High-Performance Computing

Lecture 3 OpenMP Constructs, Data Sharing and Copying

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

- **Worksharing Constructs**
  - ✓ Loop worksharing construct
    - Ordered and schedule clauses
  - ✓ Reduction
- Data Sharing
  - ✓ Private
  - ✓ Firstprivate and lastptivate
  - ✓ Threadprivate
- Data Copying
- Other Constructs
  - ✓ Single worksharing construct
  - ✓ Sections worksharing construct
  - ✓ Master construct

(Lecture 2) 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);
         }
  }
```

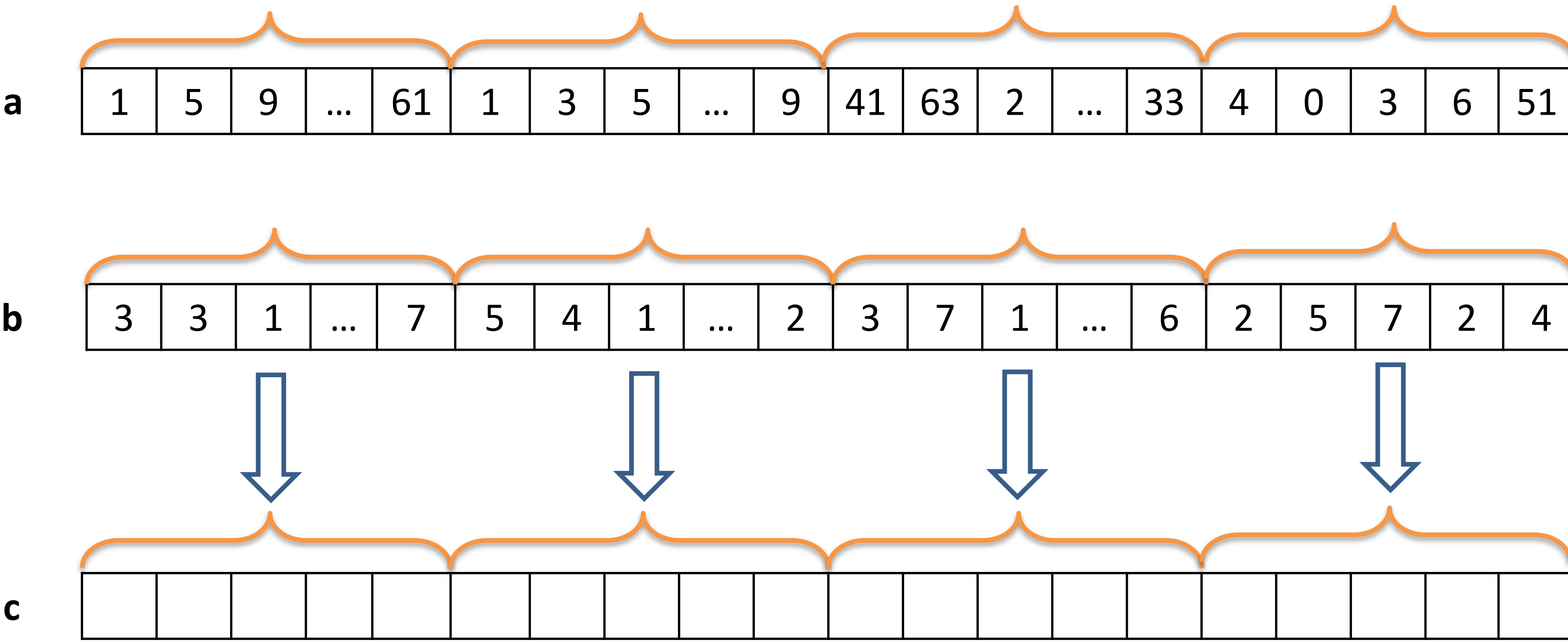
```
#pragma omp parallel
  {
     #pragma omp for
         for(i=0; i<N; i++){
             c[i] = a[i]+b[i];
         }
  }
```

Sequential code

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

Loop Worksharing Construct (Example)

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



Loop Worksharing Construct

- The loop worksharing construct splits up loop iterations among the threads in a team.
- The following are 3 equivalent examples of vector addition.

