INSTITUTE OF COMMUNICATION, INFORMATION AND PERCEPTION TECHNOLOGIES

Scuola Superiore Sant'Anna

OpenMP and GPU Programming

OpenMP

Emanuele Ruffaldi

PERCeptual RObotics Laboratory, TeCIP
Scuola Superiore Sant'Anna
Pisa,Italy
e.ruffaldi@sssup.it

April 5, 2016

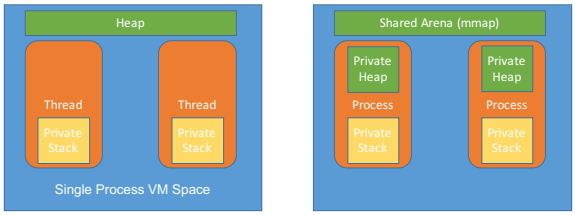
Parallel Programming

Parallel programming refers to the computation of a problem by decomposing across multiple processors performing the same task. This is complementary to concurrent programming in which different processors perform related task acting on possibly shared resources.
Parallel programming is becoming important even on Desktop or Laptop machines due to the increased availability of multicore CPUs.
Parallel programming can be organized between shared memory or message passing:

- shared memory: the processors share task memory
- message passing: all (most) data is exchanged via some form of channels

Memory Models

This diagram shows the memory models:



Parallel Programming in C/C++

There are two approaches for this type of programming in C/C++ and in general programming languages: language-based features vs library based features.

- Language-based features introduce some special syntax for expressing parallelism and concurrency (OpenMP, Cilk)
- Library-based features use library functions for expressing paralellism, at different levels of abstraction (pthreads, Intel TBB)

In this course we address OpenMP because it is standard based, easy to use and widely available on most C/C++ programming tools.

OpenMP Programming Model

Programming Model:

- Language support and library for shared memory parallel programming
- Multiple threads, one per-core, within the same address space
- Different levels of parallelization
- Specific for C/C++ and Fortran

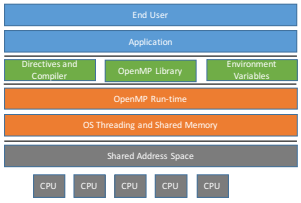
Goals:

- Minimize transformation from single process to multiprocess application
- Provide easy to use synchronization without message passing

OpenMP Solution Stack

The following picture presents the OpenMP solution stack with the User Application on top, then the OpenMP development time functionalities, the run-time, OS services and then the shared memory processors.

Stack



OpenMP History

- OpenMP has been standardized from the OpenMP architecture review board in 1997 first for Fortran then C++
- 2.0 arrived in 2000 and then 2.5 in 2005.
- 3.0 introduced Tasks in 2008
- 4.0 introduced a large set of features in 2013
- last one is 4.5 added in 2015

Compiler Support

Each compiler has adopted OpenMP at different stages of the standard:

- GCC supports OpenMP 4.0 since gcc 4.9, 3.0 since gcc 4.4, 2.5 since gcc 4.2 (enabled with -fopenmp)
- Microsoft Visual C++ supports OpenMP 2.5 since version 2005 (enabled with /openmp)
- CLang supports OpenMP 3.1 since version 3.8 (but not in stock OSX CLang)

Minimal Example

Before entering a systematic analysis of OpenMP it is worth getting an example:

```
#include <cmath>
#include <chrono>
#include <iostream>

int main()
{
const int size = 32768;
double sinTable[size];

#pragma omp parallel for
for(int n=0; n<size; ++n)
sinTable[n] = std::sin(2 * M_PI * n / size);

return 0;
}
```

The environment variable OMP_NUM_THREADS controls the number of parallel threads to be instantiated so different situations can be tested.

- Try the example using different OMP_NUM_THREADS and time it
- How we can measure time precisely under Linux?

Building Example

These examples can be easily build using gcc with "-fopemp" flag, and then the library is provided by libgomp.

```
find_package(OpenMP)
if (OPENMP_FOUND)
set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
endif()
add_definitions(--std=c++11)
link_libraries(gomp)
```

Measuring Time

Sometime we'll need time measurement. Under Linux the main option is clock_gettime that supports different types of clock sources:

- CLOCK_REALTIME
- CLOCK_MONOTONIC
- CLOCK_PROCESS_CPUTIME_ID
- CLOCK_THREAD_CPUTIME_ID

Then we'll use some code for time difference. This is good but not portable so let's use C++11 standard:

```
#include <chrono>
auto now = std::chrono::high_resolution_clock::now();
auto dt = [] (decltype(now) now) { return std::chrono::duration_cast<std::chrono::microseconds>(std::
...
std::cout << "elapsed us: " << dt(now) << std::endl;
}
```

The Linux clocks are reflected by the C++11 clock families:

- high_resolution_clock
- system_clock
- steady_clock

Luckly OpenMP provides omp_get_wtime() that returns in double the number of elapsed seconds.

