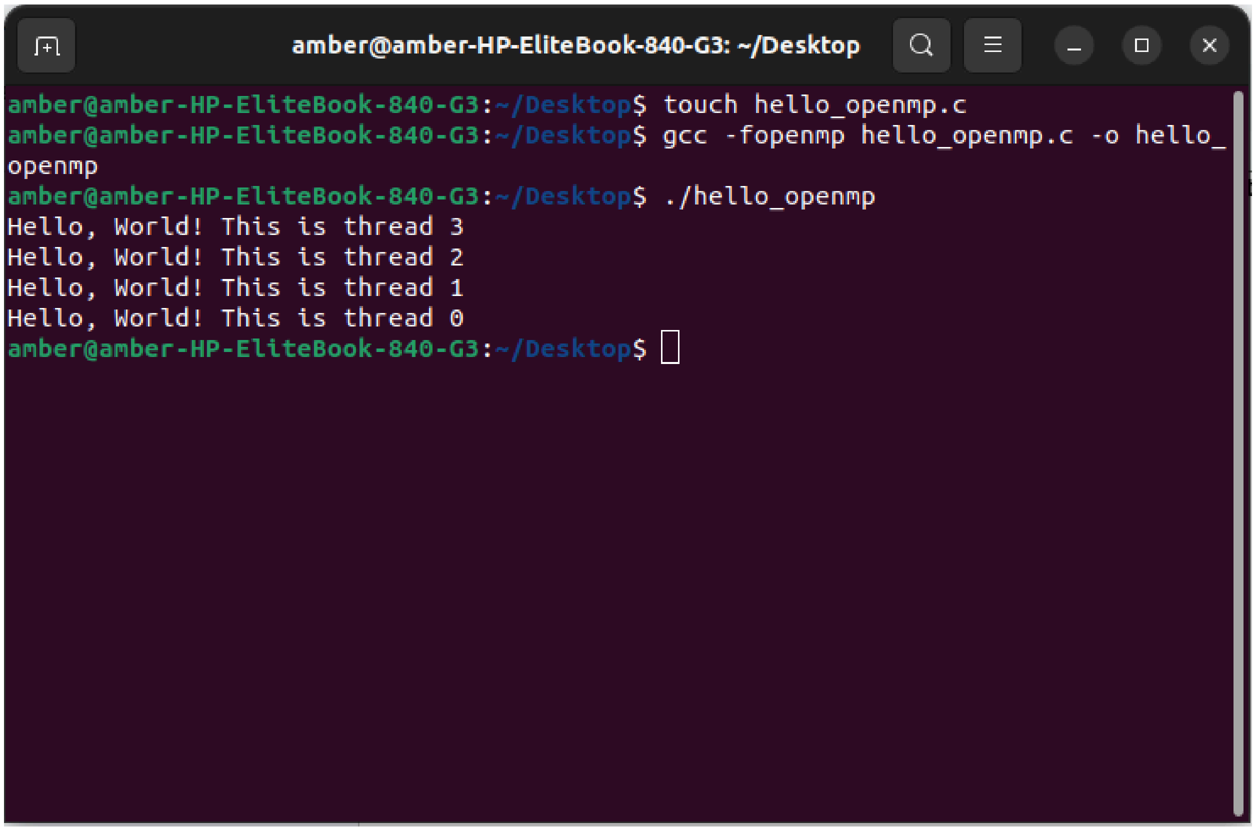
PDC ASSIGNMENT 01

Exercise 1: Setting up the Environment
Ensure you have a C/C++ compiler that supports OpenMP installed on your system. Just open the text editor or
IDE and start writing your code:

```
// Include standard input/output header file for printf function
#include <stdio.h>

// Include OpenMP header file for parallel programming support
#include <omp.h>

// Main function - program execution starts here
int main() {

        // Begin a parallel region where multiple threads execute the following block
        #pragma omp parallel
        {
        // Get the ID number of the current thread
        int thread_id = omp_get_thread_num();

        // Print "Hello, World!" along with the thread's ID
        printf("Hello, World! This is thread %d\n", thread_id);
        }

        // Return 0 to indicate that the program ended successfully
        return 0;
}
```

