DOCTYPE html>

Agenda

13:30 - 15:00: Brief introduction to Parallel Computing with OpenMP - Session 3

- OpenMP Tasking
- OpenMP Vectorization

OpenMP Tasking

The `task` directive

- Critical regions: where each thread should run the region one-at-a-time

```
#pragma omp parallel
{
    #pragma omp critical
    {
        ... code to be run by each thread, one-at-a-time ...
    }
}
```

  of course, critical regions are serialized, i.e. the runtime scales with the number of threads.

- Single regions: within a parallel region, run by one thread

```
#pragma omp parallel
{
#pragma omp single
    {
        printf("Hi\n");
    }
}
```

OpenMP Tasking

The `task` directive

- Tasks: define a block of code, a task to be run by a single thread:

```
#pragma omp task
{
    ...
}
```

- Usually run within a single region, to distribute work

```
int a = 1;
int b = 2;
#pragma omp parallel
{
    #pragma omp single
    {
           #pragma omp task
        {
            // This will be done by one thread
                a = a+1;
        }

        #pragma omp task
        {
            // This will be done by another thread
            b = b+1;
        }
    }
}
```

OpenMP Tasking

The `task` directive

- Copy ex05 as before:

```
[front01 ex04]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex05 .
[front01 tomp$ cd ex05
```

- Inspect, compile, and run a.c :

```
[front01 ex05]$ cc -std=c99 -fopenmp -o a a.c
[front01 ex05]$ vi submit_ex05.sh
...
OMP_NUM_THREADS=5 ./a
...
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
Hi, I am thread: 0 of 5
Hi, I am thread: 1 of 5
Hi, I am thread: 4 of 5
Hi, I am thread: 3 of 5
Hi, I am thread: 2 of 5
```

OpenMP Tasking

The `task` directive

- Enclose the print statement in an `omp single` region:

```
#pragma omp parallel
{
    #pragma omp single
    {
        int tid = omp_get_thread_num();
        int nth = omp_get_num_threads();
        printf("Hi, I am thread: %2d of %2d\n", tid, nth);
    }
}
```

- Compile and run a few times.

```
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
Hi, I am thread: 0 of 5
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
Hi, I am thread: 1 of 5
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
Hi, I am thread: 3 of 5
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
```

- Each time a single thread calls the `printf()` . Which thread this is, is random.

OpenMP Tasking

The `task` directive

- Now try the following:

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

int
main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            {
                int tid = omp_get_thread_num();
                int nth = omp_get_num_threads();
                printf("1: Hi, I am thread: %2d of %2d\n", tid, nth);
            }
            #pragma omp task
            {
                int tid = omp_get_thread_num();
                int nth = omp_get_num_threads();
                printf("2: Hi, I am thread: %2d of %2d\n", tid, nth);
            }
        }
    }
    return 0;
}
```

- Compile and run a few times
- What do you see ?

OpenMP Tasking

The `task` directive, another example

- The `taskwait` directive can be used to ensure the order in which tasks are run:

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

int
main()
{
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        {
            int tid = omp_get_thread_num();
            int nth = omp_get_num_threads();
            printf("1: Hi, I am thread: %2d of %2d\n", tid, nth);
        }
        #pragma omp taskwait
        #pragma omp task
        {
            int tid = omp_get_thread_num();
            int nth = omp_get_num_threads();
            printf("2: Hi, I am thread: %2d of %2d\n", tid, nth);
        }
    }
}
return 0;
}
```

- This way, the first `printf()` is always run first

OpenMP Tasking

The `task` directive, another example

- Consider the three words: one , two , and three
- Write a program that, each time it is run, prints a random permutation of these three words

The regular procedure

- Copy ex06 as before:

```
[front01 ex05]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex06 .
[front01 tomp]$ cd ex06
```

- Inspect, compile, and submit:

```
[n001 ex06]$ vi submit_ex06.sh
...
./a
...
[n001 ex06]$ more ex06.out
one two three
```

- Add task directives, so that the three words appear in a random permutation

OpenMP Tasking

The `task` directive, yet another example

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

int
main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            printf("one ");

            #pragma omp task
            printf("two ");

            #pragma omp task
            printf("three ");
        }
    }
    printf("\n");
    return 0;
}
```

Edit `submit_ex06.sh` to

```
[front01 ex06]$ vi submit_ex06.sh
...
OMP_NUM_THREADS=5 ./a
...
```

and run a few times:

```
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
three one two
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
one two three
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
two one three
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
one two three
```

OpenMP Tasking

The `task` directive, yet another example

- Now add an additional `task` to print `done!` , but insure that its *always* as the last word

OpenMP Tasking

The `task` directive, yet another example

- Copy ex07 as before:

```
[front01 ex06]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex07 .
[front01 tomp]$ cd ex07
```

OpenMP Tasking

The task directive, yet another example

- Inspect `a.c` , in particular `main()`
  - Initializes an array `a[N]`
    - The first `N-1` elements are set to a random integer, up to 8 digits long
    - The last element `a[N-1]` is set to `-1`
  - Sets a pointer p to the first element: `*p = &a[0]`
  - And then enters a loop:
    - Calls the function `process()` with argument the pointer `p` : `process(p)`
    - After `process()` returns, sets p to the next element in `a[]` : `p=p+1`
    - The loop terminates when the value in p is -1 , i.e. when it reaches the end of array `a[]`
- Inspect function `process()`
  - Takes the value pointed to by `*x` (p in the main program)
  - Sums all integers from one up to the value of `*x`
  - Sets `*x` to be equal to the sum

OpenMP Tasking

The `task` directive, yet another example

- main of `ex07.c`

```c
int
main()
{
  srand(2147483641);
  int a[N];
  for(int i=0; i<N-1; i++)
    a[i] = irand();

  a[N-1] = -1;

  double t0 = stop_watch(0);
  int *p = &a[0];
  while(*p >= 0) {
      process(p);
      p = p+1;
  }
  t0 = stop_watch(t0);

  printf(" t = %lf\n", t0);
  return 0;
}
```

- and function `process()`

```c
void
process(int *x)
{
    int sum = 0;
    for(int i=0; i<*x; i++) {
        sum += i;
    }
*x = sum;
return;
}
```

OpenMP Tasking

The `task` directive, yet another example

- Currently, the program uses an array length of 10, and takes about 1.0 seconds to complete
- Our goal is to use `omp task` directives so that the sums are distributed to different threads
- Use `omp parallel` and `omp single` to define a `parallel` region and a `scalar` region within that parallel region
- Use `omp task` to specify which block is to be distributed to different threads. Be careful, you will need to use a `firstprivate` directive somewhere
- There are many ways in which this code could enter an infinite loop, including:
    - Corrupting the array. For example by writing, in `process()` to the wrong memory address, thus corrupting the last element of `a[]` , causing the `while()` loop to loop forever
    - Not incrementing correctly (*p) . For example if the thread that increments p is not the thread that checks the `while()` condition

OpenMP Tasking

The `task` directive, reduction of overheads

Recursive approach to compute Fibonacci

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

```
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.

OpenMP Tasking

The `task` directive, reduction of overheads

First version parallelized with Tasking (omp-v1)

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

```
int fib(int n)
{
    if (n < 2) return 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;
}
```

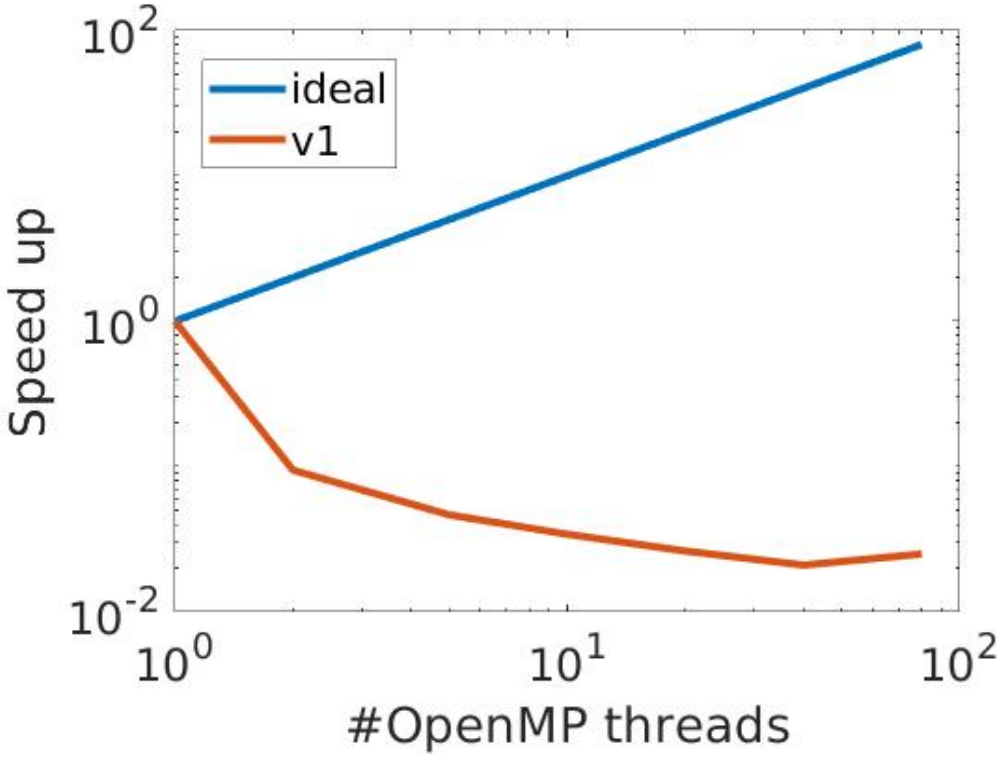
- Only one Task / Thread enters `fib()` from `main()`, it is responsable for creating the two initial work tasks
- Taskwait is required, as otherwise `x` and `y` would be lost

