THE UNIVERSITY OF WARWICK

High Performance Computing

Course Notes

Shared Memory Parallel Programming

Dr Ligang He

OpenMP

- OpenMP stands for Open specification for Multi-processing
- used to assist compilers to understand and parallelise the serial code
- Can be used to specify shared memory parallelism in Fortran, C and C++ programs
- OpenMP is a specification for
  - a set of compiler directives,
  - RUN TIME library routines, and
  - environment variables

History of OpenMP

- Started late 80s there was emergence of shared memory parallel computers with proprietary directive-driven programming environments
- Poor portability, OpenMP emerges as an industry standard

- OpenMP specifications include:

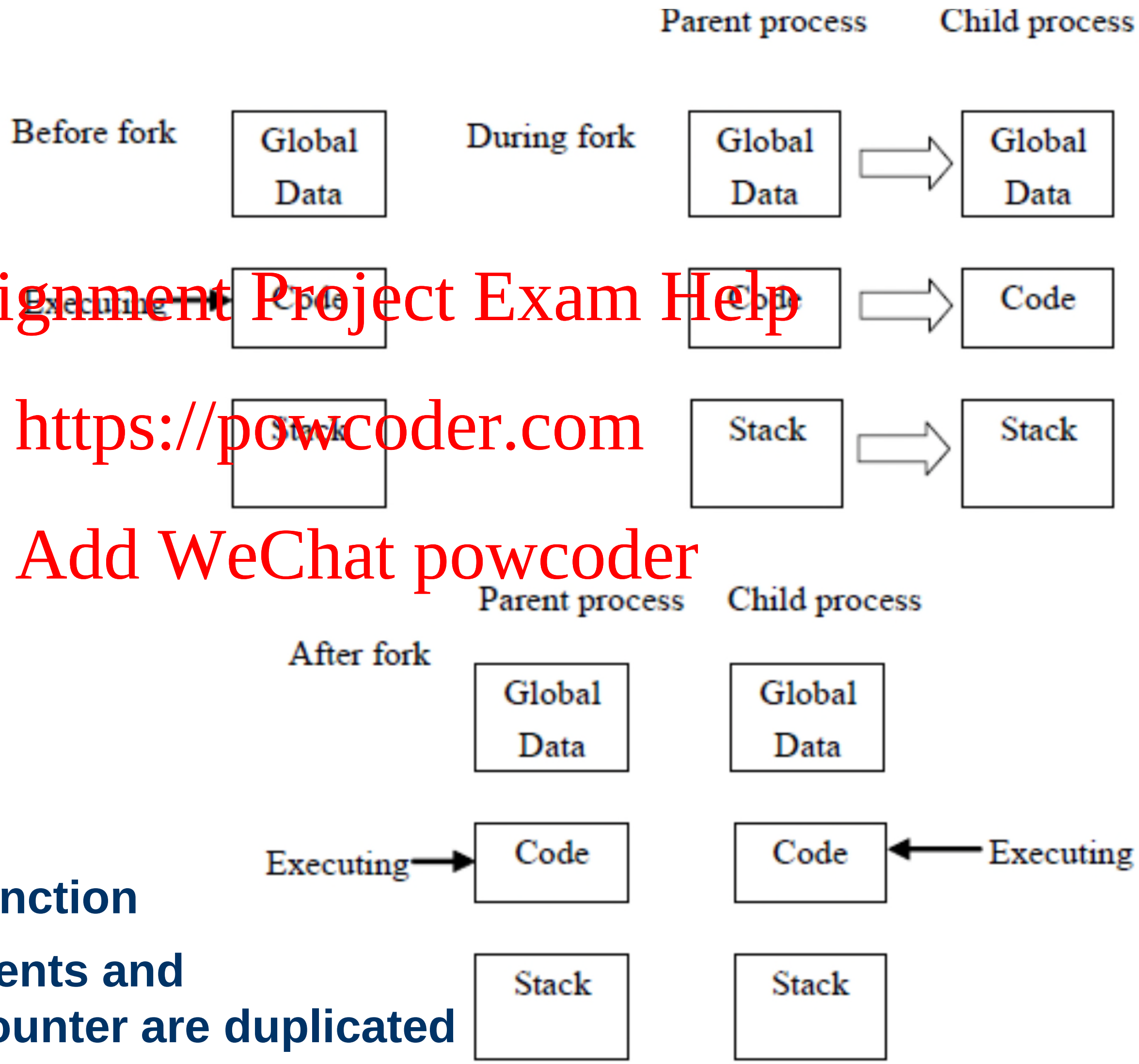
  - OpenMP 1.0 for Fortran, 1997, OpenMP 1.0 for C/C++, 1998
  - OpenMP 2.0 for Fortran, 2000, OpenMP 2.0 for C/C++ , 2002

