Introduction to OpenMP

Christian Terboven

IT Center, RWTH Aachen University
terboven@itc.rwth-aachen.de
History

- De-facto standard for Shared-Memory Parallelization.
- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN (errata)
- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.
- 05/2008: OpenMP 3.0
- 07/2011: OpenMP 3.1
- 07/2013: OpenMP 4.0
- 11/2015: OpenMP 4.5

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.
What is OpenMP?

- De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran

- Compiler Directives, Runtime routines and Environment variables
OpenMP Overview
&
Parallel Region
OpenMP: Shared-Memory Parallel Programming Model.

All processors/cores access a shared main memory.

Real architectures are more complex, as we will see later / as we have seen.

Parallelization in OpenMP employs multiple threads.
OpenMP programs start with just one thread: The Master.

Worker threads are spawned at Parallel Regions, together with the Master they form the Team of threads.

In between Parallel Regions the Worker threads are put to sleep. The OpenMP Runtime takes care of all thread management work.

Concept: Fork-Join.

Allows for an incremental parallelization!
Parallel Region and Structured Blocks

The parallelism has to be expressed explicitly.

C/C++

```c
#pragma omp parallel
{
  ...
  structured block
  ...
}
```

Fortran

```fortran
!$omp parallel
...
  structured block
  ...
!$omp end parallel
```

Structured Block

- Exactly one entry point at the top
- Exactly one exit point at the bottom
- Branching in or out is not allowed
- Terminating the program is allowed (abort / exit)

Specification of number of threads:

- Environment variable:
  
  ```
  OMP_NUM_THREADS=...
  ```

- Or: Via `num_threads` clause:
  
  ```
  add num_threads(num) to the parallel construct
  ```
Hello OpenMP World
Hello orphaned OpenMP World
From within a shell, global setting of the number of threads:

```
export OMP_NUM_THREADS=4
./program
```

From within a shell, one-time setting of the number of threads:

```
OMP_NUM_THREADS=4   ./program
```
For Worksharing Construct
If only the `parallel` construct is used, each thread executes the Structured Block.

**Program Speedup: Worksharing**

OpenMP‘s most common Worksharing construct: `for`

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int i;</code></td>
<td><code>INTEGER :: i</code></td>
</tr>
<tr>
<td><code>#pragma omp for</code></td>
<td><code>!$omp do</code></td>
</tr>
<tr>
<td><code>for (i = 0; i &lt; 100; i++)</code></td>
<td><code>DO i = 0, 99</code></td>
</tr>
<tr>
<td><code>{                          </code></td>
<td><code>a[i] = b[i] + c[i]</code></td>
</tr>
<tr>
<td><code>    a[i] = b[i] + c[i];</code></td>
<td><code>END DO</code></td>
</tr>
</tbody>
</table>

→ Distribution of loop iterations over all threads in a Team.

→ Scheduling of the distribution can be influenced.

Loops often account for most of a program‘s runtime!
Worksharing illustrated

Pseudo-Code
Here: 4 Threads

Serial

\[
\begin{align*}
\text{do } i &= 0, 99 \\
a(i) &= b(i) + c(i) \\
\text{end do}
\end{align*}
\]

Thread 1

\[
\begin{align*}
\text{do } i &= 0, 24 \\
a(i) &= b(i) + c(i) \\
\text{end do}
\end{align*}
\]

Thread 2

\[
\begin{align*}
\text{do } i &= 25, 49 \\
a(i) &= b(i) + c(i) \\
\text{end do}
\end{align*}
\]

Thread 3

\[
\begin{align*}
\text{do } i &= 50, 74 \\
a(i) &= b(i) + c(i) \\
\text{end do}
\end{align*}
\]

Thread 4

\[
\begin{align*}
\text{do } i &= 75, 99 \\
a(i) &= b(i) + c(i) \\
\text{end do}
\end{align*}
\]

Memory

\[
\begin{align*}
A(0) &\\
\quad &\\
\quad &\\
B(0) &\\
\quad &\\
\quad &\\
C(0) &\\
\quad &\\
\quad &\\
A(99) &\\
\quad &\\
\quad &\\
B(99) &\\
\quad &\\
\quad &\\
C(99) &
\end{align*}
\]
Vector Addition
Influencing the For Loop Scheduling

- **for-construct**: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the `schedule` clause:
  
  1. `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.

  2. `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.

  3. `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.

- **Default on most implementations is schedule(static)**.
Can all loops be parallelized with `for`-constructs? No!

