OpenMP

Martin Kruliš

Jakub Yaghob

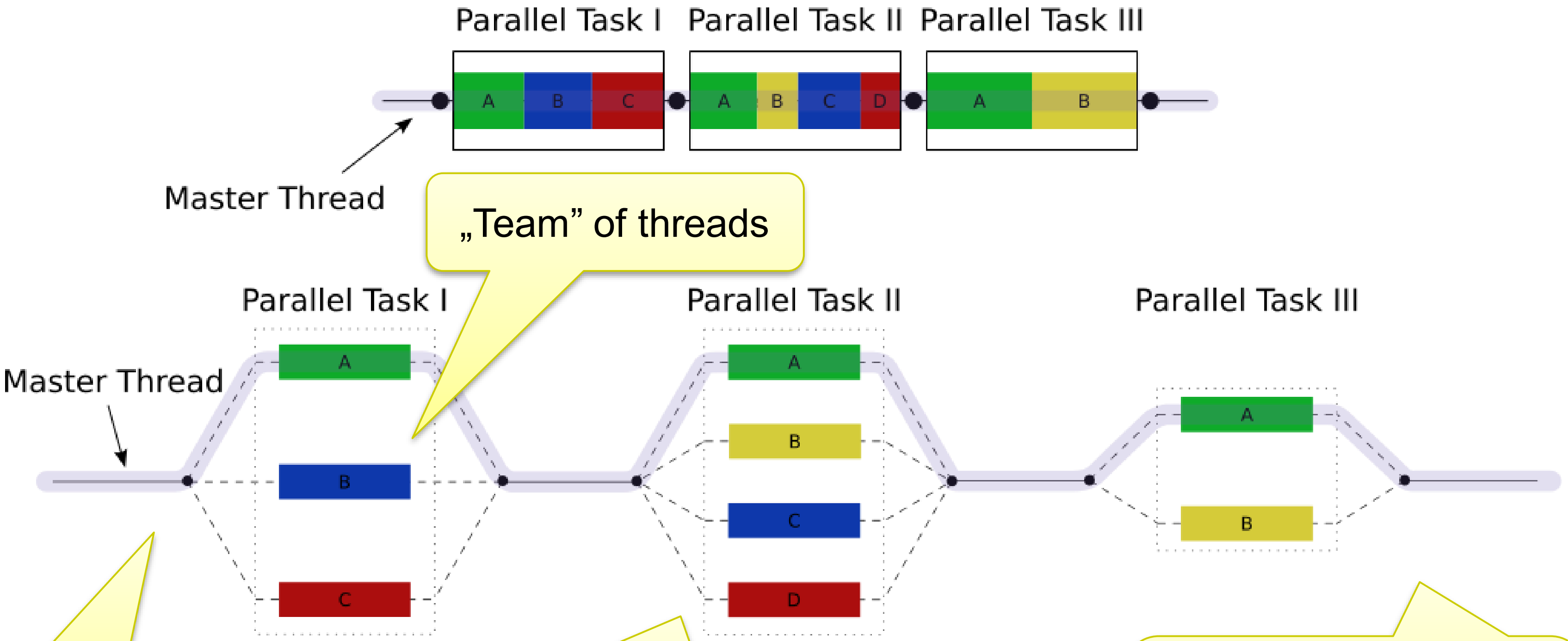
About

- OpenMP (http://www.openmp.org)
  - API for multi-threaded, shared memory parallelism
  - Supported directly by compilers
    - **C/C++** and Fortran
    - Activated by compiler directive (e.g., **`g++ -fopenmp`**)
  - Three components
    - Compiler directives (pragmas)
    - Runtime library resources (functions)
    - Environment variables (defaults, runtime configuration)

History

- Specification versions
  - 1.0 – C/C++ and FORTRAN versions (1997-1998)
  - 2.0 – C/C++ and FORTRAN versions (2000-2002)
    - Supported by MSVC
  - 2.5 – combined C/C++ and FORTRAN (2005)
  - 3.0 – tasks (2008)
  - 3.1 – additional extensions for tasks and atomics (2011)
  - 4.0 – SIMD, cancellation, array sections (2013)
  - 4.5 – Fortran 2003 (2015)
    - Widely available (gcc 6+, Intel compilers, clang)
  - 5.0 – memory model, accelerators (GPU), unified shared memory, iterators, debugger support (2018)
  - 5.1 – full C++20 and Fortran 2008 support, C++ attributes (2020)
    - Partial support (gcc 11, Intel compilers, clang)
  - 5.2 – improvements, refinements
  - 6.0 – should be released 2024

