Introduction to High-Performance Computing

Session 05

Introduction to OpenMP

What is OpenMP and why use it?

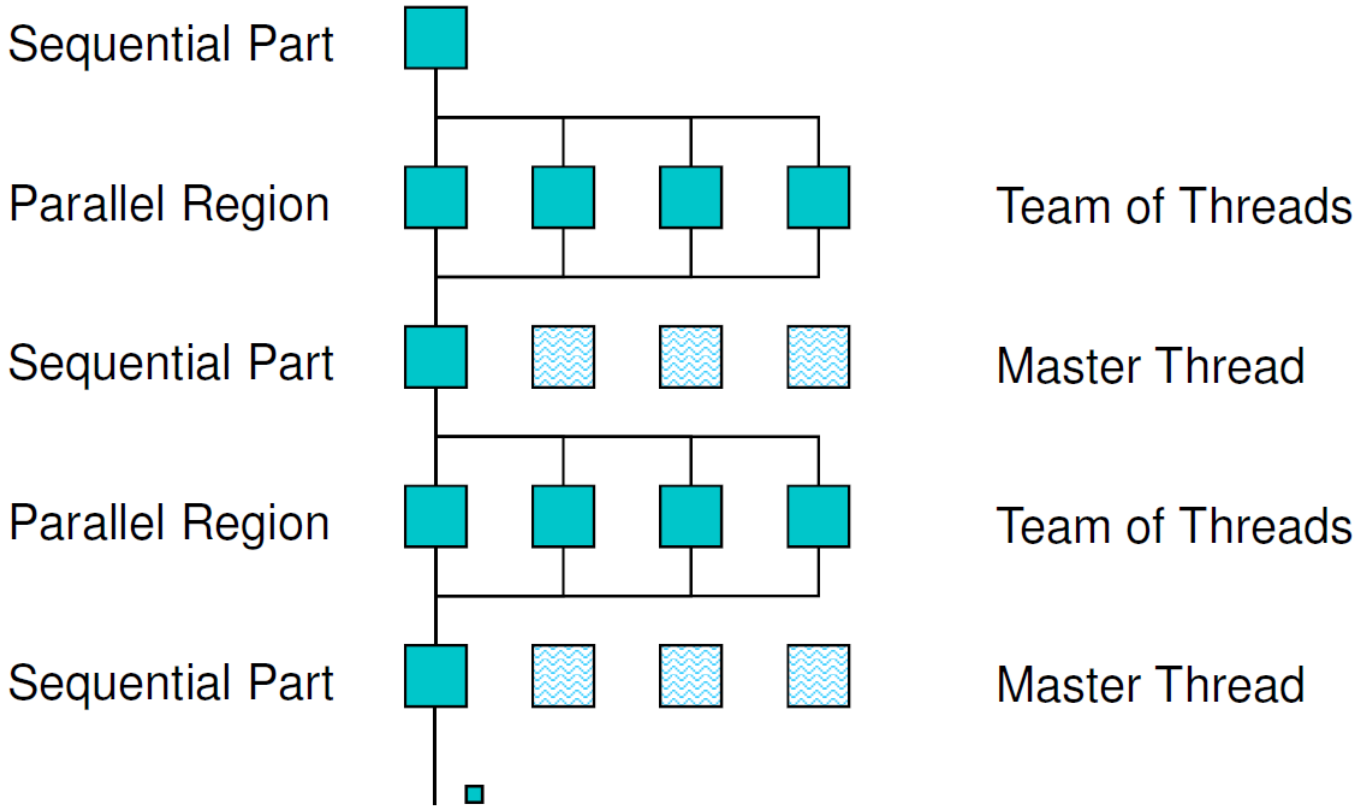
- OpenMP is a standard programming model for shared memory parallelization
  - portable across different shared memory architectures
  - allows incremental parallelization
  - based on compiler directives and a few library routines
  - supports Fortran and C/C++

- 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 ist 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

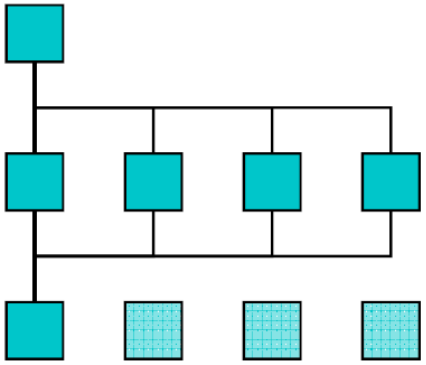
OpenMP Execution Model



OpenMP Parallel Region Construct

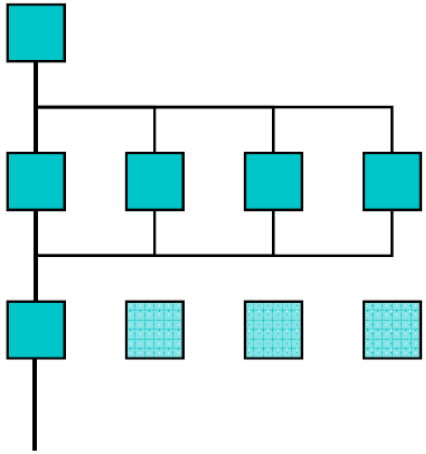
Fortran:

