Trapezoid Code Analysis

- **Lines 1-3**: These lines include the necessary header files. `stdio.h` is for standard input/output functions, `stdlib.h` provides utility functions (like `strtol`), and `omp.h` includes OpenMP functionalities.

```
5 void Trap(double a, double b, int n, double *global_result);
```

- **Line 5**: Declaration of the function `Trap`, which will perform the trapezoidal integration. It takes:
    - `a`: lower limit of integration.
    - `b`: upper limit of integration.
    - `n`: number of trapezoids (subintervals).
    - `global_result`: a pointer to a double where the final result will be stored.

```
7 int main(int argc, char *argv[])
```

- **Line 7**: The entry point of the program. It takes command-line arguments for configuration.

```
8 double global_result = 0.0;
```

- **Line 8**: A global variable `global_result` initialized to zero, used to accumulate the results from different threads.

```
9 double a, b;
10 int n;
11 int thread_count;
```

- **Lines 9-11**: Declaration of variables:
    - `a` and `b`: the limits of integration.
    - `n`: the number of trapezoids.
    - `thread_count`: the number of threads to be used.

```
13 thread_count = strtol(argv[1], NULL, 10);
```

- **Line 13**: Converts the first command-line argument to an integer to determine how many threads to use.

```
14 printf("Enter a, b, and n\n");
15 scanf("%lf %lf %d", &a, &b, &n);
```

- **Lines 14-15**: Prompts the user to enter the values for `a`, `b`, and `n`. The values are read using `scanf`.

```
16 #pragma omp parallel num_threads(thread_count)
17 Trap(a, b, n, &global_result);
```

- **Lines 16-17**: This block creates a parallel region. The `Trap` function is called within this region, allowing multiple threads to execute it. The number of threads is specified by `thread_count`.

```
19 printf("With n = %d trapezoids, our estimate\n", n);
20 printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
```

- **Lines 19-20**: Print the results, showing the number of trapezoids used and the estimated integral value.

```
22 return 0;
```

- **Line 22**: Return statement for the `main` function, indicating successful completion of the program.

Trap Function Implementation

```
25 void Trap(double a, double b, int n, double *global_result)
```

- **Line 25**: Definition of the `Trap` function.

```
26 double h, x, my_result;
27 double local_a, local_b;
28 int i, local_n;
```

- **Lines 26-28**: Local variable declarations:
    - `h`: width of each trapezoid.
    - `x`: a variable for evaluating the function.
    - `my_result`: to store the result for the current thread.
    - `local_a` and `local_b`: local interval limits for each thread.
    - `local_n`: number of trapezoids assigned to each thread.

```
29 int my_rank = omp_get_thread_num();
30 int thread_count = omp_get_num_threads();
```

- **Lines 29-30**: Get the thread's rank (ID) and the total number of threads.

```
32 h = (b - a) / n;
```

- **Line 32**: Calculate the width of each trapezoid.

```
33 local_n = n / thread_count;
```

- **Line 33**: Determine the number of trapezoids each thread will compute.

```
c
Copy code
34 local_a = a + my_rank * local_n * h;
35 local_b = local_a + local_n * h;
```

- **Lines 34-35**: Calculate the local interval `[local_a, local_b]` for each thread based on its rank and the number of trapezoids assigned.

```
36 my_result = (f(local_a) + f(local_b)) / 2.0;
```

- **Line 36**: Initialize the local result using the function values at the local boundaries.

```
37 for (i = 1; i <= local_n - 1; i++)
```

- **Line 37**: Loop through the inner trapezoids.

```
38 x = local_a + i * h;
39 my_result += f(x);
```

- **Lines 38-39**: Evaluate the function at each x and add it to `my_result`.

```
41 my_result = my_result * h;
```

- **Line 41**: Multiply the accumulated value by `h` to finalize the local integral result.

```
43 #pragma omp critical
44 *global_result += my_result;
```

- **Lines 43-44**: Use a critical section to safely update the global result. This ensures that only one thread can modify `global_result` at a time, preventing race conditions.

Linear Search

```
int Linear_search(int key, int A[], int n) {
 int found_index = -1; // Default to -1 if not found
#pragma omp parallel for num_threads(thread_count)
 for (int i = 0; i < n; i++) {
if (A[i] == key) {
 #pragma omp critical { found_index = i; // Update shared index safely
 }
 }
 }
 return found_index; // Return found index or -1 }
```

Reduction clause and its operators

1. Sum Reduction (+)

Code Example:

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

int main() {
    int n = 100;
    double sum = 0.0;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 1; i <= n; i++) {
        sum += i;
    }

    printf("Sum of first %d numbers: %.0f\n", n, sum);
    return 0;
}
```

Output:

```
Sum of first 100 numbers: 5050
```

2. Product Reduction (`*`)

Code Example:

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

int main() {
    int n = 10;
    long long product = 1;

    #pragma omp parallel for reduction(*:product)
    for (int i = 1; i <= n; i++) {
        product *= i;
    }

    printf("Product of first %d numbers: %lld\n", n, product);
    return 0;
}
```

Output:

```
Product of first 10 numbers: 3628800
```

3. Maximum Reduction (`max`)

Code Example:

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

int main() {
    int n = 10;
    int arr[10] = {1, 3, 5, 7, 2, 8, 6, 4, 9, 0};
    int max_value = arr[0];

    #pragma omp parallel for reduction(max:max_value)
    for (int i = 0; i < n; i++) {
        if (arr[i] > max_value) {
            max_value = arr[i];
        }
    }

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

Output:

```
Maximum value: 9
```

4. Minimum Reduction (`min`)

Code Example:

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

int main() {
    int n = 10;
    int arr[10] = {1, 3, 5, 7, 2, 8, 6, 4, 9, 0};
    int min_value = arr[0];

    #pragma omp parallel for reduction(min:min_value)
    for (int i = 0; i < n; i++) {
        if (arr[i] < min_value) {
            min_value = arr[i];
        }
    }

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

Output:

```
Minimum value: 0
```

5. Logical AND Reduction (`&&`)

Code Example:

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

int main() {
    int n = 5;
    int arr[5] = {1, 1, 1, 1, 0}; // Logical values (1 = true, 0 = false)
    int all_true = 1; // Assume all true

    #pragma omp parallel for reduction(&:all_true)
    for (int i = 0; i < n; i++) {
        all_true &= arr[i];
    }

    printf("All values are true: %s\n", all_true ? "Yes" : "No");
    return 0;
}
```

Output:

```
All values are true: No
```

6. Logical OR Reduction (||)

Code Example:

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

int main() {
    int n = 5;
    int arr[5] = {0, 0, 0, 0, 1}; // Logical values (1 = true, 0 = false)
    int any_true = 0; // Assume none are true

    #pragma omp parallel for reduction(||:any_true)
    for (int i = 0; i < n; i++) {
        any_true |= arr[i];
    }

    printf("Any values are true: %s\n", any_true ? "Yes" : "No");
    return 0;
}
```

Output:

```
Any values are true: Yes
```

Monte Carlo Method

Serial Code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    long long total_points = 1000000; // Total random points to generate
    long long points_inside_circle = 0;

    // Seed the random number generator
    srand(time(NULL));

    for (long long i = 0; i < total_points; i++) {
        double x = (double)rand() / RAND_MAX; // Generate random x coordinate
        double y = (double)rand() / RAND_MAX; // Generate random y coordinate

        // Check if the point is inside the quarter circle
        if (x * x + y * y <= 1.0) {
            points_inside_circle++;
        }
    }

    // Estimate pi
    double pi_estimate = 4.0 * points_inside_circle / total_points;
    printf("Estimated value of pi: %.10f\n", pi_estimate);

    return 0;
}
```

Monte Carlo Method
Parallel Code

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

int main() {
    long long total_points = 1000000; // Total random points to generate
    long long points_inside_circle = 0;

    // Seed the random number generator
    srand(time(NULL));

    // Parallel section
    #pragma omp parallel
    {
        long long local_points_inside_circle = 0; // Local count for each thread

        #pragma omp for
        for (long long i = 0; i < total_points; i++) {
            double x = (double)rand() / RAND_MAX; // Generate random x coordinate
            double y = (double)rand() / RAND_MAX; // Generate random y coordinate

            // Check if the point is inside the quarter circle
            if (x * x + y * y <= 1.0) {
                local_points_inside_circle++;
            }
        }

        // Critical section to update the global count
        #pragma omp atomic
        points_inside_circle += local_points_inside_circle;
    }

    // Estimate pi
    double pi_estimate = 4.0 * points_inside_circle / total_points;
    printf("Estimated value of pi: %.10f\n", pi_estimate);

    return 0;
}
```

Leibniz formula for π

$$\pi = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

This means that you can approximate π by summing a finite number of terms from this series. Each term alternates in sign and is defined by:

$$\text{term}_k = \frac{(-1)^k}{2k+1}$$

Steps to Calculate π Using This Series

1. **Choose a Number of Terms**: Decide how many terms of the series you want to sum. More terms yield a more accurate approximation of π.
2. **Calculate Each Term**: For each term, compute the value using the formula above.
3. **Sum the Terms**: Keep a running total of the terms.
4. **Multiply by 4**: The result of the sum will be multiplied by 4 to get the approximation of π.

```
#include <stdio.h>
int main() {
    long long terms = 1000000; // Number of terms to compute
    double pi = 0.0;
    for (long long k = 0; k < terms; k++) {
        pi += (double)(-1) * k / (2 * k + 1);
    }
    pi *= 4; // Multiply by 4 to get pi approximation
    printf("Approximation of pi using %lld terms: %.10f\n", terms, pi);
    return 0;
}
```

```
Approximation of pi using 1000000 terms: 3.1415916535
```

Parallel Code for Calculating Leibniz formula for Pi

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

int main() {
    long long terms = 1000000; // Number of terms to compute
    double pi = 0.0;

    // Parallel region with reduction
    #pragma omp parallel for reduction(+:pi)
    for (long long k = 0; k < terms; k++) {
        pi += (double)(-1) * k / (2 * k + 1);
    }

    pi *= 4; // Multiply by 4 to get pi approximation

    printf("Approximation of pi using %lld terms: %.10f\n", terms, pi);
    return 0;
}
```