Name: Shreeyash Shripad Dongarkar

PRN: 22510025

High Performance Computing Lab (B – 1)

Assignment – 4

Title: Study and Implementation of Synchronization

[GitHub Repository](https://github.com/TerminatorShri/22510025_HPCL)

Analyse and Implement Parallel Codes for below Programs using OpenMP considering Synchronization Requirements. Demonstrate use of different Clauses and Constructs wherever applicable.

PS 1 – Fibonacci Sequence

The Fibonacci Sequence is:

**Parallelization considerations**

* **Naïve recursive Fibonacci** (fib(n-1) + fib(n-2)) has **overlapping subproblems** → bad for performance unless memoized.
* **Iterative Fibonacci** is sequential by nature i.e. each term depends on the previous one → can't fully parallelize in the straightforward way.
* **We can parallelize**:
  + **Multiple independent Fibonacci computations** (e.g., computing fib(k) for many k values).
  + Recursive calls in naïve Fibonacci using **tasks**.
* **Synchronization needed**:
  + Avoid race conditions when summing results.
  + Use critical or reduction clauses.

A) Parallel Fibonacci using OpenMP Tasks

1. #include <stdio.h>

2. #include <omp.h>

3.

4. long fib(int n) {

5.     if (n < 2) return n;

6.

7.     long x, y;

8.

9.     #pragma omp task shared(x)

10.     x = fib(n - 1);

11.

12.     #pragma omp task shared(y)

13.     y = fib(n - 2);

14.

15.     #pragma omp taskwait

16.     return x + y;

17. }

18.

19. int main() {

20.     int n = 10;

21.     long result;

22.

23.     double start = omp\_get\_wtime();

24.

25.     #pragma omp parallel

26.     {

27.         #pragma omp single

28.         result = fib(n);

29.     }

30.

31.     double end = omp\_get\_wtime();

32.

33.     printf("Fibonacci(%d) = %ld\n", n, result);

34.     printf("Time: %f seconds\n", end - start);

35.

36.     return 0;

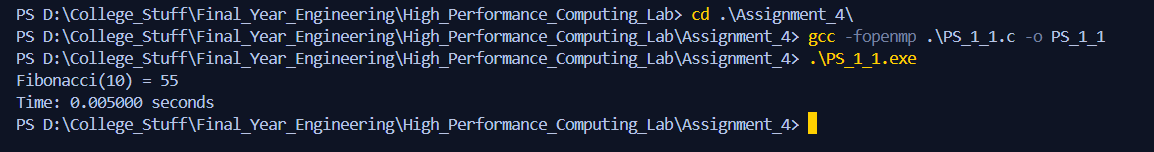
37. }

**Parallel region**:

* #pragma omp parallel starts a team of threads.
* #pragma omp single ensures that **only one thread** starts the top-level call to fib(n).  
  From there, fib will spawn parallel tasks for the other threads to work on.

**Recursive call:**

* **Base case:** Stops recursion when n < 2.
* **Tasks:**
  + #pragma omp task tells OpenMP to create a new task that runs in parallel.
  + shared(x) means all threads can see the same x variable.
* **Synchronization:**
  + #pragma omp taskwait ensures both x and y are computed before adding.



**Pros:**

* Uses multiple threads for independent subproblems.
* Easy to parallelize because Fibonacci recursion is naturally a tree.

**Cons:**

* Fibonacci is a terrible benchmark for real work — lots of repeated calculations.
* Too many small tasks can cause overhead greater than the benefit.
* For small n, the parallel version may be slower than serial.

B) Parallelizing Multiple Fibonacci Computations

1. #include <stdio.h>

2. #include <omp.h>

3.

4. long fib\_iter(int n) {

5. if (n < 2) return n;

6. long a = 0, b = 1, c;

7. for (int i = 2; i <= n; i++) {

8. c = a + b;

9. a = b;

10. b = c;

11. }

12. return b;

13. }

14.

15. int main() {

16. int nums[] = {10, 20, 25, 30};

17. int size = 4;

18. long results[size];

19.

20. double start = omp\_get\_wtime();

21.

22. #pragma omp parallel for schedule(dynamic) shared(results)

23. for (int i = 0; i < size; i++) {

24. results[i] = fib\_iter(nums[i]);

25. #pragma omp critical

26. printf("Thread %d computed fib(%d) = %ld\n",

27. omp\_get\_thread\_num(), nums[i], results[i]);

28. }

29.

30. double end = omp\_get\_wtime();

31.

32. printf("\nTotal time: %f seconds\n", end - start);

33. return 0;

34. }

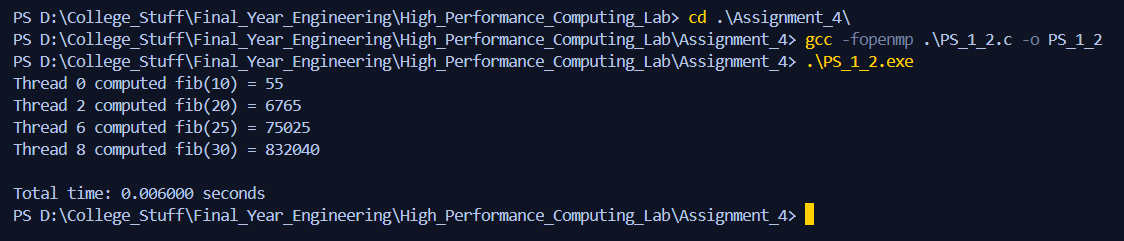
#pragma omp parallel for : Creates a pool of threads and splits the for loop iterations among them.

schedule(dynamic) : Assigns chunks of iterations to threads **on demand**.  
Useful if iterations take **different amounts of time** — faster threads grab more work after finishing their current task.

shared(results) : All threads write to the same results array (at different indexes, so no data race here).