```
//Example 1.
#pragma omp parallel
{
   int id, i, Nthrds, istart, iend;
   id = 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++)
   {
       c[i] = a[i] + b[i];
   }
}
```

- Manually assign the loop to each thread, which is complicated.
- The same work can be done succinctly by using “parallel for”.

Loop Worksharing Construct (Cont'd)

```
//Example 2.
#pragma omp parallel
#pragma omp for
    for(i=0;i<N;i++)
    {
       c[i] = a[i] + b[i];
     }
```

- OpenMP parallel region and a worksharing for construct

```
//Example 3.
#pragma omp parallel for
    for(i=0;i<N; i++)
    {
        c[i] = a[i] + b[i];
    }
```

- OpenMP shortcut: Put the "parallel" and the worksharing directive on the same line

Note: loop index "i" is private by default.

Loop Worksharing Construct (Cont'd)

Working with loops

- 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 directive and test

```
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++)
{
   j +=2;
   A[i] = big(j);
}
```

Remove loop carried dependence

```
int i, A[MAX];
#pragma omp parallel for
  for (i=0;i< MAX; i++)
  {
     int j = 5 + 2* (i+1);
     A[i] = big(j);
  }
```

Outline

- **Worksharing Constructs**
  - ✓ Loop worksharing construct
    - Ordered and schedule clauses
  - ✓ Reduction
- Data Sharing
  - ✓ Private
  - ✓ Firstprivate and lastptivate
  - ✓ Threadprivate
- Data Copying
- Other Constructs
  - ✓ Single worksharing construct
  - ✓ Sections worksharing construct
  - ✓ Master construct

Loop Worksharing Constructs: The ordered Clause

THE UNIVERSITY OF WESTERN AUSTRALIA

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?

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.

Loop Worksharing Constructs: The schedule Clause

- The **schedule** clause is sometimes used in parallel for loops.
- When the amount of computation for each iteration in the loop is not equal, simply assigning the same number of iterations to each thread will result in unbalanced computing.
- This causes some lines to execute first and some to execute later, thus causing CPU idle and affecting program performance. For example, the following code:

```
int i, j;
int a[100][100] = {0};
for ( i =0; i < 100; i++)
{
   for( j = i; j < 100; j++ )
   {
      a[i][j] = i*j;
   }
}
```

- If you parallelise the outer loop, say using 4 threads, and each thread is evenly allocated 25 iterations of the loop.
- There is a 100-fold difference in the amount of computation between i = 0 and i = 99.

Loop Worksharing Constructs: The schedule Clause (Cont'd)

The schedule clause affects how loop iterations are mapped onto threads to solve these problems.

schedule(type[,size])

- **type**
  The kind of scheduling can be: static, dynamic, guided, or runtime
- **size**
  (Optional) Specifies the size of iterations. *size* must be an integer. Not valid when *type* is runtime

Loop Worksharing Constructs: The schedule Clause (Cont'd)

- **schedule(static [,chunk])**
  – Deal-out blocks of iterations of size "chunk" to each thread.

- **schedule(dynamic[,chunk])**
  – Each thread grabs "chunk" iterations off a queue until all iterations have been handled.

- **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.

- **schedule(runtime)**
  – Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0).

Loop Worksharing Constructs: The schedule Clause

Schedule Clause	When to Use
static	Pre-determined and predictable by the programmer
dynamic	Unpredictable, highly variable work per iteration
guided	Special case of dynamic to reduce scheduling overhead

least work at run-time: scheduling done at compile-time

most work at run-time: complex scheduling logic used at run-time

Loop Worksharing Constructs: The schedule Clause (Cont'd)

```c
#include<stdio.h>
#include<omp.h>
#include<unistd.h>
#define NUM_THREADS 4
#define STATIC_CHUNK 5
#define DYNAMIC_CHUNK 5
#define NUM_LOOPS 20
#define SLEEP_EVERY_N 3
int main( )
{
int nStatic1[NUM_LOOPS], nStaticN[NUM_LOOPS];
int nDynamic1[NUM_LOOPS], nDynamicN[NUM_LOOPS];
int nGuided[NUM_LOOPS];

omp_set_num_threads(NUM_THREADS);