→ 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:

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

Data Race: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).
Synchronization: Critical Region

- A Critical Region is executed by all threads, but by only one thread simultaneously (Mutual Exclusion).

```
C/C++
#pragma omp critical (name)
{
    ... structured block ...
}
```

- Do you think this solution scales well?

```
C/C++
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    #pragma omp critical
    { s = s + a[i]; }
}
```
Data Scoping
Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.
- **Scoping in OpenMP**: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for each thread
    - *firstprivate*: Initialization with Master’s value
    - *lastprivate*: Value of last loop iteration is written back to Master
  - Static variables are *shared*
Privatization of Global/Static Variables

- **Global / static variables can be privatized with the `threadprivate` directive**
  - One instance is created for each thread
  - Before the first parallel region is encountered
  - Instance exists until the program ends
  - Does not work (well) with nested Parallel Region

- Based on thread-local storage (TLS)
  - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword `__thread` (GNU extension)

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int i;</td>
<td>SAVE INTEGER :: i</td>
</tr>
<tr>
<td>#pragma omp threadprivate(i)</td>
<td>!$omp threadprivate(i)</td>
</tr>
</tbody>
</table>
Privatization of Global/Static Variables

- Global / static variables can be privatized with the `threadprivate` directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), `pthread_key_create` (Posix-Threads), keyword `__thread` (GNU extension)

Really: try to avoid the use of `threadprivate` and static variables!

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static int i;</code></td>
<td><code>SAVE INTEGER :: i</code></td>
</tr>
<tr>
<td><code>#pragma omp threadprivate(i)</code></td>
<td><code>!$omp threadprivate(i)</code></td>
</tr>
</tbody>
</table>
The Barrier Construct
The Barrier Construct

- **OpenMP barrier (implicit or explicit)**
  - Threads wait until all threads of the current *Team* have reached the barrier

```c/c++
#pragma omp barrier
```

- **All worksharing constructs contain an implicit barrier at the end**
Back to our bad scaling example

C/C++

```c
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    #pragma omp critical
    {  
      s = s + a[i];  
    }
}
```
It’s your turn: Make It Scale!

```c
#pragma omp parallel
{

  #pragma omp for
  for (i = 0; i < 99; i++)
  {
    s = s + a[i];
  }

} // end parallel
```

```c
% It's your turn: Make It Scale!
```

```c
# It's your turn: Make It Scale!
```

```c
% It's your turn: Make It Scale!
```

```c
# It's your turn: Make It Scale!
```

```c
% It's your turn: Make It Scale!
```

```c
# It's your turn: Make It Scale!
```

```c
% It's your turn: Make It Scale!
```

```c
# It's your turn: Make It Scale!
```

```c
% It's your turn: Make It Scale!
```

```c
# It's your turn: Make It Scale!
```

```c
% It's your turn: Make It Scale!
```

```c
# It's your turn: Make It Scale!
```

```c
% It's your turn: Make It Scale!
```

```c
# It's your turn: Make It Scale!
```

```c
% It's your turn: Make It Scale!
```
In a reduction-operation the operator is applied to all variables in the list. The variables have to be shared.

\[
\text{reduction(\text{operator}:\text{list})}
\]

The result is provided in the associated reduction variable

C/C++

```c
int i, s = 0;
#pragma omp parallel for reduction(+:s)
for(i = 0; i < 99; i++)
{
    s = s + a[i];
}
```

Possible reduction operators with initialization value:

+ (0), \* (1), - (0), & (\sim 0), | (0), && (1), || (0), ^ (0), min (largest number), max (least number)
False Sharing

double s_priv[nthreads];

#pragma omp parallel num_threads(nthreads)
{
    int t=omp_get_thread_num();

    #pragma omp for
    for (i = 0; i < 99; i++)
    {
        s_priv[t] += a[i];
    }
}

} // end parallel
for (i = 0; i < nthreads; i++)
{
    s += s_priv[i];
}
Data in Caches

- When data is used, it is copied into caches.
- The hardware always copies chunks into the cache, so called \textit{cache-lines}.
- This is useful, when:
  - the data is used frequently (temporal locality)
  - consecutive data is used which is on the same cache-line (spatial locality)
False Sharing

- False Sharing occurs when
  - different threads use elements of the same cache-line
  - one of the threads writes to the cache-line

- As a result the cache line is moved between the threads, also there is no real dependency

- Note: False Sharing is a performance problem, not a correctness issue
False Sharing

- no performance benefit for more threads
- Reason: false sharing of s_priv
- Solution: padding so that only one variable per cache line is used

![Graph showing MFLOPS vs. #threads with and without false sharing]

**Standard**

<table>
<thead>
<tr>
<th>cache line 1</th>
<th>cache line 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td></td>
</tr>
</tbody>
</table>

