DOCTYPE html>

Agenda

11:45 - 12:30: Brief introduction to Parallel Computing with OpenMP - Session 2

- OpenMP Work sharing (45 min + Hands On)

OpenMP Work Sharing

- Parallel region creates an *Single Program Multiple Data* instance where each thread executes the same code

- we can one split the work between the threads of a parallel region?

    - Loop construct
    - Task construct

OpenMP Work Sharing

- Directive instructing compiler to share the work of a loop
    - C/C++: `#pragma omp for [clauses]`
    - Fortran: `!$omp do [clauses]`
        - The construct must followed by a loop construct. To be active it must be inside a parallel region
        - Combined construct with parallel:

          in **C/C++**:

          ```
          #pragma omp parallel for
          ```

          in **Fortran**

          ```
          $omp parallel do
          ```

- Loop index is `private` by default
- Work sharing dynamics can be controlled with the schedule `-clause`

For-loops

- in C/C++

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

- in Fortran

```
!$OMP PARALLEL DO
do i = 1, n
    ...
end do
!$OMP END PARALLEL DO
```

OpenMP Work Sharing

Loop construct

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

the `n` iterations will be split over the available threads accordingly

- Static scheduling, e.g.:

  ```
  #pragma omp parallel for schedule(static, 10)
  ```

  a chunk is 10 iterations. Threads receive a chunk to work in order.
- Dynamic scheduling, e.g.:

  ```
  #pragma omp parallel for schedule(dynamic, 10)
  ```

  a chunk is 10 iterations. Threads receive a chunk to work until they are exhausted.
- Guided scheduling, e.g.:

  ```
  #pragma omp parallel for schedule(guided)
  ```

  chunk size is modified as iterations are consumed.

OpenMP Work Sharing

Loop construct

- For loops

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

  use within a parallel region, i.e. when a parallel region is already open.

- Race condition:

  ```
  int sum_variable = 0;
  #pragma omp parallel for
  for(int i=0; i<n; i++){
      sum_variable += ...;
          ...
  }
  ```

- takes place when multiple threads read and write a variable simulatneously
- random results depedning on the order of threads accessing `sum_variable`

OpenMP Work Sharing

Reductions

- Summing elements of array is an example of reduction operation

$$S = \sum_{j=1}^{N} A_j = B_1 + B_2$$

- OpenMP provides support for common reductions within parallel regions and loops with the `reduction` -clause

OpenMP Work Sharing

Reductions

- Summing elements of array is an example of reduction operation

$$S = \sum_{j=1}^{N} A_j = B_1 + B_2$$

- OpenMP provides support for common reductions within parallel regions and loops with the `reduction` -clause

reduction(operator:list)

- Performs reduction on the (scalar) variables in list
- Private reduction variable is created for each thread's partial result
- Private reduction variable is initialized to operator's initial value
- After parallel region the reduction operation is applied to private variables and result is aggregated to the shared variable

OpenMP Work Sharing

Reductions within for loop:

```
int sum_variable = 0;

#pragma omp parallel for reduction(+: sum_variable)
for(int i=0; i<n; i++){
    sum_variable += ...;
        ...
}
```

Different reductions operators available like:

Operator	Initial value
+	0
-	1
*	0

OpenMP Work Sharing

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

- Copy ex03 as before:

```
[front01 ex02]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex03 .
[front01 tomp]$ cd ex03
```

- Inspect, compile axpy.c, edit the submit script and run :

```
[front01 ex03]$ cc -std=c99 -fopenmp -o axpy axpy.c
[front01 ex03]$ vi submit_ex03.sh
...
[front01 ex03]$ sbatch submit_ex03.sh
[front01 ex03]$ more ex03.out
t0 = 0.232100 sec, t1 = 0.232260 sec, diff z norm = 0.000000e+00
```

- Use an OpenMP pragma to parallelize the second occurrence of the main for loop

OpenMP Work Sharing

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

- Use an OpenMP pragma to parallelize the second occurrence of the main for loop

It's also useful to report the total number of threads:

```
printf(" t0 = %lf sec, t1 = %lf sec, diff z)

#pragma omp parallel
{
    int nth = omp_get_num_threads();
    #pragma omp single
    printf(" nth = %2d, t0 = %lf sec, t1 = %lf)
}
```

OpenMP Work Sharing

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

- Run for `OMP_NUM_THREADS` from 1,…,10. How does the runtime scale?

OpenMP Work Sharing

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

- Run for `OMP_NUM_THREADS` from 1,...,10. How does the runtime scale?

```
[front01 ex03]$ vi submit_ex03.sh
...

for n in 1 2 3 4 5 6 7 8 9 10
do
OMP_NUM_THREADS=$n ./axpy $((32*1024*1024))
done
...
[front01 ex03]$ sbatch submit_ex03.sh
```

- How the speed up depend on the number of threads ?
- Does it change with the number of operation ?

OpenMP Work Sharing

Linear algebra

Dot product operation:

$$r = x^T y = \sum_{i=0}^{n-1} x_i y_i$$

- Copy ex04 as before:

```
[front01 ex03]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex04 .
[front01 tomp]$ cd ex04
```

- Inspect, compile, and run xdoty.c :

```
[front01 ex04]$ cc -std=c99 -o xdoty xdoty.c
[front01 ex04]$ vi submit_ex04.sh
...
./xdoty $((32*1024*1024))
...
[front01 ex04]$ sbatch submit_ex04.sh
[front01 ex04]$ more ex04.out
t0 = 0.172530 sec, t1 = 0.171624 sec, norms = 8.387960e+06, 8.387960e+06
```

- Use an OpenMP pragma to parallelize the second occurrence of the main for loop

OpenMP Work Sharing

Linear algebra

Dot product operation:

$$r = x^T y = \sum_{i=0}^{n-1} x_i y_i$$

- Now run for OMP_NUM_THREADS from 1,…,10.
- edit `submit_ex04.sh` via

```
for n in 1 2 3 4 5 6 7 8 9 10; do
    OMP_NUM_THREADS=$n ./xdoty $((32*1024*1024))
done
```

Does the code scale ?