#pragma omp parallel
{
   #pragma omp for schedule(static, 1)
   for (int i = 0 ; i < NUM_LOOPS ; ++i)
   {
      if((i % SLEEP_EVERY_N) == 0)
          sleep(0);
      nStatic1[i] = omp_get_thread_num();
   }

   #pragma omp for schedule(static, STATIC_CHUNK)
   for (int i = 0 ; i < NUM_LOOPS ; ++i)
   {
      if((i % SLEEP_EVERY_N) == 0)
         sleep(0);
      nStaticN[i] = omp_get_thread_num();
   }
```

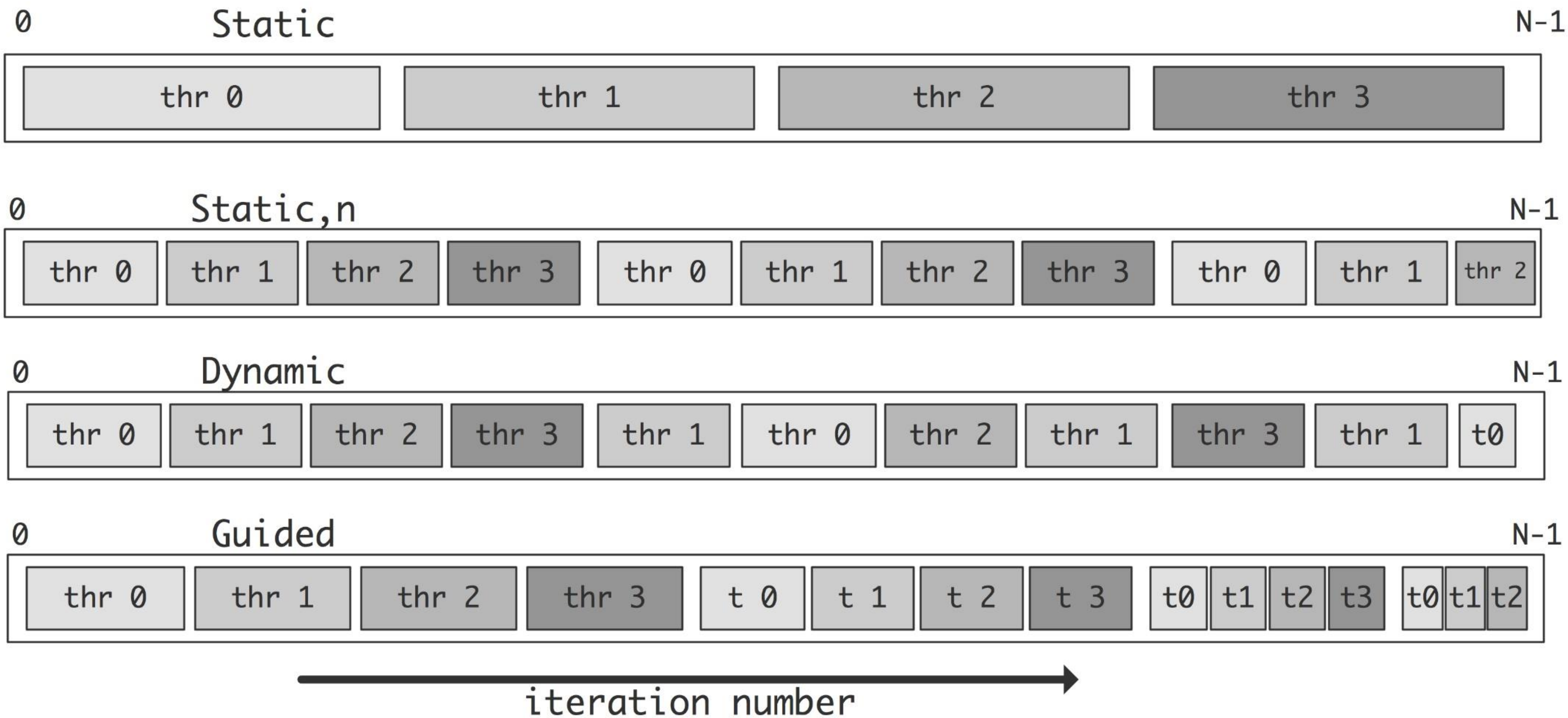
```c
#pragma omp for schedule(dynamic, 1)
for (int i = 0 ; i < NUM_LOOPS ; ++i)
{
   if((i % SLEEP_EVERY_N) == 0)
       sleep(0);
   nDynamic1[i] = omp_get_thread_num( );
}

