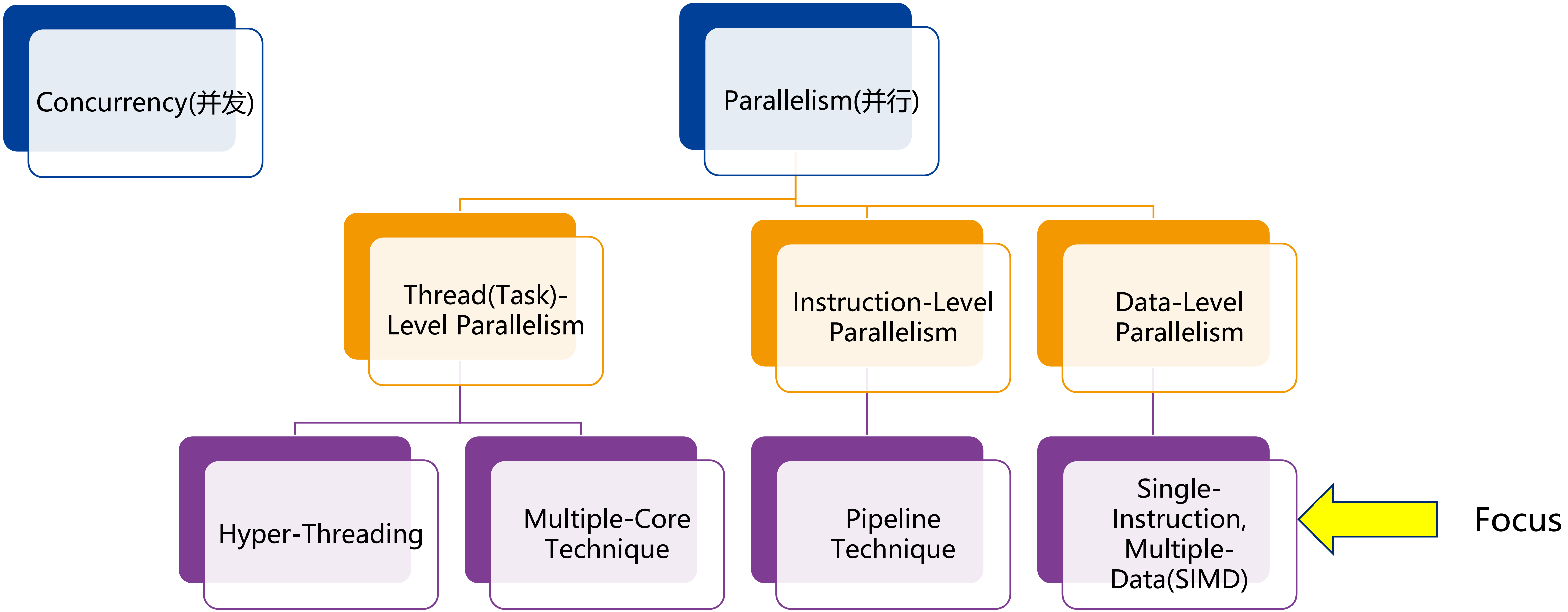
Discussion 12
SIMD and OpenMP

lizz@shanghaitech.edu.cn

上海科技大学 ShanghaiTech University

立志成才 报国裕民

Parallelism vs. Concurrency



		Data Streams	
		Single	**Multiple**
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

Scalar v.s. SIMD

The conventional sequential approach using one instruction to process each individual data is called scalar operations.