Compile the code using:
gcc -o hello_openmp -fopenmp hello_openmp.c

Output :



amber@amber-HP-EliteBook-840-G3: ~/Desktop

```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch hello_openmp.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp hello_openmp.c -o hello_
openmp
amber@amber-HP-EliteBook-840-G3:~/Desktop$ ./hello_openmp
Hello, World! This is thread 3
Hello, World! This is thread 2
Hello, World! This is thread 1
Hello, World! This is thread 0
amber@amber-HP-EliteBook-840-G3:~/Desktop$ 
```

Task 1

1. **Provide a list of run-time routines that are used in OpenMP**

i) omp_get_thread_num: Returns the ID of the current thread.
ii) omp_get_num_threads: Returns the number of threads in the current parallel region.
iii) omp_set_num_threads: Sets the number of threads for subsequent parallel regions.
iv) omp_get_max_threads: Returns the maximum number of threads available.
v) omp_get_num_procs: Returns the number of processors available.
vi) omp_in_parallel: Checks if the code is running in a parallel region.
vii) omp_set_dynamic: Enables/disables dynamic adjustment of thread count.
viii) omp_get_wtime: Returns a wall-clock time for timing parallel regions.
ix) omp_get_wtick: Returns the resolution of the timer used by omp_get_wtime.
x) omp_init_lock: Initializes a lock for synchronization.

2. **Why aren't you seing the Hello World output thread sequence as 0, 1, 2, 3 etc. Why are they disordered?**

The output is disordered because the threads run concurrently and the operating system schedules them unpredictably based on availability of CPU cores. Each thread may execute at different times thus causing a random output order.

3. What happens to the thread_id if you change its scope to before the pragma?

If thread_id is declared before the parallel region, it becomes shared among threads. Multiple threads update it simultaneously, causing a race condition and leading to incorrect or inconsistent thread IDs in the output.

4. Convert the code to serial code.

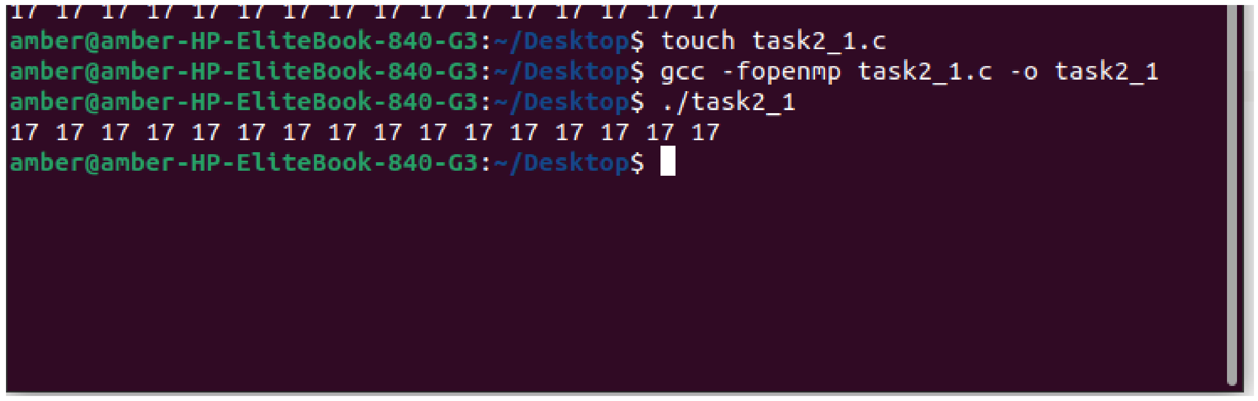
```
#include <stdio.h>

int main() {
        printf("Hello, World! This is thread 0\n");
        return 0;
}
```

Task 2

1. The following code adds two arrays of size 16 together and stores answer in result array.

```
// Include standard input/output header file for printf function
#include <stdio.h>

// Include OpenMP header file for parallel programming support
#include <omp.h>

// Main function - program execution starts here
int main() {

        // Declare and initialize the first array with 16 elements
        int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};

        // Declare and initialize the second array with 16 elements (in reverse order)
        int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

        // Declare an array to store the results
        int result[16];

        // Start a parallelized for loop where iterations are divided among threads
        #pragma omp parallel for
        for (int i = 0; i < 16; i++) {
        // Add corresponding elements of array1 and array2 and store in result array
        result[i] = array1[i] + array2[i];
        }

        // Start a parallel region where multiple threads can run
        #pragma omp parallel
        {
        // Only the master thread (thread 0) will execute the following block
        if (omp_get_thread_num() == 0) {
        // Loop through the result array and print each element
        for (int i = 0; i < 16; i++) {
                printf("%d ", result[i]);
        }
        // Print a newline after printing all elements
        printf("\n");
        }
        }

        // Return 0 to indicating  the program ended successfully
        return 0;
}
```