#pragma omp for schedule(dynamic, DYNAMIC_CHUNK)
for (int i = 0 ; i < NUM_LOOPS ; ++i)
{
   if((i % SLEEP_EVERY_N) == 0)
       sleep(0);
   nDynamicN[i] = omp_get_thread_num( );
}

#pragma omp for schedule(guided)
for (int i = 0 ; i < NUM_LOOPS ; ++i)
{
   if((i % SLEEP_EVERY_N) == 0)
       sleep(0);
   nGuided[i] = omp_get_thread_num( );
}
}
}
```

Loop Worksharing Constructs: The schedule Clause (Cont'd)

Possible Output:

```
-------------------------------------------------------------
| static | static | dynamic | dynamic | guided |
|   1    |   5    |    1    |    5    |        |
-------------------------------------------------------------
|   0    |   0    |    0    |    2    |   1    |
|   1    |   0    |    3    |    2    |   1    |
|   2    |   0    |    3    |    2    |   1    |
|   3    |   0    |    3    |    2    |   1    |
|   0    |   0    |    2    |    2    |   1    |
|   1    |   1    |    2    |    3    |   3    |
|   2    |   1    |    2    |    3    |   3    |
|   3    |   1    |    0    |    3    |   3    |
|   0    |   1    |    0    |    3    |   3    |
|   1    |   1    |    0    |    3    |   2    |
|   2    |   2    |    1    |    0    |   2    |
|   3    |   2    |    1    |    0    |   2    |
|   0    |   2    |    1    |    0    |   3    |
|   1    |   2    |    2    |    0    |   3    |
|   2    |   2    |    2    |    0    |   0    |
|   3    |   3    |    2    |    1    |   0    |
|   0    |   3    |    3    |    1    |   1    |
