OpenMP - A Very Short Introduction

Version 1.1

Paulo Pedreiras

DETI/UA/IT

September 8, 2021

Outline

1. Section 1: Introduction
2. Section 2: Overview
3. Section 3: Directives
4. Section 4: Synchronization
5. Bibliography

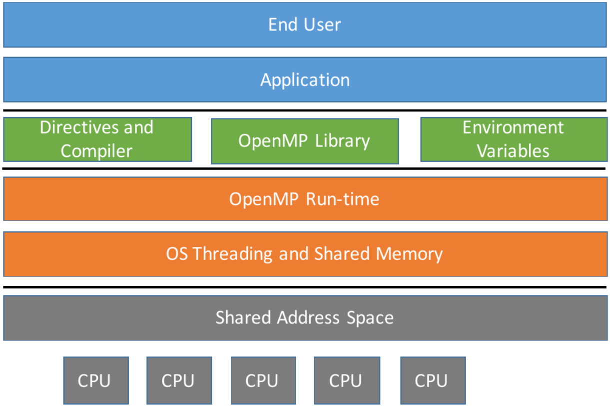
Introduction

OpenMP:An API for Writing Multithreaded Applications

- Compiler Directives + Library Functions + Shared Memory Model
- Available for C and C++ (among other languages) and multiple toolchains
- Greatly simplifies the programmer task
  - Minimizes code changes required to move from single process to multiprocess
  - Easy synchronization
  - ...

OpenMP structure

OpenMP stack



General aspects

A few aspects ...

- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the general form:
    #pragma omp construct [clause [clause]...]
- Include file
  - “#include ”omp.h“
- Compiling with GCC: just add ”-fopenmp“
  - If missed pragmas are ignored
- Additional library functions
  - Control number of threads, get thread ID, etc.
  - E.g. omp_get_num_threads(), omp_get_thread_num();

Simple example

Lets start with a simple example ...

```
// Summs two vectors, sequential execution
#include <stdio.h>
#include <omp.h>
...

main() {
    ... (init and fill vectors)

    start = omp_get_wtime();
    // Sum the vector elements
    for(i=0;i<size-1;i++)
        V3[i] = V1[i]+V2[i];
    stop = omp_get_wtime();
    printf("Time: %f (ms)\n",(stop-start)*1000);
}
```

Note:

For 1000000 size float vectors it takes 4.929833 ms on my computer (Intel Core I7, 2.6 GHz).

Simple example

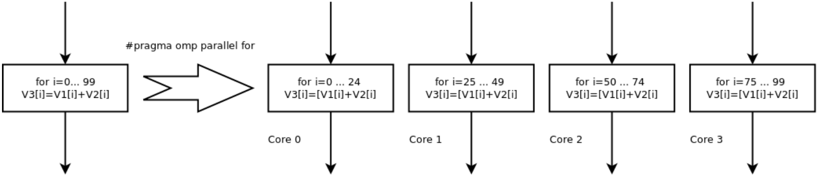
Using OpenMP ...

```
// Summs two vectors, parallel execution
#include <stdio.h>
#include <omp.h>
...

main() {
    ... (init and fill vectors)

    start = omp_get_wtime();
    // Sum the vector elements
    #pragma omp parallel for
    for(i=0;i<size-1;i++)
        V3[i] = V1[i]+V2[i];
    stop = omp_get_wtime();
    printf("Time: %f (ms)\n",(stop-start)*1000);
}
```

Note:

- In the same conditions I get 0.899622 ms, instead of 4.929833 ms!
- One single line added!

Simple example



The magic comes from ...

- The "For" cycle is **automatically** split by NTHREADS threads (by default in GCC set to number of cores)
- Code executed in parallel
- OpenMP handles the counters, creates the threads, synchronizes the execution, ...

OpenMP Directives

OpenMP is based on the use of directives that in C/C++ do correspond to preprocessor pragma commands starting with "omp".

#pragma omp construct [clause [clause] ...]

Example:

```
#pragma omp parallel numthreads(4)
```

These directives apply to the next statement. The syntax allows the serial execution just by ignoring the directives.

The OpenMP directive acts over **structured block**

- A structured block is an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom
- A structured block may contain another OpenMP construct.
- It is allowed to use exit()
- The compiler transforms those directives in calls to the OpenMP runtime API

A few commonly used OpenMP functions

- **omp_set_num_threads(NTHREADS);**
  - Sets the number of threads to NTHREADS.
    Default for GCC is the number of cores.
- **omp_get_num_threads(NTHREADS);**
  - Returns the number of threads currently set.
- **omp_get_thread_num();**
  - Returns the thread number.
- **omp_in_parallel();**
  - Detects if code is inside a parallel region
- **omp_num_procs();**
  - Returns the number of processors
- **omp_get_wtime();**
  - Returns the current time in seconds (double precision float). Useful for measuring execution times.