  - OpenMP 2.5 for C/C++ and Fortran, 2005
  - OpenMP 3.0 for C/C++ and Fortran, 2008
  - OpenMP 3.1, 2011
  - OpenMP 4.0, 2013, OpenMP 4.5, 2015
- OpenMP Architecture Review Board: Compaq, HP, IBM, Intel, SGI, SUN

OpenMP programming model

- An implementation of thread models
  - Multiple threads running in parallel
- Used for shared memory architecture
- Fork-join model

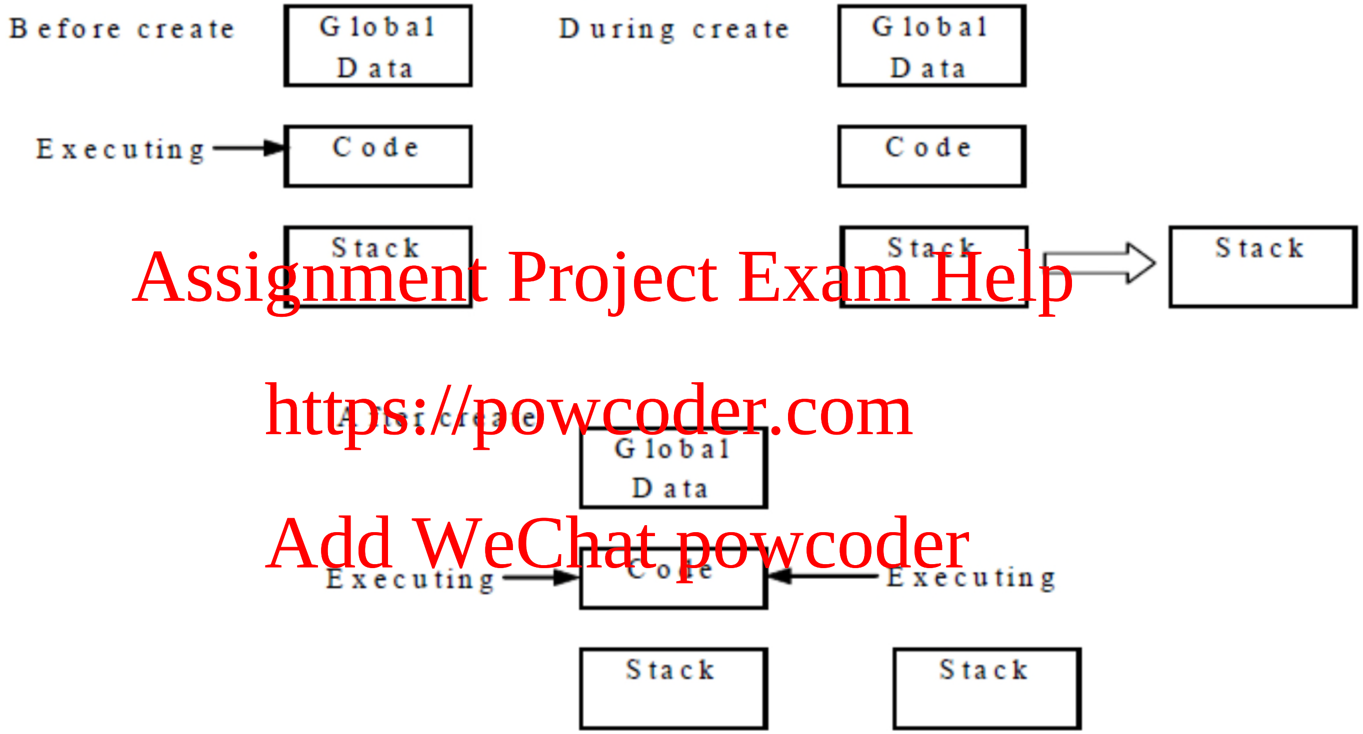
How a new process is created



Use the fork function

All three segments and the program counter are duplicated

How Threads are Created?



Only the stack segment and the program counter are duplicated

Threads

- Used to split a program into separate tasks, one per thread, that can execute concurrently
- “Light weight process”: multiple threads exist within the context of a single process, sharing the process’s code, global information, other resources
- Threads usually communicate by processing shared global data values

global shared space – global data accessed from single global address space (heap) shared among the threads

local private space – each thread also has its own local private data (stack) that is not shared

OpenMP code structure in C

```
#include <omp.h>
main () {
        int var1, var2, var3;
        Serial code
/*Beginning of parallel section. Fork a team of threads. Specify variable scoping*/
        #pragma omp parallel private(var1, var2) shared(var3)
{
        Parallel section executed by all threads
        ...
        All threads join master thread and disband
}
Resume serial code
}
```

OpenMP code structure in Fortran

PROGRAM HELLO

INTEGER VAR1, VAR2, VAR3

Serial code ...