OpenMP Tasking

The `task` directive, reduction of overheads

Scalability measurements (1/3)

Overhead of task creation prevents better scalability



OpenMP Tasking

The `task` directive, reduction of overheads *`if` Clause*

- If the expression of an `if` clause on a task evaluates to `false`
  - The encountering task is suspended
  - The new task is executed immediately
  - The parent task resumes when the new task finishes
- Used for optimization, e.g., avoid creation of small tasks

OpenMP Tasking

The `task` directive, reduction of overheads

Second version parallelized with Tasking (omp-v2)

- Improvement: Don‘t create yet another task once a certain (small enough) n is reached

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

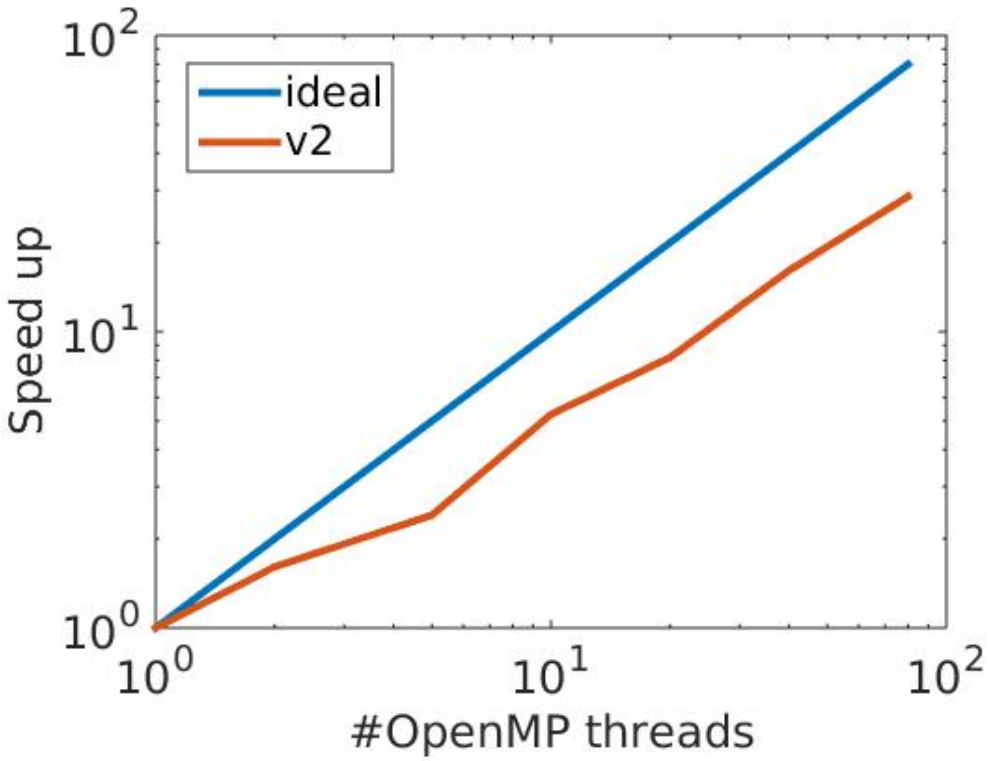
```
int fib(int n)
{
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x) \
    if(n > 30)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y) \
    if(n > 30)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
    return x+y;
}
```

OpenMP Tasking

The `task` directive, reduction of overheads

Scalability measurements (2/3)

Speedup is ok, but we still have some overhead when running with 4 or 8 threads



OpenMP Tasking

The `task` directive, Third version parallelized with Tasking (omp-v3)

- Improvement: Skip the OpenMP overhead once a certain `n` is reached (no issue w/ production compilers)

