**UNIT 4 OPEN MP PROGRAMS**

PROGRAM 1

|  |
| --- |
| %%writefile hello.c  #include <stdio.h>  #include <omp.h>  int main()  { #pragma omp parallel  { printf("Hello from thread %d\n", omp\_get\_thread\_num()); }  return 0; } |

a simple OpenMP program in C that prints a message from each thread.

(%%writefile)—just a heads-up, that's used in Jupyter notebooks to save the contents of the cell into a file.

**🧠 What it does:**

* Uses OpenMP to run a #pragma omp parallel region. The #pragma omp parallel directive is how you tell OpenMP to start a **parallel region** — it's where the program forks into multiple threads. Each thread runs the code inside the following block.
* Each thread prints its thread number with omp\_get\_thread\_num().

**🛠 If you want to compile and run it:**

You'll need to compile it with OpenMP support using GCC (or another compiler that supports OpenMP):

gcc -fopenmp hello.c -o hello

./hello

**✅ Example output (depending on number of threads):**

Hello from thread 0

Hello from thread 1

Hello from thread 2

Hello from thread 3

(The order might be jumbled because threads can run independently.)

If you want to control the number of threads:

export OMP\_NUM\_THREADS=4

./hello

PROGRAM 2

|  |
| --- |
| %%writefile hello.c  #include <stdio.h>  #include <omp.h>  int main() { omp\_set\_num\_threads(4); // Set number of threads to 4  #pragma omp parallel  {  printf("Hello from thread %d\n", omp\_get\_thread\_num());  }  return 0; } |

You've written a clean OpenMP program that:

* Sets the number of threads to 4 explicitly using omp\_set\_num\_threads(4)
* Spawns 4 threads to print their thread ID with omp\_get\_thread\_num()

**🔧 To compile and run:**

If you're in a terminal, do:

gcc -fopenmp hello.c -o hello

./hello

**🔄 Example Output (may vary in order due to concurrency):**

bash

Hello from thread 2

Hello from thread 0

Hello from thread 1

Hello from thread 3

The order isn’t guaranteed because threads may execute out of sequence.

Top of Form

COMMMAND 1

!lscpu | grep '^CPU(s):'

This command (when run in a Jupyter notebook or shell with !) fetches the number of logical CPUs (threads) available on your system.

🔍 Example Output:

CPU(s): 8

This means your system has 8 logical CPUs (could be 4 physical cores with hyperthreading).

To get **physical cores only**:

bash

lscpu | grep 'Core(s) per socket'

To check OpenMP default threads from within a C program:

printf("Default max threads: %d\n", omp\_get\_max\_threads());

COMMAND 2

os.cpu\_count()

Returns the number of **logical processors** (same as lscpu | grep '^CPU(s):').

So if you see 8, that could mean:

* 4 physical cores with hyperthreading (2 threads per core), or
* 8 physical cores without hyperthreading

PROGRAM 3

|  |
| --- |
| %%writefile hello\_openmp.c  #include <stdio.h>  #include <omp.h>  int main()  {  #pragma omp parallel { int ID = omp\_get\_thread\_num(); //Assigned once per thread printf("hello(%d)", ID); // Uses the thread's ID  printf("world(%d) \n", ID); //Uses the same ID  }  return 0; } |

Each thread:

* Gets its own thread ID using omp\_get\_thread\_num()
* Prints hello(ID) and world(ID) on the same line

**🖨️ Example output (order may vary):**

hello(3)world(3)

hello(2)world(2)

hello(1)world(1)

Note: Output order may be jumbled due to how threads are scheduled. If you want to print lines cleanly (e.g. one line per thread), it’s already handled nicely here because each thread finishes both printfs before releasing the CPU.

**🧼 Optional cleanup:**

If you want the output to look tidier, you could combine the two printf calls:

printf("hello(%d) world(%d)\n", ID, ID);

PROGRAM 4

|  |
| --- |
| %%writefile shared\_var.c  #include <stdio.h>  #include <omp.h>  int main() {      int sharedVar = 0;      #pragma omp parallel shared(sharedVar)      {          #pragma omp critical          sharedVar += 1;      }      printf("Final value of sharedVar: %d\n", sharedVar);      return 0;} |

sharedVar is shared among all threads.

