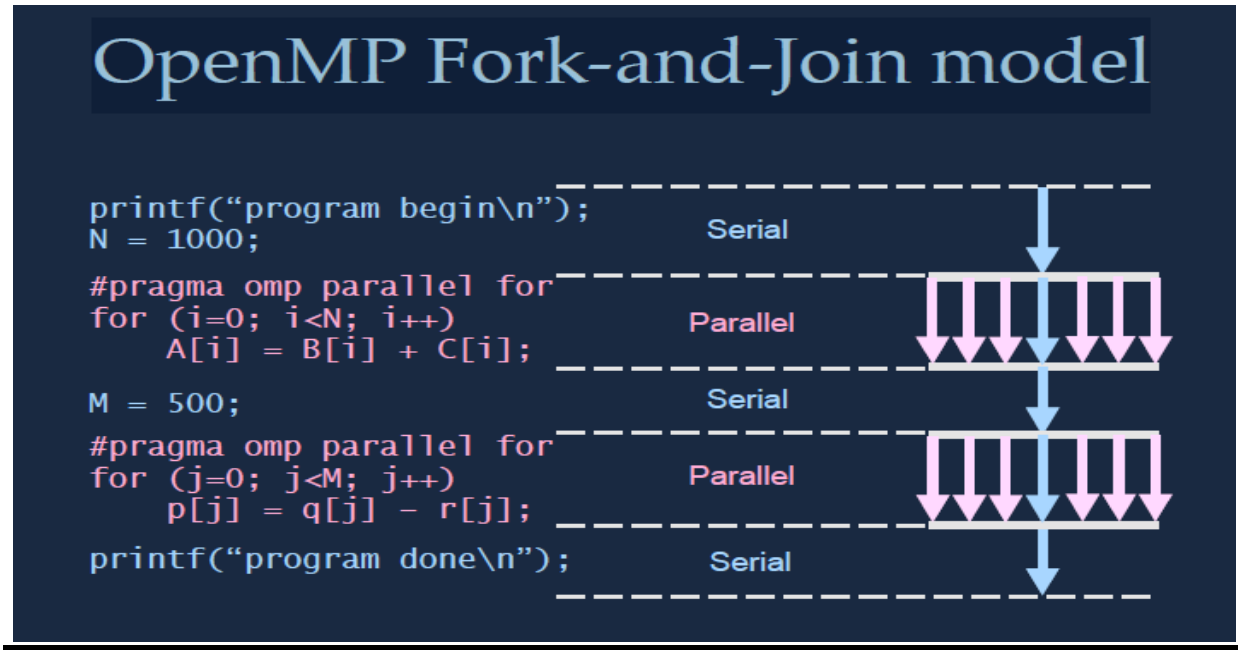
LAB 02: SHARED MEMORY PARALLEL PROGRAMMING (SMPP)

CS 3006 - Parallel and Distributed Computing

NATIONAL UNIVERSITY - FAST (FALL 2023)

Lab#2: Shared Memory Parallel Programming

OpenMP Fork Join Model:



Parallel Region and Synchronization:

Creating additional threads to execute the current construct

#pragma omp parallel

In order to correspond to the join part of the fork join paradigm, all threads will wait for each other before passing the pragma

#pragma omp barrier

The Omp Single Clause:

#pragma omp single is an OpenMP directive that is used to define a code segment that should be executed only once by any one thread from the team.

Example#1: Printing messages using synchronization clause

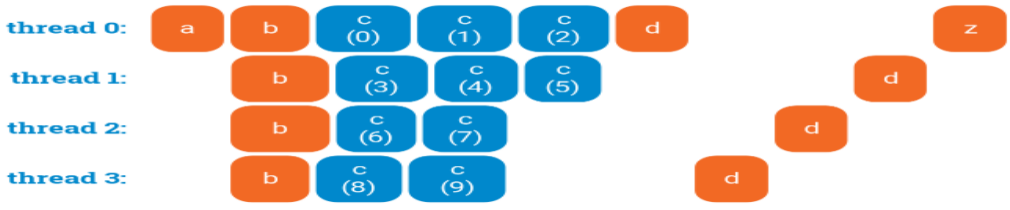
```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

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

```
{
	omp_set_num_threads(4);

	#pragma omp parallel
	{
		// Threads print their first message
		printf("[Thread %d] I print my first message.\n", omp_get_thread_num());

		// Make sure all threads have printed their first message before moving on.
		#pragma omp barrier

		// One thread indicates that the barrier is complete.
		#pragma omp single
		{
			printf("The barrier is complete, which means all threads have printed their
first message.\n");
		}