Execution Model



„Team” of threads

Fork/join model

Threads allocated dynamically, but thread pool may be used to optimize management

Nested (recursive) fork/joins may be allowed optionally

Memory model

- Relaxed consistency, shared memory model
- Memory
  - Storage location may by associated with one or more devices
  - Only threads on devices may access it
- Thread private memory
  - Not accessible by other threads
- Load/store are not guaranteed to be atomic
- Memory operations are considered unordered, unless some defined cases

Pragmas

- Basic Syntax

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

  - Code should work without the pragmas (as serial)
  - Pragmas may be used to
    - Spawn a parallel region
    - Divide workload among threads
    - Serialize sections of code
    - Synchronization

Directives as C++ attributes

- Effort to remove C++ preprocessor

  [[omp :: directive(*directive [clause, …]* **)]]**

  or

  [[using omp : directive(*directive [clause, …]* **)]]**

  - Attribute directives that apply to the same statement are unordered
  - Ordering can be imposed

  [[omp :: sequence(*[omp::]directive , [omp::]directive* **)]]**

Directives as C++ attributes – demo

```
[[ omp::sequence(
directive(parallel),
directive(for)) ]]
for(...) {}

#pragma omp parallel
#pragma omp for
for(...) {}
```

Parallel Region

- Spawning a thread team

```
#pragma omp parallel
{
   int tid = omp_get_thread_num();
   // use tid to do thread's bidding
}
```

Spawns one thread per CPU code by default

Thread index within its team (master == 0)

  - Implicit barrier at the end
  - No branching/goto-s that will cause the program to jump in/out to/of parallel blocks
  - Regular branching or function calls are OK

Variable Scope

- Variables Scope
  - Private – a copy per each thread
    - Local variables in a parallel block are implicitly private
  - Shared – all threads share the same variable
    - Synchronization may be required

```
int x, y, id;
#pragma omp parallel private(id), shared(x,y)
{
  id = x + omp_get_thread_num();
  ...
}
```

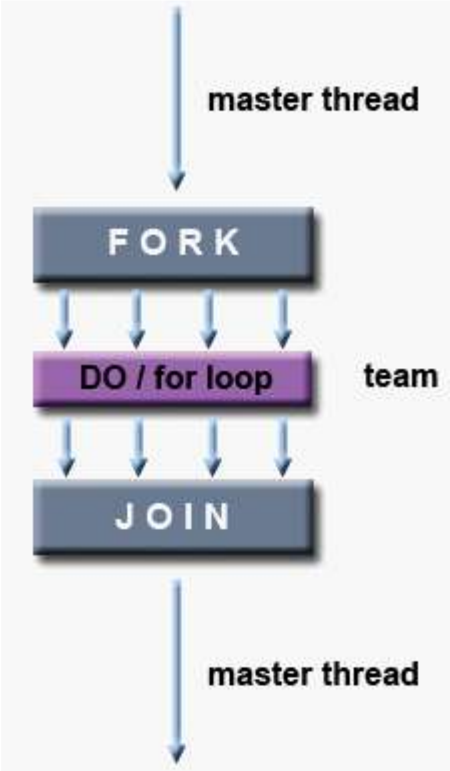
Variable Scope

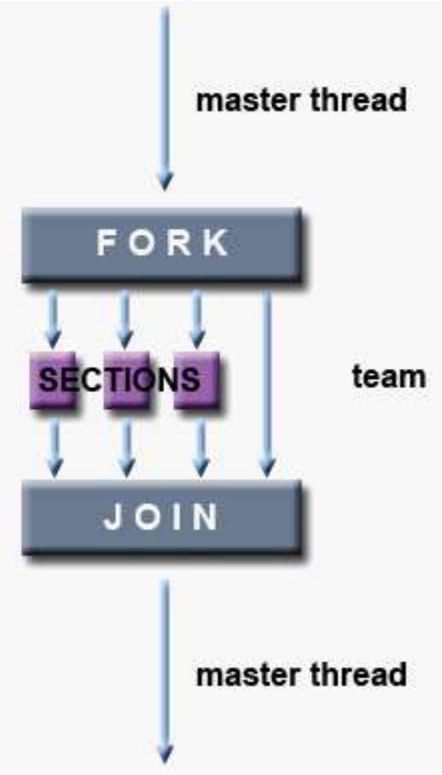
- Private Scope
  - Most variables are private by default
    - Variables declared inside the parallel block
    - All non-static variables in called functions
    - Values are not initialized (at the beginning and the end of the block)
      - Except for classes (default constructor must be accessible)