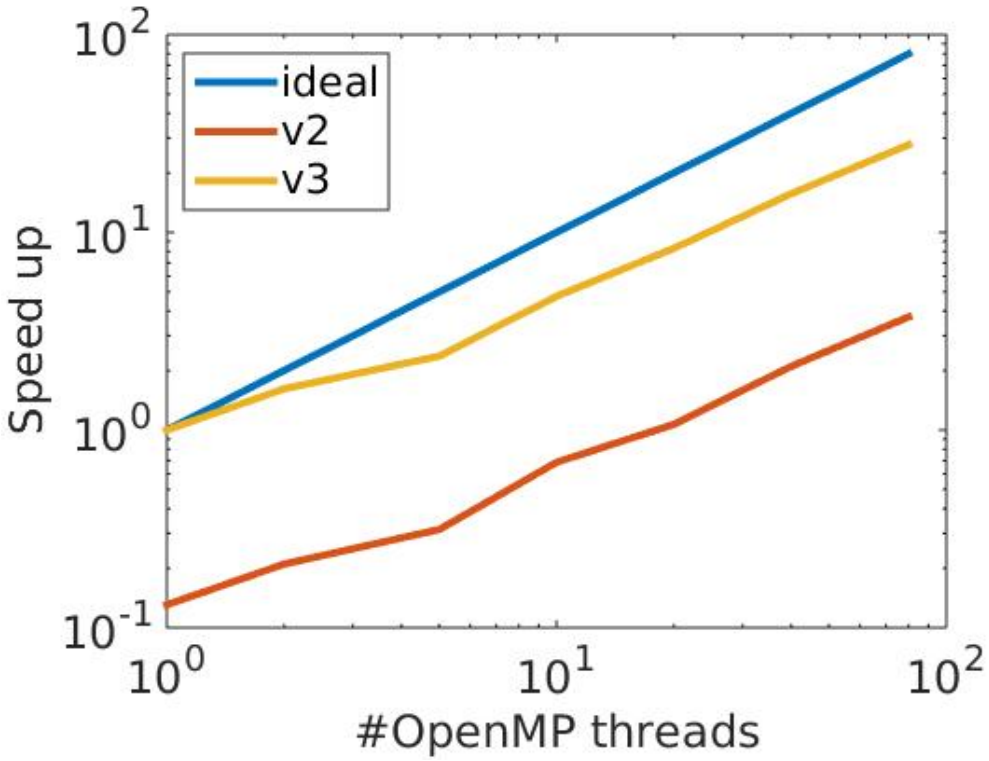
```
int main(int argc,
char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(input);
        }
    }
    [...]
}
```

```
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;
}
```

OpenMP Tasking

The `task` directive, reduction of overheads

Scalability measurements (3/3)



OpenMP Tasking

The `task` directive, reduction of overheads

- Can you achieve better scaling ?
- Check the dependence on the `if clause`
- Can you come up with a faster version for calculating Fibonaci numbers ?

Vectorization in OpenMP

In OpenMP 4.0, SIMD directives were added to help compilers generate efficient vector code.

- SIMD directives explicitly enable vectorization in the compiler
- can support the autovectorization of the compiler

SIMD loop directives

- can be placed above for loops with the syntax

  ```
  #pragma omp simd [clause[[,]clause] ...]
  ```

  which marks the loop as a SIMD enabled loop or SIMD region.
- OpenMP loop directives only apply to for loops that are in a canonical form, where the number of iteration is known

Vectorization in OpenMP

SAXPY example

- Now, lets declare a `simd` region for the second loop

```
...
#pragma omp simd
    for(int i=0; i<n; i++) {
    z_1[i] = a*x[i] + y[i];
}
...
```

compile with `-O1` and run

```
[front01 ex08]$ more ex03.out
t0 = 0.099114 sec, t1 = 0.098404 sec, diff z norm = 0.000000e+00
```

Vectorization in OpenMP

SAXPY example

- Now, lets declare a `simd` region for the second loop

```
...
#pragma omp simd
    for(int i=0; i<n; i++) {
    z_1[i] = a*x[i] + y[i];
}
...
```

compile with `-O1` and run

```
[front01 ex08]$ more ex03.out
t0 = 0.099114 sec, t1 = 0.098404 sec, diff z norm = 0.000000e+00
```

- compile with `-O2` and check the change
- what happens with `-Ofast` ? Be very careful with `-Ofast`, it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules for math function

Vectorization in OpenMP

SAXPY example

```
Lets try to parallelize it
```

```
...
#pragma omp parallel for simd
 for(int i=0; i<n; i++) {
    z_1[i] = a*x[i] + y[i];
}
 ...
```

- Note that the default gnu compiler will not work, reload another version

```
[front01 ex03]$ module load GCC/8.2.0-2.31.1
[front01 ex03]$ cc -std=c99 -fopenmp -O2 -o axpy axpy.c
```