Output:

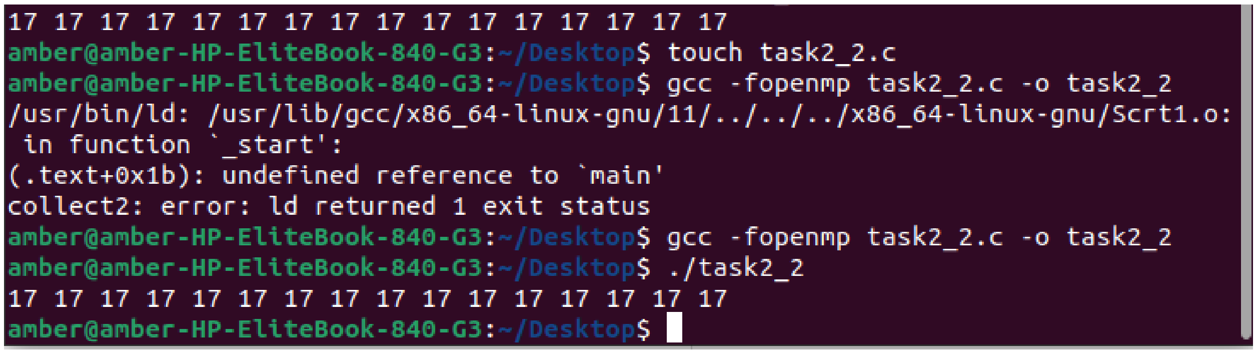


```
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch task2_1.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task2_1.c -o task2_1
amber@amber-HP-EliteBook-840-G3:~/Desktop$ ./task2_1
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Q1. Convert it into Parallel, such that only the addition part is parallelized.

```
// Include standard input/output header file for printf function
#include <stdio.h>

// Main function - program execution starts here
int main() {

        // Declare and initialize the first array with 16 elements
        int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};

        // Declare and initialize the second array with 16 elements (in reverse order)
        int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

        // Declare an array to store the results
        int result[16];

        // Loop through each index from 0 to 15
        for (int i = 0; i < 16; i++) {
        // Add corresponding elements of array1 and array2 and store the sum in result array
        result[i] = array1[i] + array2[i];
        }

        // Loop through each element in the result array and print it
        for (int i = 0; i < 16; i++) {
        printf("%d ", result[i]); // Print the current element followed by a space
        }

        // Print a newline after the entire array has been printed
        printf("\n");

        // Return 0 to indicate that the program ended successfully
        return 0;
}
```

Output:



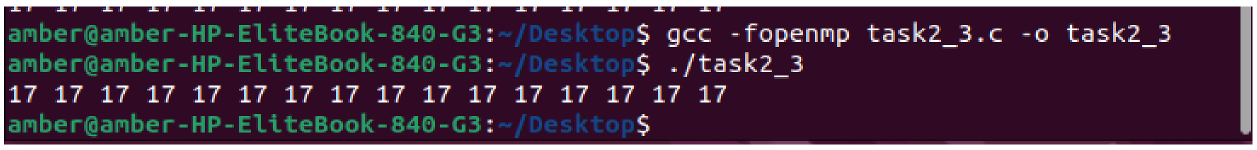
```
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch task2_2.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task2_2.c -o task2_2
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o:
 in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task2_2.c -o task2_2
amber@amber-HP-EliteBook-840-G3:~/Desktop$ ./task2_2
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

Q2: Modify the code such that this is also parallel, but only thread of id 0 is able to display the entire loop. The others should not do anything. When making it parallel, make sure its the old threads and new threads are not created. What output do you see?