The #pragma omp critical ensures only **one thread at a time** updates sharedVar.

So if you run with 4 threads, sharedVar should end up being 4.

**Example output:**

Final value of sharedVar: 4

PROGRAM 5

|  |
| --- |
| %%writefile shared\_var.c  #include <stdio.h>  #include <omp.h>  int main() {  omp\_set\_num\_threads(100);  int sharedVar = 0;  #pragma omp parallel shared(sharedVar)  {  #pragma omp critical  sharedVar += 1;  }  printf("Final value of sharedVar: %d\n", sharedVar);  return 0;  }  Bottom of Form |

This program tests your shared variable with a much larger number of threads — omp\_set\_num\_threads(100) — which is awesome for seeing how #pragma omp critical protects against race conditions even under high contention.

Note:

omp\_set\_num\_threads(100);

Requests 100 threads for the parallel region.

Each thread increments sharedVar once, but the critical section ensures they do so one at a time.

**Expected output:**

Final value of sharedVar: 100

✅ You should always get **100** as the result (as long as your system can support that many threads)

PROGRAM 6

|  |
| --- |
| %%writefile matrix\_addition.c  #include <stdio.h>  #include <omp.h>  #define N 3  int main() {      int A[N][N] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};      int B[N][N] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};      int C[N][N];      #pragma omp parallel for collapse(2)      for (int i = 0; i < N; i++) {          for (int j = 0; j < N; j++) {              C[i][j] = A[i][j] + B[i][j];          }      }      printf("Resultant Matrix:\n");      for (int i = 0; i < N; i++) {          for (int j = 0; j < N; j++) {              printf("%d ", C[i][j]);          }          printf("\n");      }      return 0;  } |

#pragma omp parallel for creates a team of threads to perform the loop in parallel.

collapse(2) flattens the two loops into a single loop of N\*N iterations, distributing the entire 2D workload across threads.

This is ideal when both loops are small (as in your 3x3 example).

**Output:**

Resultant Matrix:

10 10 10

10 10 10

10 10 10

**⚙️ Compile & Run:**

bash

CopyEdit

gcc -fopenmp matrix\_addition.c -o matrix\_add

./matrix\_add

* You can print which thread handled which element if you want to visualize parallelism:

printf("C[%d][%d] = %d (Thread %d)\n", i, j, C[i][j], omp\_get\_thread\_num());

PROGRAM 7

|  |
| --- |
| %%writefile matrix\_add\_timed.c  #include <stdio.h>  #include <omp.h>  #define N 3  int main() {  int A[N][N] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  int B[N][N] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};  int C[N][N];  double start\_time = omp\_get\_wtime(); // Start timer  // Matrix addition (currently serial)  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  C[i][j] = A[i][j] + B[i][j];  }  }  printf("Resultant Matrix:\n");  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  printf("%d ", C[i][j]);  }  printf("\n");  }  double end\_time = omp\_get\_wtime(); // End timer  printf("Execution Time = %f seconds\n", end\_time - start\_time);  return 0;  } |

* Adds two 3×3 matrices **serially**
* Prints the result
* Measures execution time in **seconds** using OpenMP’s high-resolution wall clock

**🧾 Sample Output:**

Resultant Matrix:

10 10 10

10 10 10

10 10 10

Execution Time = 0.000001 seconds

PROGRAM 8

|  |
| --- |
| %%writefile parallel\_sum.c  #include <stdio.h>  #include <omp.h>  #define SIZE 1000000  int main() {  int i;  double sum = 0.0;  double array[SIZE];  // Initialize array with values  for (i = 0; i < SIZE; i++) {  array[i] = 1.0; // Simple case where sum = SIZE  }  double start\_time = omp\_get\_wtime(); // Start timer  #pragma omp parallel for reduction(+:sum)  for (i = 0; i < SIZE; i++) {  sum += array[i];  }  double end\_time = omp\_get\_wtime(); // End timer  printf("Sum = %.2f\n", sum);  printf("Execution Time = %f seconds\n", end\_time - start\_time);  return 0;  } |

This is a clean and efficient OpenMP program to **compute the sum of a large array in parallel**, with proper timing. This is exactly the kind of task where OpenMP shines. 🚀

**🔍 Quick Recap of What You're Doing:**

1. **Initialize** a big array (SIZE = 1,000,000) with 1.0 — so expected sum = 1000000.00.
2. Use OpenMP with:

#pragma omp parallel for reduction(+:sum)

* + Each thread computes part of the sum
  + reduction(+:sum) avoids race conditions by giving each thread a private copy of sum, then combining them safely at the end.

1. Time the execution using omp\_get\_wtime().

**✅ Sample Output:**

Sum = 1000000.00

Execution Time = 0.012345 seconds

(Your exact execution time will vary depending on your system and number of threads.)

**💡 Pro Tips:**

* Want to know how many threads were used? Add inside the parallel region:

#pragma omp parallel

{

#pragma omp single

printf("Using %d threads\n", omp\_get\_num\_threads());

}

PROGRAM 9

|  |
| --- |
| %%writefile shared\_var\_critical.c  #include <stdio.h>  #include <omp.h>  int main() {  int sharedVar = 0;  // Optional: Set number of threads  omp\_set\_num\_threads(8); // You can change this value  #pragma omp parallel shared(sharedVar)  {  #pragma omp critical  sharedVar += 1;  }  printf("Final value of sharedVar: %d\n", sharedVar);  return 0;  } |

This is another clean and correct OpenMP example using a **shared variable with a critical section** to ensure safe updates in parallel.

**✅ What This Program Does:**

* **Initializes** sharedVar to 0
* **Sets** number of threads to 8 (or any other number you choose)
* Each thread enters a #pragma omp critical section and increments sharedVar once
* Finally, the program prints the value of sharedVar

**🧾 Expected Output:**

If all threads run as expected:

Final value of sharedVar: 8

Because each of the 8 threads adds 1 to sharedVar — one at a time, safely.

**🧠 Why #pragma omp critical Is Important:**

Without the critical section, you’d have a **race condition**, where multiple threads might read/update/write the variable at the same time, causing incorrect results.

PROGRAM 10

|  |
| --- |
| %%writefile omp\_single\_example.c  #include <stdio.h>  #include <omp.h>  void add(int a, int b, int \*result) {      #pragma omp parallel      {          #pragma omp single          {          \*result = a + b;          }  }  }  int main() {      int x = 5, y = 10, sum = 0;      add(x, y, &sum);      printf("Sum: %d\n", sum);      return 0;  } |

The code provided is a simple example of using OpenMP with the single directive, which ensures that a block of code is executed by only one thread.

**Explanation:**

1. The add function takes two integers a and b, and a pointer result where the sum will be stored.
2. The #pragma omp parallel directive creates a parallel region, meaning multiple threads will be available for execution.
3. Inside the parallel region, #pragma omp single ensures that only one thread executes the code inside the block (\*result = a + b;), which avoids potential race conditions by preventing multiple threads from writing to result simultaneously.
4. In the main function, two integers (x = 5 and y = 10) are passed to the add function, and the sum is printed.

When you run this program, you will get the output:

Sum: 15

This program will use OpenMP for parallel execution, but only a single thread will be responsible for performing the addition due to the single directive.

PROGRAM 11

|  |
| --- |
| %%writefile modified.c  #include <stdio.h>  #include <omp.h>  void modify(int a) {  #pragma omp parallel  {  // Each thread will print its own copy of 'a'  printf("Thread %d: Original a = %d\n", omp\_get\_thread\_num(), a);  // Modify 'a' inside the thread  a += 10;  // Show modified value (thread-local)  printf("Thread %d: Modified a = %d\n", omp\_get\_thread\_num(), a);  }  }  int main() {  int x = 5;  modify(x);  printf("Back in main: x = %d\n", x); // Still 5  return 0;  } |

This program demonstrates how OpenMP works with shared and private variables.

**Explanation of Behavior:**

1. **Function modify:**
   * The function modify accepts an integer a. It prints the value of a before and after modification.
   * Inside the #pragma omp parallel directive, multiple threads will be spawned. Each thread gets its own local copy of a (this is the default behavior for arguments passed by value in OpenMP, making a private to each thread).
   * Each thread modifies its own local copy of a, adding 10 to it, and prints the modified value.
2. **Thread Outputs:**
   * The omp\_get\_thread\_num() function is used to print the thread number, allowing you to observe how each thread works independently on its own copy of a.
   * Since a is private to each thread, the main thread and other threads will not interfere with each other. Thus, each thread's modification will be isolated.
3. **Main Function Behavior:**
   * After the parallel region completes, the value of x in the main function remains unchanged because x is passed by value to the modify function. The modifications made inside the parallel region are done to local copies of a within the threads.
   * The final print statement in the main function shows that x is still 5, which is the original value passed.

**Output:**

You would see something like the following when you run the program (the thread numbers may vary depending on the system):

Thread 0: Original a = 5

Thread 0: Modified a = 15

Thread 1: Original a = 5

Thread 1: Modified a = 15

Thread 2: Original a = 5

Thread 2: Modified a = 15

Thread 3: Original a = 5

Thread 3: Modified a = 15

Back in main: x = 5

**Key Points:**

* The value of x in the main function does not change because it was passed by value to the modify function.
* Each thread operates on its own local copy of a, and the changes are local to the threads. Therefore, the modification does not affect the original value of x.
* OpenMP ensures that each thread has its own private copy of the variables passed by value, which prevents race conditions in this context.

PROGRAM 12

|  |
| --- |
| %%writefile newfile.c  #include <stdio.h>  #include <omp.h>  void add(int a, int b, int \*result) {  #pragma omp parallel  {  printf("Thread %d: Inside parallel region\n", omp\_get\_thread\_num());  // All threads execute this  \*result = a + b;  a=a+10;  b=b+10;  printf("Thread %d: Calculated result = %d\n", omp\_get\_thread\_num(), \*result);  }  }  int main() {  int x = 5, y = 10, sum = 0;  add(x, y, &sum);  printf("Back in main: sum = %d\n", sum);  return 0;  } |

The code provided contains an OpenMP parallel region, but there are some important aspects of how OpenMP works that could lead to potential issues, especially with shared and private variables. In this case, the variable sum (result) is shared among threads, but the way a and b are modified inside the parallel block won't have any effect on the final result as they are local to each thread. Here's an explanation of potential issues and a correction:

**Key Issues:**

1. **Shared/Private Variables**:
   * The variable sum is shared across threads, so the threads might overwrite each other's values, leading to incorrect results.
   * Variables a and b should either be marked as private if you don't want them to affect other threads' calculations, or you should handle the accumulation of sum in a thread-safe way (using critical sections or reduction).
2. **Thread Synchronization**:
   * The way you're setting \*result in all threads could lead to race conditions. If two threads simultaneously modify the result, it can cause unpredictable behavior.

**Suggestions for Modification:**

1. Make a and b private to each thread.
2. Accumulate the results in a thread-safe manner (for example, using reduction or critical sections).
3. Consider handling how you accumulate sum safely in a parallel region.

Here's the corrected version of the code:

#include <stdio.h>

#include <omp.h>

void add(int a, int b, int \*result) {

#pragma omp parallel

{

int thread\_sum = 0;

int tid = omp\_get\_thread\_num();

printf("Thread %d: Inside parallel region\n", tid);

// Private variables for each thread

thread\_sum = a + b;

a = a + 10; // Changes will not affect other threads

b = b + 10;

printf("Thread %d: Calculated result = %d\n", tid, thread\_sum);

// Critical section to safely accumulate result

#pragma omp atomic

\*result += thread\_sum;

}

}

int main() {

int x = 5, y = 10, sum = 0;

add(x, y, &sum);

printf("Back in main: sum = %d\n", sum);

return 0;

}

**Explanation of Changes:**

* **Private a and b**: Each thread works with its own local copies of a and b, so they don't interfere with each other.
* **Thread-Safe Accumulation**: #pragma omp atomic ensures that only one thread can modify \*result at a time, avoiding race conditions.
* **thread\_sum**: Each thread computes its own sum independently of others.

Now the program will compute the sum correctly, and the output should be more predictable.

Top of Form

Bottom of Form