!Beginning of parallel section. Fork a team of threads. Specify variable scoping

!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)

Parallel section executed by all threads

...

All threads join master thread and disband

!$OMP END PARALLEL

Resume serial code

...

END

OpenMP Directives Format

C/C++

#pragma omp	directive-name	[clause, ...]
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.

Fortran

sentinel	directive-name	[clause ...]
All Fortran OpenMP directives must begin with a sentinel. The accepted sentinels depend upon the type of Fortran source. Possible sentinels are: !OMP, COMP, *$OMP	A valid OpenMP directive. Must appear after the sentinel and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.

OpenMP features

- OpenMP directives are ignored by compilers that don't support OpenMP. In this case, codes are run as serial codes
- Compiler directives used to specify

  - sections of code that can be executed in parallel

  - critical sections

  - Scope of variables (private or shared)
- Mainly used to parallelize loops, e.g. separate threads to handle separate iterations of the loop
- There is also a run-time library that has several useful routines for checking the number of threads and number of processors, changing the number of threads, etc

Fork-Join Model

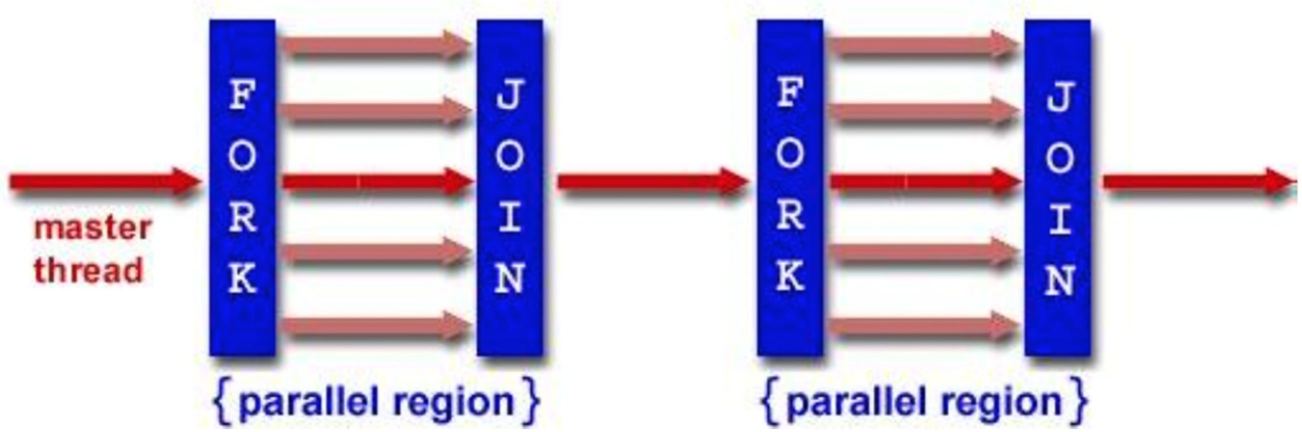
Multiple threads are created using the parallel construct

For C and C++

```
#pragma omp parallel
{
    ... do stuff
}
```

For Fortran

```
!$OMP PARALLEL
    ... do stuff
!$OMP END PARALLEL
```

How many threads are generated

The number of threads in a parallel region is determined by the following factors, in order of precedence:

- Use of the omp_set_num_threads() library function
- Setting of the OMP_NUM_THREADS environment variable
- Implementation default - the number of CPUs on a node

Threads are numbered from 0 (master thread) to N-1

Parallelizing loops in OpenMP

- Compiler directive specifies that loop can be done in parallel

For C and C++:

```
#pragma omp parallel for
for (i=0;i++;i<N) {
      value[i] = compute(i);
}
```


For Fortran:

```
!$OMP PARALLEL DO
      DO (i=1:N)
         value(i) = compute(i);
      END DO
!$OMP END PARALLEL DO
```

Partition and allocation of loop iterations

- Can use thread scheduling to specify partition and allocation of iterations to threads

```
#pragma omp parallel for schedule(static,4)
```


- schedule(static [,chunk])

  - Partition the loop into blocks of iterations of size chunk
  - Before execution, deal out the blocks to each thread