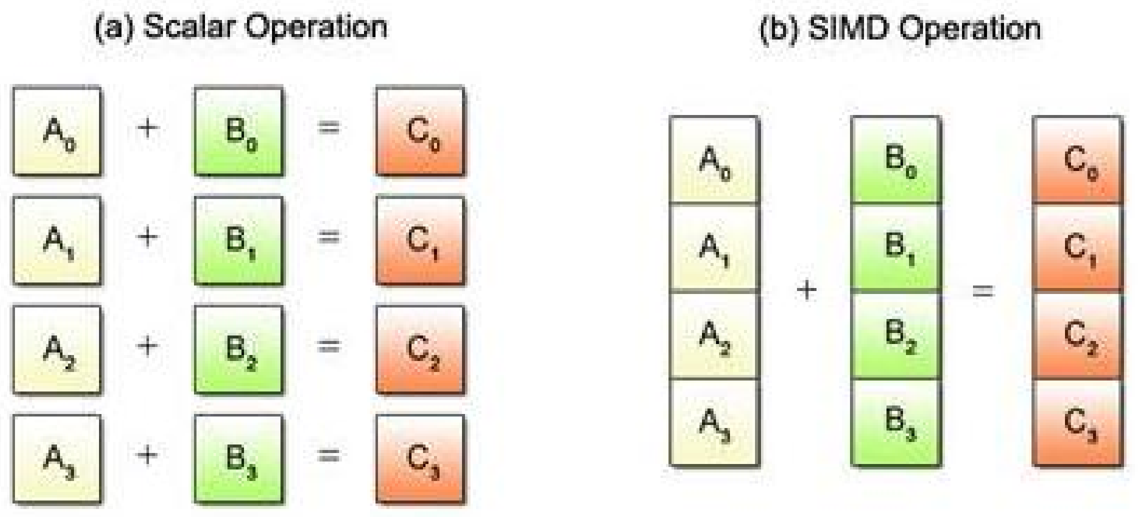


Fig. 2.1: Scalar vs. SIMD Operations

Despite the advantage of being able to process multiple data per instruction, SIMD operations can only be applied to certain predefined processing patterns.

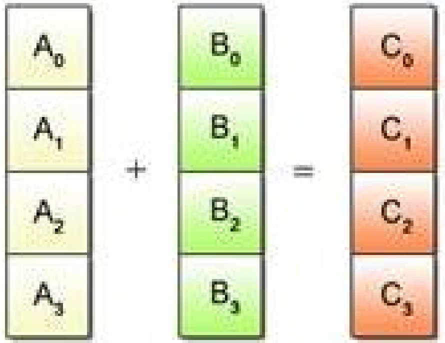


Fig. 2.2: Example of SIMD Processable Patterns

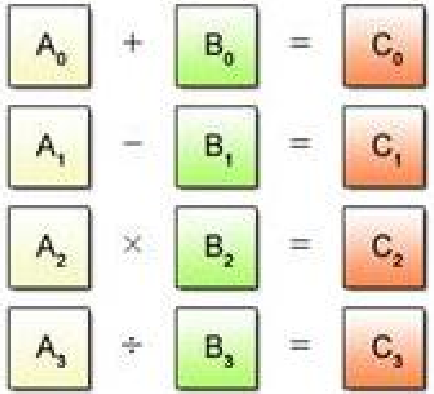


Fig. 2.3: Example of SIMD Unprocesable Patterns

If you are a C++ programmer, you are probably familiar with the basic types: char, short, int, float, and so on. Each of these have specific sizes: 8 bits for a char, 16 for short, 32 for int and float. Bits are just bits, and therefore the difference between a float and an int is in the interpretation.

```
int a;
float& b = (float&)a;
```

An alternative way to achieve this is using a union:

```
union { int a; float b; };
```

```
union { unsigned int a4; unsigned char a[4]; };
```

This time, a small array of four chars overlaps the 32-bit integer value a4. We can now access the individual bytes in a4 via array a[4]. Note that a4 now basically has four 1-byte ‘lanes’, which is somewhat similar to what we get with SIMD.

We could even use a4 as 32 1-bit values, which is an efficient way to store 32 boolean values.

```
#include "nmmintrin.h" // for SSE4.2
#include "immintrin.h" // for AVX
```

A __m128 variable contains four floats, so we can use the union trick again:

```
union { __m128 a4; float a[4]; };
```

Now we can conveniently access the individual floats in the __m128 vector.

```
__m128 a4 = _mm_set_ps( 4.0f, 4.1f, 4.2f, 4.3f );
__m128 b4 = _mm_set_ps( 1.0f, 1.0f, 1.0f, 1.0f );
```

To add them together, we use _mm_add_ps:

```
__m128 sum4 = _mm_add_ps( a4, b4 );
```

