HPC2N, UmU

First steps in OpenMP

Joachim Hein
LUNARC & Centre of Mathematical Sciences
Lund University

Pedro Ojeda May
HPC2N
Umeå University

Introduction

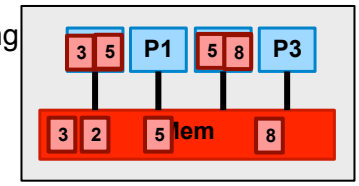
- Parallel and serial regions
- Master thread and teams of threads
- Directives and library functions
- Controlling the number of threads
- Timing OpenMP codes

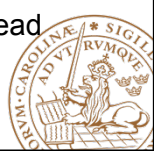
Shared Memory Programming

- Huge renaissance recently in scientific computing:
  - Multicore processors
  - Multithreaded processing cores
- Shared memory hardware:
  - Very expensive 15 years ago
  - Inexpensive these days
    - E.g.: Contemporary laptop has between 2 and 8 cores
- Number of ways to program shared memory, including:
  - Posix threads (typically C, C++)
  - OpenMP (C, C++, Fortran)
  - Threaded languages (e.g.: Java)

Shared Memory Architecture

- Threads placed on several processing elements manipulate the same, shared memory space
- Easy to move data between threads
  - Write result to shared memory
  - Read on different processor
- Care is needed regarding order:
  - P0 needs to write before P2 can read
- Read/write to shared memory has typically higher cost than manipulating registers/cache ➔ communication overhead





OpenMP resources

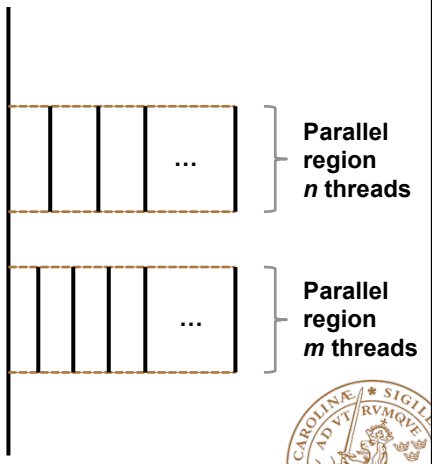
- OpenMP™ is trademarked by the OpenMP ARB
- The OpenMP ARB is a not for profit corporation
  http://openmp.org
- You can get the standard specifications from there (free)
  http://www.openmp.org/specifications/
- Free tutorials:
  https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG
  https://computing.llnl.gov/tutorials/openMP/

Textbooks

- Textbook (OpenMP 2.5): “Using OpenMP”
  B Chapman, G Jost, R van der Pas
  MIT press, Cambridge, Massachusetts, 2008
- Textbook (OpenMP 4.5): “Using OpenMP – The next Step”
  R van der Pas, E Stolzer, C Terboven
  MIT press, Cambridge, Massachusetts, 2017

Threads in OpenMP

- Program execution starts single threaded (master thread)
- Start parallel region:
  - Team of threads created (fork)
  - Each thread: independent instruction stream
- End of parallel region:
  - Team of threads join
  - Synchronisation occurs
  - Single thread continues
- Next parallel region might have different number of threads



OpenMP: Directive based

- OpenMP is based on directives
  - In Fortran they are special comments
  - In C/C++ they are `#pragma`
- In both cases these directives are ignored
  - Compiler without OpenMP support (rare these days)
  - OpenMP support not enabled in compiler (option)
- Makes it easy to have a single version of the source
  - Serial execution
  - Parallel execution
- OpenMP also facilitates conditional compilation

OpenMP directives in Fortran

- In free format Fortran a directive looks as follows

```
!$omp  directive_name [clause […]]
```

- In fixed format Fortran directives look as follows. They **always** start in column 1

```
!$omp  directive_name [clause […]]
c$omp  directive_name [clause […]]
*$omp  directive_name [clause […]]
```

- The first piece (e.g. `!$omp`) is called *"sentinel"*

Examples for line continuation in Fortran

- Line continuation as per language standard
- Continuation line needs to start with sentinel (e.g. `!$omp`)
- Example in free format:

```
!$omp parallel do &
!$omp   shared(a,b)
```

