Carl von Ossietzky
Universität
Oldenburg

Betriebseinheit für
technisch-wissenschaftliche
Infrastruktur

Introduction to High-Performance Computing

Session 05
Introduction to OpenMP

Parallel Programming Models

- two dominating programming models:
  - OpenMP: uses directives to define work decomposition
  - MPI: standardized message-passing interface
- other programming models
  - HPF (high-performance Fortran)
  - PGAS (Partitioned Global Address Space), e.g. Co-Array Fortran UPC (Unified Parallel C)
- programming models for compute devices
  - CUDA
  - OpenCL
  - OpenACC
  - …

What is OpenMP and why use it?

- OpenMP is a standard programming model for shared memory parallelization
  - portable across different shared memory architectures
  - allows incremental parallelization
  - based on compiler directives and a few library routines
  - supports Fortran and C/C++

- easy approach to multi-threaded programming
  - allows to exploit modern multi-core CPUs
  - good performance gain for invested effort
  - hybrid-parallelization with MPI-OpenMP

OpenMP Programming Model

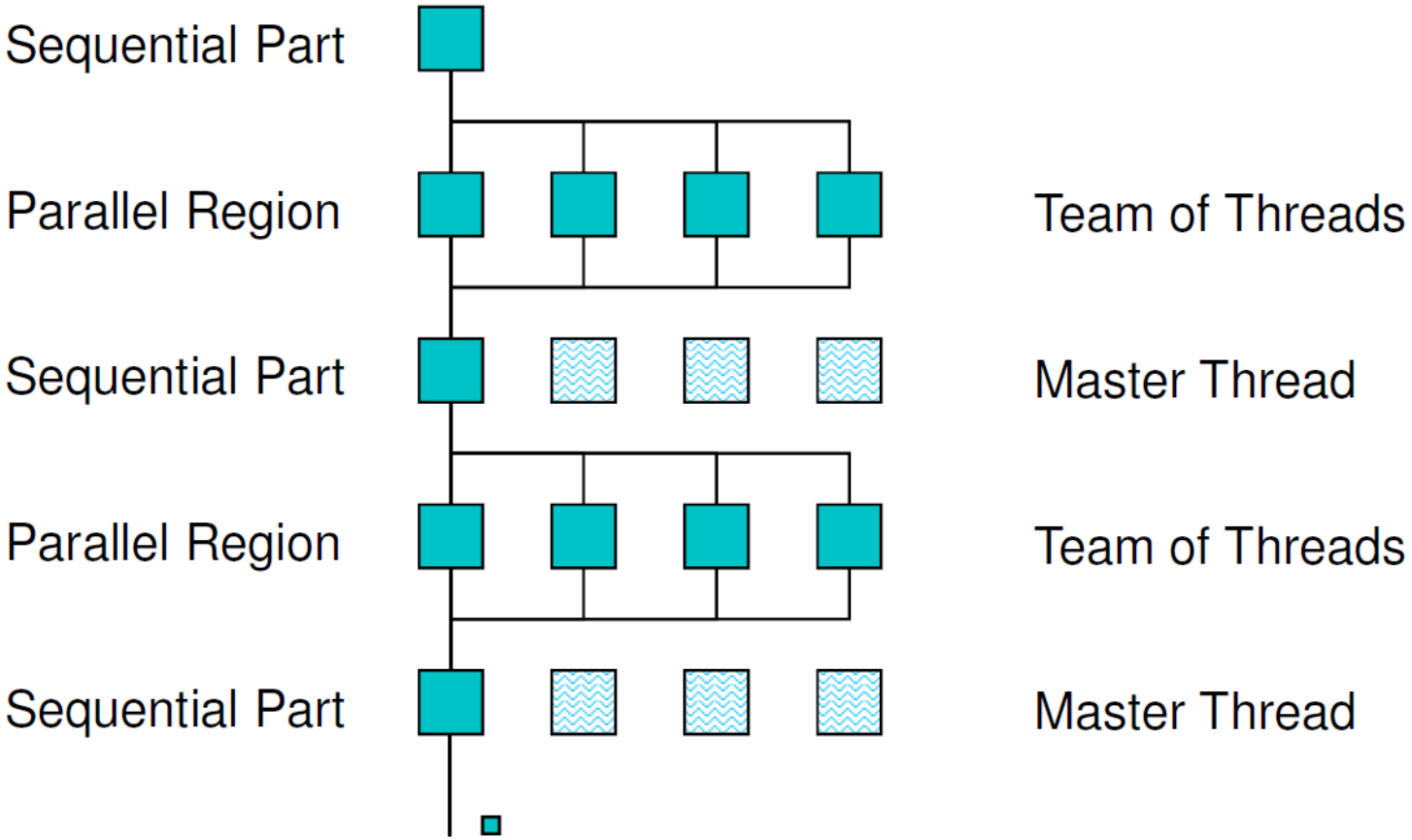
- OpenMP is a shared memory model
- workload is distributed among threads
- variables can be
  - shared among all threads
  - duplicated for each thread (private)
- threads communicate by sharing variables
  - unintended sharing can lead to race condition
- synchronization for execution control and to avoid data conflicts