Code:

```
// Include standard input/output header file for printf function
#include <stdio.h>

// Include OpenMP header file for parallel programming support
#include <omp.h>

// Main function - program execution starts here
int main() {

        // Declare and initialize the first array with 16 elements
        int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};

        // Declare and initialize the second array with 16 elements (in reverse order)
        int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

        // Declare an array to store the results of element-wise addition
        int result[16];

        // Start a parallelized for loop where iterations are divided among threads
        #pragma omp parallel for
        for (int i = 0; i < 16; i++) {
        // Add corresponding elements of array1 and array2 and store the result in result array
        result[i] = array1[i] + array2[i];
        }

        // Loop through each element in the result array and print it
        for (int i = 0; i < 16; i++) {
        printf("%d ", result[i]); // Print the current element followed by a space
        }

        // Print a newline after the entire array has been printed
        printf("\n");

        // Return 0 to indicate that the program ended successfully
        return 0;
}
```

Output:



```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task2_3.c -o task2_3
amber@amber-HP-EliteBook-840-G3:~/Desktop$ ./task2_3
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

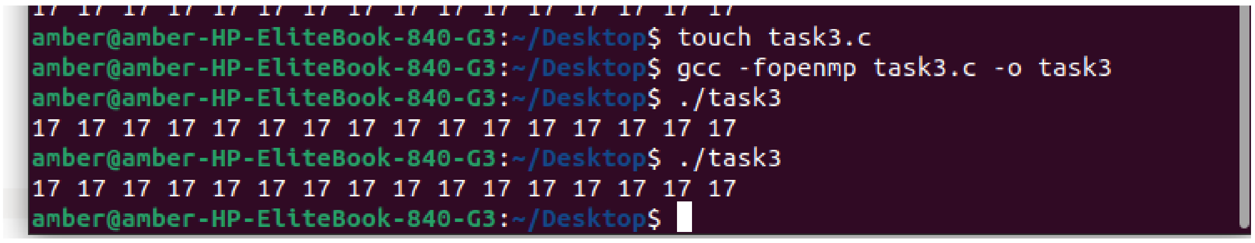
Task 3
1. Modify the code of Task 2 and do the job in half of the threads.

Code:

```
// Include standard input/output header file for printf function
#include <stdio.h>

// Include OpenMP header file for parallel programming support
#include <omp.h>

// Main function - program execution starts here
int main() {

        // Declare and initialize the first array with 16 elements
        int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};

        // Declare and initialize the second array with 16 elements (in reverse order)
        int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

        // Declare an array to store the results of element-wise addition
        int result[16];

        // Set the number of threads to 4 for parallel processing
        omp_set_num_threads(4);

        // Start a parallelized for loop where iterations are divided among threads
        #pragma omp parallel for
        for (int i = 0; i < 16; i++) {
        // Add corresponding elements of array1 and array2 and store the result in result array
        result[i] = array1[i] + array2[i];
        }

        // Start a parallel region where multiple threads can run
        #pragma omp parallel
        {
        // Only the master thread (thread 0) will execute the following block
        if (omp_get_thread_num() == 0) {
        // Loop through each element in the result array and print it
```

```
            for (int i = 0; i < 16; i++) {
                        printf("%d ", result[i]); // Print the current element followed by a space
            }
            // Print a newline after the entire array has been printed
            printf("\n");
            }
            }

            // Return 0 to indicate that the program ended successfully
            return 0;
}
```

Output:



```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch task3.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task3.c -o task3
amber@amber-HP-EliteBook-840-G3:~/Desktop$ ./task3
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
amber@amber-HP-EliteBook-840-G3:~/Desktop$ ./task3
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
amber@amber-HP-EliteBook-840-G3:~/Desktop$ 
```

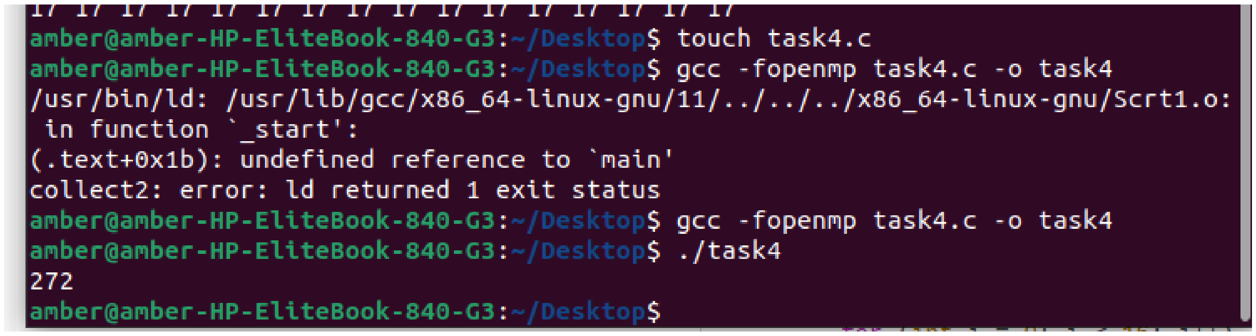
Task 4

1. The following code adds the contents of the array array1 and array2.