- Example in fixed format:

```
c$omp parallel do
c$ompa shared(a,b)
c$ompb shedule(dynamic)
```

  – Non-blank in column 6 marks continuation line

OpenMP directives in C

- In C/C++ a directive looks as follows

```
#pragma omp directive_name [clause […]]
```

- Use backslash "\" for line continuation
- Directive name specifies the action
- The clause(s) allow further specification

Library functions

- In addition to directives: OpenMP offers **library functions**
  - Mainly to control operating environment
- Utilising them requires header functions
  - In C:

```
#include omp.h
```

  - In Fortran:

```
include "omp_lib.h"
```

or

```
use omp_lib
```

Conditional compilation

- OpenMP compilers define the pre-proc. macro `_OPENMP`

```
#ifdef _OPENMP
#include omp.h
#endif
```

- In Fortran, lines starting with `!$` (free format) or `!$`, `*$`, `c$` are only compiled if OpenMP is active

```
!$    use omp_lib
```

- **Rem:** The above guard is required if code needs to be compiled serially

Construct: parallel in Fortran

- The most important construct in OpenMP

```
!$omp parallel
  structured block of Fortran
!$omp end parallel
```

- This will start a team of threads
  - working on the block between the directives
- End of parallel region: Wait for the last thread
  - implicit synchonisation

A first example in Fortran

- First portion of code: executed on the master
  - Printing of 3+5 once
- Construct: parallel
  - Create number of threads
  - Each thread: does the addition
  - Each thread: prints 6+7
- **Rem:** also compiles serially

```
program example
  Implicit None

  print *, "3+5=", 3+5

!$omp parallel
  print *, "6+7=", 6+7
!$omp end parallel

end program example
```

Construct: parallel in C

- The most important construct in OpenMP

```
#pragma omp parallel
  {
    structured block of C instructions
  }
```

- This will start a team of threads
  - working on the block enclosed with { } in parallel
- End of parallel region: Wait for the last thread
  - implicit synchonisation

A first example in C

- First portion of code: executed on the master
  - Printing of 3+5 once
- Construct: parallel
  - Create number of threads
  - Each thread: does the addition
  - Each thread: prints 6+7

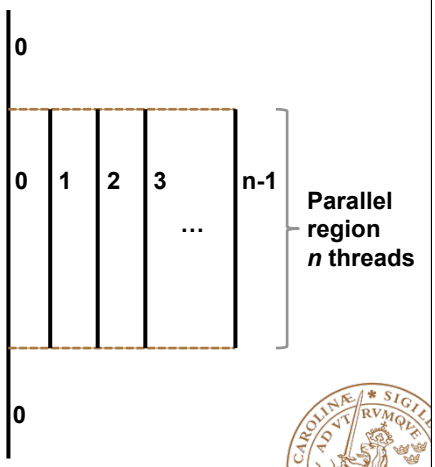
```
int main()
{
  printf("3+5=%i\n", 3+5);

#pragma omp parallel
  {
    printf("6+7=%i\n", 6+7);
  }
  return 0;
}
```

Controlling the number of threads

- Number of threads started by parallel construct can be controlled in a number of ways
  1. Environment variable
     `OMP_NUM_THREADS`
  2. Function call during program execution:
     `omp_set_num_threads(n)`
  3. Clause `num_threads` on `parallel` construct:
     `!$omp parallel num_threads(n)`
     or
     `#pragma omp parallel num_threads(n)`
- Priority increases top to bottom (clause highest)

Thread number and thread id

- Query functions require header files
- Query number of threads:
  `omp_get_num_threads()`
- Query thread id:
  `omp_get_thread_num()`



F90-example: Printing thread number

```
program FortanHello

!$ use omp_lib
  implicit none

!$omp parallel
  print *,"I am thread", omp_get_thread_num(), &
          " out of ", omp_get_num_threads()
!$omp end parallel

end program FortanHello
```