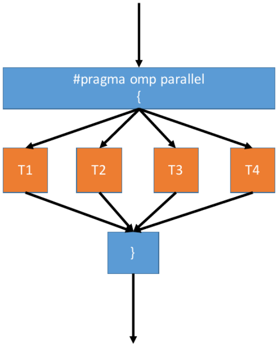
The Parallel Directive

- The parallel directive introduces a parallel region executed by all the OpenMP processes.
- The contained structured block is affected by that directive.

```
#pragma omp parallel
{
    <code of structured block>
}
```

- By default the number of threads created in is equal to the available processors. Optionally it can be specified by the programmer.

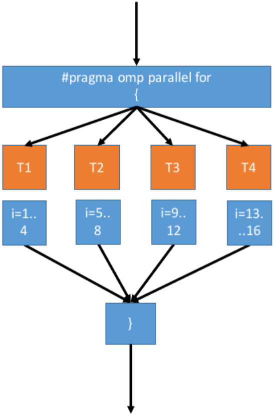
```
#pragma omp parallel numthreads(4)
{
    < code of structured block>
}
```



The For Directive

- Used to parallelize a for-loop
- It is the most common directive. It can be specified inside a parallel directive, or via the short-hand "parallel for".
- Loop index managed automatically

```
#pragma omp parallel for
{
    <for loop structured block>
}

or,

#pragma omp parallel
#pragma omp for
{
    <for loop structured block>
}
```



The For Directive

For Directive - additional aspects

- There are a few options for executing the threads that can be controlled via clauses in the directive
  - schedule(static [,k]) - divides the loop in chunks of size k. If omitted k is set to $n/NTHREADS$.
    - The set of loop iterations assigned to each thread is computed a priori
    - Default behavior.Non optimal!
  - schedule(dynamic [,k]) - divides the loop in chunks as before.
    - Threads are assigned dynamically. When a thread finishes it is assigned a new chunck
    - Better use of processors when iteration take different amounts of time, but higher overhead
  - schedule(guided) - similar to a dynamic schedule, but the chunk size changes as the program runs. Begins with big chunks but adjusts to smaller chunk sizes if the workload is imbalanced
  - etc.
- There is no guaranteed that the elements of the loop are executed in a specific order, unless the ordered clause is added.
- The single directive dictates that a piece of code is executed only by a single thread.

Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

A few synchronization primitives

critical Only one thread can enter the critical region

atomic Atomic provides mutual exclusion but only applies to the update of a memory location

barrier Barrier: Each thread waits until all threads arrive.

- There is an implicit barrier at the end of "parallel for" constructs and parallel regions
- Implicit barriers can be override by a "nowait" caluse

Synchronization Example

Parallelizing the computation of the average of the elements of a vector ...

```
...
#pragma omp parallel for
for(k=0;k<VSIZE;k++)
    #pragma omp critical
    avg += V1[k];
vg = avg / VSIZE;
```

Reduction

- Combining values into a single accumulation variable (as in the previous slide) creates a dependency among threads that must be removed
- It a common situation called **reduction** .
- OpenMP provides a **reduction(op:list)** clause that handles this issue
- Inside a parallel or a work-sharing construct:
  - A local copy of each **list** variable is made. Initialization is set by **op** . E.g. 0 for "+").
  - Compiler finds standard reduction expressions containing "op" and uses them to update the local copy.
  - Local copies are reduced into a single value and combined with the original global value.

Reduction example

Code gets greatly simplified!

```
...
#pragma omp parallel for reduction(+:avg)
for(k=0;k<VSIZE;k++)
    avg += V1[k];
avg = avg / VSIZE;
```

Data environment

Default storage attributes

- Global variables are SHARED among threads
- Stack variables in sub-programs called from parallel regions are PRIVATE
- Automatic variables within a statement block are PRIVATE.
- "for" index in "parallel for" constructs are PRIVATE.
- Shared variables can be made PRIVATE via a private clause

Example:

```
...
int main()
 {
     int i, j; /* PRIVATE */
     int V[100]; /* SHARED */
     #pragma omp parallel private(j)
     {
         int k = 1; /* PRIVATE */
         #pragma omp for
             for (i=0; i<VSZIE; i++)
 ...
```

Bibliography

- OpenMP Application Programming Interface Version 5.1 (November 2020), available at https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf
- Hands-On Introduction to OpenMP, Mattson and Meadows, available at https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf