Outline

OpenMP Standard

- **OpenMP** is a portable **standard** for the programming of **shared memory** systems.
- Standardization similar to the **MPI** standard for the distributed address space programming
  originally designed in 1997
- Naming: Open – *open standard*; MP – *multiprocessing*
- OpenMP provides an API (application programming interface) for C, C++ and Fortran
  OpenMP Application Program Interface, Version 2.5, May 2005; Version 3.0, May 2008; Version 4.0, July 2013
- OpenMP extends sequential programming languages by **language constructs**, **library functions** and **environment variables**
- More information: `http://www.openmp.org`

Literature

- Parallel Programming for Multicore and Cluster Systems, Rauber, Rünger, Springer 2010
- Parallele Programmierung, Rauber, Rünger, Springer 2008
- Multicore: Parallele Programmierung, Rauber, Rünger, 2008
- OpenMP, Informatik im Fokus, Hoffmann, Lienhart, Springer, 2008.
- Using OpenMP - Portable Shared Memory Parallel Programming. Barbara Chapman, Gabriele Jost, Ruud van der Pas, MIT Press, 2008.
- OpenMP Application Program Interface, Version 2.5, May 2005
- OpenMP Application Program Interface, Version 3.0, May 2008, OpenMP Architecture Review Board, http://www.openmp.org

SPEC OMP (OpenMP Benchmark Suite):
`http://www.spec.org/hpg/omp`

Overview

Programming Model

- An OpenMP program consists of a set of **cooperating threads**, created and destroyed by a **fork-join pattern**.
  - The execution of an OpenMP program begins with a **master thread**.
  - The master thread works sequentially until a `parallel` construct is encountered. At the parallel construct, the master thread creates a **team of threads** and becomes the **master** of that team.
- All threads execute the **same program text** (SPMD, specified in the `parallel` construct.
- At the end of the `parallel` **construct**, all threads perform an **implicit barrier synchronization**.
- All threads can access the **shared address space**
  ⇝ **synchronization** on **concurrent accesses** is needed

OpenMP Programming

- For using OpenMP, the **header file** `omp.h` is needed.
- There are constructs for the work sharing between parallel **threads**.
- There are constructs for the synchronization of parallel threads.
- Declaration of **shared** and **private variables**
  Important: The **programmer** has to provide a **correct parallel program specification**, the compiler does not verify the correctness.

A First Example

A vector `array` of length 1000 is initialized **in parallel** by multiple threads:

```
1 const int n = 1000;
2 int array[size];
3 #pragma omp parallel for
4 for (int i = 1 ; i < n ; ++i)
5    array[i] = i;
```

Compiler directive (also called **pragma**):
`#pragma omp parallel for`
⇝ the statement (Line 4) after the `for` loop is executed in parallel, i.e. the statement `array[i] = i;` is executed in parallel.

- Implicit assignment of loop indices to threads

General Form of Compiler Directives

```
#pragma omp <directive> [clauses [[,] clauses] ...]
```

- A compiler directive has to end with a line break.
- The specification of clauses is optional.
- Clauses influence the behavior of directives.
- Different clauses are used for different directives.

OpenMP Compiler Support

- GCC supports OpenMP since Version 4.2
  command line compiler switch **-fopenmp** .
- Intel C++ Compiler
  supports the OpenMP standard and additionally Intel specific directives (since Version 8).
- Sun Studio for Solaris OS supports OpenMP since Version 2.5.

The compiler variable `_OPENMP` tells, whether the compiler supports OpenMP and OpenMP support is switched on.

OpenMP
General Information about OpenMP

Error Handling in OpenMP

- When an an error occurs within an OpenMP directive, the execution of the program is stopped.
- Program behavior is undefined according to the OpenMP standard description in situations like:
  for a parallel region no new threads are started, or a critical section is not protected by synchronization.
- An explicit error message is not necessarily provided.

Overview

OpenMP
Controlling Parallel Computations

Work Sharing Constructs

- Parallelization of `for` loops
- Parallelization of non-iterative parallel regions (`section`)
- Parallel `task` (new concept in OpenMP 3.0)
- Execution explicitly not in parallel (`single`)

Overview

Parallel Region

For the **definition of a parallel region** the following directive is used:

```
#pragma omp parallel [parameter [parameter] …]
          structured block of statements;
```

- This directive creates a **team of threads**
- The **master** of the team is the thread that calls the directive
- Master thread: thread number 0, remaining threads: consecutive thread numbers starting at one
- Each thread of the team (including the master) executes the **code block after the directive**.
- After the execution of the code block all threads except for the master are **destroyed**.
- The **number of threads in a team** is specified with the clause:
  `num_threads (expression)`
  The expression has to return a positive integer value.

