COMS4040A&COMS7045A: High Performance Computing & Scientific Data Management Introduction to OpenMP: Part I

Hairong Wang

School of Computer Science,
University of the Witwatersrand, Johannesburg

Semester one 2025

WITS UNIVERSITY

Contents

Outline

What is OpenMP?

- OpenMP: Open specifications for Multi Processing, is a commonly used shared memory parallel programming model.
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- API is mainly specified for C, C++ and Fortran.
- OpenMP programs accomplish parallelism exclusively through the use of threads.

OpenMP parallel programming model

- 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.
  - A compiler directive in C/C++ is called a pragma.
  - A pragma is a way to communicate information to the compiler.

OpenMP parallel programming model cont.

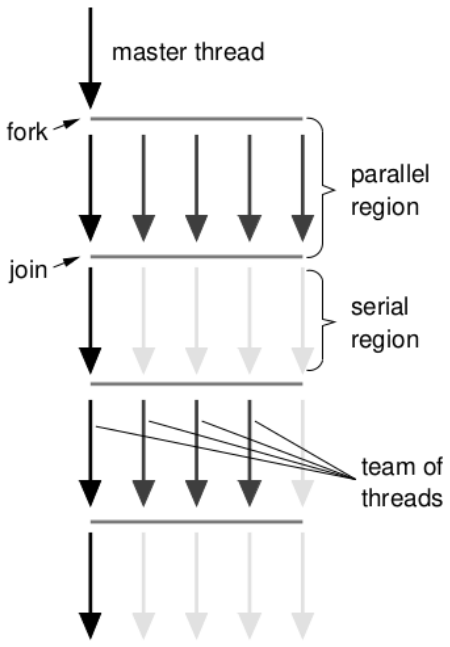

Figure: OpenMP parallel computing model – **fork-join model**

- 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;
    - Asssigned the thread number 0 within the group.
  - JOIN: When the team of threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

WITS UNIVERSITY

OpenMP parallel programming model

- In C/C++, an OpenMP directive applies to the statement or structured block that follows the directive.
- For example,

```
1 #include <omp.h>

3 #pragma omp parallel
4 {
5   do_work(omp_get_thread_num(), omp_get_num_threads());
6 }
```

or

```
1 #include <omp.h>

3 #pragma omp parallel
4   do_work(omp_get_thread_num(), omp_get_num_threads());
```

Compiling OpenMP Programs

Example 1

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

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main() {
4    //Parallel region with default number of threads
5    #pragma omp parallel
6    //Start of the parallel region
7    {
8      //Runtime library function to return a thread number
9      int ID = omp_get_thread_num();
10     printf("Hello World! (Thread %d)\n", ID);
11   }//End of the parallel region
12 }
```

- To compile, in a terminal, type

  `gcc -fopenmp hello_omp.c -o hello_omp`

- To run, in a terminal, type `./hello_omp`

WITS UNIVERSITY

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>)`
- Conditional parallelization:
  - `if (<scalar expression>)`

  determines whether the `parallel` construct creates threads.

Parallel Region Construct Cont.

- Interpreting an OpenMP parallel directive

```
1    #pragma omp parallel if (is_parallel==1) num_threads(8) shared
         (b) private(a) \
2      firstprivate(c) default(none)
3    {
4      /* structured block */
5    }
```

Parallel Region Construct Cont.

- You create threads in OpemMP with the parallel construct.

Example 2

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

```
1   #include <stdio.h>
2   #include <omp.h>
3   int main() {
4     //Parallel region with default number of threads
5     #pragma omp parallel num_threads(4)
6     //Start of the parallel region
7     {
8       //Runtime library function to return a thread number
9       int ID = omp_get_thread_num();
10      printf("Hello World! (Thread %d)\n", ID);
11    }//End of the parallel region
12  }
```

Parallel Region Construct Cont.

Example 3

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

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main() {
4    omp_set_num_threads(4);
5    #pragma omp parallel
6    {
7      int ID = omp_get_thread_num();
8      printf("Hello World! (Thread %d)\n", ID);
9    }
10 }
```

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`
- Good practice: An OpenMP program should always be written so that it does not assume a specific number of threads

Example 4

Compile and run “parallel_region.c”. Change the number of threads in the parallel region using different ways.

Data scoping

- Any variables that existed before a parallel region still exist inside parallel region, and are by default shared.
- Private variables
  - Using **private** clause in **#pragma** `omp parallel` directive
  - The variables declared inside a parallel region (or structured block) is also **private** by default.
  - The index variable of a worksharing loop is automatically **private**.
- After the parallel region, the original values of the privatized variables are retained if they are not modified.

OpenMP worksharing for loops

- Loops are natural candidates for parallelization if individual iterations are independent.

Computing the π Using Integration

Example 5

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.

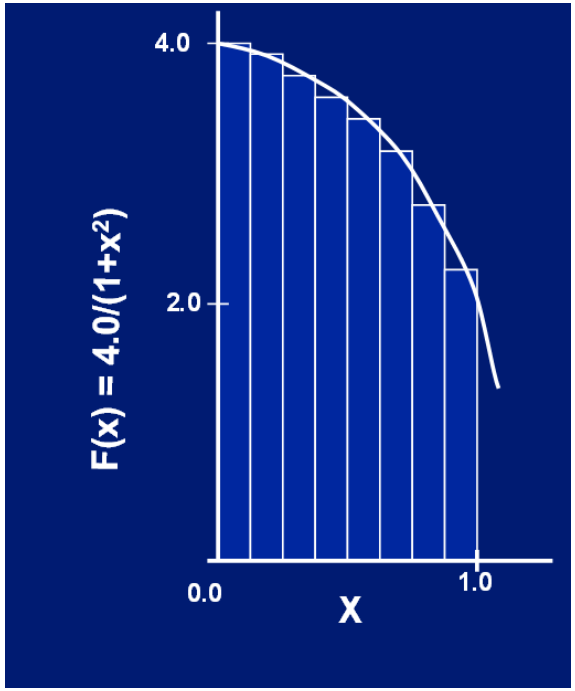

Figure: Approximating the π using numerical integration

Serial Program for Computing π

Serial Program for Computing π:

```
1  static long num_steps = 1000000;
2  double step;
3  int main () {
4    int i;
5    double x, pi, sum = 0.0;
6    step = 1.0/(double) num_steps;
7    for (i=0;i< num_steps; i++){
8      x = (i+0.5)*step;
9      sum = sum + 4.0/(1.0+x*x);
10   }
11   pi = step * sum;
12 }
```

Parallel Program for Computing π — Method I

Parallel computation of number π.

```
1  static long num_steps = 1000000;
2  double step;
3  #define NUM_THREADS 2
4  int main (){
5    int i, nthreads;
6    double pi=0.0, sum[NUM_THREADS];
7    step = 1.0/(double)num_steps;
8    omp_set_num_threads(NUM_THREADS);
9    #pragma omp parallel
10   {
11     int i, id, tthreads; double x;
12     tthreads = omp_get_num_threads();
13     id = omp_get_thread_num();
14     if(id==0) nthreads=tthreads;
15     for (i=id, sum[id]=0.0;i< num_steps; i=i+tthreads) {
16       x = (i+0.5)*step;
17       sum[id] = sum[id] + 4.0/(1.0+x*x);
18     }
19   }
20   for(i=0, pi=0.0;i<nthreads;i++)
21     pi += step * sum[i];
22 }
```

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.
- Avoiding false sharing - having each thread use its own private storage, and then update the shared variable when they are done.

False sharing cont.

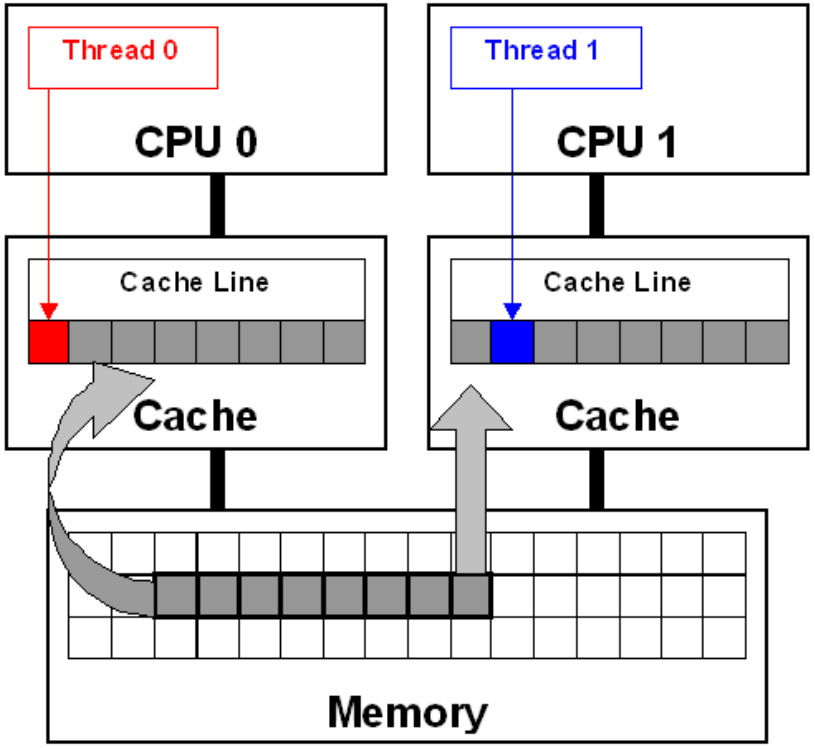

Figure: False sharing

Parallel Program for Computing Π — Method II

```
1  static long num_steps = 1000000;
2  double step;
3  #define NUM_THREADS 2
4  int main () {
5    int i, nthreads, tthreads, id;
6    double pi = 0.0, sum = 0.0, x;
7    step = 1.0/(double)num_steps;
8    omp_set_num_threads(NUM_THREADS);
9    #pragma omp parallel
10   {
11     tthreads = omp_get_num_threads();
12     id = omp_get_thread_num();
13     if(id==0) nthreads=tthreads;
14     for (i=id;i< num_steps; i=i+tthreads) {
15       x = (i+0.5)*step;
16       sum = sum + 4.0/(1.0+x*x);
17     }
18   }
19   pi += step * sum;
20 }
```

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

```
#pragma omp critical
```

```
1  float result;
2  ......
3  #pragma omp parallel
4  {
5    float B; int i, id, nthrds;
6    id = omp_get_thread_num();
7    nthrds = omp_get_num_threads();
8    for(i = id; i < nthrds; i+= nthrds) {
9      B = big_job(i);
10   }
11   #pragma omp critical
12     result += calc(B);
13   ......
14 }
```

High level synchronizations in OpenMP Cont.

- atomic: 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 $op \in (+=, -=, *=, /=, \%=)$
- $x++$
- $++x$
- $x--$
- $--x$

```
1   #pragma omp parallel
2   {
3     ......
4     double tmp, B;
5     B = calc();
6     tmp = big_calc(B);
7     #pragma omp atomic
8       X += tmp;
9     ......
10  }
```

WITS UNIVERSITY

High level synchronizations in OpenMP Cont.

- barrier: Each thread waits until all threads arrive.

```
#pragma omp barrier
```

```
1  #pragma omp parallel
2  {
3    int id=omp_get_thread_num();
4    A[id]=calc1(id);
5    #pragma omp barrier
6    B[id]=calc2(id, A);
7    ......
8  }
```

Outline

Worksharing Construct

- A parallel construct by itself creates an SPMD program, i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team? Worksharing.
- Worksharing
  - Loop construct: Splits up a loop iterations among the threads in a team. The name of the loop construct is `for`.
  - sections/section constructs
  - single construct
  - task construct

Loop Construct

Example 6

Given arrays $A[N]$ and $B[N]$. Find $A = A + B$.

```
1  ......
2  //Serial
3  for(i=0; i< N; i++)
4    a[i]=a[i]+b[i];
```

```
1  ......
2  //Using loop construct
3  #pragma omp parallel
4  #pragma omp for
5  {
6    for(i=0; i<N; i++)
7      a[i] = a[i] + b[i];
8  }
```

```
1  //Using parallel construct
2  #pragma omp parallel
3  {
4    int id, i, nthrds, istart,
         iend;
5    id i= omp_get_thread_num();
6    nthrds = omp_get_num_threads
         ();
7    istart = id * N/nthrds;
8    iend = (id+1) * N/nthrds;
9    if(id==nthrds-1)
10     iend = N;
11   for(i=istart; i<iend; i++)
12     a[i] = a[i] + b[i];
13 }
```

Loop Construct Cont.

- The **schedule** clause specifies how the iterations of the loop are assigned to the threads in a team.
- The syntax is **#pragma omp for schedule(kind [, chunk])**.
- Schedule kinds:
  - **schedule(static [,chunk])**: Deals out blocks of iterations of size `chunk` to each thread in a round robin fashion.
    - The iterations can be assigned to the threads before the loop is executed.
  - **schedule(dynamic [,chunk])**: Each thread grabs `chunk` size of iterations off a queue until all iterations have been handled. The default is 1.
    - The iterations are assigned while the loop is executing

Schedule clause

- **schedule(guided [,chunk])**: Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size `chunk` as the calculation proceeds. The default is 1.
- **schedule(runtime)**: Schedule and chunk size taken from the OMP_SCHEDULE environment variable.
  - For example, `export OMP_SCHEDULE="static,1"`
- **schedule(auto)**: Left up to the runtime or compiler to choose.

More on schedule

- Most OpenMP implementations use a roughly block partition.
- There is some overhead associated with schedule.
- The overhead for `dynamic` is greater than `static`, and the overhead for `guided` is the greatest.
- If each iteration of a loop requires roughly the same amount of computation, then it is likely that the default distribution will give the best performance.
- If the cost of the iterations decreases linearly as the loop executes, then a static schedule with small chunk size will probably give the best performance.
- If the cost of each iteration can not be determined in advance, then `schedule(runtime)` can be used.

The number of threads active

The `omp_set_dynamic` routine enables or disables dynamic adjustment of the number of threads available for the execution of subsequent parallel regions.

- `omp_set_dynamic()` – A call to this function with nonzero argument allows OpenMP to choose any number of threads between 1 and the set number of threads.

Example 7

```
 omp_set_dynamic(1);
#pragma omp parallel num_threads(8)
```

allows the OpenMP implementation to choose any number of threads between 1 and 8.

The number of threads active cont.

Example 8

```
omp_set_dynamic(0);
#pragma omp parallel num_threads(8)
```

only allows the OpenMP implementation to choose 8 threads. The action in this case is implementation dependent.

- `omp_get_dynamic()` – You can determine the default setting by calling this function.

Example

Example 9

Add various `schedule` clause in `Example_ploop.c` to see the number of threads created in the parallel region.

Example 10

Use `omp_set_dynamic()` in `Example_ploop.c` to inspect the interaction with `num_threads` clause.

Loop Construct Cont.

Basic approach to parallelize a loop:

- Find compute intensive loops
- Make the loop iterations independent, so they can be safely executed in any order without loop carried dependencies.
- Place the appropriate OpenMP directives and test.

Example 11

Removing a loop carried dependency.

```
1  //Loop dependency
2  int i, j, A[MAX];
3  j=5;
4  for(i=0; i<MAX; i++){
5    j+=2;
6    A[i]=big(j);
7  }
8  //Removing loop dependency
9  int i, A[MAX];
10 //Shortcut: "parallel for"
11 #pragma omp parallel for
12   for(i=0; i<MAX; i++){
13     int j=5+2*(i+1);
14     A[i]=big(j);
15   }
```

Reduction

```
1  ......
2  double ave=0.0, A[MAX];
3  int i;
4  for(i=0;i<MAX;i++){
5    ave+=A[i];
6  }
7  ave = ave/MAX;
8  ......
```

We are aggregating multiple values into a single value—**reduction**. Reduction operation is supported in most parallel programming environments.

- OpenMP reduction clause: *reduction(op:list)*.
- Inside a parallel or a work-sharing construct
  - A local copy of each list variable is made and initialized depending on the operation specified by the operator “op”.
  - Each thread updates its own local copy
  - Local copies are aggregated into a single value.

Reduction Cont.

```
1  ......
2  double ave=0.0, A[MAX];
3  int i;
4  #pragma omp parallel for
       reduction(+:ave)
5    for(i=0;i<MAX;i++){
6      ave+=A[i];
7    }
8  ave = ave/MAX;
9  ......
```

- Table below shows associative operands that can be used with reduction (for C/C++) and their common initial values.

Op	Initial value	Op	Initial value
+	0	&	∼0
*	1	|	0
-	0	^	0
min	Large number (+)	&&	1
max	Most neg. number	||	0

Data dependences

- Data dependences occur in loops in which the computation in one iteration depends on the results of one or more previous iterations.

```
1       fibo[0]=1;
2       fibo[1]=1;
3       for(i=2;i<n;i++)
4           fibo[i]=fibo[i-1]+fibo[i-2];
```

- If we parallelize this code segment as follows, what happens?

```
1       fibo[0]=1;
2       fibo[1]=1;
3       #pragma omp parallel for num_threads(thread count)
4           for(i=2;i<n;i++)
5               fibo[i]=fibo[i-1]+fibo[i-2];
```

Data dependences cont.

- A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.
- The dependence of the computation of `fibo[6]` on the computation of `fibo[5]` is called a data dependence.
- Since the value of `fibo[5]` is calculated in one iteration, and the result is used in a subsequent iteration, the dependence is also called a loop carried dependence.

Data dependence cont.

- At least one of the statements must write or update the variable in order for the statements to represent a dependence;
- In order to detect a loop carried dependence, we should only concern ourselves with variables that are updated by the loop body;
- That is, we should look for variables that are read or written in one iteration, and written in another.

Summary

- OpenMP parallel programming model
- OpenMP core features: parallel construct and worksharing

References

- Introduction to High Performance Computing for Scientists and Engineers by Georg Hager and Gerhard Wellein, Chapter 6.
- Parallel Programming in C with MPI and OpenMP, by Michael J. Quinn, Chapter 17.
- Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation), by Barbara Chapman, Gabriele Jost and Ruud van der Pas. The MIT Press, 2007.
- `https://hpc-tutorials.llnl.gov/openmp/`
- Various resources are avaialble at `https://www.openmp.org/resources/`
- OpenMP Application Programming Interface Examples, `https://www.openmp.org/wp-content/uploads/openmp-examples-5.0.0.pdf`

WITS UNIVERSITY