OpenMP Programming Workshop @LRZ

Manuel Arenaz | February 11-13, 2020

©Appentra Solutions S.L.

Agenda

8:30 - 9:00	*Setup and welcome participants*
9:00 - 9:15	Overview
9:15 - 10:30	**The OpenMP Common Core** **Decomposing code into patterns for parallelization** **Using Parallelware Trainer: A walk-through with PI example**
10:30 - 11:00	Coffee
11:00 - 12:40	Practicals: Examples codes PI, MANDELBROT, HEAT and LULESHmk Worksheet: Parallelizing PI and LULESHmk with OpenMP
12:40 - 13:00	*Close*

What is the OpenMP Common Core?

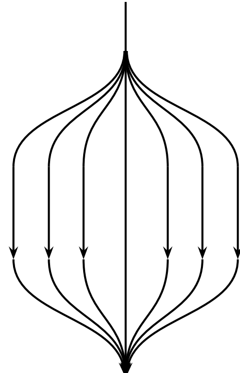
- OpenMP is an API designed for programming shared memory parallel computers.
- OpenMP uses the concepts of threads and tasks
- OpenMP is a set of extensions to Fortran, C and C++
- The extensions consist of:
  - Compiler directives, *e.g. #pragma omp parallel*
  - Runtime library routines, *e.g. omp_get_num_threads()*
  - Environment variables, *e.g. OMP_NUM_THREADS*
- **The OpenMP Common Core consists of the most widely used OpenMP constructs:**
  - While the OpenMP specification contains dozens of constructs, most programs only use 19
  - The first version and the majority of the "Common Core" slides were developed by Tim Mattson, Intel Corp. Many others have contributed

What is the OpenMP common core?

OMP pragma, function, or clause	Concepts	Notes about Parallelware Trainer 1.2
`#pragma omp parallel`	Parallel region, teams of threads, structured block, interleaved execution across threads.	Support for the definition of the parallel region.
`int omp_get_thread_num()` `int omp_get_num_threads()`	Create threads with a parallel region and split up the work using the number of threads and thread ID.	n/a
`double omp_get_wtime()`	Speedup and Amdahl's law. False Sharing and other performance issues.	n/a
`setenv OMP_NUM_THREADS N`	Internal control variables. Setting the default number of threads with an environment variable.	Support for control of number of threads.
`#pragma omp barrier` `#pragma omp critical`	Synchronization and race conditions. Revisit interleaved execution.	Support for synchronization using '*atomic*'.
`#pragma omp for` `#pragma omp parallel for`	Worksharing, parallel loops, loop carried dependencies.	Support for workload sharing.
`reduction(op:list)`	Reductions of values across a team of threads.	Support for parallel scalar/sparse reductions
`schedule(dynamic [,chunk])` `schedule (static [,chunk])`	Loop schedules, loop overheads and load balance.	
`private(list),` `firstprivate(list), shared(list)`	Data environment.	Support for data scoping with private & shared.
`nowait`	Disabling implied barriers on workshare constructs, the high cost of barriers. The flush concept (but not the concept).	
`#pragma omp single`	Workshare with a single thread.	Support for the tasking paradigm.
`#pragma omp task` `#pragma omp taskwait`	Tasks including the data environment for tasks.	Support for the tasking paradigm.

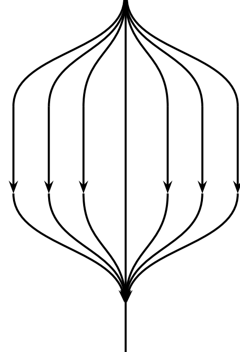
Sequential

Parallel



Sequential

Parallel



```
program mycode

!$omp parallel
.
.
.
.
.
!$omp end parallel
.
.
!$omp parallel
.
.
.
.
!$omp end parallel
```

```
void mycode() {

#pragma omp parallel
{
.
.
.
.
}
.
.
#pragma omp parallel
{
.
.
.
}
```

Directive: *parallel*

- Directive that defines a parallel region, identifying the code region to be parallelized by the compiler.
- It starts parallel execution on the current processor.
- By itself of limited use. It needs to be combined with the work-sharing loop directive, which actually indicates to the compiler how to schedule the loop iterations on the processor.

```
#pragma omp parallel
  for (i=0; i<N; i++)
  {
    y[i] = 2.0f * x[i] + y[i];
  }
```

C and C++:

```
#pragma omp parallel [clause [[,] clause]…]
```

Fortran:

```
!$omp parallel [clause [[,] clause]…]
```

Directive: *for*

- Directive for *work-sharing* in OpenMP
- By itself a parallel region is of limited use, but when paired with the *for* directive the compiler will generate a parallel version of the loop for the processor.
- By using the *for* directive the programmer asserts that the affected loop is safe to parallelize and allows the compiler to select how to schedule the loop iterations on the target processor.
- Clauses are used for correctness or performance

```
#pragma omp parallel for
  for (i=0; i<N; i++)
  {
    y[i] = 2.0f * x[i] + y[i];
  }
```

C and C++:

```
#pragma omp for [clause [[,] clause]…]
```

Fortran:

```
!$omp do [clause [[,] clause]…]
```

Directive: *atomic*

- Ensures only one thread can read/write (i.e. any form of access) a variable at any given time
- Use case: when one or more loop iterations need to access an element in memory at the same time data races can occur.
  - Use when a reduction is present, but the `reduction` operator cannot be used (e.g. for a sparse reduction)

```
#pragma omp parallel for
  for(int i=0;i<N;i++) {
    #pragma omp atomic update
    h[a[i]]+=1;
  }
```

C and C++:

```
#pragma omp atomic [clause [[,] clause]…]
```

Fortran:

```
!$omp atomic [clause [[,] clause]…]
```

Directives: *task & taskwait*

- The directive *task* defines an explicit task that will execute a given code region.
  - The task is added to a pool of tasks managed by the runtime.
- The directive *taskwait* forces a task to wait on the completion of child tasks.
  - Typically the master thread of a parallel region creates child tasks and waits on their completion.
- The programmer is responsible for managing the data scoping of the variables (e.g., private, shared, reduction,...)

```
#pragma omp parallel
#pragma omp master
{
for (int i = 0; i < N; i++) {
#pragma omp task
   { …
   }
}
#pragma omp taskwait
} // end parallel master
```

C and C++:

```
#pragma omp task [clause [[,] clause]…]
#pragma omp taskwait
```

Fortran:

```
!$omp task [clause [[,] clause]…]
!$omp taskwait
```