Parallel Region: Shared/Private Variables

The following optional **parameters** can be used for the declaration of **shared** or **private** variables:

* **Private variables** are declared through:

  `private (list_of_variables)`

  Effect: Creation of **uninitialized copies** of the listed variables on the **runtime stack** of each thread.
  Every thread accesses only its own **local copy**.
* **Shared Variables** are declared through:

  `shared (list_of_variables)`

  Effect: Every thread of the team will access the **same variable** (same memory position) in the parallel region.

Parallel Region: Default Parameter

The **default parameter** is used for assigning a **default value** (`shared` or `private`) in a parallel region:

- `default (shared)`
  specifies that all variables are **per default shared variables** .
- `default (none)`
  specifies that each variable has to be **explicitly** declared **shared** or **private**.

More OpenMP Functions

- Determination of the **number of threads in a team**:
  Runtime function:
  `int omp_get_num_threads()`
- Obtain the unique **thread identifier** of the calling thread using a **consecutive numbering** of threads:
  `int omp_get_thread_num()`
  The **master thread** has the number **0**.
- **Nesting parallel regions** is possible.
  Default: Only **one thread** executes the **parallel region at the inner-most nesting level**.
  The call of
  `void omp_set_nested (int nested)`
  with `nested != 0` allows for the usage **of more than one thread for the parallel region on the inner-most nesting level**.

OpenMP
Controlling Parallel Computations

Example: Parallel Array Calculations

Array x is distributed to the threads of a parallel region:

```
#include <stdio.h>
#include <omp.h>
int npoints, iam, np, mypoints;
double *x;
int main() {
    scanf("%d", &npoints);
    x = (double *) malloc(npoints * sizeof(double));
    initialize();
    #pragma omp parallel shared(x,npoints) private(iam,np,mypoints)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        mypoints = npoints / np;
        compute_subdomain(x, iam, mypoints);
    }
}
```


OpenMP
Controlling Parallel Computations

Overview

OpenMP
Controlling Parallel Computations

Parallel Loop (I)

Inside a parallel region, iterations of **parallel loops** can be distributed to threads of the team of the parallel region:

```
#pragma omp for [parameter [parameter] …]
for (i = lower_bound; i op upper_bound; incr_expr) {
    //loop body
}
```

- The **iterations** of the loop have to be **independent of each other**.
- The **number of iterations** must be known in advance.
- It is **forbidden to modify** the **iteration variable** `i` in the **loop body**.
  In the loop body, `i` is treated as a **private variable** of the threads involved.

Parallel Loop (II)

- The expression `op` denotes a **boolean operator** from the set $\{<, <=, >, >=\}$.
- The expression `incr_expr` can be one of the following: `++i`, `i++`, `--i`, `i--`, `i=i+incr`, `i=incr+i`, `i=i-incr`
- At the end of a parallel loop, all participating threads are **synchronized implicitly**, which can be **avoided** by using **the parameter** `nowait`.
- The nesting of **for directives** within a parallel region is **forbidden**. But it is possible to nest **parallel regions**.

Scheduling of Loop Iterations (I)

The **mapping of loop iterations** to threads of a parallel region can be specified precisely by using the **scheduling parameter**:

* `schedule (static, block_size)`:
  Static distribution – the iterations are assigned in **blocks** of size `block_size` in a **round-robin** fashion to the threads of the team. If `block_size` is not given, blocks of **almost equal size** are formed and assigned in a **blockwise distribution**.
* `schedule (dynamic, block_size)`:
  Dynamic distribution of **blocks** of size `block_size` to threads. A thread gets a **new block** as soon as the thread finishes the computation of the previously assigned block.
  When the block size is not specified, **blocks of the size of 1** are used.

Scheduling of Loop Iterations (II)

* `schedule (guided, block_size)`:
  Dynamic scheduling of blocks with **decreasing size**. Iterations are assigned in blocks of size

$$\text{block size} = \frac{\text{number of remaining iterations}}{\text{number of threads}} \quad .$$

  For `block_size` > 1, a block contains **never less** than `block_size` iterations, except for the last block.
  Default: `block_size` = 1;
* `schedule (runtime)`:
  The **scheduling** is **specified at runtime**
  Control: **environmental variable** `OMP_SCHEDULE`

Example: Matrix Multiplication with For Directive (I)

Static scheduling: every thread computes a **row block** of the matrix
Realization in **two steps** (initialization and calculation) with **implicit synchronization**.

```
#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;
```