The __mm_set_ps and _mm_add_ps keywords are called *intrinsics*. SSE and AVX intrinsics all compile to a single assembler instruction; using these means that we are essentially writing assembler code directly in our program. There is an intrinsic for virtually every scalar operation:

```
_mm_sub_ps( a4, b4 );
_mm_mul_ps( a4, b4 );
_mm_div_ps( a4, b4 );
_mm_sqrt_ps( a4 );
_mm_rcp_ps( a4 ); // reciprocal
```

OpenMP

Parallel Programming Standards

Thread Libraries :
- Win32 API / POSIX threads

Compiler Directives :
- OpenMP (Shared memory programming) ⇦ Our focus

Message Passing Libraries :
- MPI (Distributed memory programming)

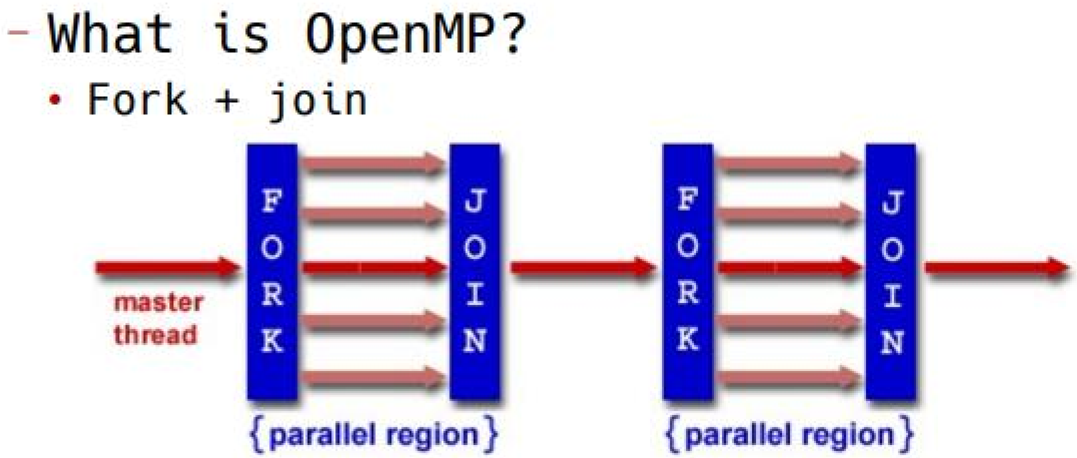
Most of the constructs in OpenMP are compiler directives or pragmas.

– For C and C++, the pragmas take the form:

```
#pragma omp construct [clause [clause]…]
```

Include files :

```
#include   "omp.h"
```



OpenMP is usually used to parallelize loops:

- Find your most time consuming loops.
- Split them up between threads.