Sample output from code (F90-version)

```
 I am thread           0  out of           8
 I am thread           3  out of           8
 I am thread           4  out of           8
 I am thread           2  out of           8
 I am thread           1  out of           8
 I am thread           7  out of           8
 I am thread           6  out of           8
 I am thread           5  out of           8
```

- Example using 8 threads
- Each threads prints:
  - its thread number
  - total number of threads

C-example: Printing thread number

```
#include <stdio.h>
#include <omp.h>
int main()
{
#pragma omp parallel
    {
       printf("I am thread %i of %i\n",
         omp_get_thread_num(),
         omp_get_num_threads() );
    }
  return 0;
}
```

Use thread number for task farm (Fortran)

- Situation:
  - We have three serial programs
  - Want to run them on different threads
- Preparation work:
  - Make the programs into functions:

source2.f90

```
Program Prog2

  ! statements
End Program Prog2
```

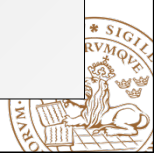
source2.f90

```
subroutine sub2()

  ! statements
End subroutine sub2
```

- Write a new main using OpenMP to run these functions

Simple version of new main program:

```
Program farm
 use omp_lib
 call omp_set_num_threads(3)
!$OMP parallel
  if (omp_get_thread_num().eq.0) call sub0()
  if (omp_get_thread_num().eq.1) call sub1()
  if (omp_get_thread_num().eq.2) call sub2()
!$OMP end parallel

End program farm
```



Use thread number for task farm (C-version)

- Situation:
  - We have three serial programs
  - Want to run them on different threads
- Preparation work:
  - Make the programs into functions:

source2.c

```
int main()
{
  // statements
}
```

source2.c

```
int funct2()
{
  // statements
}
```

- Write a new main using OpenMP to run these functions

Simple version of new main function:

```
int main()
{
 omp_set_num_threads(3);
#pragma omp parallel
 {
  if (omp_get_thread_num() == 0) funct0();
  if (omp_get_thread_num() == 1) funct1();
  if (omp_get_thread_num() == 2) funct2();
 }
 return 0;
}
```

Timer in OpenMP in Fortran

- Parallel programming is all about speed
- Timer `omp_get_wtime`
- Returns seconds
  - `double` in C
  - `double precision` in Fortran
- Accuracy can be queried with `omp_get_wtick`
- Bound to thread!

```
double precision :: stime, ftime

...

stime =  omp_get_wtime()

 ! code segement to
 ! be timed

ftime = omp_get_wtime()

print *,"time: ", ftime-stime
```


Timer in OpenMP for C

- Parallel programming is all about speed
- Timer `omp_get_wtime`
- Returns seconds
  - `double` in C
  - `double precision` in Fortran
- Accuracy can be queried with `omp_get_wtick`
- Bound to thread!

```
double stime =
 omp_get_wtime();

 // code segements to
 //  be timed

double ftime =
 omp_get_wtime()-stime;

printf("time: %f\n",
    ftime );
```

Compiling OpenMP code

- Most modern compilers support OpenMP
- Simply add a compiler flag to enable OpenMP
- Example for GCC:

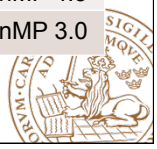
```
gfortran –O3 –fopenmp –o prog_omp prog_omp.f90
```

Compiler	Flag for OpenMP	Standard Implemented (_OPENMP)
GNU	`-fopenmp`	version 4.8.5: OpenMP 3.1 version 4.9.3 OpenMP 4.0 version 5.4.0 OpenMP 4.0 version 6.2.0 OpenMP 4.5

INTEL	`-openmp`
	`-qopenmp`
	`-qopenmp`
		version 16.0.1: OpenMP 4.0
		version 16.0.3: OpenMP 4.0
		version 17.0: OpenMP 4.5

PGI	`-mp`	version 15.10, 16.4, 16.7: OpenMP 3.0

Rem: Some features of newer standards may be available



Summary

- Introduction to teams of threads in OpenMP
- Controlling and querying basic properties of a thread
  - Number of threads
  - Thread number
- Timing code