Partition and allocation of loop iterations

- Can use thread scheduling to specify partition and allocation of iterations to threads

#pragma omp parallel for schedule(static,4)

→ schedule(static [,chunk])

→ schedule(dynamic [,chunk])

- →Partition the loop into blocks of iterations of size chunk
- →Put the blocks into a queue
- →During the execution, Each thread grabs a block off the queue until all are done
- →Which thread runs which block of iterations depends on the thread's execution pace

Partition and allocation of loop iterations

- **Can use thread scheduling to specify partition and allocation of iterations to threads**

#pragma omp parallel for schedule(static,4)

- **schedule(static [,chunk])**

- **schedule(dynamic [,chunk])**

- **schedule(runtime): Find schedule from an environment variable OMP_SCHEDULE**

Synchronisation in OpenMP

Critical construct

Fortran	!$OMP CRITICAL *[name]* *block* !$OMP END CRITICAL
C/C++	#pragma omp critical *[name]* *structured block*

Barrier construct

Fortran	!$OMP BARRIER
C/C++	#pragma omp barrier

Example of Critical Section

```
#include <omp.h>
main() {
        int x;
        x = 0;
        #pragma omp parallel shared(x)
        {
                …
                #pragma omp critical
                x = x+1;
        } /* end of parallel section */
}
```

Example of Barrier in OpenMP

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
        int th_id, nthreads;
        #pragma omp parallel private(th_id)
{
        th_id = omp_get_thread_num();
        printf("Hello World from thread %d\n", th_id);
        #pragma omp barrier
        if ( th_id == 0 ) {
                nthreads = omp_get_num_threads();
                printf("There are %d threads\n",nthreads);
        }
return 0;
}
```

Data Scope Attributes in OpenMP

- OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be viewed by threads
- These clauses are used in conjunction with several directives (e.g. PARALLEL, DO/for) to control the scoping of enclosed variables
- Three often encountered clauses:
  - Shared
  - Private
  - Reduction

Shared and private data in OpenMP

- **private(var)** creates a local copy of var for each thread
- **shared(var)** states that var is a global variable to be shared among threads


```
!$OMP PARALLEL DO
!$OMP& PRIVATE(xx,yy) SHARED(u,f)
    DO j = 1,m
      DO i = 1,n
        xx = -1.0 + dx * (i-1)
        yy = -1.0 + dy * (j-1)
        u(i,j) = 0.0
        f(i,j) = -alpha * (1.0-xx*xx) * & (1.0-yy*yy)
      END DO
    END DO
!$OMP END PARALLEL DO
```

Reduction Clause

- **Reduction –** `reduction (op : var)`
- The exemplar op is add, logical OR (commutative operations)

- A local copy of the variable is made for each thread

- The local values of the variable can be updated by the threads.

- At the end of parallel region, the local values are combined to create global value through Reduction operation

An Example of Reduction Clause

```
double ZZ, res=0.0;
    #pragma omp parallel for reduction (+:res) private(ZZ)
    for (i=1;i<=N;i++) {
        ZZ = i;
        res = res + ZZ:
    }
```

Run-Time Library Routines

- Can perform a variety of functions, including

Query the number of threads/thread no.

Set the number of threads to be generated

Query the number of processors in the computer

Changing the number of threads

Run-Time Library Routines

- **query routines allow you to get the number of threads and the ID of a specific thread**

id = omp_get_thread_num(); //thread no.

Nthreads = omp_get_num_threads(); //number of threads

- **Can specify number of threads at runtime**

omp_set_num_threads(Nthreads);

Environment Variable

- Controlling the execution of parallel code
- Four environment variables

  - OMP_SCHEDULE: how iterations of a loop are scheduled

  - OMP_NUM_THREADS: maximum number of threads

  - OMP_DYNAMIC: enable or disable dynamic adjustment of the number of threads
  - OMP_NESTED: enable or disable nested parallelism

Lab session today

- Practice OpenMP

- Download lab instructions and code from here:

https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs402/cs402_seminar2_openmp.pdf

https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs402/cs402_seminar2_code.zip

- Move down to Lab 001 and 003

Assignment 1 - OpenMP

- **Use OpenMP to parallelize the deqn code**
  - The overall objective is to achieve good speedup
- **Write a report**
  - Explain in detail what you did with the sequential code
  - benchmark the runtime of each relevant loop and the runtime of the whole parallel program against the number of threads; present the runtimes in graph or table; analyze the results
  - Discuss the iteration scheduling in your program
  - Analyze the overhead of OpenMP
  - Presentation skills, spelling, punctuation and grammar
  - Up to four A4 pages