Introduction to OpenMP

part I of III

Christian Terboven <terboven@rz.rwth-aachen.de>
30.07.2013 / Aachen, Germany
Stand: 22.07.2013
Version 2.3
History

- De-facto standard for Shared-Memory Parallelization.
- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN (errata)
- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.
- 05/2008: OpenMP 3.0 release
- 07/2011: OpenMP 3.1 release
- 07/2013: OpenMP 4.0 released

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.
Agenda

- Basic Concept: *Parallel Region*
- The *For* Construct
- The *Single* Construct
- The *Task* Construct
- *Scoping*: Managing the Data Environment
- Synchronization: the *Critical* and *Reduction* Constructs
- More Components of OpenMP
Parallel Region
OpenMP Execution Model

- OpenMP programs start with just one thread: The *Master*.

- *Worker* threads are spawned at *Parallel Regions*, together with the Master they form the *Team* of threads.

- In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.

- Concept: *Fork-Join*.
- Allows for an incremental parallelization!
OpenMP’s machine model

- **OpenMP: Shared-Memory Parallel Programming Model.**

  All processors/cores access a shared main memory.

  Real architectures are more complex, as we will see later / as we have seen.

  Parallelization in OpenMP employs multiple threads.
Parallel Region and Structured Blocks

- The parallelism has to be expressed explicitly.

C/C++

```c
#pragma omp parallel
{
    ...
    structured block
    ...
}
```

Fortran

```fortran
!$omp parallel
    ...
    structured block
    ...
$!omp end parallel
```

- **Structured Block**
  - Exactly one entry point at the top
  - Exactly one exit point at the bottom
  - Branching in or out is not allowed
  - Terminating the program is allowed (abort / exit)

- **Specification of number of threads:**
  - Environment variable:
    ```
    OMP_NUM_THREADS=...
    ```
  - Or: Via `num_threads` clause:
    ```
    add num_threads(num) to the parallel construct
    ```
Hello OpenMP World
Hello orphaned World
Starting OpenMP Programs on Linux

- From within a shell, global adjustment of the number of threads:
  
  ```
  export OMP_NUM_THREADS=4
  ./program
  ```

- From within a shell, one-time adjustment of the number of threads:
  
  ```
  OMP_NUM_THREADS=4  ./program
  ```

- Intel Compiler on Linux: asking for more information:
  
  ```
  export KMP_AFFINITY=verbose
  export OMP_NUM_THREADS=4
  ./program
  ```
For Construct
For Worksharing

- If only the *parallel* construct is used, each thread executes the Structured Block.
- **Program Speedup: Worksharing**
- OpenMP’s most common Worksharing construct: *for*

```
C/C++
int i;
double a[N], b[N], c[N];
#pragma omp parallel for
for (i = 0; i < N; i++)
{
    a[i] = b[i] + c[i];
}

Fortran
INTEGER :: i
INTEGER, DIMENSION(N) :: a,b,c
!$omp parallel do
DO i = 1, N
    a[i] = b[i] + c[i];
END DO
```

- Distribution of loop iterations over all threads in a Team.
- Scheduling of the distribution can be influenced.

- Loops often account for most of a program’s runtime!
Worksharing illustrated

Pseudo-Code
Here: 4 Threads

Thread 1
\[
\text{do } i = 0, 24 \\
\quad a(i) = b(i) + c(i) \\
\text{end do}
\]

Thread 2
\[
\text{do } i = 25, 49 \\
\quad a(i) = b(i) + c(i) \\
\text{end do}
\]

Thread 3
\[
\text{do } i = 50, 74 \\
\quad a(i) = b(i) + c(i) \\
\text{end do}
\]

Thread 4
\[
\text{do } i = 75, 99 \\
\quad a(i) = b(i) + c(i) \\
\text{end do}
\]

Serial
\[
\text{do } i = 0, 99 \\
\quad a(i) = b(i) + c(i) \\
\text{end do}
\]
Vector Addition
The Single Construct
### The Single Construct

**C/C++**

```c
#pragma omp single [clause]
... structured block ...
```

**Fortran**

```fortran
!$omp single [clause]
... structured block ...
!$omp end single
```

- **The single construct specifies that the enclosed structured block is executed by only on thread of the team.**
  - It is up to the runtime which thread that is.