		// Threads print their second message
		printf("[Thread %d] I print my second message.\n", omp_get_thread_num());
	}

	return EXIT_SUCCESS;
}
```

Race Conditions - Interaction with critical sections:

If you need a critical section after a loop, note that normally OpenMP will first wait for all threads to finish their loop iterations before letting any of the threads to enter a critical section:

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    #pragma omp critical
    {
        d();
    }
}
z();
```



Example#2: Sum of an array elements using critical

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

#define ARRAY_SIZE 1000

int main() {
   int array[ARRAY_SIZE];
   int sum = 0;

   // Initialize the array
   for (int i = 0; i < ARRAY_SIZE; i++) {
      array[i] = i + 1;
   }

   // Parallel region
   #pragma omp parallel shared(sum)
   {
      int local_sum = 0;

      // Calculate the local sum for each thread
      #pragma omp for
      for (int i = 0; i < ARRAY_SIZE; i++) {
         local_sum += array[i];
      }

      // Use a critical section to update the global sum
      #pragma omp critical
      {
         sum += local_sum;
      }
   } // End of parallel region

   printf("Sum of array elements: %d\n", sum);

   return 0;
}
```

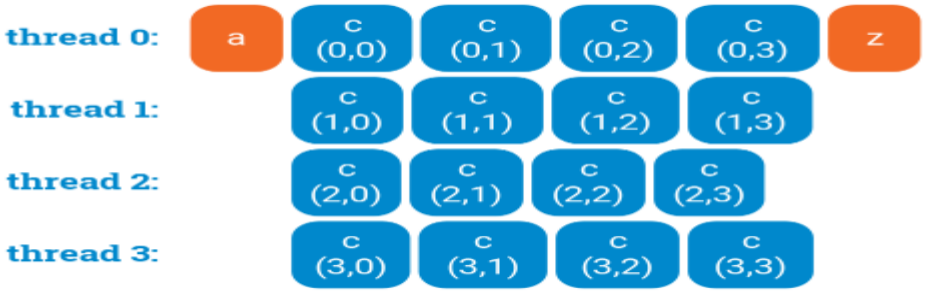
Example#3: Generating Prime numbers using Critical Clause

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

int is_prime(int num) {
   if (num <= 1) return 0; // 0 and 1 are not prime numbers
   if (num == 2) return 1; // 2 is a prime number

   if (num % 2 == 0) return 0; // Even numbers (except 2) are not prime

   for (int i = 3; i * i <= num; i += 2) {
      if (num % i == 0) {
         return 0;
      }
   }

   return 1;
}

int main() {
   int start = 2;
   int end = 100;

   #pragma omp parallel for
   for (int num = start; num <= end; num++) {
      if (is_prime(num)) {
         #pragma omp critical
         {
            printf("%d is prime.\n", num);
         }
      }
   }

   return 0;
}
```

Parallelizing Nested for Loops:

If we have nested for loops, it is often enough to simply **parallelize the outermost loop**:

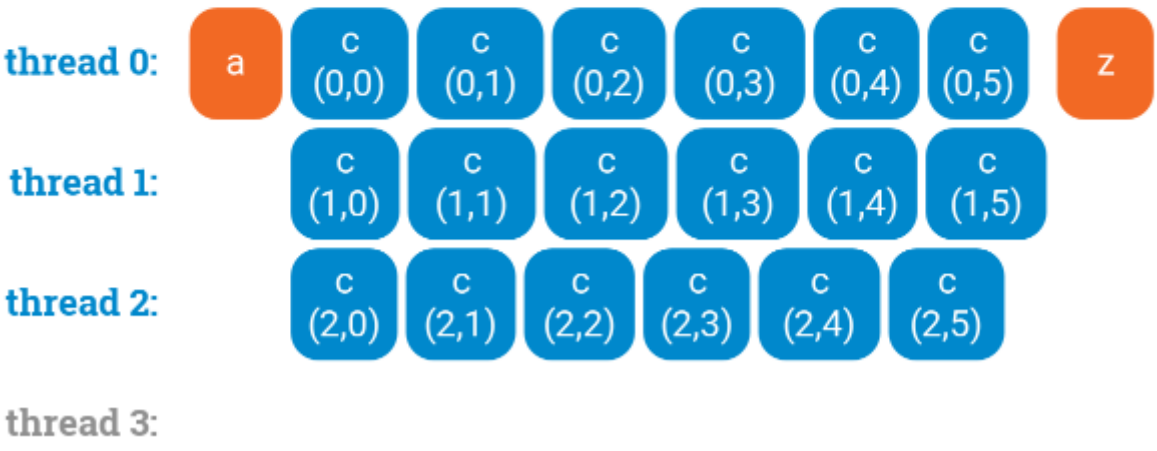
```
a();
#pragma omp parallel for
for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
        c(i, j);
    }
}
z();
```



This is all that we need most of the time. You can safely stop reading this part now; in what follows we will just discuss what to do in some rare corner cases.

