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

Hairong Wang

School of Computer Science,
University of the Witwatersrand, Johannesburg

2019-2-21

Contents

Outline

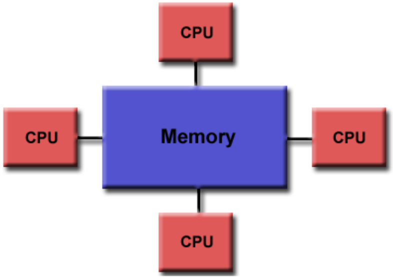
What is OpenMP?

Open specifications for Multi Processing

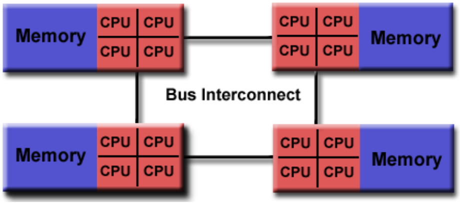
- An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism.
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- API is specified for C, C++ and Fortran.
- Portable
-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

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;
    - Asssigned 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.

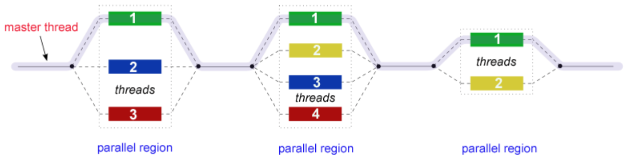


Figure: Fork-join model

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 constructs [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**⟩

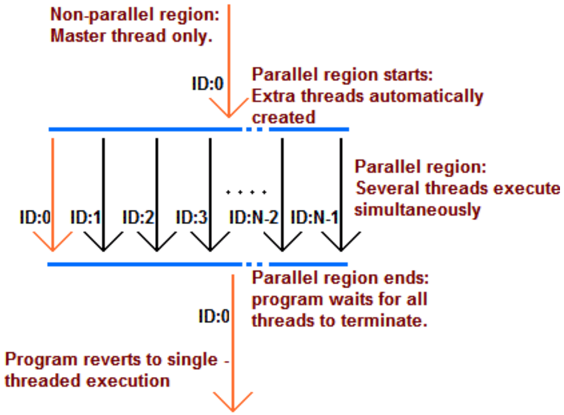

Non-parallel region:
Master thread only.
ID:0
Parallel region starts:
Extra threads automatically created
Parallel region:
Several threads execute simultaneously
ID:0
ID:1
ID:2
ID:3
....
ID:N-2
ID:N-1
Parallel region ends:
program waits for all threads to terminate.
ID:0
Program reverts to single - threaded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

```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>)`
- Conditional parallelization:`if (<scalar expression>)`, determines whether the `parallel` construct creates threads.

Parallel Region Construct Cont.

- Interpreting an OpenMP parallel directive

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

Parallel Region Construct Cont.

- You create threads in OpemMP with the parallel construct.

Example (1)

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

-

```c
#include <stdio.h>
#include <omp.h>
int main()
{
  //Parallel region with default 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()
{
  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 environmental variable OMP_NUM_THREADS, e.g., export OMP_NUM_THREADS=8

Exercise 2: Compile and run “parallel_region.c”. change the number of threads in the parallel region using different ways.

Computing the π Using Integration

Exercise 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.

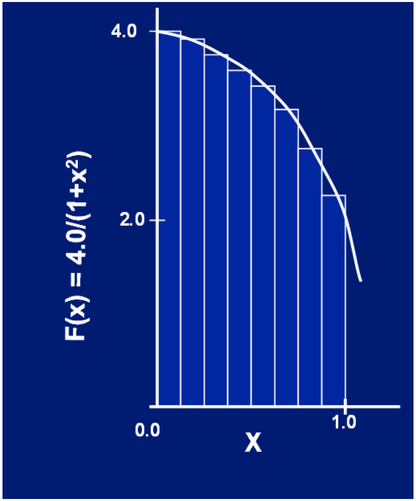


Figure: Approximating the π using numerical integration

Serial Program for Computing Π

```c
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;
}
```

Exercise 4: Parallelize the serial pi program `pi.c` using OpenMP parallel construct.

Parallel Program for Computing Π — Method I

```
static long num_steps = 1000000;
double step;
#define NUM_THREADS 2
int main (){
  int i, 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];
```

False Sharing (1)

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.**

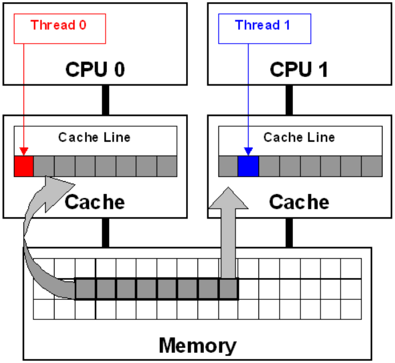


Figure: False sharing

False Sharing (2)

- 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 variable, what will be the value of `my_z`?

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

Cache Coherence (1)

Example (3)

Suppose $x = 2$ is a shared variable, $y0, y1, z1$ are private variable. 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 (2)

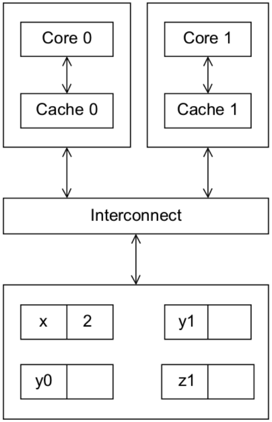


Figure: Example - cache coherence cont.

Cache Coherence (3)

Cache coherence problem - The caches for single processor systems provide no mechanism for insuring that when the caches of multiple processors store the same variable, an update by one processor to the cache variable is ‘seen’ by the other processors. That is, the cached value stored by the other processors is also updated.

False Sharing (3)

- 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.

Parallel Program for Computing Π — Method II

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

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 (4)

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

```c
......
//Serial
for(i=0; i< N; i++)
  a[i]=a[i]+b[i];
```

```c
......
//Using loop construct
#pragma omp parallel
#pragma omp for
{
  for(i=0; i<N; i++)
    a[i] = a[i] + b[i];
}
```

```c
//Using parallel construct
#pragma omp parallel
{
  int id, i, nthrds, istart,
      iend;
  id i= omp_get_thread_num()
      ;
  nthrds =
       omp_get_num_threads();
  istart = id * N/nthrds;
  iend = (id+1) * N/nthrds;
  if(id==nthrds-1)
    iend = N;
  for(i=istart; i<iend; i++)
    a[i] = a[i] + b[i];
}
```

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 (1)

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

```
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 (2)

Example

```
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 (5)

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

Example (6)

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 safely execute in any order without loop carried dependencies.
- Place the appropriate OpenMP directives and test.

Example

Removing a loop carried dependency.

```
//Loop dependency
int i, j, A[MAX];
j=5;
for(i=0; i<MAX; i++){
  j+=2;
  A[i]=big(j);
  }
//Removing loop dependency
int i, A[MAX];
//Shortcut: "parallel for"
#pragma omp parallel for
  for(i=0; i<MAX; i++){
    int j=5+2*(i+1);
    A[i]=big(j);
  }
```

Reduction

```
......
double ave=0.0, A[MAX];
int i;
for(i=0;i<MAX;i++){
  ave+=A[i];
}
ave = ave/MAX;
......
```

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.

```
......
double ave=0.0, A[MAX];
int i;
#pragma omp parallel for
    reduction(+:ave)
  for(i=0;i<MAX;i++){
    ave+=A[i];
  }
ave = ave/MAX;
......
```

- 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

Exercise

- **Exercise 6:** Parallelize the serial pi program with a loop construct. Your goal is to minimize the number of changes made to the serial code.
- **Exercise 7:** Parallelize *saxpy*:

$$y(i) = a * x(i) + y(i), 1 \leq i \leq n. \tag{1}$$

  Parallelize the serial version `saxpy_v0.c` of *saxpy*.
- **Exercise 8:** Parallelize the dot product of two vectors. That is, given two vectors X and Y with n elements each,

$$\mathbf{X} \cdot \mathbf{Y} = \sum_{i=1}^{n} x_i y_i. \tag{2}$$

  Explore using different OpenMP parallelization methods. You may start with the serial version `dotProd_v0.c`

Data dependences

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

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

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

```
fibo[0]=1;
fibo[1]=1;
#pragma omp parallel for num threads(thread count)
for(i=2;i<n;i++)
  fibo[i]=fibo[i-1]+fibo[i-2];
```

WITS
UNIVERSITY

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.

Sections/Section Construct

- `sections` directive enables specification of **task parallelism**
- The `sections` worksharing construct gives a different structured blocks to each thread.
- Syntax:

```
#pragma omp sections [clause[[,] clause]...]
{
  [#pragma omp section]
    structured block
  [#pragma omp section
    structured block]
  ...
}
```

- Each section must be a structured block of code that is independent of other sections.

Sections/Section Construct Cont.

An example:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      x_calc();
    #pragma omp section
      y_calc();
    #pragma omp section
      z_calc();
  }
}
```

Exercises

- Complete all the exercises in the class.
- Read Chapter 17 in Quinn's book

References

- 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://computing.llnl.gov/tutorials/openMP/#Introduction`
- OpenMP Application Programming Interface, `http://www.openmp.org/mp-documents/openmp-4.5.pdf`
- OpenMP Application Programming Interface Examples, `http://www.openmp.org/mp-documents/openmp-examples-4.0.2.pdf`