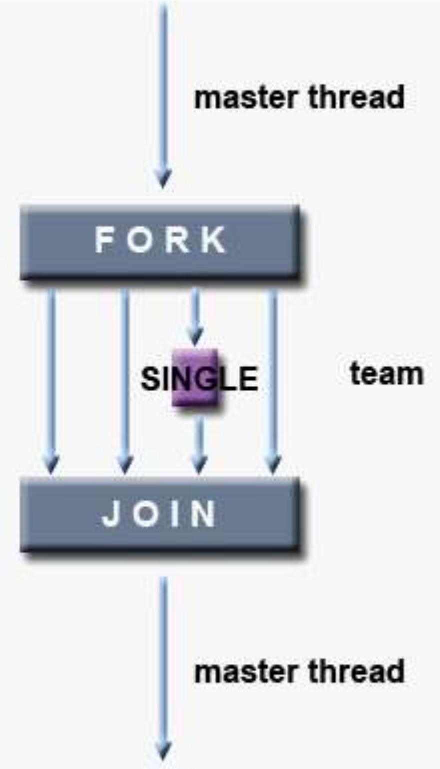
- Other scopes and additional clauses
  - Will be presented later

Work-sharing Constructs

- Divide the work in parallel block







Work-sharing Constructs

- For-loop

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

Spawns the threads

Divide for-loop workload among the threads

  - Additional clauses of **`for`** directive
    - **`schedule(strategy)`** – scheduling (work division)
      - static, dynamic, guided, runtime, auto
    - **`collapse(n)`** – encompass nested for-loops

Work-sharing Constructs

- For-loop
  - **`#pragma omp ordered`**
    - A block inside for-loop that must be executed in exactly the same order, as if the code was serial
  - **`#pragma omp parallel for`**
    - Shorthand for both creating parallel block and apply it on a parallel for-loop
    - May have clauses applied to both **`parallel`** and **`for`** directives
    - Probably the most often used construct in OpenMP

Work-sharing Constructs

- For-loop Pitfalls
  - Use **`#pragma omp for`** outside of **`#pragma omp parallel`** block
    - Has no effect, there is only one thread available!
  - Forgetting the **`for`** itself

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

    - The entire loop is executed by **ALL** threads

Work-sharing Constructs

- Sections
  - Independent blocks of code executed concurrently

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    load_player_data();
    #pragma omp section
    load_game_maps();
    ...
  }
}
```

Work-sharing Constructs

- Single
  - Code executed by single thread from group

```
#pragma omp parallel
{
  #pragma omp for
  for (...) ...

  #pragma omp single
  report_progress();

  #pragma omp for
  ...
}
```

Only one thread reports the progress

Work-sharing Constructs

- Synchronization
  - Implicit barrier at the end of each construct
    - for, sections, single
  - **`nowait`** clause
    - Removes the barrier

Synchronization Constructs

- Synchronization Directives
  - To be used within a parallel block
  - **`#pragma omp master`**
    - Region being executed only by the master thread
      - Similar to **`#pragma omp single`**
  - **`#pragma omp critical`** *`[name]`*
    - Standard critical section with lock guard
      - Only one thread may be in the section at a time
    - If a name is provided, all sections with the same name are interconnected (use the same lock)

Synchronization Constructs

- Synchronization Directives
  - **`#pragma omp barrier`**
    - All threads in a team must meet on a barrier
  - **`#pragma omp atomic`**
    - Followed by a statement like **`x += expr;`** or **`++x;`**
    - Allowed operations: +, *, -, /, &, ^, |, <<,or >>
  - **`#pragma omp flush`** *`[memory-ordering]`*
    - Make sure changes of shared variables become visible
    - Executed implicitly for many directives
      - barrier, critical, parallel, …

Tasks

- Tasks
  - Pieces of code that may be executed by a different thread (within a parallel section)

    ```
    #pragma omp task
    ```

  - Additional clauses
    - **`untied`** – different thread may resume task after yield
    - **`priority(p)`** – scheduling hint
    - **`depend(list)`** – tasks that must conclude first
    - And few other that control when and who can execute the task

Tasks

- Tasks Synchronization
  - **`#pragma omp taskwait`**
    - Wait for all child tasks (spawned in this task)
  - **`#pragma omp taskyield`**
    - Placed as statement inside a task
    - The processing thread may suspend this task and pick up another task
  - **`#pragma omp taskgroup`**
    - Spawning thread will not continue unless all tasks in the group are completed

Data Sharing Management

- Data-related Clauses
  - We already know **`private`** and **`shared`**
  - **`firstprivate`** – similar to private, but the value is initialized using the value of the main thread
  - **`lastprivate`** – value after parallel block is set to the last value, that would be computed in serial processing
  - **`copyprivate`** – used with single block, last value is broadcasted to all threads

Data Sharing Management

- Reduction
  - Variable is private and reduced at the end
  - **`#pragma omp ... reduction(`***`op`***`:`***`list`***`)`**
    - Op represents operation (+, *, &, |, …)
      - Each operation has its own default (e.g., + has 0)
    - List of variables
  - Custom reducers