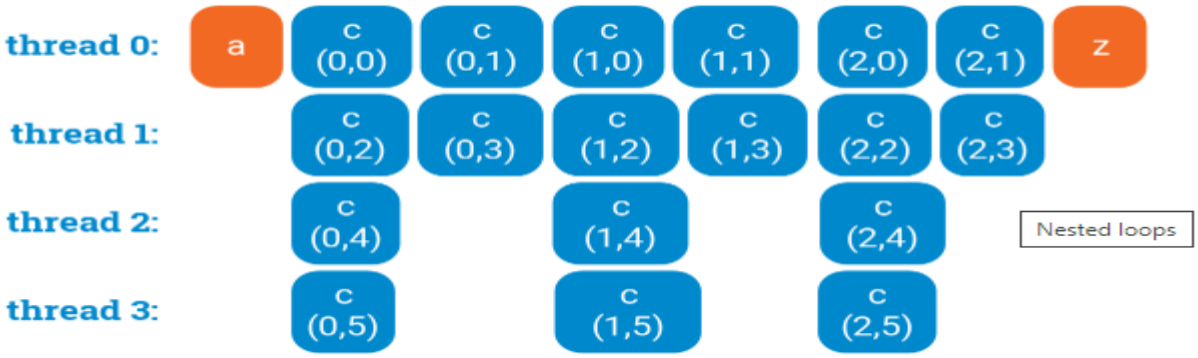
Challenges: Sometimes the outermost loop is so short that not all threads are utilized:

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```



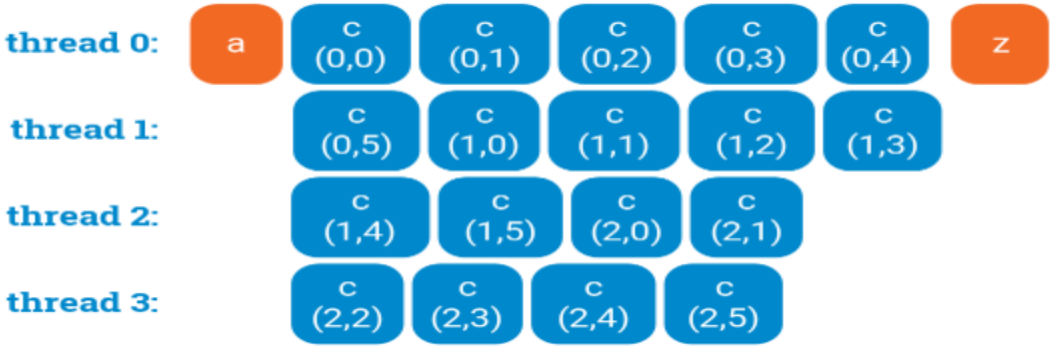
We could try to parallelize the *inner* loop. However, then we will have more overhead in the inner loop, which is more performance-critical, and there is no guarantee that the thread utilization is any better:

```
a();
for (int i = 0; i < 3; ++i) {
    #pragma omp parallel for
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```



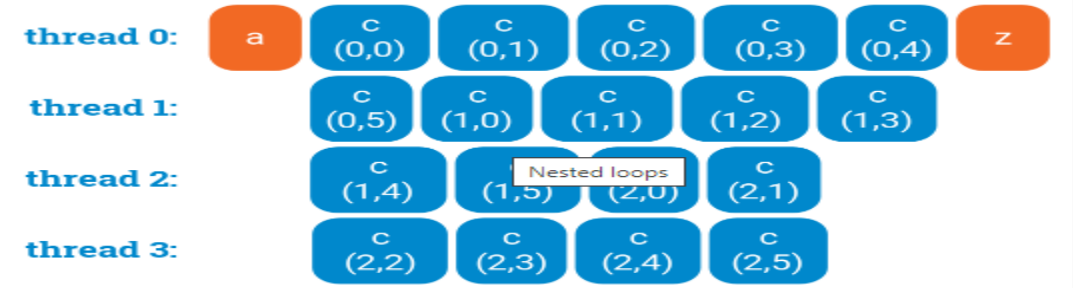
Good Ways to Do it: In essence, we have got here 3 × 6 = 18 units of work, and we would like to spread it evenly among the threads. The correct solution is to **collapse it into one loop** that does 18 iterations. We can do it manually:

```
a();
#pragma omp parallel for
for (int ij = 0; ij < 3 * 6; ++ij) {
    c(ij / 6, ij % 6);
}
z();
```



Or we can ask OpenMP to do it for us:

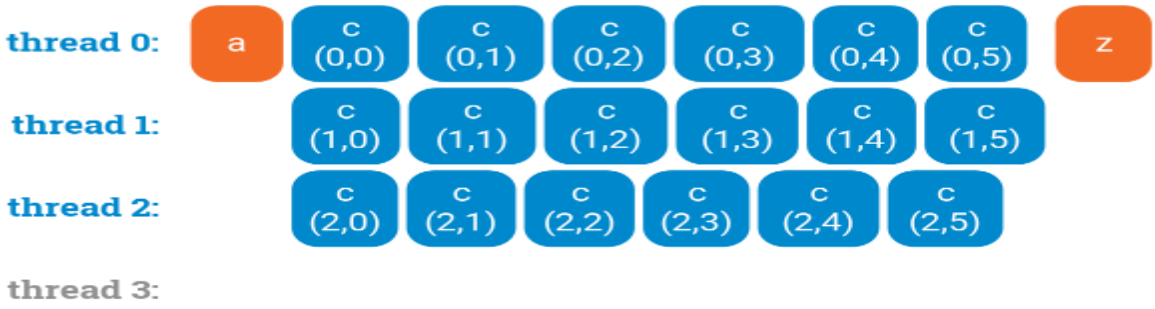
```
a();
#pragma omp parallel for collapse(2)
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```



Wrong Ways to Do it (part 1): Unfortunately, one often sees failed attempts of parallelizing nested for loops. This is perhaps the most common version:

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
    #pragma omp parallel for
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```

This code **does not do anything meaningful**. "Nested parallelism" is disabled in OpenMP by default, and the *second pragma is ignored at runtime*: a thread enters the inner parallel region, a team of only one thread is created, and each inner loop is processed by a team of one thread. The end result will look, in essence, identical to what we would get without the second pragma — but there is just more overhead in the inner loop:



On the other hand, if we tried to enable "nested parallelism", things would get much worse. The inner parallel region would create more threads, and overall we would have *3 × 4 = 12 threads* competing for the resources of 4 CPU cores — not what we want in a performance-critical application.

Wrong Ways to Do it (part 2): One also occasionally sees attempts of using multiple nested `omp for` directives inside one `parallel` region. This is **seriously broken**; OpenMP specification does not define what this would mean but simply forbids it:

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
    #pragma omp for
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```

In the system that we have been using here as an example, the above code thankfully gives a **compilation error**. However, if we manage to trick the compiler to compile this, e.g. by hiding the second `omp for` directives inside another function, it turns out that the program **freezes** when we try to run it.

Collapse Clause:

In the following example, the **k** and **j** loops are associated with the loop construct. So the iterations of the **k** and **j** loops are collapsed into one loop with a larger iteration space, and that loop is then divided among the threads in the current team. Since the **i** loop is not associated with the loop construct, it is not collapsed, and the **i** loop is executed sequentially in its entirety in every iteration of the collapsed **k** and **j** loop. The variable **j** can be omitted from the **private** clause when the **collapse** clause is used since it is implicitly private. However, if the **collapse** clause is omitted then **j** will be shared if it is omitted from the **private** clause. In either case, **k** is implicitly private and could be omitted from the **private** clause.

```
#pragma omp for collapse(2) private(i, k, j)
 for (k=kl; k<=ku; k+=ks)
        for (j=jl; j<=ju; j+=js)
                for (i=il; i<=iu; i+=is)
                        bar(a,i,j,k);
```

***Task#1**: Write a program in openMP C to find the factorial of any number less than 105 the program will save the answer in one shared variable and will solve it. Once the program will end it will ask again to find the factorial until the user say no. Each time the answer is saved in an array and will calculate for new values only. Previous values which are stored in array will directly be accessed from that array.*

***Task#2**: Write a program in openMP C to solve the matrix addition. You can have two large number of matrices A and B say 16 x 16. You can divide them in 2x2 matrices and add them and place the answer in the resultant Matrix C. Each of 2x2 matrices will be solved by separate thread and their result storing in the resultant will be only saved via critical section.*

***Task#3**: Write a program in openMP C to generate the Binomial Coefficient Series. You can calculate the time of finding each coefficient separately.*

***Task#3**: Write a program in openMP C to find number of permutation words created by any given string. Example "Happy" is 5 letter word and can generate 5! Words. Now each combination will be generated by different thread and calculate the time individually.*