Parallel Programming

OpenMP

Jan Schönherr
Technische Universität Berlin
School IV – Electrical Engineering and Computer Sciences
Communication and Operating Systems (KBS)
Einsteinufer 17, Sekr. EN6, 10587 Berlin

OpenMP API

- API for programming shared memory architectures
  - Specified for C, C++ and Fortran
- Contents
  - Compiler directives
  - Library routines (e. g. locks, query/set defaults)
  - Environment variables (to change some defaults)
- Features
  - Partial/incremental parallelization of (existing) programs
  - Portable source code
  - Source code still compiles without OpenMP support

OpenMP History

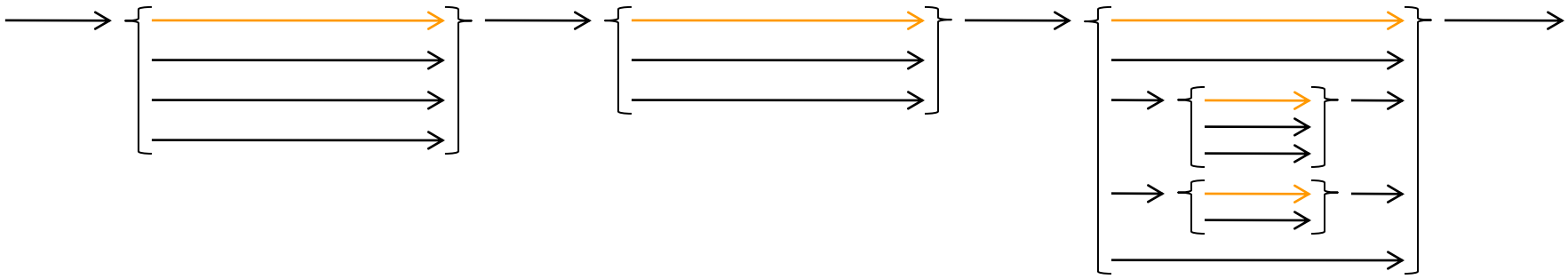
- Early history
  - OpenMP 1.0 in 1997 (Fortran) and 1998 (C/C++)
  - OpenMP 2.0 in 2000 (Fortran) and 2002 (C/C++)
  - OpenMP 2.5 in 2005
- OpenMP 3.0 in 2008
  - Adds *tasks* construct
  - Supported with GCC 4.4, ICC 11.0
  - Currently at OpenMP 3.1 in 2011
- OpenMP 4.0 (currently RC2)
  - Adds support for SIMD, coprocessors

General C/C++ Syntax

- Directives look like
  - #pragma omp <directive> [clause, …]
- Most directives affect the next structured block
  - One statement
  - One compound statement (e. g. multiple statements enclosed in {})
  - Another OpenMP directive
- Library functions are defined in omp.h

Execution Model

- OpenMP is based on Fork/Join parallelism
  - Program execution starts with a single *initial thread*
  - When told so, a thread creates a *team* of itself (the *master*) and zero or more additional threads
  - Later, the team is joined
  - In between is a *parallel region*



Parallel Region

- The *parallel* directive defines a parallel region
  - A team of threads is created
  - Each thread executes the structured block after the directive
  - Only the master continues execution after the structured block
  - Implicit barrier at the end
- Number of threads depends on several factors
  - E. g. your wishes, run-time system, implementation, …
  - It is fixed during the parallel region
- Nesting of parallel regions possible, but potentially problematic
  - Load balancing and overhead might become a problem
  - Implementations may not support arbitrary nesting
    - In that case, no additional threads are created

Worksharing Constructs

- Used inside of parallel regions
- Work is shared among a team
  - i. e. not all members execute everything inside the construct
- Collective character, all members or none must reach it
- Four constructs
  - *for:* Divides loop iterations between members
  - *sections:* Several structured blocks are distributed among members and executed in parallel
  - *single:* A structured block is executed only by one member
  - *workshare* (Fortran only)
- Implicit barrier at the end (can be switched off)

Parallel Loops

- Iterations of a *for*-loop are divided among the team
- Some restrictions regarding the loop
- Different schedules possible
  - *static*: equal-sized chunks are assigned round-robin
  - *dynamic*: equal-sized chunks are assigned to free threads
  - *guided*: chunks are assigned to free threads, starting large and becoming smaller
- Selection of schedule may be delegated to the compiler and/or runtime system

Parallel Statements

- Execute (independent) statements in parallel

- A *sections* construct is divided into multiple *sections*
  - Sections may be executed concurrently, in any order
  - Each section is executed by only one team member
  - Distribution of sections among threads is implementation specific

- Enables static distribution of work

Tasks (since OpenMP 3.0)

- Generalization of Sections, much more flexible
- (Implicit) Shared task queue
  - Every team member can create tasks
  - Every team member can execute tasks
- Dynamic work distribution
  - Degree of parallelism limited by team size
- Limited support for synchronization between tasks
  - Team members may switch between multiple tasks
  - Task may yield execution explicitly
  - Task can wait for completion of all child tasks
- For, e. g., divide & conquer and other cases of nested parallelism

Data-sharing (i)

- Data-sharing clauses specify what happens to variables before, during and after an OpenMP construct
- (Not all clauses are accepted on all directives)

- Variables declared before a construct can be set to:
  - *shared*: all threads access the same memory
  - *private*: each thread accesses its own uninitialized memory
  - *firstprivate*: like *private*, but initialized
  - *reduction*: each thread gets a local, appropriately initialized copy; origin is updated afterwards with the result of a reduction
  - *copyprivate*: update all private origins after a single construct
  - **...** (some more for more specialized uses)

Data-sharing (ii)

- Data-sharing defaults
  - Parallel region: *shared,* with some exceptions
    - e. g. a parallel loop variable is *private*
  - Tasks: mostly *firstprivate*

- Default can be disabled with *default(none)*
  - Prevents errors as you must now specify each attribute explicitly

- Data-sharing attributes on pointers/arrays might not do what you think

Synchronization Constructs

- *barrier*: Execution continues only after all members have reached the barrier
- *master*: Block is only executed by the master of the team
- *critical*: Block is executed by at most one thread at the same time
- *atomic*: A memory location is updated atomically (and since OpenMP 3.1 optionally returned)
- *flush*: Synchronizes a thread's view of shared memory

Behind the scenes of GNU OpenMP

```
#pragma omp parallel
{
        body;
}
```

```
void subfunction (void *data)
{
        use data;
        body;
}

setup data;
GOMP_parallel_start (subfunction, &data, num_threads);
subfunction (&data);
GOMP_parallel_end ();
```

```
#pragma omp for schedule(runtime)
 for (i = 0; i < n; i++)
        body;
```

```
{
        long i, _s0, _e0;
        if (GOMP_loop_runtime_start (0, n, 1, &_s0, &_e0))
                do {
                        for (i = _s0, i < _e0; i++)
                                body;
                } while (GOMP_loop_runtime_next (&_s0, _&e0));
        GOMP_loop_end ();
}
```