```
 // Include standard input/output header file for printf function
#include <stdio.h>

// Main function - program execution starts here
int main() {

            // Declare and initialize the first array with 16 elements
            int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};

            // Declare and initialize the second array with 16 elements (in reverse order)
            int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

            // Declare variables to store the results of summing elements from the arrays
            int result1 = 0, result2 = 0;

            // Loop through each element of array1 and accumulate the sum into result1
            for (int i = 0; i < 16; i++) {
```

```
        result1 += array1[i]; // Add each element of array1 to result1
        }

        // Check if result1 is greater than 10
        if (result1 > 10) {
        result2 = result1;  // If the condition is true, assign result1 to result2

        // Loop through each element of array2 and accumulate the sum into result2
        for (int i = 0; i < 16; i++) {
        result2 += array2[i]; // Add each element of array2 to result2
        }
        }

        // Print the final value of result2
        printf("%d\n", result2);

        // Return 0 to indicate that the program ended successfully
        return 0;
}
```

Output:



```
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch task4.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task4.c -o task4
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o:
 in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task4.c -o task4
amber@amber-HP-EliteBook-840-G3:~/Desktop$ ./task4
272
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

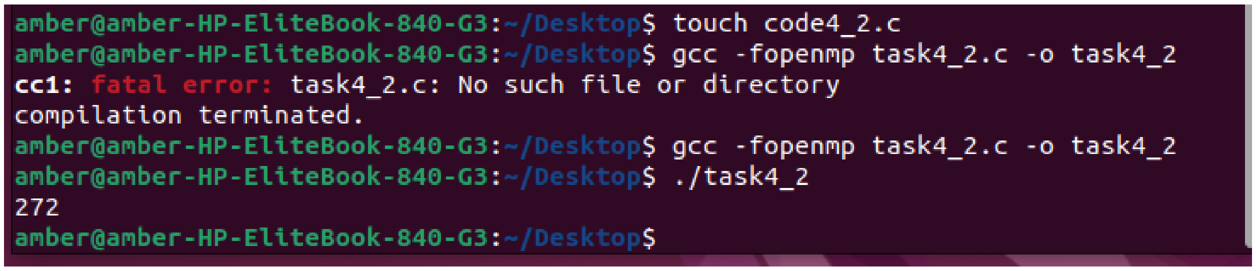
2. Convert it into Parallel using 16 threads.

Code:

```
// Include standard input/output header file for printf function
#include <stdio.h>

// Include OpenMP header file for parallel programming support
#include <omp.h>
```

```
// Main function - program execution starts here
int main() {

        // Declare and initialize the first array with 16 elements
        int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};

        // Declare and initialize the second array with 16 elements (in reverse order)
        int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

        // Declare variables to store the results of summing elements from the arrays
        int result1 = 0, result2 = 0;

        // Set the number of threads to 16 for parallel processing
        omp_set_num_threads(16);

        // Parallelize the summing process of array1 using reduction to avoid race conditions
        #pragma omp parallel for reduction(+:result1)
        for (int i = 0; i < 16; i++) {
        result1 += array1[i]; // Add each element of array1 to result1
        }

        // Check if the sum of array1 elements (result1) is greater than 10
        if (result1 > 10) {
        result2 = result1;  // If the condition is true, assign result1 to result2

        // Parallelize the summing process of array2 using reduction to avoid race conditions
        #pragma omp parallel for reduction(+:result2)
        for (int i = 0; i < 16; i++) {
        result2 += array2[i]; // Add each element of array2 to result2
        }
        }

        // Print the final value of result2
        printf("%d\n", result2);

        // Return 0 to indicate that the program ended successfully
        return 0;
}
```