- **Useful for:**
  - I/O
  - Memory allocation and deallocation, etc. (in general: setup work)
  - Implementation of the single-creator parallel-executor pattern as we will see now…
Task Construct
Sudoku for Lazy Computer Scientists

- Lets solve Sudoku puzzles with brute multi-core force

1. Find an empty field
2. Insert a number
3. Check Sudoku
4. a) If invalid: Delete number, Insert next number
4. b) If valid: Go to next field

![Sudoku puzzle grid]

RZ: Christian Terboven
The Task Construct

Each encountering thread/task creates a new Task

- Code and data is being packaged up
- Tasks can be nested
  - Into another Task directive
  - Into a Worksharing construct

Data scoping clauses:

- shared(list)
- private(list)  firstprivate(list)
- default(shared / none)
This parallel algorithm finds all valid solutions

1. Search an empty field
2. Insert a number
3. Check Sudoku
   - If invalid: Delete number, Insert next number
   - If valid: Go to next field

```
#pragma omp task
#pragma omp taskwait
```
OpenMP parallel region creates a team of threads

```c
#pragma omp parallel
{
  #pragma omp single
  solve_parallel(0, 0, sudoku2, false);
} // end omp parallel
```

- Single construct: One thread enters the execution of `solve_parallel`
- the other threads wait at the end of the `single` ...
  - ... and are ready to pick up tasks „from the work queue“

Syntactic sugar (either you like it or you do not)

```c
#pragma omp parallel sections
{
  solve_parallel(0, 0, sudoku2, false);
} // end omp parallel
```
The actual implementation
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
    if (!sudoku->check(x, y, i)) {
        #pragma omp task firstprivate(i,x,y,sudoku)
        {
            // create from copy constructor
            CSudokuBoard new_sudoku(*sudoku);
            new_sudoku.set(y, x, i);
            if (solve_parallel(x+1, y, &new_sudoku)) {
                new_sudoku.printBoard();
            }
        }
    }
} // end omp task

#pragma omp taskwait
wait for all child tasks

#pragma omp taskwait
Performance Evaluation

Sudoku on 2x Intel® Xeon® E5-2650 @2.0 GHz

- Intel C++ 13.1, scatter binding

Graph showing the runtime [sec] for 16x16 Sudoku with varying #threads.
Scoping
Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.

- Scoping in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region, *firstprivate* for Tasks.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for each thread
    - *firstprivate*: Initialization with Master’s value
    - *lastprivate*: Value of last loop iteration is written back to Master
  - Static variables are *shared*
Privatization of Global/Static Variables

- Global / static variables can be privatized with the \textit{threadprivate} directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword \_
      \_
      \_thread (GNU extension)

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int i;</td>
<td>SAVE INTEGER :: i</td>
</tr>
<tr>
<td>#pragma omp threadprivate(i)</td>
<td>!$omp threadprivate(i)</td>
</tr>
</tbody>
</table>
Privatization of Global/Static Variables

- Global / static variables can be privatized with the `threadprivate` directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - `TlsAlloc` (Win32-Threads), `pthread_key_create` (Posix-Threads), keyword `__thread` (GNU extension)

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static int i;</code></td>
<td><code>SAVE INTEGER :: i</code></td>
</tr>
<tr>
<td><code>#pragma omp threadprivate(i)</code></td>
<td><code>!$omp threadprivate(i)</code></td>
</tr>
</tbody>
</table>

Really: try to avoid the use of threadprivate and static variables!
Tasks in OpenMP: Data Scoping

- Some rules from *Parallel Regions* apply:
  - Static and Global variables are shared
  - Automatic Storage (local) variables are private

- If shared scoping is not derived by default:
  - Orphaned Task variables are firstprivate by default!
  - Non-Orphaned Task variables inherit the shared attribute!

→ Variables are firstprivate unless shared in the enclosing context

- So far no verification tool is available to check Tasking programs for correctness!
Synchronization
Can all loops be parallelized with `for`-constructs? No!

- Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent. BUT: This test alone is not sufficient:

```c/c++
int i;
double s, a[N];
#pragma omp parallel for
for (i = 0; i < N; i++)
{
    s = s + a[i];
}
```

- **Data Race:** If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).
Synchronization: Critical Region

- A Critical Region is executed by all threads, but by only one thread simultaneously (Mutual Exclusion).

C/C++

```c
#pragma omp critical (name)
{
    ... structured block ...
}
```

- Do you think this solution scales well?

