COMS3008A: Parallel Computing
Introduction to OpenMP: Part I

Hairong Wang

School of Computer Science,
University of the Witwatersrand, Johannesburg

March 12, 2025

Contents

Outline

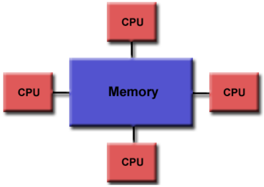
What is OpenMP?

Open specifications for Multi Processing

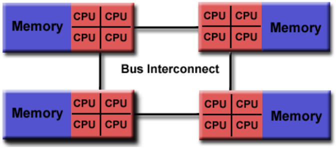
- An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism.
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- API is specified for C, C++ and Fortran.
- Portable
- Easy to use

OpenMP Programming Model

- OpenMP is designed for multi-processor/core, shared memory machines.



(a) UMA



(b) NUMA

Figure: The shared memory model of parallel computation.

OpenMP Programming Model Cont.

- Thread Based Parallelism:
  - OpenMP programs accomplish parallelism exclusively through the use of threads.
  - A thread of execution is the smallest unit of processing that can be scheduled by an operating system.
  - Threads exist within the resources of a single process. Without the process, they cease to exist.
  - Typically, the number of threads match the number of processors/cores of a machine, but it can differ.

OpenMP Programming Model Cont.

- Explicit Parallelism:
  - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Compiler Directive Based: Most OpenMP parallelism is specified through the use of compiler directives which are embedded in C/C++ or Fortran source code.

OpenMP Programming Model Cont.

- Fork - Join Model: OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
    - FORK:
        - The master thread then creates a team of parallel threads;
        - Becomes the master of this group of threads;
        - Assigned the thread number 0 within the group.
    - JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

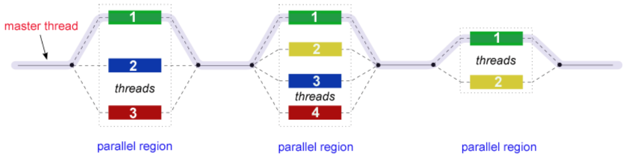


Figure: Fork-join model

OpenMP API Overview

- Three components: Compiler Directives, Runtime Library Routines, Environment Variables.
  - Compiler Directives: Appear as comments in your source code to guide the compilers.
  - The OpenMP API includes an ever-growing number of run-time library routines.
  - OpenMP provides several environment variables for controlling the execution of parallel code at run-time.

OpenMP API Overview Cont.

- OpenMP core syntax:
  - Most of the constructs in OpenMP are compiler directives: **#pragma omp ⟨construct name⟩ [clause [clause]. . .]**
  - OpenMP code structure: Structured block. A block of one or more statements with one point of entry and one point of exit at the end.
  - Function prototypes and types in the file: **#include ⟨omp.h⟩**

OpenMP API Overview Cont.

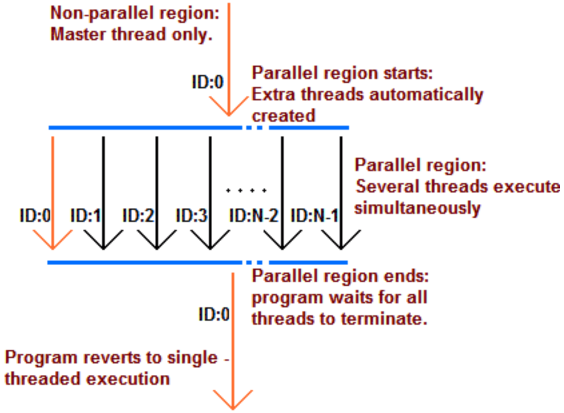

Non-parallel region:
Master thread only.
ID:0
Parallel region starts:
Extra threads automatically created
ID:0
ID:1
ID:2
ID:3
ID:N-2
ID:N-1
Parallel region:
Several threads execute simultaneously
Parallel region ends:
program waits for all threads to terminate.
ID:0
Program reverts to single - threaded execution

Figure: OpenMP parallel construct execution model

Compiling OpenMP Programs

Exercise 1: Write a multi-threaded C program that prints “Hello World!”. (hello_omp.c)

```
#include <stdio.h>
#include <omp.h>
int main(){
  //Parallel region with default number of threads
  #pragma omp parallel
  //Start of the parallel region
  {
    //Runtime library function to return a thread
       number
    int ID = omp_get_thread_num();
    printf("Hello World! (Thread %d)\n", ID);
  }//End of the parallel region
}
```

Compiling OpenMP Programs Cont.

- To compile, type **gcc -fopenmp hello_omp.c -o hello_omp**.
- To run, type **./hello_omp**

Outline

Outline

Parallel Region Construct

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

- Syntax:

```
#pragma omp parallel [clause[[,] clause] ... ]
    structured block
```

Parallel Region Construct Cont.

Typical clauses in `clause` list:

- Degree of concurrency:
  `num_threads(<integer expression>)`
- Data scoping:
  - `private(<variable list>)`
  - `firstprivate(<variable list>)`
  - `shared(<variable list>)`
  - `default(<data scoping specifier>)` — (shared or none)
- Conditional parallelization: `if (<scalar expression>)`, determines whether the `parallel` construct creates threads.

Parallel Region Construct Cont.

- Interpreting an OpenMP parallel directive

```
......
int b, a=10, c=5;
#pragma omp parallel if (is_parallel==1) \
num_threads(8) shared(b) private(a) \
firstprivate(c) default(none)
{
/* structured block */
}
```

Parallel Region Construct Cont.

- Create threads in OpemMP using the parallel construct.

Example 1

Create a 4-thread parallel region using *num_threads* clause.

```
#include <stdio.h>
#include <omp.h>
int main()
{
  //Parallel region with 'num_threads' clause to set
     number of threads
  #pragma omp parallel num_threads(4)
  //Start of the parallel region
  {
    //Runtime library function to return a thread number
    int ID = omp_get_thread_num();
    printf("Hello World! (Thread %d)\n", ID);
  }//End of the parallel region
}
```

Parallel Region Construct Cont.

Example 2

Create a 4-thread parallel region using runtime library routine.

```c
#include <stdio.h>
#include <omp.h>
int main()
{
  //Runtime library function to request the number of
     threads in the parallel region
  omp_set_num_threads(4);
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("Hello World! (Thread %d)\n", ID);
  }
}
```

Parallel Region Construct Cont.

Different ways to set the number of threads in a parallel region:

- Using runtime library function `omp_set_num_threads()`
- Setting clause `num_threads`, e.g.,
  ```
  #pragma omp parallel num_threads (8)
  ```
- Specify at runtime using environment variable `OMP_NUM_THREADS`, e.g.,
  ```
  export OMP_NUM_THREADS=8
  ```

Computing the π Using Integration

Example 3

Compute the number π using numerical integration: $\int_0^1 \frac{4.0}{1+x^2} = \pi$.

1. Write a serial program for the problem.
2. Parallelize the serial program using OpenMP directive.
3. Compare the results from 1 and 2.

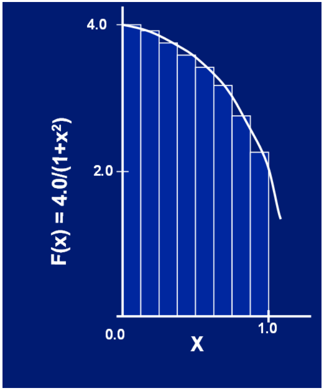


Figure: Approximating the π using numerical integration

Serial Program for Computing Π

```
static long num_steps = 1000000;
double step;
int main (){
  int i;
  double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=0;i< num_steps;i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Parallel Program for Computing Π — Method I

```
static long num_steps = 1000000;
double step;
#define NUM_THREADS 2
int main (){
  int nthreads;
  double pi, sum[NUM_THREADS];
  step = 1.0/(double)num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
    int i, id, tthreads; double x;
    tthreads = omp_get_num_threads();
    id = omp_get_thread_num();
    if(id==0) nthreads=tthreads;
    for (i=id, sum[id]=0.0;i< num_steps; i=i+tthreads){
      x = (i+0.5)*step;
      sum[id] = sum[id] + 4.0/(1.0+x*x);
    }
  }
  for(i=0, pi=0.0;i<nthreads;i++)
    pi += step * sum[i];
}
```

WITS UNIVERSITY

False Sharing

If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads...this is called **false sharing.**

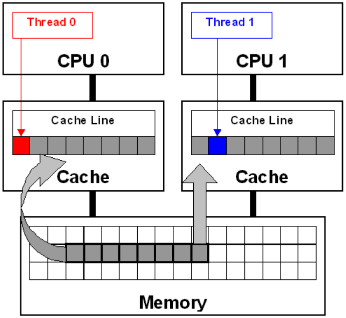


Figure: False sharing

Cache

- Processors usually execute operations faster than they access data in memory.
- To address this problem, a block of relatively fast memory is added to a processor — cache.
- The design of cache considers the temporal and spatial locality.
- For example, x is a shared variable and $x = 5$, `my_y`, `my_z` are private variables, what will be the value of `my_z`?

```
if myId==0{
    my_y = x;
    x++;
}else if myId==1{
    my_z = x;
}
```

WITS
UNIVERSITY

Cache Coherence

Example 4

Suppose $x = 2$ is a shared variable, $y0, y1, z1$ are private variables. If the statements in Table 1 are executed at the indicated times, then $x =?$, $y0 =?$, $y1 =?$, $z1 =?$

Time	Core 0	Core 1
0	$y0 = x;$	$y1 = 3 * x;$
1	$x = 7;$	Statements not involving x
2	Statements not involving x	$z1 = 4 * x;$

Table: Example - cache coherence

Cache Coherence cont.

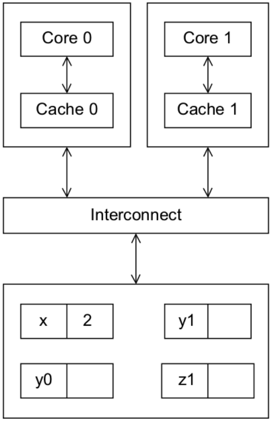


Figure: Example - cache coherence cont.

Cache Coherence cont.

Cache coherence problem - When the caches of multiple processors store the same variable, an update by one processor to the cached variable is ‘seen’ by the other processors. That is, the cached value stored by the other processors is also updated.

False Sharing

- Suppose two threads with separate caches access different variables that belong to the same cache line. Further suppose at least one of the threads updates its variable.
- Even though neither thread has written to a shared variable, the cache controller invalidates the entire cache line and forces the other threads to get the values of the variables from main memory.
- The threads aren't sharing anything (except a cache line), but the behaviour of the threads with respect to memory access is the same as if they were sharing a variable. Hence the name false sharing.

Parallel Program for Computing Π — Method II

```c
static long num_steps = 1000000;
int main (){
  int i, tthreads, id;
  double step, pi = 0.0, sum = 0.0, x;
  step = 1.0/(double)num_steps;
  #pragma omp parallel
  {
    tthreads = omp_get_num_threads();
    id = omp_get_thread_num();
    for (i=id; i<num_steps; i+=tthreads){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
    }
  }
  pi = step * sum;
}
```

Race Condition

- When multiple threads update a shared variable, the computation exhibits non-deterministic behaviour — race condition. That is, two or more threads attempt to access the same resource.
- If a block of code updates a shared resource, it can only be updated by one thread at a time.

Outline

Synchronization

- Synchronization: Bringing one or more threads to a well defined and known point in their execution. *Barrier* and *mutual exclusion* are two most often used forms of synchronization.
  - Barrier: Each thread wait at the barrier until all threads arrive.
  - Mutual exclusion: Define a block of code that only one thread at a time can execute.
- Synchronization is used to impose order constraints and to protect access to shared data.
- In Method II of computing π,

  ```
  sum = sum + 4.0/(1.0+x*x);
  ```

  is called a critical section.
- Critical section - a block of code executed by multiple threads that updates a shared variable, and the shared variable can only be updated by one thread at a time.

High level synchronizations in OpenMP

- `critical` construct: Mutual exclusion. Only one thread at a time can enter a *critical* section.

  `#pragma omp critical`

```
float result;
......
#pragma omp parallel
{
  float B; int i, id, nthrds;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads();
  for(i = id; i < nthrds; i+= nthrds) {
    B = big_job(i);
  }
  #pragma omp critical
    result += calc(B);
  ......
}
```

High level synchronizations in OpenMP Cont.

- `atomic` construct: Basic form. Provides mutual exclusion but only applies to the update of a memory location.
- `#pragma omp atomic`

The statement inside the *atomic* must be one of the following forms:

- $x\ op = expr$, where $op \in (+=, -=, *=, /=, \% =)$
- $x++$
- $++x$
- $x--$
- $--x$

```
#pragma omp parallel
{
  ......
  double tmp, B;
  B = calc();
  tmp = big_calc(B);
  #pragma omp atomic
    X += tmp;
  ......
}
```

High level synchronizations in OpenMP Cont.

- `barrier` construct: Each thread waits until all threads arrive.

  `#pragma omp barrier`

```
    #pragma omp parallel
    {
      int id=omp_get_thread_num();
      A[id]=calc1(id);
      #pragma omp barrier
      B[id]=calc2(id, A);
      ......
    }
```

Summary

- OpenMP execution model
- Create a parallel region in OpenMP programs
- We have studied the following constructs:
  - `parallel`
  - `critical`
  - `atomic`
  - `barrier`
- Usage of OpenMP library functions and environment variables
- False sharing and race condition.

References

Sources in Chapman, Jost, and Pas, *Using OpenMP: Portable Shared Memory Paralle Programming* and Trobec et al., *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*

Chapman, Barbara, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Paralle Programming*. The MIT Press, Cambridge, 2007. ISBN: 9780262533027.

Trobec, Roman et al. *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Springer, 2018.