Sequential Program

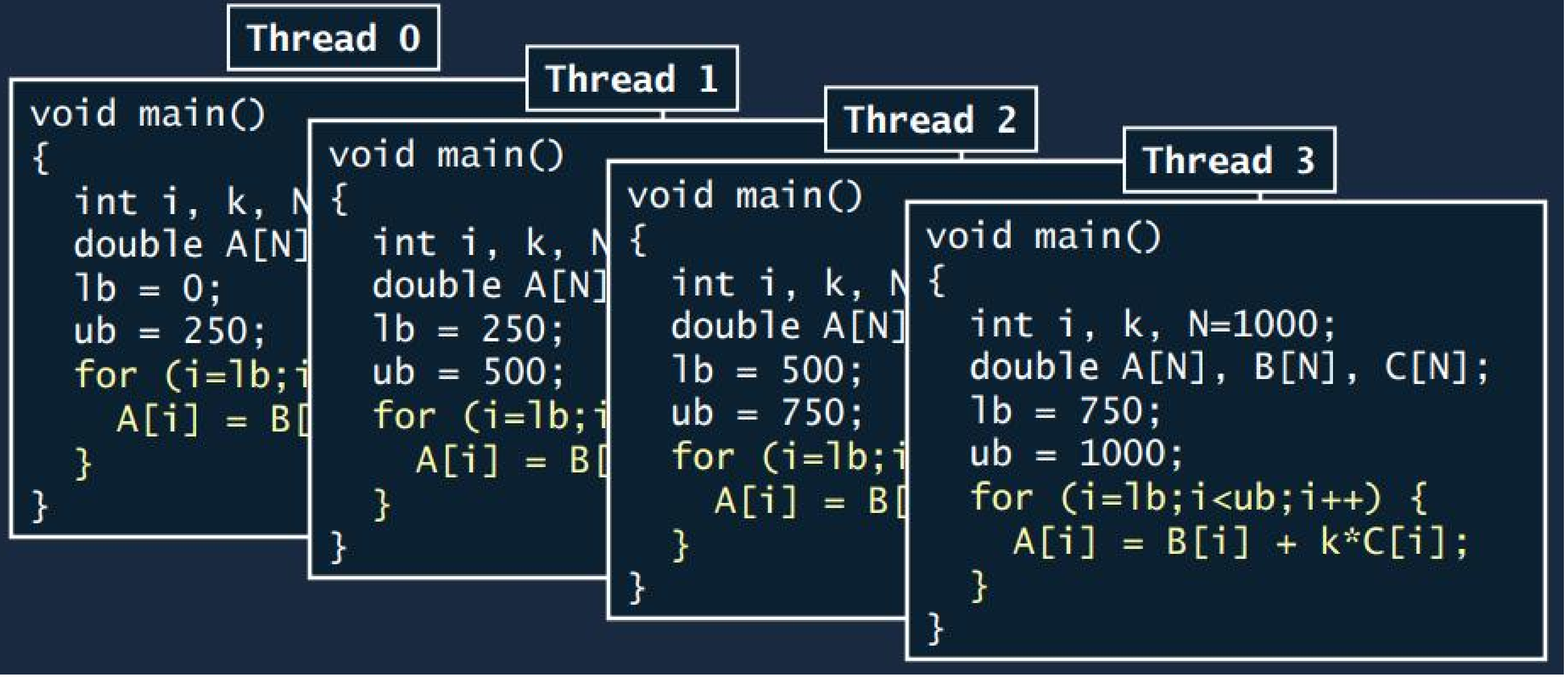
```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

→

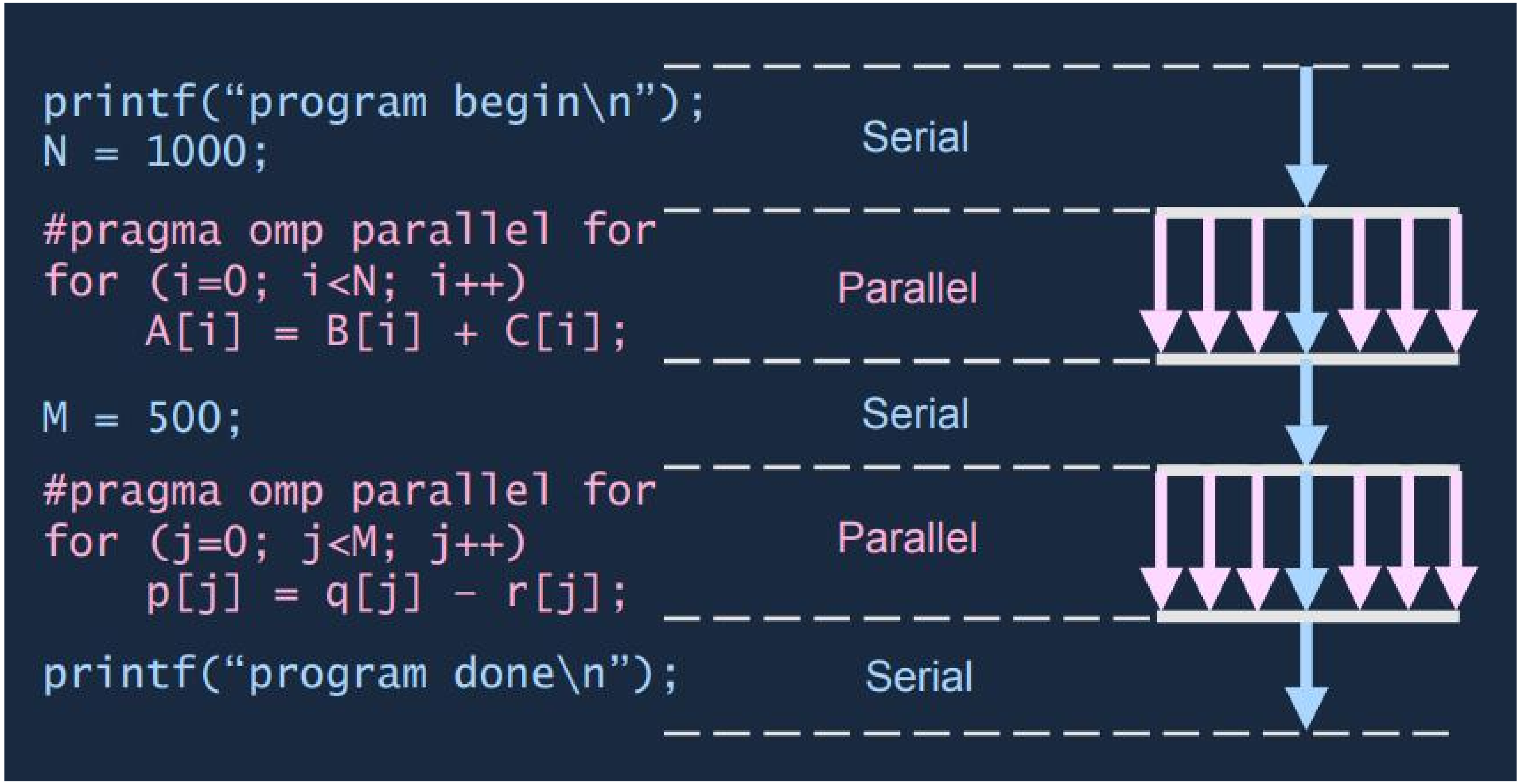
Parallel Program