Example: Matrix Multiplication with For Directive (II)

```
int main() {
   read_input(MA, MB);
   #pragma omp parallel shared(MA,MB,MC,size) private(row,col,i) {
      #pragma omp for schedule(static)
      for (row = 0; row < size; row++) {
         for (col = 0; col < size; col++)
            MC[row][col] = 0.0;
      }
      #pragma omp for schedule(static)
      for (row = 0; row < size; row++) {
         for (col = 0; col < size; col++)
            for (i = 0; i < size; i++)
               MC[row][col] += MA[row][i] * MB[i][col];
      }
   }
   write_output(MC);
}
```


Example: Matrix Multiplication – Double Parallel Loop

```
int main() {
   read_input(MA, MB);
   #pragma omp parallel private(row,col,i) {
      #pragma omp for schedule(static)
      for (row = 0; row < size; row++) {
         #pragma omp parallel shared(MA, MB, MC, size) {
            #pragma omp for schedule(static)
            for (col = 0; col < size; col++) {
               MC[row][col] = 0.0;
               for (i = 0; i < size; i++)
                  MC[row][col] += MA[row][i] * MB[i][col];
            }
         }
      }
   }
   write_output(MC);
}
```


Overview

Non-Iterative Worksharing Constructs

Computations of the parallel section can be **assigned explicitly** to threads, with the **section directive**:

```
#pragma omp sections [parameter [parameter]...]
{
   [#pragma omp section]
      structured block
   [#pragma omp section
      structured block
      ...
   ]
}
```

The `sections` directive ends with an **implicit synchronization**.

Single Execution

If instructions in a parallel region may **only** be **executed once**, the `single` can be used:

```
#pragma omp single [parameter[parameter]...]
    structured block
```

Effect: The structured block is **executed by only one thread** of the team.

Syntactic Abbreviations

A parallel region with **only one for directive** can be abbreviated with:

```
#pragma omp parallel for [parameter[parameter]...]
   for (i = lower_bound; i op upper_bound; incr_expr) {
       loop body
   }
```

A parallel region with **only one sections directive** can be abbreviated with:

```
#pragma omp parallel sections [parameter[parameter]...]
{
    [#pragma omp section]
        structured block
    [#pragma omp section
        structured block
        ...
    ]
}
```


Overview

Controlling the Number of Threads (I)

The **number of threads** used for a parallel region can be controlled with the help of **runtime functions**:

- Dynamic adjustment of the number of threads through the runtime system:
  `void omp_set_dynamic (int dynamic_threads)`
  - The value `dynamic_threads` $\neq 0$ **uses a dynamic adjustment**;
  - The value `dynamic_threads` $= 0$ **disables the dynamic adjustment**
    $\rightsquigarrow$ The runtime system will use the **specified number of threads**.

  The function must be called **outside of all parallel regions**.

Controlling the Number of Thread (II)

- Query of the **current status**:
  `int omp_get_dynamic (void)`
  Return value 0, if no dynamic adjustment is set.
- **Set the number of threads** for the subsequent parallel regions:
  `void omp_set_num_threads (int num_threads)`
- If dynamic thread adjustment is enabled: `num_threads` specifies the **maximum number of threads** to be used.
- If dynamic thread adjustment is disabled: `num_threads` denotes the **number of threads** in subsequent parallel regions.

Controlling the Nesting of Threads

- Controlling the **nesting in parallel regions**:
  `void omp_set_nested (int nested)`
  `nested` $= 0$: The inner parallel region is executed by one thread **sequentially**.
  `nested` $\neq 0$: The runtime system can use **more than one** thread for executing the inner parallel region.
- **Query the current status** of the nesting strategy:
  `int omp_get_nested (void)`

Overview

Overview

Critical Section (I)

A **parallel region** is executed by multiple threads accessing the same **shared data**. Thus, there is the need for synchronization in order to protect critical sections for avoiding race conditions. OpenMP offers several possibilities:

- Definition of a **critical section**, which is executed by **only one thread** at a time:

  ```
  #pragma omp critical [(name)]
      structured block
  ```

  Effect: When a thread encounters a critical section, it waits until no other thread executes the corresponding code section.

Critical Section (II)

- A critical section can be executed by only one thread at a time. This refers to all threads of the program, not only to the current team of threads.
- All code sections that are marked with `critical` are considered to be **one critical section**.
  $\leadsto$ Exclusion of parallel execution of different code sections where synchronization is not necessary.
- Finer Granularity:
  - **Critical sections with names** (in round braces): Threads have to wait for entering a **named critical section** until no other thread is executing the critical section with the same name.
  - **Unnamed critical sections** are handled like one critical section with an **unspecified** name.
- **Exiting** a structured block of a critical region with `break`, `continue` or `return` is **not allowed**. Otherwise, a proper release of a critical section is impossible.

Atomic Operations

An **atomic operation** is specified by:

```
#pragma omp atomic
    statement
```

The statement must be of the following form: `x binop= E`
`x++, ++x, x--, --x` where `x` denotes a **variable** and `E` denotes a **scalar expression, not containing** `x` ;
`binop` $\in \{+, -, *, /, \&, \wedge, \backslash, \ll, \gg\}$
Effect: After evaluating the expression `E`, the update of `x` becomes an **atomic operation**, i.e. a **non-interruptable operation**.
Note: The evaluation of `E` is **not an atomic operation**.

Global Reduction Operation

- **Global reduction operations** can be realized by the `reduction` clause:

  `reduction (op:list)`

  with a reduction operator `op` $\in \{+, -, *, /, \&, \wedge, |, \&\&, \|\}$, `list` is a list of **reduction variables** (shared variables).
- **Effect**:
  - Each thread gets a **private copy** of each reduction variable. A thread can assign values to its private copy.
  - At the **end** of the surrounding region, a reduction operation of the local copies of each reduction variable is performed and stored into the corresponding **shared variable**.
  - This means, the shared result variable is computed by performing the given operation `op` with all local copies of the reduction variable.

Overview

5. OpenMP

Barrier-Synchronization

An explicit **barrier synchronization** can be achieved by using the pragma:

```
#pragma omp barrier
```

Effect: All threads of the team wait at the barrier until all other threads reach it. Only when all threads have reached that barrier, the program continues.

Data Consistency

- A **consistent view** of the memory of shared data structures can be obtained explicitly by using

  `#pragma omp flush [(list)]`

  A **list of variables** (`list`) can be provided. Each of these variables gets a consistent value, i.e. all threads see the same value. If the list is empty, all variables get a consistent value.
- A `flush` directive is executed implicitly after executing:
  - a `barrier` directive,
  - when **entering** and after **leaving** a `critical` directive,
  - after **leaving** a `parallel` directive,
  - after leaving a `for`, `sections` or `single` directive, if no `nowait` parameter is given.

Example: Flush Directive

```
#pragma omp parallel private (iam, neighbor) shared (work, sync)
{
  iam = omp_get_thread_num();
  sync[iam] = 0;
  #pragma omp barrier
  work[iam] = do_work();
  #pragma omp flush (work)
  sync[iam] = 1;
  #pragma omp flush (sync)
  neighbor = (iam != 0) ? (iam - 1) : (omp_get_num_threads() - 1);
  while (sync[neighbor] == 0) {
    #pragma omp flush (sync)
    { }
  }
  combine (work[iam], work[neighbor]);
}
```

Master-Thread

Execution of a structured block within a parallel region **only by the master thread**:

```
#pragma omp master
    structured block
```

No other thread executes the specified structured block.

Overview

Simple Lock-Variables (I)

- Locking mechanism with **simple lock variables**
  - **simple lock variables** of type
    `omp_lock_t`
    Simple lock variables can be **locked once**.
  - **Nestable lock variables** of type
    `omp_nest_lock_t`
    Nestable lock variables can be **locked several times**.
- **Initialization** of lock variables:

```
void omp_init_lock (omp_lock_t *lock);
void omp_init_nest_lock (omp_nest_lock_t *lock);
```


Simple Lock Variables (II)

- **Destruction** of lock variables:

```
void omp_destroy_lock (omp_lock_t *lock)
void omp_destroy_nest_lock (omp_nest_lock_t *lock)
```

- **Setting** a lock variable:

```
void omp_set_lock (omp_lock_t *lock)
void omp_set_nest_lock (omp_nest_lock_t *lock)
```

Effect: The executing thread is **blocked until** the lock variable is **available**. Then, the lock variable is set to *locked* by this thread. A **simple lock variable** is available, when **no thread** currently holds it. A **nestable lock variable** is available, when at most the calling thread is holding it.

Simple Lock Variables (III)

- **Releasing** a lock variable:

  void omp_unset_lock (omp_lock_t *lock)
  void omp_unset_nest_lock (omp_nest_lock_t *lock)

  A lock variable can only be **released** by the thread **holding** it.
  A **nestable** lock variable **has to be released as many times** as it was **locked**.
- **Test** of a lock variable:

  int omp_test_lock (omp_lock_t *lock)
  int omp_test_nest_lock (omp_nest_lock_t *lock)

  If the issued lock variable is available, it gets locked.
  If the issued lock variable is locked, the calling thread is not blocked.
  Return value: 1 on successful locking,
  otherwise 0

Time measurement

- **Time measurement** in OpenMP:
  `double omp_get_wtime();`
- Example:

```
{
      double d, time;
          ...
      d=omp_get_time();
      // part to measure
      time=omp_get_time()-d;
}
```