|   1    |   3    |    3    |    1    |   1    |
|   2    |   3    |    3    |    1    |   1    |
|   3    |   3    |    0    |    1    |   3    |
-------------------------------------------------------------
```

The schedule Clause: Overview



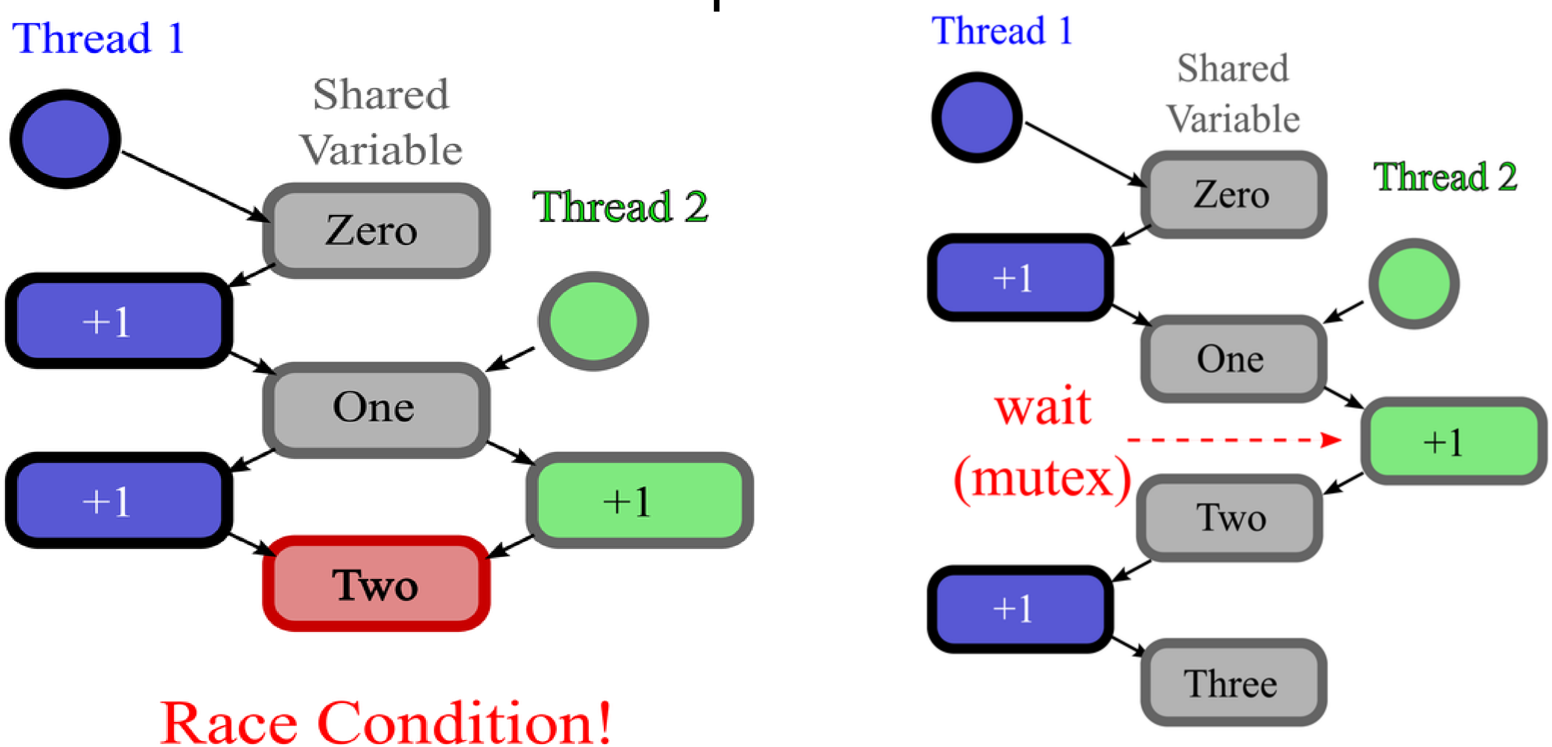
Outline

- **Worksharing Constructs**
  - ✓ Loop worksharing construct
    - Ordered and schedule clauses
  - ✓ Reduction
- Data Sharing
  - ✓ Private
  - ✓ Firstprivate and lastptivate
  - ✓ Threadprivate
- Data Copying
- Other Constructs
  - ✓ Single worksharing construct
  - ✓ Sections worksharing construct
  - ✓ Master construct

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.



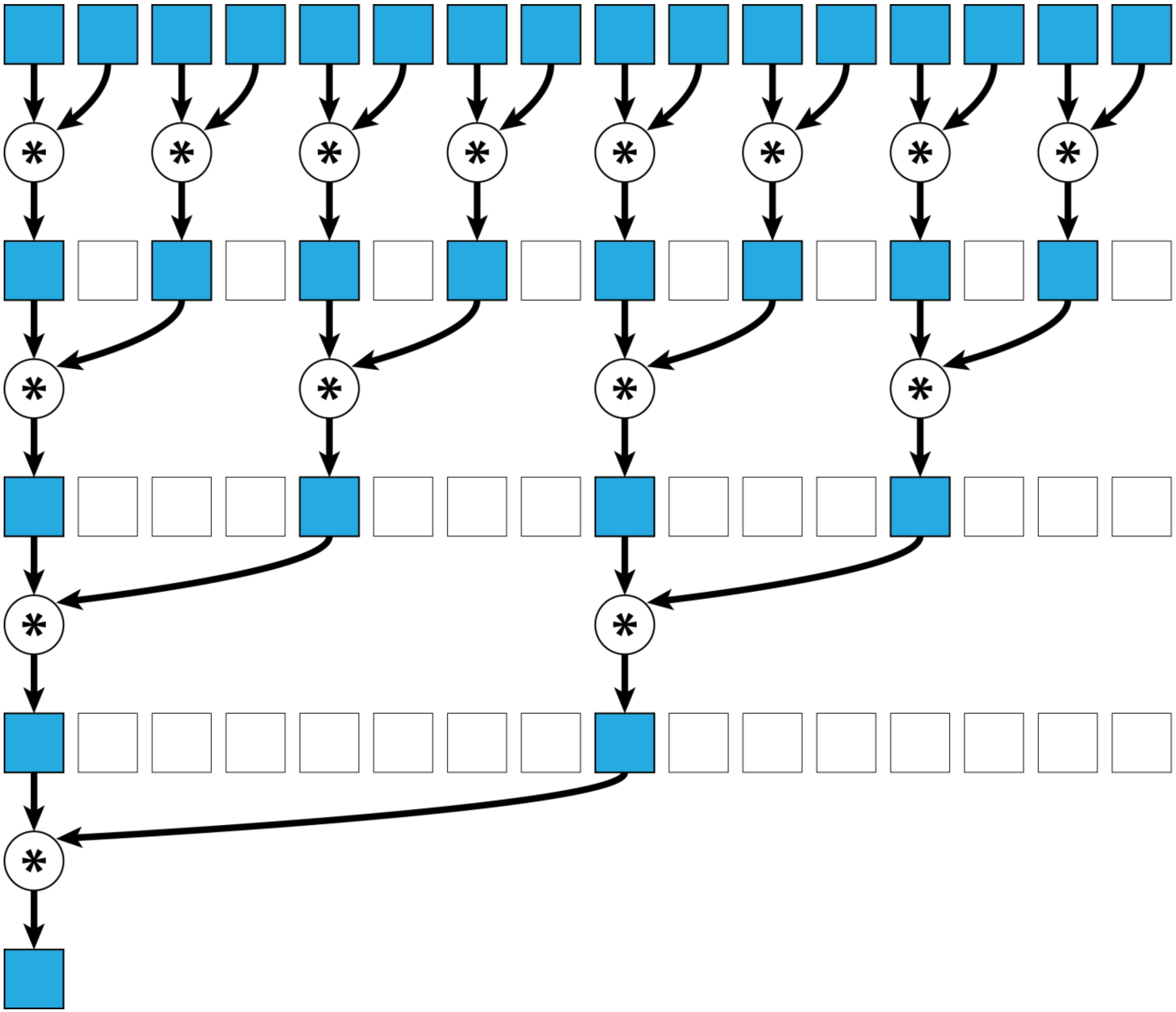
Race Condition

- **What happen in this case?**