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 num_threads(4)
```

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

- In particular the OpenMP directive acts over a "structured block" that is: an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom, or another OpenMP construct.

An OpenMP enabled compiler uses these directives to transform the code also by adding the invocation to the run-time API.
In the following we'll discuss the different directives.

Compilation and Run-time Detection

The presence of OpenMP can be detected at compile-time via the _OPENMP define. The OpenMP run-time can be accessed via the include file omp.h.
The most classic solution is the estimation of the number of available threads via *_get_thread_num*().

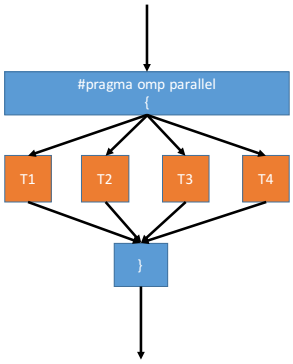
Parallel Directive

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 >
}
```

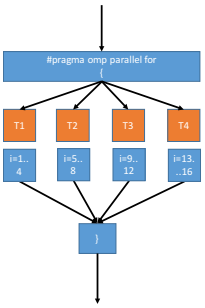
At high-level OpenMP works by dealing with a number of threads organized in a Team. At start there is only one thread in the team and then the parallel directive activates other threads in the team. When not specified there is a number of thread in the team equal to the available processors. Otherwise it can be specified.

```
#pragma omp parallel num_threads(3)
{
< code of structured block >
}
```



For Directive

The for directive is used to parallelize a for-loop with some control over the loop is assigned to the tasks. This is the most common directive and it can be specified inside a parallel directive, or via a short-hand "parallel for". The for loop following the directive is split among the threads of the Team.



For Directive

- There are different options for executing the threads in the Team and they can be controlled via clauses in the directive
  - schedule(static [,k]) - divides the loop in chunks of size k, if k is not specified k is n/nthreads. Tasks are pre-assigned to the threads
  - schedule(dynamic [,k]) - divides the loop in chunks as before, but tasks are assigned dynamically
  - schedule(guided) - k starts large and it is automatically decreased
  - schedule(auto) is compiler dependent
  - schedule(runtime) - uses the variable OMP_SCHEDULE
- Also there is no guaranteed that the elements of the loop are executed in a specific order, unless the **ordered** clause is added.
- The **ordered** directive dictates that a piece of code is executed in the correct order.
- The **single** directive dictates that a piece of code is executed only by a single thread.

Sections

Parallel for is not enough for many tasks. For this reasons the sections construct has been introduced. A sections construct is a structured block containing multiple sections that needs to be executed in parallel. Each section has to be preceded by "omp section" except the first. At the end of the sections block there is a implicit barrier, unless the nowait clause is used.

```
#pragma omp parallel // starts a new team
{
    //Work0(); // this function would be run by all threads.
    #pragma omp sections // divides the team into sections
    {
        // everything herein is run only once.
        { Work1(); }
        #pragma omp section
        { Work2();
        Work3(); }
        #pragma omp section
        { Work4(); }
    }
    //Work5(); // this function would be run by all threads.
}
```


Sections Single

The following is the example for single

```
#include <iostream>

int main()
{
    #pragma omp parallel
    {
        std::cout << "multiple␣\n";
        #pragma omp single
        {
            std::cout << "only␣one\n";
        }
    }
    return 0;
}
```

Tasks

Task have been introduce in OpenMP 3.0 to provide a simpler to use control of tasking along the approach of the fork-join model. The approach of section was cumbersome requiring a FIXED, non-RECURSIVE number of parallel sections.
In OpenMP 3.0 every task can generate other task, that is then enqueued to the tasks list and decided by the runtime when it will be executed. Synchronization among tasks allow to collect execution results.

- "#pragma omp task" - creates a new task
- "#pragma omp barrier" - blocks for all the threads in the Team
- "#pragma omp taskwait" - waits for the completion of the children tasks

Task elements

Tasks are made of the following elements:

- Code
- Data environment
- Assigned real thread

In addition they are characterized by two phases:

- Packaging - when the context is captured
- Execution

Task data environment

Variables whose lexical scope is visible at the construct are considered SHARED by default. Static variables declared inside the block are SHARED, while local variable are PRIVATE. This behavior can be modified by specifying a specific behavior for each variable

- shared(list) - variables are shared
- private (list) - one copy per thread default constructed. The original variable cannot be used because it can be unreliable
- lastprivate (list) - as private but at the end of the execution the variable is updatedd
- firstprivate (list) - the variable is copy constructed

The general behavior can be modified by using the clause default:

- default(shared) - all variables are shared (DEFAULT)
- default(none) - each variable has to be specified

Note that these clauses do apply also to other worksharing primitives such as for, parallel and section.

Task Example

```
#include <cmath>
#include <chrono>
#include <memory.h>
#include <iostream>
#include "omp.h"
int main()
{
    const int size = 10;
    double sinTable[size];
    memset(sinTable,0,sizeof(sinTable));
    auto before1 = omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp single
        {
            std::cout << "single " << omp_get_thread_num() << std::endl;
            for(int n=0; n<size; ++n)
            {
                #pragma omp task default(none) shared(sinTable) firstprivate(n) shared(std::cout)
                {
                    #pragma omp critical
                    {
                        std::cout << "thread " << omp_get_thread_num() << " iter " << n << " ptr "
                                  << sinTable << std::endl;
                    }
                    sinTable[n] = std::sin(2 * M_PI * n / size);
                }
            }
            #pragma omp taskwait
        }
    }
    auto after1 = omp_get_wtime();

    std::cout << "elapsed omp(s): " << (after1-before1) << std::endl;
    for(int n=0; n<size; ++n)
    {
        std::cout << "result " << n << " " << sinTable[n] << std::endl;
    }
    return 0;
}
```

OpenMP Implementation

How OpenMP can be implemented?

- Compiler support
- Code Transformations
- Run-time library

Discussion about the possible implementation of the simple "parallel for". Check also the ideas present in the paper "Ruffaldi E., Dabisias G., Brizzi F. & Buttazzo G. (2016). SOMA: An OpenMP Toolchain For Multicore Partitioning. In 31st ACM/SIGAPP Symposium on Applied Computing . ACM.