#pragma omp critical : Ensures that the printf statements don’t interleave and garble the output. Printing is relatively slow, so you don’t want multiple threads writing at once.

omp\_get\_thread\_num() : Returns the **ID** of the current thread (0, 1, 2, ...).



**Pros**

* **Parallelism with OpenMP –** Multiple Parallel Computations, potentially speeding up the execution multiple inputs. schedule(dynamic) is a reasonable choice here since different Fibonacci numbers have different execution times (though for iterative it’s not huge).
* **Thread-safe output –** critical clause avoids mixed output when threads print result.

**Cons**

* **Overhead of parallelism for small workloads -** Since fib\_iter() is **O(n)** and nums[] array is tiny ({10, 20, 25, 30}), the **thread creation + scheduling overhead** will outweigh parallelism benefits. For such small tasks, we might even get slower performance compared to single-thread execution.
* **Unnecessary dynamic scheduling -** For equal complexity tasks, schedule(static) would have less scheduling overhead. dynamic is more useful when task durations vary significantly (e.g., recursive Fibonacci or variable workloads).
* **Critical section bottleneck -** The #pragma omp critical section forces threads to serialize when printing. This can hurt scalability if there are many tasks and a lot of output.

PS 2 – Producer Consumer Problem

The **Producer-Consumer** problem involves:

* **Producer** → generates data items and places them in a buffer.
* **Consumer** → retrieves data from the buffer.
* **Synchronization** → ensures:
  + Producer doesn't add when buffer is full.
  + Consumer doesn't remove when buffer is empty.
  + No race conditions (both trying to modify buffer at once).

1. #include <stdio.h>

2. #include <omp.h>

3. #include <unistd.h>

4.

5. #define BUFFER\_SIZE 5

6. #define NUM\_ITEMS 20

7.

8. int buffer[BUFFER\_SIZE];

9. int count = 0;

10. int in = 0;

11. int out = 0;

12.

13. void produce\_item(int item) {

15. usleep(100000);

16. printf("Produced: %d\n", item);

17. }

18.

19. void consume\_item(int item) {

21. usleep(150000);

22. printf("Consumed: %d\n", item);

23. }

24.

25. int main() {

26. int i;

27.

28. omp\_set\_num\_threads(2);

29.

30. #pragma omp parallel shared(buffer, count, in, out)

31. {

32. int thread\_id = omp\_get\_thread\_num();

33.

34. if (thread\_id == 0) {

35. for (i = 1; i <= NUM\_ITEMS; i++) {

36. int produced = 0;

37. while (!produced) {

38. #pragma omp critical

39. {

40. if (count < BUFFER\_SIZE) {

41. buffer[in] = i;

42. in = (in + 1) % BUFFER\_SIZE;

43. count++;

44. produce\_item(i);

45. produced = 1;

46. }

47. }

48. }

49. }

50. }

51. else if (thread\_id == 1) {

52. for (i = 1; i <= NUM\_ITEMS; i++) {

53. int consumed = 0;

54. while (!consumed) {

55. #pragma omp critical

56. {

57. if (count > 0) {

58. int item = buffer[out];

59. out = (out + 1) % BUFFER\_SIZE;

60. count--;

61. consume\_item(item);

62. consumed = 1;

63. }

64. }

65. }

66. }

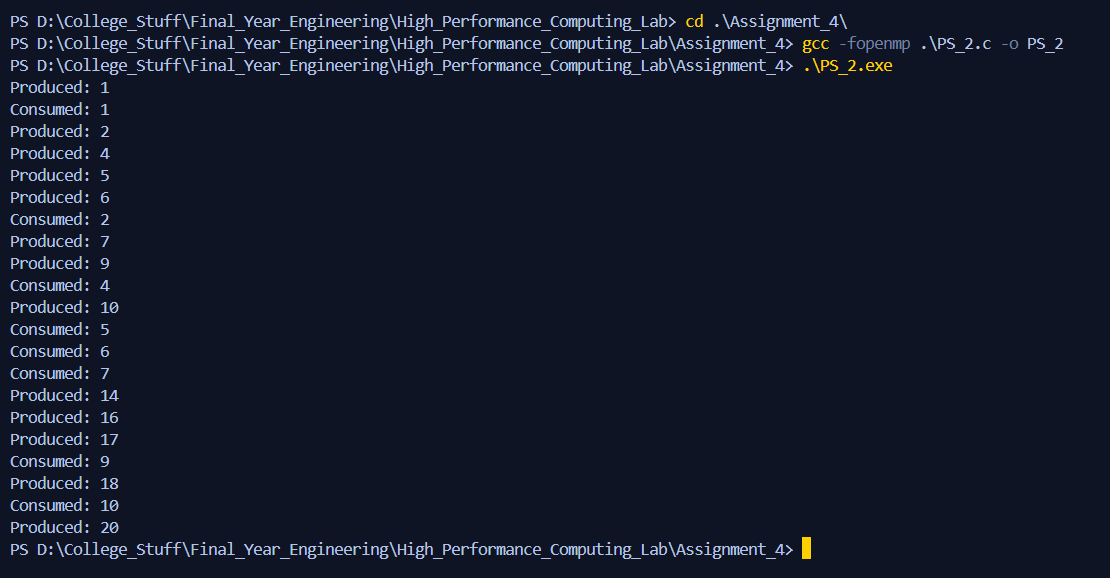
67. }

68. }

69.

70. return 0;

71. }



**#pragma omp critical** ensures that only **one thread** can modify count, in, out, or the buffer at a time.