```
#include “omp.h”
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

- Single Program Multiple Data (SPMD)



OpenMP Fork-and-Join model



Implicit “barrier” synchronization at end of for loop.

Directives

Private: This directive is used to create a private copy of a variable for each thread, so that each thread can work on its own copy of the variable without interfering with other threads.

Reduction: This directive is used to perform a reduction operation on a variable across multiple threads, and then store the result in a single variable. The supported reduction operations include addition, subtraction, multiplication, bitwise AND/OR/XOR, and logical AND/OR.

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

int main() {
    int count = 0;

    #pragma omp parallel for private(count)
    for (int i = 0; i < 10; i++) {
        count++;
        printf("Count = %d\n", count);
    }

    printf("Count = %d\n", count);
    return 0;
}
```

```
Count = 1930265
Count = 1930266
Count = 1930537
Count = 9
Count = 10
Count = 1940553
Count = 1940809
Count = 1942345
Count = 1943753
Count = 1943625
Count = 15630105
Count = 15628809
Count = 15633609
Count = 9
Count = 10
Count = 0
```

```
Count = 7047193
Count = 9
Count = 10
Count = 7042281
Count = 7042073
Count = 7042074
Count = 7046777
Count = 7047369
Count = 7043609
Count = 7045817
Count = 0
```

```
Count = 15894505
Count = 15891001
Count = 15889705
Count = 15894713
Count = 15892265
Count = 9
Count = 10
Count = 15900425
Count = 15889433
Count = 15889434
Count = 0
```

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

int main() {
    int count = 0;

    omp_set_num_threads(10);

    #pragma omp parallel for private(count)
    for (int i = 0; i < 10; i++) {
        count++;
        printf("Count = %d\n", count);
    }

    printf("Count = %d\n", count);
    return 0;
}
```

```
Count = 15563065
Count = 15568249
Count = 15561961
Count = 15561962
Count = 15563225
Count = 15562745
Count = 9
Count = 10
Count = 15562905
Count = 15568377
Count = 0
```

```
Count = 7113433
Count = 7108889
Count = 7118649
Count = 7107817
Count = 7107818
Count = 9
Count = 10
Count = 7114393
Count = 7109097
Count = 7110441
Count = 0
```

```
Count = 16359177
Count = 16349769
Count = 16348457
Count = 16353257
Count = 9
Count = 10
Count = 16351001
Count = 16353465
Count = 16348185
Count = 16348186
Count = 0
```

```
Count = 9
Count = 10
Count = 14786313
Count = 14775321
Count = 14775322
Count = 14776889
Count = 14778153
Count = 14780601
Count = 14775593
Count = 14780393
Count = 0
```

The private directive declares that a variable is private to each thread, meaning that each thread has its own instance of the variable and does not share the value of the variable among all parallel parts in each thread. The firstprivate directive also declares a variable to be private, but it also initializes the variable to its value in the serial part.