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



C / C++:

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



Example: OMP_HelloWorld

- code available on Stud.IP / HPC Wiki

```
#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
$ 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 -p carl.p
#SBATCH -n 1                    # single task with
#SBATCH -c 8                    # cpus-per-task

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

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

OpenMP Programming

- include library

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

- available library routines
  - setting number of threads
  - getting number of threads
  - getting thread ID
  - wall clock time

```
omp_set_num_threads()
omp_get_num_threads()
omp_get_thread_num()
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;

    // wait for all threads
    #pragma omp barrier
    if (threadID==0) {
      nthreads = omp_get_num_threads();
      cout << "Using " << nthreads << " threads!" << endl;
    }
  } /* end omp parallel */
}
```

Shared and Private Variables

- in OMP_HelloWorld2 threadID is shared among all threads
- race condition
  - every thread is writing to the same memory address
  - final value unpredictable

- 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

Shared and Private Variables

- in OMP_HelloWorld2 threadID is shared among all threads
- race condition
  - every thread is writing to the same memory address
  - final value unpredictable

- solution is to make threadID private

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

Work Sharing Directives

- parallel region to create a team of threads
  - every thread executes the same code
  - example

```
const int N=1000000;
double x[N];
#pragma omp parallel
{
  int threadID = omp_get_thread_num();

  for(int i=0; i<N; i++)
    x[i] = 1./double(threadID+1);
}
```

  - every thread does the same work (and there is a race condition)

Work Sharing Directives

- parallel region to create a team of threads
  - every thread executes the same code
  - example

```
const int N=1000000;
double x[N];
#pragma omp parallel
{
  int threadID = omp_get_thread_num();
  #pragma omp for
  for(int i=0; i<N; i++)
    x[i] = 1./double(threadID+1);
}
```

  - But now every thread does a chunk of work

Work Sharing Directives

- parallel region to create a team of threads
  - every thread executes the same code
  - example

```
const int N=1000000;
double x[N];
#pragma omp parallel for
{
  for(int i=0; i<N; i++)
    x[i] = 1./double(i+1);
}
```

  - Directive can be seperated or combines as needed

Work Sharing Directives

- usable in parallel regions
- directives to specify how the work is distributed
- no synchronization at entry, only at exit (disable with nowait)
- directives
  - for — splits a loop into parallel tasks
  - sections / section — defines a task for one thread
  - single / master — one thread only, no syncronization
  - critial — only one thread at a time
  - …
- Additional clauses e.g. to further specify distribution of work

Example: Mean of Random Numbers

- how to parallelize the program Random.cpp with OpenMP?
  - e.g. the calculation of the mean value

```
// calculate mean value
double mean=0;
for (int i=0; i<NSIZE; i++)
   mean += vec[i];
mean /= NSIZE;
```

OpenMP Directive `critical`

- only one thread at a time can execute critical code block
  - in the example

```
#pragma omp critical
mean += mean_loc;
```

    this ensures mean is calculated without race condition
  - overhead for synchronization and serialization of code block
  - a faster alternative is provided by the atomic directive

```
#pragma omp atomic
mean += mean_loc;
```

  - has limitation on the expressions (critical is more general)

OpenMP Clauses

- the behavior of OpenMP directives can be adjusted using clauses
  - e.g. the following clauses can be used with the for directive:

```
private(list)
firstprivate(list)
lastprivate(list)
```
} how data is treated

```
reduction(reduction-identifier:list)
```
} compiler creates reduction operation

```
schedule([modifier [,modifier]:]kind[, chunk_size])
collapse(n)
ordered[(n)]
```
} how work of loop is distributed among threads

```
nowait
```
} no implicit barrier at the end of loop construct

OpenMP **reduction** Clause

- an alternative (optimal?) solution can be obtained with the reduction clause

```
// calculate mean value
double mean=0;
#pragma omp parallel reduction(+:mean)
{
   #pragma omp for
   for (int i=0; i<NSIZE; i++)
      mean += vec[i];
}
mean /= NSIZE;
```

  - no need of critical section and private variable mean_loc

Code Portability

- it is often desirable to have the same code file being used for serial and OpenMP parallel code
  - use conditional compilation, e.g.

```
#ifdef _OPENMP
  double wt1 = omp_get_wtime();
#endif
```

  - pragmas only have effect when OpenMP option is used at compile time
  - code becomes more difficult to read

OpenMP Summary

- standard for easy shared memory parallelization
- uses compiler directives and some library functions
- based on threads and a fork-join model
- incremental parallelization
- serial and parallel code in one source file
- difference between shared and private data is important
- be careful about race conditions

Practicing

Calculate Pi in Parallel

- modify the program Pi.cpp so that it parallelizes the computation of Pi with OpenMP
  - add a parallel region to the code
  - parallelize the loop so that each thread computes a part of sum (integral)
  - combine the partial sums for the final answer
  - also add a wall clock timer (omp_get_wtime()) and compare the change in CPU and wall clock time for different number of threads

  - All files are on the WIKI page of this course