- Checkout OpenMP enviroment variables:
  - Set `OMP_DYNAMIC=true` and compare it with `OMP_NUM_THREADS=$n`

Vectorization in OpenMP

SAXPY example

- Now does it work ?
- Intel compiler can provide a report

```
[front01 ex08]$ module load icc
[front01 ex08]$ icc -std=c99 -qopt-report=2  -qopenmp -O2 -o axpy axpy.c
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
[front01 ex08]$ more axpy.optrpt
...
LOOP BEGIN at axpy.c(76,3)
remark #15300: LOOP WAS VECTORIZED
LOOP END
...
LOOP BEGIN at axpy.c(83,3)
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END
...
```

- in more complex code, the auto-vectorization of the compiler is likely to fail, for that OpenMP provides causes which will help the compiler to identify vectorizable regions, like loop and delvare directives (see next pages for an overview)

Vectorization in OpenMP

SIMD loop directives

- SIMD aligned `#pragma omp simd aligned([ptr] : [alignment], . . . )`
  - data aligned speed up memory access and help the compiler to determine property of data
  - programmer has to ensure the data alignment

- SIMD reduction `#pragma omp simd reduction([operation] : [variable],. . . )`
  - instructs the compiler to perform a vector reduction on a variable
  - some compilers have difficulties to detecting reductions automatically

- SIMD safelen `#pragma omp simd safelen([value])`

  - for data dependency within the loop
  - assure that only data get accessed which are not exceeding the specified value

- SIMD collapse `#pragma omp simd collapse([value]`

  - try collapse nested loops into one, suitable for auto-vectorization

- SIMD private/lastprivate `#pragma omp simd private([variable],...)`

  - `private` and `lastprivate` clauses control data privatization and sharing of variables for a SIMD Loop
  - `private` clause creates an uninitialized vector inside the SIMD loop for the given variable
  - `lastprivate` clause provides the same semantics but also copies out the values produced from the last iteration to outside the loop

Vectorization in OpenMP

SIMD declare directives

SIMD enabled functions can be declared by

```
#pragma omp declare simd [clause[[,] clause] ...]
```

- compiler will create several versions of SIMD declared functions
  - different vectorization used depend on from which region function is called
  - function has its own type of vectorized arguments, `uniform`, `vector` and `linear`
- SIMD declare aligned `#pragma omp declare simd aligned([argument] : [alignment],. . . )`
  - `aligned clause` instructs the compiler that the pointers passed as function arguments are aligned by the given alignment value
- SIMD declare simdlen `#pragma omp declare simd simdlen([value])`
  - `simdlen clause` specifies the number of packed arguments the vectorized function will execute
- SIMD declare uniform `#pragma omp declare simd uniform([argument],. . . )`
  - indicates that value will not change is shared between the SIMD lanes of the loop
- SIMD declare linear `#pragma omp declare simd linear([argument] : [linearstep],. . . )`
  - `linear` clause will be increased linear between each successive function call

Vectorization in OpenMP

SIMD declare directives

Lets take a look to our last exercise

```
[front01 ex07]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex08 .
[front01 tomp$ cd ex08
```

The file is based on exercise 3, only that the function is outsourced. Compile it via:

```
[front01 ex08]$ gcc -fopenmp -O0 -c fnt.c
[front01 ex08]$ gcc -fopenmp -O0 -o axpy fnt.o axpy.c
```

and check the timings.

- check if compiler optimization can speed up the problem
- does something change if you compile the different files with different flags
- does it help to the function axpy as `declare simd`

SUMMARY

- OpenMP Introduction
  - Compiler directives, i.e., language extensions for shared memory parallelization

```
- Syntax: *directive*, **construct**, `clauses`
- C/C++: *#pragma* **omp parallel** `shared(data)`
```

  - OpenMP API functions can give back informations on the thread ID with `omp_get_thread_num()` and `omp_get_num_threads()`
- OpenMP Data sharing
  - Private variables can be indicated as `private(list)`
  - Shared variables can be indicated as `shared(list)`
- OpenMP Work sharing
  - `for-loop` construct
    - use `#pragma omp parallel for` to parallize a for-loop
    - use reduction clauses to avoid race-conditions like `#pragma omp parallel for reduction(sum_var:+)`
  - tasking
    - task can be assigned to a single thread in a parallel region
    - simplies to paralellize code but still need care to avoid overheads
- OpenMP Vectorization
  - openMP can indicate vectorizable regions for auto-vectorization of the compiler
    - regions with `#pragma omp simd` or for function `#pragma omp declare simd`