C/C++

```c
int i;
double s, a[N];
#pragma omp parallel for
for (i = 0; i < N; i++)
{
    #pragma omp critical
    { s = s + a[i]; }
}
```
It's your turn: Make It Scale!

```c
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < N; i++)
    {
        s = s + a[i];
    }
}
```

- do i = 0, 24
  s = s + a(i)
  end do

- do i = 25, 49
  s = s + a(i)
  end do

- do i = 0, 99
  s = s + a(i)
  end do

- do i = 50, 74
  s = s + a(i)
  end do

- do i = 75, 99
  s = s + a(i)
  end do

} // end parallel
In a reduction-operation the operator is applied to all variables in the list. The variables have to be shared.

- reduction(operator: list)

- The result is provided in the associated reduction variable

```
C/C++
int i;
double s, a[N];
#pragma omp parallel for reduction(+:s)
for(i = 0; i < N; i++)
{
    s = s + a[i];
}
```

- Possible reduction operators with initialization value:
  + (0), * (1), − (0),
  & (~0), | (0), && (1), || (0),
  ^ (0), min (least number), max (largest number)
The Barrier and Taskwait Constructs

- **OpenMP barrier (implicit or explicit)**
  - All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit
  ```c/c++
  #pragma omp barrier
  ```

- **Task barrier: taskwait**
  - Encountering Task suspends until child tasks are complete
    - Only direct childs, not descendants!
  ```c/c++
  #pragma omp taskwait
  ```
More Components of OpenMP
## Components of OpenMP

<table>
<thead>
<tr>
<th>Directives</th>
<th>Runtime Functions</th>
<th>Env. Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Section Expressions</td>
<td>Number of Threads</td>
<td>Number of Threads</td>
</tr>
<tr>
<td>Parallel Region</td>
<td>Thread ID</td>
<td>Scheduling Type</td>
</tr>
<tr>
<td>Worksharing Constructs</td>
<td>Dynamic Thread Adjustment</td>
<td>Dynamic Thread Adjustment</td>
</tr>
<tr>
<td>SIMD Constructs</td>
<td>Cancellation Status</td>
<td>Thread Affinity Places</td>
</tr>
<tr>
<td>Device Constructs</td>
<td>Nested Parallelism</td>
<td>Nested Parallelism</td>
</tr>
<tr>
<td>Tasking</td>
<td>Schedlue</td>
<td>Stacksize</td>
</tr>
<tr>
<td>Synchronization Constructs</td>
<td>Active Levels</td>
<td>Idle Thread</td>
</tr>
<tr>
<td>Cancellation Constructs</td>
<td>Device Selection</td>
<td>Active Levels</td>
</tr>
<tr>
<td>Declaration Constructs</td>
<td>Nesting Level</td>
<td>Thread Limit</td>
</tr>
<tr>
<td>Memory Flush</td>
<td>Ancestor Thread</td>
<td>Cancellation</td>
</tr>
<tr>
<td>Data-sharing attributes</td>
<td>Team Size</td>
<td>Default Printout</td>
</tr>
<tr>
<td></td>
<td>Wallclock Timer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Locking</td>
<td></td>
</tr>
</tbody>
</table>

**Red color indicates new addition to OpenMP 4.0**
The Worksharing Constructs

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or non at all
- No implied barrier on entry; implied barrier on exit
- A worksharing construct does not launch any new threads
### Some Additional Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma omp master {&lt;code-block&gt;}</code></td>
<td>There is no implied barrier on entry or exit !</td>
</tr>
<tr>
<td><code>!$omp master &lt;code-block&gt; !$omp end master</code></td>
<td></td>
</tr>
<tr>
<td><code>#pragma omp critical [(name)] {&lt;code-block&gt;}</code></td>
<td>Very useful to avoid data races</td>
</tr>
<tr>
<td><code>!$omp critical [(name)] &lt;code-block&gt; !$omp end critical [(name)]</code></td>
<td>Also supports fine tuning controls</td>
</tr>
<tr>
<td><code>#pragma omp atomic</code></td>
<td></td>
</tr>
<tr>
<td><code>!$omp atomic</code></td>
<td></td>
</tr>
</tbody>
</table>
Appendix A: make/gmake
**make / gmake**

- **make**: “smart” utility to manage compilation of programs and much more
  - automatically detects which parts need to be rebuild
  - general rules for compilation of many files
  - dependencies between files can be handled

- **Usage:**
  ```make <target>  or  gmake <target>```

- **Rules:**
  ```
  target ... : prerequisites ...
  < tab > command
  < tab > ...
  ```

  - target: output file (or only a name)
  - prerequisites: input files (e.g. source code files)
  - command: action to be performed