THE UNIVERSITY OF WESTERN AUSTRALIA

High-Performance Computing

Lecture 4 OpenMP Tasking and Synchronisation

CITS5507

Zeyi Wen

Computer Science and Software Engineering

School of Maths, Physics and Computing

Acknowledgement: The lecture slides are adapted from many online sources.

Outline

- **OpenMP Task**
  - ✓ What is task
  - ✓ Task generation and execution
  - ✓ Data environment of tasks
  - ✓ Task Switching
- Synchronisation
  - ✓ Why synchronisation
  - ✓ Synchronisation constructs
  - ✓ Critical and atomic
  - ✓ Flush and lock

Tasks in OpenMP

- Key concept: OpenMP always had tasks, the parallel computing community just never called them that.
  - ➢ Thread encountering *parallel* construct packages up a set of implicit tasks, one per thread.
  - ➢ Team of threads is created. Each thread in team is assigned to one of the tasks (and tied to it).
  - ➢ Barrier holds original master thread until all implicit tasks are finished.

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

Assume used 4 threads

Sequential code

```
for(i=0; i<N; i++){
   c[i] = a[i]+b[i];
}
```


(Lecture 3) Loop Worksharing Construct

- The loop worksharing construct splits up loop iterations among the threads in a team.

```
#pragma omp parallel
 {
   #pragma omp for
      for(i=0; i<N; i++){
         do_something(i);
      }
 }
```

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

Assume used 4 threads

Sequential code

```
for(i=0; i<N; i++){
   c[i] = a[i]+b[i];
}
```

(Lecture 3) Loop Worksharing Example

Vector Addition: **c = a + b**

a	1	5	9	...	61	1	3	5	...	9	41	63	2	...	33	4	0	3	6	51
b	3	3	1	...	7	5	4	1	...	2	3	7	1	...	6	2	5	7	2	4
c																				

Tasks in OpenMP

- Key concept: OpenMP always had tasks, the parallel computing community just never called them that.
  - Thread encountering *parallel* construct packages up a set of implicit tasks, one per thread.
  - Team of threads is created. Each thread in team is assigned to one of the tasks (and tied to it).
  - Barrier holds original master thread until all implicit tasks are finished.
- OpenMP 3.0 has simply added a way to create a task explicitly for the team to execute.
- Every part of an OpenMP program is part of one task or another!

OpenMP 3.0 and Task

The official introduction of the task construct:

- The **task** construct defines an **explicit task**, which may be executed by the encountering thread or **deferred** for execution by any other thread in the team.
- The data environment of the task is determined by the data sharing attribute clauses.
- Task execution is subject to task scheduling.
  - ✓ Also see the OpenMP 3.1 documentation for the associated **taskyield** and **taskwait** directives.

Syntax of Task

The syntax of task construct

```
#pragma omp task [clause [ [,] clause] …]
        //structured code block
```

where clause can be one of:

```
if (expression)
untied
shared(list)
private(list)
firstprivate(list)
default(shared | none)
```

We will talk later…

Outline

- **OpenMP Task**
  - ✓ What is task
  - ✓ Task generation and execution
  - ✓ Data environment of tasks
  - ✓ Task Switching
- Synchronisation
  - ✓ Why synchronisation
  - ✓ Synchronisation constructs
  - ✓ Critical and atomic
  - ✓ Flush and lock

Task Generation

Example: how many tasks generated, and why?

```
/* Create threads */
#pragma omp parallel num_treads(2)
{
#pragma omp task
  t0();
#pragma omp task
  t1();
}
// Implicit barrier
```

4, as two threads generate two tasks respectively.

(Lecture 3) Single Worksharing Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a nowait clause).

```
#pragma omp parallel
{
     do_many_things();

#pragma omp single
     {
       exchange_boundaries();
     }

     do_many_other_things();
}
```

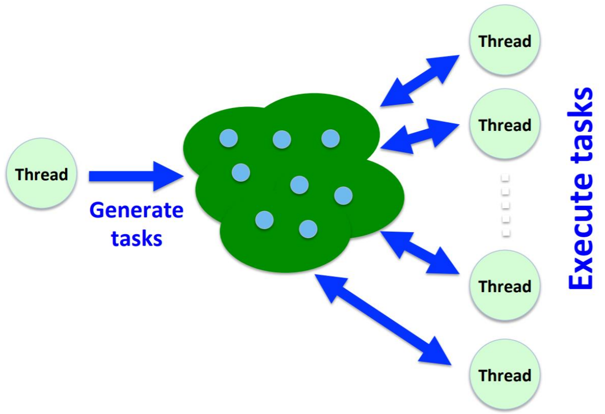

Task Generation (Continued)

The typical skeleton of a task construct

```
/* Create threads */
#pragma omp parallel num_treads(2)
{
#pragma omp single
  {
#pragma omp task
    t0();
#pragma omp task
    t1();
  }
}
// Implicit barrier
```

One generator, multiple tasks.

Task Generation and Execution



Why We Need Task?

"For" and "sections" directive defects :

- Traditional implicit tasks within "for" and "section" constructs **cannot** be dynamically partitioned based on the runtime environment;
- The schedule of tasks in "for" and "sections" must be known in advance and the "**task**" directive are used to solve this problem.

Task Execution Scheduling

- When a thread encounters a **task** construct, it may choose to execute the task immediately or defer execution until some later time.
- If a task is delayed, it is placed in the conceptual task pool associated with the current parallel region. Threads in the current group pull tasks out of the pool and execute them until the pool is empty.

(The thread executing the task may be different from the thread that initially encountered the task.)

Limitation of the For Construct

Example: the defects of "for"

```
int main()
{
int a[N];
init(a, N);
#pragma omp parallel num_threads(2)
{
#pragma omp for
for(int i = 0;i < N; i++)
    {
        process(a[i]);
    }
}
return 0;
}
```

The "for" directive can deal with this problem well, but what if we change the iteration to the following:

```
for(int i = 0;i < N; i=i+a[i])
```

Then, the "for" may not work, as the iteration loop depends on the values held in the array a[].

Limitation of the For Construct (Continued)

THE UNIVERSITY OF WESTERN AUSTRALIA

Using the task construct to solve it

```
int main(int argc, char* argv[])
{
    int a[N];
    init(a);
#pragma omp parallel num_threads(2)
{
#pragma omp single
    {
        for(int i = 0;i < N; i=i+a[i])
        {
#pragma omp task
                process(a[i]);
        }
    }
}
   return 0;
}
```

```
void init(int* a, N)
{
for(int i=0; i<N; i++)
        a[i] = i;
}
```

process(a[i]) can be printf("%d\n", a[i]);

Section v.s. Task

THE UNIVERSITY OF WESTERN AUSTRALIA

Example: Difference between “section” and “task”

```
// sections
#pragma omp sections
{
#pragma omp section
    foo();
#pragma omp section
    bar();
}
```

Sections are enclosed within the sections construct and (unless the nowait clause was specified) threads will not leave it until all sections have been executed :

```
                  [     sections      ]
Thread 0: -------< section 1 >---->*------
Thread 1: -------< section 2      >*------
Thread 2: ------------------------>*------
...                                 *
Thread N-1: ---------------------->*------
```


Section v.s. Task (Continued)

THE UNIVERSITY OF WESTERN AUSTRALIA

Example: Difference between “section” and “task”

```
// tasks
#pragma omp single nowait
{
#pragma omp task
    foo();
#pragma omp task
    bar();
}
#pragma omp taskwait
//taskwait work like barrier
```

Tasks are queued and executed whenever possible at the so-called task scheduling points, which we will talk later.

In fact, If no task scheduling points are present inside the region’s code (like above), the OpenMP runtime might start the tasks whenever it deems appropriate. Therefore, task is wiser than sections while distributing computing resources most of the time.

Section v.s. Task (Continued)

THE UNIVERSITY OF WESTERN AUSTRALIA

Here are possible scenarios of what might happen if there are three threads in last example:

```
                 +--+-->[ task queue ]--+
                 |  |                   |
                 |  |      +-----------+
                 |  |      |
Thread 0: --< single >-|  v  |-----
Thread 1: -------->|< foo() >|-----
Thread 2: -------->|< bar() >|-----
```

```
                  +--+-->[ task queue ]--+
                  |  |                   |
                  |  |     +------------+
                  |  |     |
Thread 0: --< single >-| v              |---
Thread 1: --------->|< foo() >< bar() >|---
Thread 2: --------------------->|       |---
```

```
                 +--+-->[ task queue ]--+
                 |  |                   |
                 |  |     +------------+
                 |  |     |
Thread 0: --< single >-| v < bar() >|---
Thread 1: --------->|< foo() >      |---
Thread 2: ----------------->|       |---
```

```
Thread 0: --< single: foo(); bar() >*---
Thread 1: ------------------------->*---
Thread 2: ------------------------->*---
```

(Lecture 3) Data Sharing

Data Environment:

- An important consideration for OpenMP programming is the understanding and use of data scoping.
- As OpenMP is based upon the shared memory programming model, **most variables are shared by default**.
- Global variables include:
  - ✓ File scope variables, static
- **But not everything is shared...**
- Private variables include:
  - ✓ Loop index variables
  - ✓ Stack variables in subroutines called from parallel regions

Outline

- **OpenMP Task**
  - ✓ What is task
  - ✓ Task generation and execution
  - ✓ Data environment of tasks
  - ✓ Task Switching
- Synchronisation
  - ✓ Why synchronisation
  - ✓ Synchronisation constructs
  - ✓ Critical and atomic
  - ✓ Flush and lock

Data Environment of Tasks

- Most variables are shared by default in OpenMP.
- However, the default for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope).
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared, because the barrier guarantees task completion.

Data Environment of Tasks (Continued)

The **task** directive takes the following data attribute clauses that define the data environment of the task:

- **default (shared | none)**
- **private (list)**
- **firstprivate (list)**
- **shared (list)**

Data Sharing Attribute Clauses Review

shared clause: The **shared** clause declares one or more list items to be shared by tasks generated by a parallel, teams, or task generating construct.

private clause: The **private** clause declares one or more list items to be **private** to a task or to a **SIMD** lane.

firstprivate clause: The **firstprivate** clause declares one or more list items to be **private** to a task, and initialises each of them with the value that the corresponding original item has when the construct is encountered.

default clause: Specifies the behavior of **unscoped** variables in a parallel region.

Data Scoping Example (Continued)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
#pragma omp parallel private(b)
      {
      int d = 4;
      #pragma omp task
      {
      int e = 5;
      // Scope of a: shared
      // Scope of b: firstprivate
      // Scope of c: shared
      // Scope of d: firstprivate
      // Scope of e: private
      }
    }
}
```


OpenMP Task with If-Clause

The if clause:

- When the if clause argument is false, the task is executed immediately by the encountering thread.

```
#pragma omp task if(0) //this task is undeferred
   foo();
```

It's a user directed optimisation

- When the cost of deferring the task is too high, compared to the cost of executing the task code
- To control cache and memory affinity

Barriers of Task

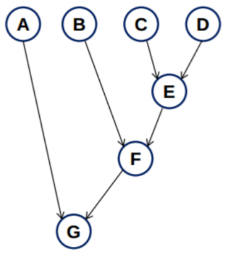
When/where are tasks complete?

- **At thread barriers, explicit or implicit**
  - ✓ applies to all tasks generated in the current parallel region up to the barrier matches user expectation

- **At task barriers**
  i.e. Wait until all tasks defined in the current task have completed.
  #pragma omp taskwait
  Note: applies only to tasks generated in the current task, not to "descendants" .

The taskwait Clause

Example: taskwait pseudocode




```
#pragma omp parallel
{
   #pragma omp single
   {
      #pragma omp task A();
      #pragma omp task if (0)
      {
             #pragma omp task B();
             #pragma omp task if (0)
             {
                #pragma omp task C();
                D();
                #pragma omp taskwait
                E();
             }
             #pragma omp taskwait
             F();
      }
      #pragma omp taskwait
      G();
   }
}
```


Task Example: Fibonacci Numbers

Example: Computing Fibonacci Numbers

```
#include <stdio.h>
#include <omp.h>
int fib(int n)
{
  int i, j;
  if (n<2)
      return n;
  else
  {
#pragma omp task shared(i) firstprivate(n)
      i=fib(n-1);
#pragma omp task shared(j) firstprivate(n)
      j=fib(n-2);
#pragma omp taskwait
      return i+j;
  }
}
```

Task scheduling point

```
int main()
{
  int n = 10;
  omp_set_dynamic(0);
  omp_set_num_threads(4);
#pragma omp parallel shared(n)
{
#pragma omp single
  printf ("fib(%d) = %d\n", n, fib(n));
}
  return 0;
}
```


Task Example: Traverse a Linked List

Example: Parallel iterate through a linked list

```
#pragma omp parallel
{
#pragma omp for private(p)
         for ( int i =0; i <numlists; i++)
         {
              p = listheads[i] ;
              while (p)
              {
#pragma omp task firstprivate(p)
                  {
                           process(p);
                  }
                  p=next(p);
              }
         }
}
```


Task Example: Tree Traversal

Example: Postorder tree traversal

```
void postorder(node *p)
{
         if (p->left)
#pragma omp task
            postorder(p->left);
         if (p->right)
#pragma omp task
            postorder(p->right);
#pragma omp taskwait // wait for descendants

         process(p->data);
}
```

Task scheduling point

- Parent task suspended until children tasks complete

Outline

- **OpenMP Task**
  - ✓ What is task
  - ✓ Task generation and execution
  - ✓ Data environment of tasks
  - ✓ Task Switching
- Synchronisation
  - ✓ Why synchronisation
  - ✓ Synchronisation constructs
  - ✓ Critical and atomic
  - ✓ Flush and lock

Task Switching: Tied and Untied

Tied and Untied

- If the code is executed by the same thread from start to finish, the task is **tied**.
- A task is **untied** if the code can be executed by multiple threads, causing different threads to execute different parts of the code.
- **By default, tasks are tied** and can be referred to as untied by using the untied clause in conjunction with the task directive.

Task Switching

Task Switching

- Certain constructs have task scheduling points at **defined** locations within them
- When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (called **task switching**)
- It can then return to the original task and resume

Task Switching (Continued)

Where are scheduling points

- The point where task construction is encountered;
- The point where the taskwait construct is encountered;
- The point where an implicit or explicit barrier is encountered;
- The "mission accomplished" point.

Task Switching Example

- Too many tasks generated in an eye-blink
- Generating task will have to **suspend** for a while

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
  {
  #pragma omp task
     process(item[i]);
  }
}
```

- With task switching, the executing thread can:
  - ✓ Execute an already generated task (draining the "**task pool**")
  - ✓ Dive into the encountered task (could be very **cache-friendly**)

Task Switching

Thread Switching

```
#pragma omp single
{
  #pragma omp task untied
  for (i=0; i<ONEZILLION; i++)
  #pragma omp task
       process(item[i]);
}
```

- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…
- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- The programmer is responsible for synchronisation!

Comments on Task Clause

Conclusions on tasks

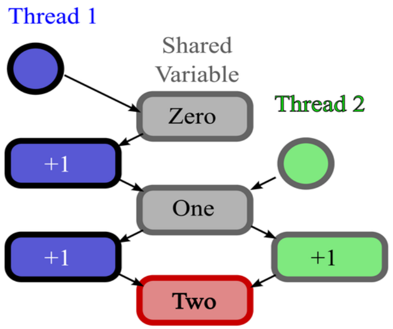
- Enormous amount of work by many people
- Tightly integrated into **3.0** spec
- Flexible model for irregular parallelism
- Provides balanced solution despite often conflicting goals
- Appears that performance can be reasonable

Outline

- **OpenMP Task**
  - ✓ What is task
  - ✓ Task generation and execution
  - ✓ Data environment of tasks
  - ✓ Task Switching
- Synchronisation
  - ✓ Why synchronisation
  - ✓ Synchronisation constructs
  - ✓ Critical and atomic
  - ✓ Flush and lock

Race Condition

- A data race occurs when two threads access the same memory without proper **synchronisation**.
- This can cause the program to produce non-deterministic results in parallel mode.



Race Condition!

Race Condition Example

- **Race Condition is discussed in Week 3 lecture.**
- **It can be elaborated by following intuitive example**

```
THREAD 1:
update(x)
{
    x = x + 1
}

x = 0
update(x)
print(x)
```

```
THREAD 2:
update(x)
{
    x = x + 1
}

x = 0
update(x)
print(x)
```


Race Condition Example (Continued)

- One possible execution sequence:
  - ✓ Thread 1 initialises x to 0 and calls update(x).
  - ✓ Thread 1 adds 1 to x and x now equals 1.
  - ✓ Thread 2 initialises x to 0 and calls update(x). x now equals 0.
  - ✓ Thread 1 prints x, which is equal to 0 instead of 1
  - ✓ Thread 2 adds 1 to x. x now equals 1.
  - ✓ Thread 2 prints x as 1.

```
THREAD 1:
update(x)
{
    x = x + 1
}

x = 0
update(x)
print(x)
```

```
THREAD 2:
update(x)
{
    x = x + 1
}

x = 0
update(x)
print(x)
```


Outline

- **OpenMP Task**
  - ✓ What is task
  - ✓ Task generation and execution
  - ✓ Data environment of tasks
  - ✓ Task Switching
- Synchronisation
  - ✓ Why synchronisation
  - ✓ Synchronisation constructs
  - ✓ Critical and atomic
  - ✓ Flush and lock

Synchronisation Constructs

- To avoid a situation like this, the updating of x must be **synchronised** between the two threads to ensure that the correct result is produced.
- OpenMP provides a variety of **Synchronisation Constructs** that control how the execution of each thread proceeds relative to other team threads.

- **OpenMP Synchronisation**:
  - Single/Master (Week 3 lecture)
  - Ordered (Week 3 lecture)
  - Barriers
  - Critical
  - Atomic
  - Flush (memory subsystem synchronisation)
  - Locks

(Lecture 3) Single Worksharing Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).

- A barrier is implied at the end of the single block (can remove the barrier with a nowait clause).

```
#pragma omp parallel
{
      do_many_things();

#pragma omp single
      {
        exchange_boundaries();
      }

      do_many_other_things();
}
```


(Lecture 3) Master Construct

- The master construct denotes a structured block that is only executed by the master thread.

- The other threads just skip it (no synchronisation is implied).

```
#pragma omp parallel
{
         do_many_things();
#pragma omp master
         {
           exchange_boundaries();
         }
#pragma omp barrier
          do_many_other_things();
}
```


(Lecture 3) Loop Worksharing Constructs: The ordered Clause

The **ordered** region executes in the sequential order.

```
void test(int first, int last)
{
#pragma omp parallel
#pragma omp for schedule(static) ordered
   for (int i = first; i <= last; ++i)
   {
      // Do something here.
      if (i % 2)
      {
         #pragma omp ordered
         printf_s("test() iteration %d\n", i);
      }
   }
}
int main(int argc, char *argv[])
{
    test(1, 8);
}
```

Output:
test() iteration 1
test() iteration 3
test() iteration 5
test() iteration 7

Exercise:
- Delete "#pragma omp ordered", compile and run the program multiple times.
- Do you see any difference?

(Lecture 3) Loop Worksharing Constructs: The ordered Clause (Cont'd)

The omp ordered directive must be used as follows:

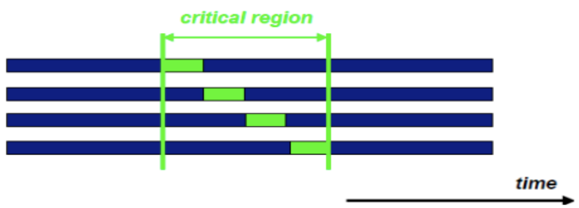
- It must appear within the extent of an "**omp for**" or "**omp parallel for**" construct containing an ordered clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.
- An iteration of a loop must not execute the same omp ordered directive more than once.
- An iteration of a loop must not execute more than one distinct omp ordered directive.

Outline

- **OpenMP Task**
  - ✓ What is task
  - ✓ Task generation and execution
  - ✓ Data environment of tasks
  - ✓ Task Switching
- Synchronisation
  - ✓ Why synchronisation
  - ✓ Synchronisation constructs
  - ✓ Critical and atomic
  - ✓ Flush and lock

Synchronisation Constructs CRITICAL Directive

- **Mutual exclusion**: the CRITICAL directive specifies a region of code that must be executed by only one thread at a time.
- If a **thread** is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.



Synchronisation Constructs CRITICAL Directive

Usage:

```
#pragma omp critical [ name ] newline
```

- **The optional name enables multiple different CRITICAL regions to exist:**
  - Names act as **global identifiers**. Different CRITICAL regions with the same name are treated as the same region.
  - All CRITICAL sections which are unnamed, are treated as the same section.

CRITICAL Construct/Example

Example:

```
#include <omp.h>
main(int argc, char *argv[]) {
   int x;
   x = 0
#pragma omp parallel shared(x)
   {
   #pragma omp critical
       x = x + 1;
   }
/* end of parallel region */
   return 0;
}
```

Notes:

- All threads in the team will attempt to execute in parallel, however, because of the CRITICAL construct surrounding the increment of **x**, only one thread will be able to read/increment/write **x** at any time.

CRITICAL Construct/Example

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
 int main() {
int i, max,  a[SIZE];
for (i = 0; i < SIZE; i++)
{
    a[i] = rand();
    printf("%d\n", a[i]);
}
max = a[0];
#pragma omp parallel for num_threads(4)
   for (i = 1; i < SIZE; i++)
   {
        if (a[i] > max)
        {
            #pragma omp critical
              if (a[i] > max) {
                  max = a[i];
              }
        }
   }
   printf("max = %d\n", max);
   return 0;
}
```

Compare ***a[i]*** and ***max*** again because max could have been changed by another thread after the comparison outside the critical section.

CRITICAL Construct/Example

Output:

```
41
18467
6334
26500
19169
15724
11478
29358
26962
24464
max = 29358
```

Notes:

- You might not get the same output because we have used random number.

Synchronisation Constructs
ATOMIC Directive

- **ATOMIC** provides **mutual exclusion** but only applies to the load/update of a memory location.
- In essence, this directive provides a **mini-CRITICAL** section.
- The directive applies only to a single, immediately following statement
- An **ATOMIC** statement must follow a specific syntax.
  - **ATOMIC** construct may only be used together with an expression statement with one of operations: **+, *, -, /, &, ^, |, <>.**

ATOMIC Construct/Example

THE UNIVERSITY OF WESTERN AUSTRALIA

Usage:

```
#pragma omp atomic [ read | write | update | capture ] newline
```

Example:

```
#include <stdio.h>
#include <omp.h>
#define MAX 10
int main() {
    int count = 0;
    #pragma omp parallel num_threads(MAX)
    {
            #pragma omp atomic
            count++;
    }
    printf("Number of threads: %d\n", count);
    return 0;
}
```

Output:

```
Number of threads: 10
```


CRITICAL v.s. ATOMIC

THE UNIVERSITY OF WESTERN AUSTRALIA

- **Critical section:**
  - Ensures serialisation of blocks of code.
  - Can be extended to serialise groups of blocks with proper use of "name" tag.
  - Slower!
- **Atomic operation:**
  - Is much faster!
  - Only ensures the serialisation of a particular operation.

Outline

THE UNIVERSITY OF WESTERN AUSTRALIA

- **OpenMP Task**
  - ✓ What is task
  - ✓ Task generation and execution
  - ✓ Data environment of tasks
  - ✓ Task Switching
- Synchronisation
  - ✓ Why synchronisation
  - ✓ Synchronisation constructs
  - ✓ Critical and atomic
  - ✓ Flush and lock

Synchronisation Constructs FLUSH Directive

THE UNIVERSITY OF WESTERN AUSTRALIA

- The **FLUSH** directive identifies a synchronisation point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point.
- Usage: `#pragma omp flush (list) newline`
- The optional **list** contains a list of named variables that will be flushed in order to avoid flushing all variables.
- For **pointers** in the list, note that the pointer itself is flushed, not the object it points to.

Synchronisation Constructs
FLUSH Directive

- Implementations must ensure any prior modifications to **thread-visible** variables are visible to all threads after this point;
  - ie. compilers must restore values from registers to memory, hardware might need to flush write buffers, etc.
- The **FLUSH** directive is implied for the directives shown in the table below. The directive is not implied if a **NOWAIT** clause is present.

C / C++
barrier **parallel** - upon entry and exit **critical** - upon entry and exit **ordered** - upon entry and exit **for** - upon exit **sections** - upon exit **single** - upon exit

Synchronisation Constructs
FLUSH Directive

```c
#include <stdio.h>
#include <omp.h>
void read(int *data) {
  printf("read data\n");
  *data = 1;
}
void process(int *data) {
  printf("process data\n");
  (*data)++;
}
int main()
{
  int data;
  int flag;
  flag = 0;
  #pragma omp parallel sections num_threads(2)
{
  #pragma omp section
{
  printf("Thread %d: ", omp_get_thread_num( ));
  read(&data);
  #pragma omp flush(data)
  flag = 1;
  #pragma omp flush(flag)
// Do more work.
}
#pragma omp section
{
  while (!flag) {
  #pragma omp flush(flag)
  }
  #pragma omp flush(data)

  printf("Thread %d: ", omp_get_thread_num());
  process(&data);
  printf("data = %d\n", data);
}
}
  return 0;
}
```

Output:

```
Thread 0: read data
Thread 1: process data
data = 2
```


Synchronisation Constructs
Lock Directive

- A **lock** implies a memory fence of all thread visible variables.
- These routines are used to guarantee that only one thread accesses a variable at a time to avoid **race conditions**.
- C/C++ lock variables must have type "**omp_lock_t**" or "**omp_nest_lock_t**".
- All lock functions require an argument that has a pointer to **omp_lock_t** or **omp_nest_lock_t**.
- Simple Lock routines:
  - **omp_init_lock(omp_lock_t*);**
  - **omp_set_lock(omp_lock_t*);**
  - **omp_unset_lock(omp_lock_t*);**
  - **omp_test_lock(omp_lock_t*);**
  - **omp_destroy_lock(omp_lock_t*);**

Synchronisation Constructs
Lock Directive

General Procedure to Use Locks

- Define the **lock** variables.
- Initialise the **lock** via a call to **omp_init_lock.**
- Set the **lock** using **omp_set_lock** or **omp_test_lock**. The latter checks whether the **lock** is actually available before attempting to set it. It is useful to achieve asynchronous thread execution
- Unset a **lock** after the work is done via a call to **omp_unset_lock**.
- Remove the **lock** association via a call to **omp_destroy_lock**.

Synchronisation Constructs
Lock Directive

Example:

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main()
{
  int x;
  omp_lock_t lck;

  omp_init_lock (&lck);
  omp_set_lock (&lck);
  x = 0;

  #pragma omp parallel shared (x)
  {
    #pragma omp master
    {
    x = x + 1;
    omp_unset_lock (&lck);
    }
  }
  omp_destroy_lock (&lck);
  return 0;
}
```


References

- Readings
  - Section vs Task
  - OpenMP Tasking Tutorial
  - Lock vs nested lock