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

int main() {
    int count = 0;

    omp_set_num_threads(10);

    #pragma omp parallel for firstprivate(count)
    for (int i = 0; i < 10; i++) {
        count++;
        printf("Count = %d\n", count);
    }

    printf("Count = %d\n", count);
    return 0;
}
```

```
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 0
```

```
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 0
```

```
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 0
```

```
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 0
```

```
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 1
Count = 0
```

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

int main() {
    int count = 0;

    #pragma omp parallel for reduction(+:count)
    for (int i = 0; i < 10; i++) {
        count++;
    }

    printf("Count = %d\n", count);
    return 0;
}
```

Count = 10

Single: This directive is used to specify a block of code that should be executed by only one thread, while the other threads wait for it to complete.

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

int main(int argc, char **argv){
    int i =0 ;
    omp_set_num_threads (4) ; //Maximum 4 threads
    #pragma omp parallel private(i)
    {
        printf ( "thread %d start\n" , omp_get_thread_num ( )) ;
        #pragma omp single
        {
        for (i = 0; i <6; i++){
            printf ( "single, thread %d execute i = %d\n" ,
            omp_get_thread_num () , i) ;
        }
        }
    }
}
```

```
thread 0 start
single, thread 0 execute i = 0
single, thread 0 execute i = 1
single, thread 0 execute i = 2
single, thread 0 execute i = 3
single, thread 0 execute i = 4
single, thread 0 execute i = 5
thread 2 start
thread 1 start
thread 3 start
```

```
thread 0 start
single, thread 0 execute i = 0
single, thread 0 execute i = 1
single, thread 0 execute i = 2
single, thread 0 execute i = 3
single, thread 0 execute i = 4
single, thread 0 execute i = 5
thread 2 start
thread 3 start
thread 1 start
```

Critical: This directive is used to specify a block of code that should be executed by only one thread at a time, while the other threads wait for it to complete. The purpose of the 'critical' directive is to protect a shared resource (e.g., a variable, a file, or a device) from being accessed by multiple threads simultaneously, which can lead to race conditions and data inconsistencies.

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

int main() {
    int count = 0;

    #pragma omp parallel for
    for (int i = 0; i < 10; i++) {
        #pragma omp critical
        {
            count++;
        }
    }

    printf("Count = %d\n", count);
    return 0;
}
```

Count = 10

Barrier: This directive is used to synchronize all the threads in a parallel region, so that no thread can proceed until all the other threads have reached the barrier.

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

int main() {
    #pragma omp parallel
    {
        printf("Thread %d is doing some work.\n", omp_get_thread_num());
        #pragma omp barrier
        printf("Thread %d has finished its work.\n", omp_get_thread_num());
    }

    return 0;
}
```

```
Thread 7 is doing some work.
Thread 3 is doing some work.
Thread 4 is doing some work.
Thread 6 is doing some work.
Thread 1 is doing some work.
Thread 2 is doing some work.
Thread 0 is doing some work.
Thread 5 is doing some work.
Thread 7 has finished its work.
Thread 6 has finished its work.
Thread 1 has finished its work.
Thread 3 has finished its work.
Thread 4 has finished its work.
Thread 2 has finished its work.
Thread 0 has finished its work.
Thread 5 has finished its work.
```

```
Thread 2 is doing some work.
Thread 6 is doing some work.
Thread 1 is doing some work.
Thread 3 is doing some work.
Thread 5 is doing some work.
Thread 0 is doing some work.
Thread 4 is doing some work.
Thread 7 is doing some work.
Thread 2 has finished its work.
Thread 4 has finished its work.
Thread 6 has finished its work.
Thread 3 has finished its work.
Thread 5 has finished its work.
Thread 0 has finished its work.
Thread 7 has finished its work.
Thread 1 has finished its work.
```