**With padding**

<table>
<thead>
<tr>
<th>cache line 1</th>
<th>cache line 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>2 3</td>
</tr>
</tbody>
</table>
False Sharing avoided

double s_priv[nthreads * 8];

#pragma omp parallel num_threads(nthreads)
{
    int t=omp_get_thread_num();
    
    //pragma omp for
    for (i = 0; i < 99; i++)
    {
        s_priv[t * 8]  += a[i];
    }
}

// end parallel
for (i = 0; i < nthreads; i++)
{
    s += s_priv[i * 8];
}
Example

PI
Example: Pi (1/2)

double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

    #pragma omp parallel for
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
Example: Pi (1/2)

double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

    #pragma omp parallel for private(fX,i) reduction(+:fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
Example: Pi (2/2)

- Results:

<table>
<thead>
<tr>
<th># Threads</th>
<th>Runtime [sec.]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.11</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.14</td>
<td>7.93</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Scalability is pretty good:
  
  → About 100% of the runtime has been parallelized.
  
  → As there is just one parallel region, there is virtually no overhead introduced by the parallelization.
  
  → Problem is parallelizable in a trivial fashion ...
Single and Master Construct
The Single Construct

- **C/C++**
  ```c
  #pragma omp single [clause]
  ... structured block ...
  ```

- **Fortran**
  ```fortran
  !$omp single [clause]
  ... structured block ...
  !$omp end single
  ```

- **The single construct specifies that the enclosed structured block is executed by only on thread of the team.**
  - It is up to the runtime which thread that is.

- **Useful for:**
  - I/O
  - Memory allocation and deallocation, etc. (in general: setup work)
  - Implementation of the single-creator parallel-executor pattern as we will see now…
The Master Construct

- The master construct specifies that the enclosed structured block is executed only by the master thread of a team.

- Note: The master construct is no worksharing construct and does not contain an implicit barrier at the end.

C/C++

```c
#pragma omp master[clause]
... structured block ...
```

Fortran

```fortran
!$omp master[clause]
... structured block ...
!$omp end master
```
Runtime Library
**Runtime Library**

**C and C++:**

→ If OpenMP is enabled during compilation, the preprocessor symbol `_OPENMP` is defined. To use the OpenMP runtime library, the header `omp.h` has to be included.

→ `omp_set_num_threads(int)`: The specified number of threads will be used for the parallel region encountered next.

→ `int omp_get_num_threads`: Returns the number of threads in the current team.

→ `int omp_get_thread_num()`: Returns the number of the calling thread in the team, the Master has always the id 0.

**Additional functions are available, e.g. to provide locking functionality.**
Tasking
Recursive approach to compute Fibonacci

```c
int main(int argc,
         char* argv[])
{
    [...]  
    fib(45); 
    [...] 
}

int fib(int n)   {
    if (n < 2) return n; 
    int x = fib(n - 1); 
    int y = fib(n - 2); 
    return x+y;  
}
```

- On the following slides we will discuss three approaches to parallelize this recursive code with Tasking.
The task construct

- Deferring (or not) a unit of work (executable for any member of the team)
  - Always attached to a structured block

```c
#pragma omp task [clause[[], clause]...] {structured-block}
```

- Where clause:
  - private(list),
  - firstprivate(list),
  - shared(list)
  - default(shared | none)
  - in_reduction(r-id: list) ≥ 5.0
  - untied

```c
!$omp task [clause[[], clause]...] ...
...structured-block...
!$omp end task
```

Cutoff Strategies

- if(scalar-expression)
- mergeable
- final(scalar-expression)

Dependencies

- depend(dep-type: list)
- priority(priority-value)
Some rules from *Parallel Regions* apply:

- Static and Global variables are shared
- Automatic Storage (local) variables are private

If shared scoping is not derived by default:

- Orphaned Task variables are *firstprivate* by default!
- Non-Orphaned Task variables inherit the *shared* attribute!
- Variables are *firstprivate* unless *shared* in the enclosing context
Only one Task / Thread enters \texttt{fib()} from \texttt{main()}, it is responsible for creating the two initial work tasks. Taskwait is required, as otherwise \texttt{x} and \texttt{y} would be lost.
Fibonacci Illustration

- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
Fibonacci Illustration

- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- ...
Overhead of task creation prevents scalability!

