Sequential program

- From programmers perspective: Statements are executed in the order in which they appear
- On the CPU level this is not true
  - Instruction reordering due to compiler optimizations
  - Inside the CPU: Out of order execution to better utilize processing units
Sequential program

- From programmers perspective: Statements are executed in the order in which they appear
- On the CPU level this is not true
  - Instruction reordering due to compiler optimizations
  - Inside the CPU: Out of order execution to better utilize processing units

Parallel program

- For the CPU completely independent programs executed on different cores
- Order of memory writes in one thread can be different when observed from another thread
- Programmer is responsible for all synchronization
- Statements from different threads can be interleaved in millions of ways even for small programs
Example

What can happen if these statements are executed in parallel?

```c
int x = 0;
int y = 0;
...
if (y == 1)
    printf("x is %i \n", x);
```

- Sequentially executed this program can only print “x is 1”
- When executed in parallel “x is 0” is possible
Example

What can happen if these statements are executed in parallel?

```c
int x = 0;
int y = 0;

if (y == 1)
    printf("x is %d\n", x);
```

- Sequentially executed this program can only print “x is 1”
- When executed in parallel “x is 0” is possible

Memory operations might not be visible in the same order when observed from the other thread!

- Safe abstractions and memory model required for parallel programs
OpenMP

- OpenMP standard provides annotations for C, C++, and Fortran languages
- If compiled without OpenMP the program is still a valid sequential program
- Compiler takes care of thread handling and provides abstractions for synchronization

```c
double sum = 0.0;
double vec[N];
// initialize vec
#pragma omp parallel for reduction (+: sum)
for (int i = 0; i < N; ++i)
    x += vec[i];
```
OpenMP

- OpenMP standard provides annotations for C, C++, and Fortran languages
- If compiled without OpenMP the program is still a valid sequential program
- Compiler takes care of thread handling and provides abstractions for synchronization

```c
double sum = 0.0;
double vec[N];
// initialize vec
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; ++i)
    x += vec[i];
```
Parallel section and work sharing

- `#pragma omp parallel` starts parallel section
- Parallel section is executed by all threads

```c
#pragma omp parallel numthreads(4)
// implicit flush
{
    // executed by all 4 threads
    printf("thread_%d: hello world!\n", omp_get_thread_num());

#pragma omp single
{
    // executed on one thread
} // implicit barrier and flush

// execute do_work1() and do_work2() in parallel
#pragma omp sections
{
    #pragma omp section
do_work1();
    #pragma omp section
do_work2();
}
} // implicit barrier and flush
```
Parallel section and work sharing

- `#pragma omp parallel` starts parallel section
- Parallel section is executed by all threads
- Work sharing constructs are used to distribute work
  - `#pragma omp for`
  - `#pragma omp sections`
  - `#pragma omp single`

```c
#pragma omp parallel numthreads(4)
// implicit flush
{
    // executed by all 4 threads
    printf("thread_%d: hello world!\n", omp_get_thread_num());

    #pragma omp single
    {
        // executed on one thread
    } // implicit barrier and flush

    // execute do_work1() and do_work2() in parallel
    #pragma omp sections
    {
        #pragma omp section
        do_work1();
        #pragma omp section
        do_work2();
    }
} // implicit barrier and flush
```
Parallel section and work sharing

- `#pragma omp parallel` starts parallel section
- Parallel section is executed by all threads
- Work sharing constructs are used to distribute work
  - `#pragma omp for`
  - `#pragma omp sections`
  - `#pragma omp single`
- Memory consistency enforced with `#pragma omp flush` operation
- Wait for threads to reach specified point with `#pragma omp barrier`

```
// #pragma omp parallel numthreads(4)
// implicit flush
{
    // executed by all 4 threads
    printf("thread_%i: hello world!\n", omp_get_thread_num());

    #pragma omp single
    {
        // executed on one thread
    } // implicit barrier and flush
     // execute do_work1() and do_work2() in parallel
    #pragma omp sections
    {
        #pragma omp section
        do_work1();
        #pragma omp section
        do_work2();
    } // implicit barrier and flush
```
Sharing state

- Default uses scoping rules to determine whether variables are shared.
- Can also be specified explicitly with clauses:
  - `firstprivate(var)`
  - `private(var)`
  - `lastprivate(var)`
  - `shared(var)`
- `shared(var)` clause only useful with `default(none)` or `default(private)` clause.
Sharing state

- Default uses scoping rules to determine whether variables are shared.
- Can also be specified explicitly with clauses:
  - `firstprivate(var)`
  - `private(var)`
  - `lastprivate(var)`
  - `shared(var)`
- `shared(var)` clause only useful with `default(none)` or `default(private)` clause.

```c
int x = 0; // x is shared (scope)
int y = 0; // y is private (clause)
#pragma omp parallel numthreads(4), firstprivate(y)
// private(y) would make y uninitialized
{
    int z = 0; // z is private (scope)
}
```
Other Features

- Many new features with each new version
- `#pragma omp task` for recursive work sharing (since OpenMP 3.0)
- `#pragma omp simd` for instruction-level parallelism (since OpenMP 4.0)
- Directives for GPU computing (since OpenMP 4.0)
- ...

Thank you for your attention!
Other Features

▶ Many new features with each new version
▶ #pragma omp task for recursive work sharing (since OpenMP 3.0)
▶ #pragma omp simd for instruction-level parallelism (since OpenMP 4.0)
▶ Directives for GPU computing (since OpenMP 4.0)
▶ ...

Thank you for your attention!