Output:



```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch code4_2.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task4_2.c -o task4_2
cc1: fatal error: task4_2.c: No such file or directory
compilation terminated.
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task4_2.c -o task4_2
amber@amber-HP-EliteBook-840-G3:~/Desktop$ ./task4_2
272
amber@amber-HP-EliteBook-840-G3:~/Desktop$
```

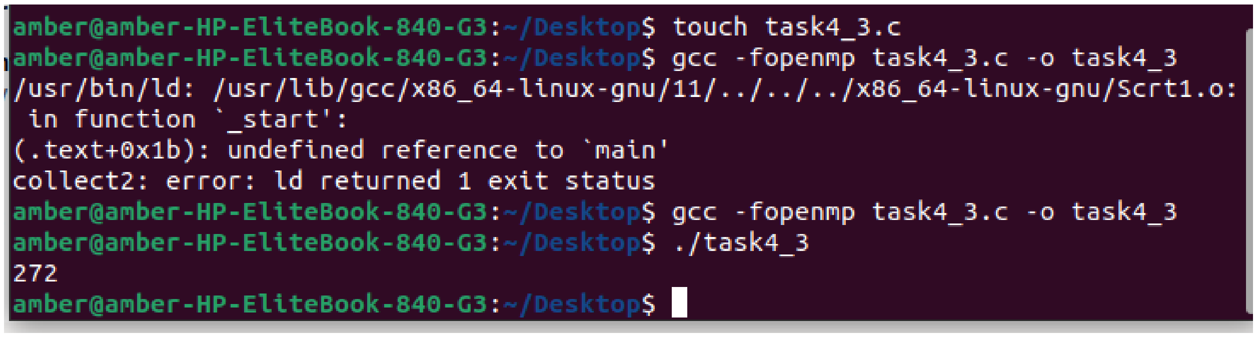
3. Try removing the reduction() clause and add #pragma omp atomic just beore the +=. What is the effect on result? Explain.

Code:

```
// Include standard input/output header file for printf function
#include <stdio.h>

// Include OpenMP header file for parallel programming support
#include <omp.h>

// Main function - program execution starts here
int main() {

        // Declare and initialize the first array with 16 elements
        int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};

        // Declare and initialize the second array with 16 elements (in reverse order)
        int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

        // Declare variables to store the results of summing elements from the arrays
        int result1 = 0, result2 = 0;

        // Set the number of threads to 16 for parallel processing
        omp_set_num_threads(16);

        // Parallelize the summing process of array1 using the atomic directive to prevent race
conditions
        #pragma omp parallel for
        for (int i = 0; i < 16; i++) {
        #pragma omp atomic
        result1 += array1[i]; // Safely add each element of array1 to result1
        }

        // Check if the sum of array1 elements (result1) is greater than 10
        if (result1 > 10) {
```

```
        result2 = result1;  // If the condition is true, assign result1 to result2

        // Parallelize the summing process of array2 using the atomic directive to prevent race
conditions
        #pragma omp parallel for
        for (int i = 0; i < 16; i++) {
        #pragma omp atomic
        result2 += array2[i]; // Safely add each element of array2 to result2
        }
        }

        // Print the final value of result2
        printf("%d\n", result2);

        // Return 0 to indicate that the program ended successfully
        return 0;
}
```

Output:



```
amber@amber-HP-EliteBook-840-G3:~/Desktop$ touch task4_3.c
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task4_3.c -o task4_3
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o:
 in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
amber@amber-HP-EliteBook-840-G3:~/Desktop$ gcc -fopenmp task4_3.c -o task4_3
amber@amber-HP-EliteBook-840-G3:~/Desktop$ ./task4_3
272
amber@amber-HP-EliteBook-840-G3:~/Desktop$ 
```

Effect on result:

Using atomic ensures thread-safe updates but is less efficient than reduction because it serializes updates, causing threads to wait for access to result1 or result2. The result remains 272 as atomic prevents race conditions, but performance may degrade due to contention.