Speedup of Fibonacci with Tasks

The OpenMP runtime optimizes when there is only one thread!
if Clause

- The if clause of a task construct
  - allows to optimize task creation/execution
  - reduces parallelism but also reduces the pressure in the runtime's task pool
  - for “very” fine grain tasks you may need to do your own (manual) if

```c
#pragma omp task if(expression)
{structured-block}
```

- If the expression of the “if” clause evaluates to false
  - the encountering task is suspended
  - the new task is executed immediately
  - the parent task resumes when the task finishes

- This is known as *undeferred* task
Improved parallelization with Tasking (omp-v2)

- Improvement: Don’t create yet another task once a certain (small enough) \( n \) is reached

```c
int main(int argc, char* argv[]) {
    [...] 
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(45);
        }
    } 
    [...] 
}
```

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x) \
       if(n > 30)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y) \
       if(n > 30)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
    return x+y;
}
```
Scalability measurements (2/3)

- Speedup is better, but still not great

![Graph showing speedup of Fibonacci with tasks]

Small tasks still have to be allocated in omp-v2!

- Speedup vs. number of threads
- omp-v1, omp-v2, optimal tracks
Improved parallelization with Tasking (omp-v3)

- Improvement: Skip the OpenMP overhead once a certain $n$ is reached

```c
int main(int argc, char* argv[]) {
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(45);
        }
    }
    [...]
}
```

```c
int fib(int n) { 
    if (n < 2) return n;
    if (n <= 30) return serfib(n);
    int x, y;
    #pragma omp task shared(x) 
    { 
        x = fib(n - 1);
    } 
    #pragma omp task shared(y) 
    { 
        y = fib(n - 2);
    } 
    #pragma omp taskwait 
    return x+y;
} 
```
Scalability measurements (3/3)

- Looks promising...

![Speedup of Fibonacci with Tasks](chart.png)

- **Speedup of Fibonacci with Tasks**
  - # Threads: 1, 2, 4, 8, 16
  - Speedup: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

Graph showing speedup of Fibonacci calculations with different numbers of threads for various OpenMP versions:
- omp-v1
- omp-v2
- omp-v3
- optimal
First two versions were slow because of overhead!

Runtime of Fibonacci with Tasks

- omp-v1
- omp-v2
- omp-v3

Runtime [s] vs # Threads

1 2 4 8 16

Runtime measurements (1/2)
Third version is comparable to serial version w/o OpenMP 😊

Runtime of Fibonacci with Tasks

- omp-v3
- serial

Runtime [s] vs # Threads

login-t, E5-2650 v4, 2x 12 cores @ 2.20 GHz
Intel Compiler 16.0.2, fib(45) = 1134903170
Tasking Overheads

- Typical overheads in task-based programs are:
  - Task creation: populate task data structure, add task to task queue
  - Task execution: retrieve a task from the queue (may including work stealing)

- If tasks become too fine-grained, overhead becomes noticeable
  - Execution spends a higher relative amount of time in the runtime
  - Task execution contributing to runtime becomes significantly smaller

- A rough rule of thumb to avoid (visible) tasking overhead
  - OpenMP tasks: 80-100k instructions executed per task
  - TBB tasks: 30-50k instructions executed per task
  - Other programming models may have another ideal granularity!
Threads vs Tasks

- **Threads do not compose well**
  - Example: multi-threaded plugin in a multi-threaded application
  - Composition usually leads to oversubscription and load imbalance

- **Task models are inherently composable**
  - A pool of threads executes all created tasks
  - Tasks from different modules can freely mix

- **Task models make complex algorithms easier to parallelize**
  - Programmers can think in concurrent pieces of work
  - Mapping of concurrent execution to threads handled elsewhere
  - Task creation can be irregular (e.g., recursion, graph traversal)
Sometimes You’re Better off with Threads...

- Some scenarios are more amenable for traditional threads
  - Granularity too coarse for tasking
  - Isolation of autonomous agents

- Static allocation of parallel work is typically easier with threads
  - Controlling allocation of work to cache hierarchy

- Graphical User Interfaces (event thread + worker threads)

- Request/response processing, e.g.,
  - Web servers
  - Database servers
Data Scoping
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a:
            // Scope of b:
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;

            // Scope of a: shared
            // Scope of b:
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d:
            // Scope of e:
        }
    }
}
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e:
        }
    }
}
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e: private
        }
    }
}
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;

            // Scope of a: shared, value of a: 1
            // Scope of b: firstprivate, value of b: 0 / undefined
            // Scope of c: shared, value of c: 3
            // Scope of d: firstprivate, value of d: 4
            // Scope of e: private, value of e: 5
        }
    }
}
Task Synchronization
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
  ```
Task Synchronization explained:

```c
#pragma omp parallel num_threads(np)
{
    #pragma omp task
    function_A();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        function_B();
    }
}
```

- **np Tasks created here, one for each thread**
- **All Tasks guaranteed to be completed here**
- **1 Task created here**
- **B-Task guaranteed to be completed here**
Questions?