Summary: Introduction to OpenMP
Fork-Join Execution Model

```
#pragma omp parallel
{
  Master Thread
  Fork of threads
  Worker Threads
  Join of threads
  Thread Team
  serial part
  parallel region
}
```
Data Sharing Attributes

```c
int a;

#pragma omp parallel shared(a)
{
  // parallel region
}
```

The same memory is used for all threads in the parallel region.
Data Sharing Attributes

```c
int a, b;

#pragma omp parallel shared(a) //
     private(b)
{
    int c;
}
```

uninitialized private copies of `b` and `c` for every thread
Data Sharing Attributes

```c
int d=2;

#pragma omp parallel firstprivate(d)
{
    #pragma omp single
    {d=6;}
}
```

Diagram illustrating the sharing of variable `d` during a parallel loop.
For Worksharing

```c
#pragma omp parallel
#pragma omp for
for (int i=0; i<100; i++){
    // Worksharing logic here
}
```

The diagram shows a parallel region that distributes loop iterations across threads. The serial part is divided into three regions:
- Serial part 0-24
- Parallel region 25-49
- Serial part 50-74
- Serial part 75-99
Parallelizable Loops

- Loop iterations must be independent to parallelize a loop!

No loop dependencies => parallelizable

```c
#pragma omp parallel for
for ( i=0 ; i<100 ; i++ ){
    a[i] = b[i] + c[i];
}
```

Loop dependencies => not parallelizable

```c
#pragma omp parallel for
for ( i=1 ; i<100 ; i++ ){
    a[i] = a[i] + a[i-1];
}
```

- Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent.

But: This test alone is not sufficient
Reduction Operations

```c
int a=0;

#pragma omp parallel
#pragma omp for reduction(+:a)
for (int i=0; i<100; i++)
{
    a+=i;
}
```

- Local copies for computation
- Update is written to the shared variable
- Reduction computes final result in the shared variable
Tasks

```
#pragma omp parallel
#pragma omp single
while (work()){
    #pragma omp task
    {
        ...
    }
} // implicit barrier here
```

A task is some code together with a data environment. Tasks can be executed by any thread in any order.
The Barrier and Taskwait Constructs

OpenMP barrier (implicit or explicit)

→ All tasks created by any thread of the current Team are guaranteed to be completed at barrier exit

C/C++
#pragma omp barrier

Task barrier: taskwait

→ Encountering Task suspends until child tasks are complete

→ Only direct childs, not descendants!

C/C++
#pragma omp taskwait
Questions?