```
#pragma omp declare reduction (identifier :
typename-list : combiner) [initializer-clause]
```

Synchronization Constructs

- Local Thread Storage

  ```
  #pragma omp threadprivate(list)
  ```

  - Variables are made private for each thread
    - No connection to explicit parallel block
    - Values persist between blocks
  - Variables must be either global or static
  - `copyin(list)` clause (of parallel block)
    - Similar to `firstprivate`
    - Threadprivate variables are initialized by master value

Runtime Library

- Runtime Library
  - `#include <omp.h>` usually required
- Functions
  - `void omp_set_num_threads(int threads)`
    - Set # of threads in next parallel region
  - `int omp_get_num_threads(void)`
    - Get # of threads in current team
  - `int omp_get_max_threads(void)`
    - Current maximum of threads in a parallel region

Runtime Library

- Functions
  - `int omp_get_thread_num(void)`
    - Current thread index within a team (master == 0)
  - `int omp_get_num_procs(void)`
    - Actual number of CPU cores (available)
  - `int omp_in_parallel(void)`
    - True, if the code is executed in parallel
  - `omp_set_dynamic()`, `omp_get_dynamic()`
    - Dynamic ~ whether # of threads in a block can be changed when the block is running

Runtime Library

Synchronization pragmas and atomic operations are better

- Locks
  - Regular and nested locks

    `omp_lock_t`, **`omp_nest_lock_t`**
  - Initialization and destruction

    `omp_init_lock()`, **`omp_init_nest_lock()`**

    `omp_destroy_lock()`, **`omp_destroy_nest_lock()`**
  - Acquiring (blocking) and releasing

    `omp_set_lock()`, **`omp_unset_lock()`**, …
  - Acquiring the lock without blocking

    `omp_test_lock()`, **`omp_test_nest_lock()`**

Environmental Variables

- Environmental Variables
  - May affect the application without recompilation
  - **`OMP_NUM_THREADS`**
    - Max number of threads during execution
  - **`OMP_SCHEDULE`**
    - Scheduling strategy for for-loop construct
    - The loop must have the strategy set to **`runtime`**
  - **`OMP_DYNAMIC`**
    - Enables dynamic adjustment of threads in a block

Thread Team Configuration

- Actual number of threads in parallel region
  1. `if(`*`condition`*`)` clause is evaluated
  2. `num_threads(`*`n`*`)` clause is used if present
  3. `omp_set_num_threads()` value is used
  4. `OMP_NUM_THREADS` env. value is used
  5. System default (typically # of CPU cores)
- Affinity
  - Whether threads are bound to CPU cores
  - `proc_bind(`*`policy`*`)` clause
  - `OMP_PROC_BIND`, `OMP_PLACES`

Nested Parallelism

- Nested Parallel Blocks
  - Implementations may not support it
  - Must be explicitly enabled

    ```
    omp_set_nested(), OMP_NESTED
    ```

  - Nesting depth may be limited

    ```
    omp_set_max_active_levels(),
    OMP_MAX_ACTIVE_LEVELS
    ```

  - More complex to get the right thread ID

    ```
    omp_get_level(), omp_get_active_level(),
    omp_get_ancestor_thread_num()
    ```

SIMD Support

- SIMD Instructions
  - Generated by compiler when possible
  - Hints may be provided by pragmas to loops
    `#pragma omp simd`, **`#pragma omp for simd`**

Use both SIMD and parallel for

  - Important clauses
    - **`safelen`** – max. safe loop unroll width
    - **`simdlen`** – recommended loop unroll width
    - **`aligned`** – declaration about array(s) data alignment
    - **`linear`** – declaration of variables with linear relation to iteration parameter

Accelerator Support

- Offload Support
  - Execute code on accelerator (GPU, FPGA, …)

    ```
    #pragma omp target ...
    ```
  - Slightly more complicated
    - Memory has to be allocated on target device
      - And data transferred there and back
    - Or memory has to be mapped to target device
  - Target device may have more complex thread structure
    - OpenMP introduce thread **`teams`** directive