Summary: Introduction to OpenMP
Fork-Join Execution Model

```c
#pragma omp parallel
{
    Worker Threads
}
```

- Master Thread
- Fork of threads
- Join of threads
- Thread Team

OpenMP and Performance
C. Terboven | IT Center der RWTH Aachen University
Data Sharing Attributes

```c
int a;

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

The same memory is used for all threads in the parallel region, while the serial part remains unchanged.
Data Sharing Attributes

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

The diagram illustrates the sharing of variables in a parallel region. Here, `a` is shared among threads, while `b` and `c` are private to each thread, ensuring each thread has its own copy.

- **Parallel Region:**
  - `a` is shared and visible across all threads.
  - `b` and `c` are private copies, one per thread.

- **Serial Part:**
  - Before the parallel region, `a` is visible, and `b` is private.
  - After the parallel region, `a` is no longer visible, and `b` remains private.

The diagram shows the progression of variable visibility through the parallel region, emphasizing the importance of data sharing attributes for optimizing parallel performance.
Data Sharing Attributes

```c
int d=2;

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

The diagram illustrates the behavior of the `firstprivate` attribute in OpenMP. In the parallel region, each thread has a separate copy of the variable `d`, initialized with the value `2`. The single section updates the value of `d` to `6`, which is then visible in the parallel region but not in the serial part. This shows how variables declared with `firstprivate` are copied to each thread's private section and then updated, ensuring data consistency and avoiding race conditions.
For Worksharing

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

}
```

distributes loop iterations across threads
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;
}
```

Update is written to the shared variable.

Reduction computes final result in the shared variable.

Local copies for computation.
Tasks

```c
#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?