OpenMP Standard

http://www.openmp.org/

- standard since 1997 (Fortran version 1.0)
- current standard is 5.1 (Nov 2020)
  - partially supported since GCC 9 and Intel 2019
  - version 4.5 supported in GCC 6.1, Intel 2017 and others
  - older versions of OpenMP have more compilers to choose from
- active development to improve performance and to adapt to new hardware technologies
  - support for SIMD parallelism was added
  - OpenMP on devices/accelerators (e.g. GPUs)

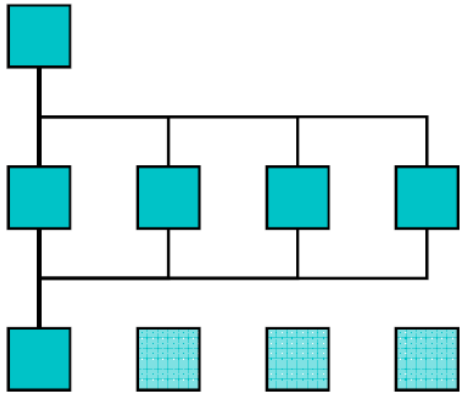
OpenMP Execution Model



OpenMP Parallel Region Construct

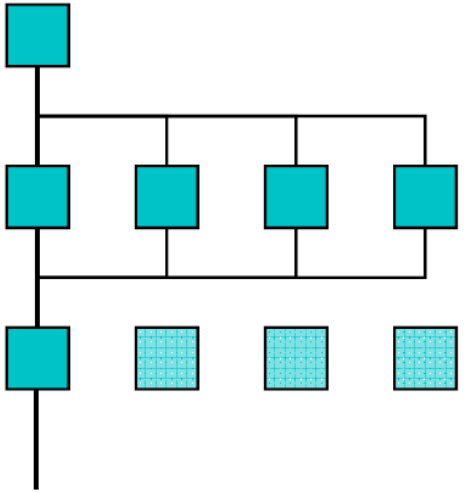
Fortran:

```
!$OMP PARALLEL
      block
!$OMP END PARALLEL
```



C / C++:

```
#pragma omp parallel
      structured block
/* omp end parallel */
```



Example: `OMP_HelloWorld`

- code available on Stud.IP

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

using namespace std;

int main () {

   #pragma omp parallel
   {
        cout << "Hello World from thread "
             << omp_get_thread_num() << endl;
   } /* end omp parallel */
}
```

Compiling and Running OpenMP Programs

- compilation with an extra option, e.g.

```
$ g++ -fopenmp OMP_HelloWorld.cpp -o OMP_HelloWorld
```
or
```
$ icpc -qopenmp OMP_HelloWorld.cpp -o OMP_HelloWorld
```

  - different compilers use different options

- before running may set environment for control

```
$ export OMP_NUM_THREADS=4
```

  - default is to use all available cores

- running the program as usual

```
$ ./OMP_HelloWorld
```

Running OpenMP Programs with SLURM

- basic job script

```
#!/bin/bash

#SBATCH --partition carl.p
#SBATCH --ntasks 1                  # single task with
#SBATCH --cpus-per-task 8           # cores to use

# execute code
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun ./OMP_HelloWorld
```

  - OpenMP programs run as single task (and single node)
  - number of cores set by **`--cpus-per-task=<n>`** or **`-c <n>`** (default is **`1`**)
  - environment variable **`$SLURM_CPUS_PER_TASK`** is available, but only if **`--cpus-per-task`** has been set
  - **`srun`** may used to create a separate job step (better accounting)

OpenMP Compiler Directives

- OpenMP uses compiler directives of the form

```
#pragma omp <directive> [clause [clause] … ]
```

  - in C/C++ this applies to the following structured block,
    in Fortran an **`END`**-directive must be used
  - different **`<directive>`** are available to control parallel program flow
  - optional one or more **`clause`**s for additional settings

OpenMP Programming

- include library

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

- available library routines
  - setting number of threads `omp_set_num_threads()`
  - getting number of threads `omp_get_num_threads()`
  - getting thread ID `omp_get_thread_num()`
  - wall clock time `omp_get_wtime()`
  - …

OMP_HelloWorld2

- what will happen here?

```
int main () {

   int threadID, nthreads;
   #pragma omp parallel
   {
      threadID = omp_get_thread_num();
      cout << "Hello World from thread " << threadID << endl;

      #pragma omp barrier     // wait for all threads

      if (threadID==0) {      // have one thread print extra info
         nthreads = omp_get_num_threads();
         cout << "Using " << nthreads << " threads!" << endl;
      }
   } /* end omp parallel */
}
```

Shared and Private Variables

- in **`OMP_HelloWorld2`** the variable **`threadID`** is shared among all threads
- this results in a so-called race condition
  - every thread is writing to the same memory address
  - final value unpredictable

- one solution is to make threadID private

```
#pragma omp parallel private(threadID)
```

Clauses for Parallel Regions

- **`private(variable list)`**
  - each thread has its own copy of the variables in the list
  - variables are not initialized (**`firstprivate`** does that)
  - no change to variable outside of parallel region (**`lastprivate`** does that)
- **`shared(variable list)`**
  - all threads shared the same variable
  - typically initialized outside of the parallel region
  - changes persist outside the parallel region
  - be careful to avoid race conditions