```
sum = 0;
#pragma omp parallel for
    for(int i = 0; i < 10; i++)
    {
        sum += a[i]
    }
```

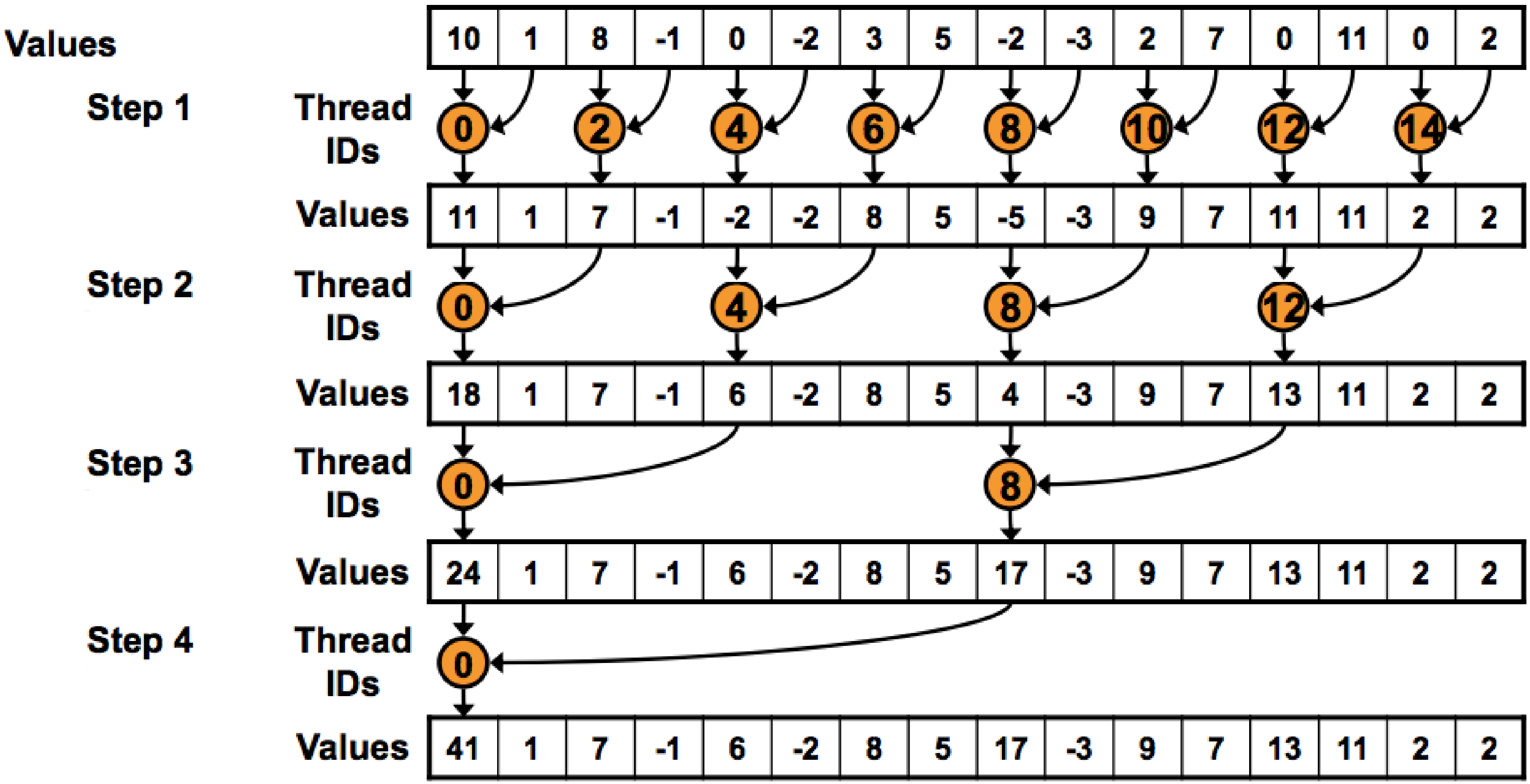
- A shared variable (sum) is modified in every iteration.
- Resulting in race condition, because multiple threads could try to update the shared variable at the same time if we parallelise the “for” loops by simply adding **#pragma omp parallel for**.
- The problem can be solved by reduction clause in OpenMP.

Parallel Reduction



Parallel Reduction (Continued)

- Many problems can be modelled as a reduction problem.
  - ✓ Sum, Min, Max, …



Reduction in OpenMP

THE UNIVERSITY OF WESTERN AUSTRALIA

- OpenMP has the special reduction clause which can express the reduction of a for loop.
- OpenMP reduction clause:
  - ✓ reduction (op : list)
- Implementation:
  - ✓ OpenMP creates a team of threads and then shares the iterations of the for loop between the threads.
  - ✓ Each thread has its own **local** copy of the reduction variable, and the thread modifies only the local copy of this variable.
  - ✓ Therefore, there is no data race.
  - ✓ When the threads join together, all the local copies of the reduction variable are combined to the global shared variable.

Reduction in OpenMP

THE UNIVERSITY OF WESTERN AUSTRALIA

Implementation:

```
sum = 0;
#pragma omp parallel for reduction(+: sum)
        for (int i = 0; i < 9; i++)
        {
              sum += a[i]
        }
```

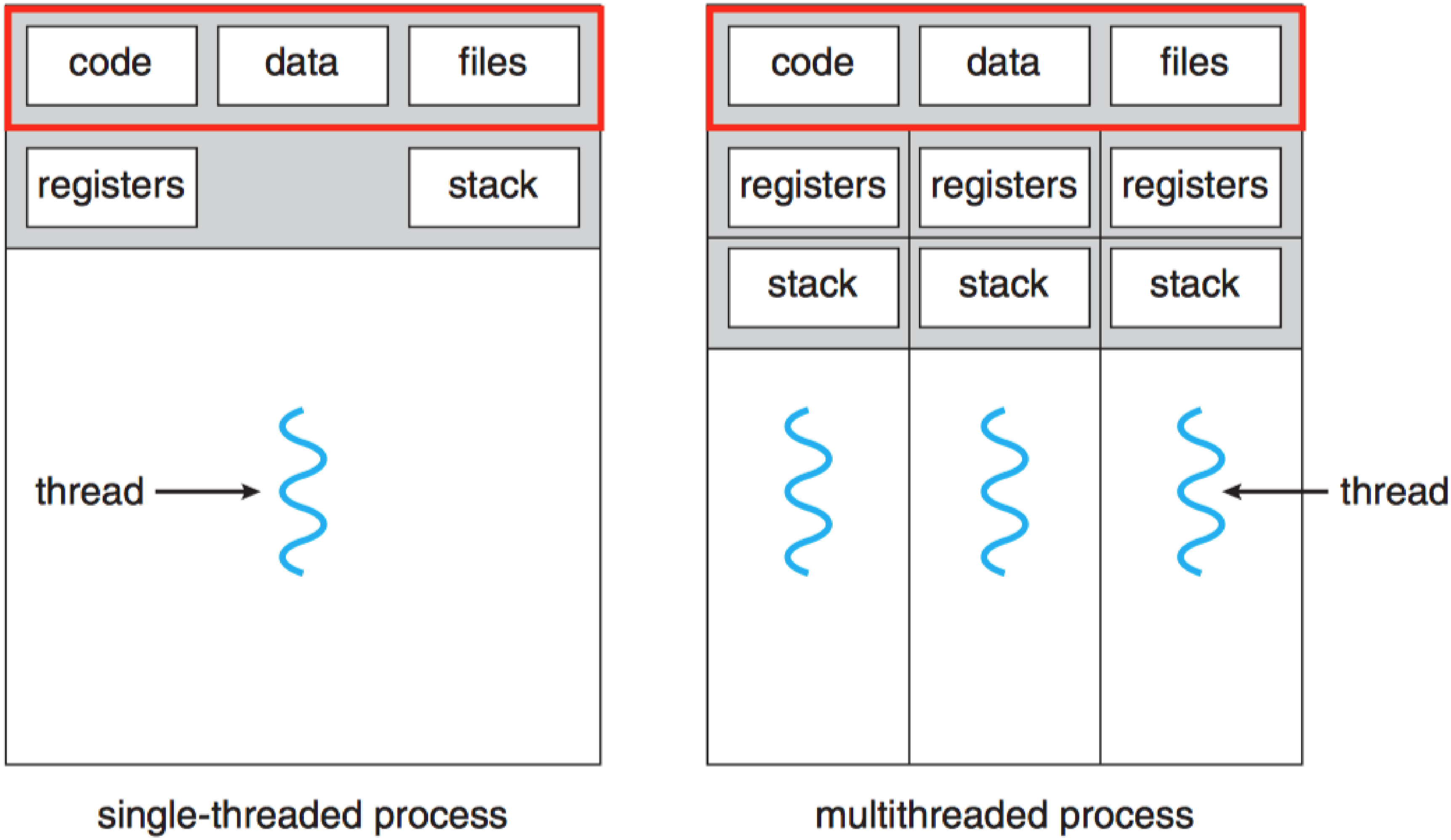
Assume each thread has **sum**, which is a local copy of the reduction variable. The threads then perform the following computations:

- Thread 1 : sum1 = a[0] + a[1] + a[2]
- Thread 2 : sum2 = a[3] + a[4] + a[5]
- Thread 3 : sum3 = a[6] + a[7] + a[8]
- sum = sum1 + sum2 + sum3

Outline

- **Worksharing Constructs**
  - ✓ Loop worksharing construct
    - o Ordered and schedule clauses
  - ✓ Reduction
- Data Sharing
  - ✓ Private
  - ✓ Firstprivate and lastptivate
  - ✓ Threadprivate
- Data Copying
- Other Constructs
  - ✓ Single worksharing construct
  - ✓ Sections worksharing construct
  - ✓ Master construct

(Lecture 1) Threads in a Process



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

Data Sharing: private Clause

Purpose:

- ✓ **private(var)** creates a new local copy of var for each thread.

Implementation:

```
int main(int argc, _TCHAR* argv[])
{
        int B;
#pragma omp parallel for private(B)
        for(int i=0; i<10;i++)
        {
            B = 100;
        }
        printf("%d\n",B);
        return 0;
}
```

May have problem here.

Notes:

- A new object of the same type is declared once for each thread in the team
- All references to the original object are replaced with references to the new object
- Should be assumed to be uninitialized for each thread

Data Sharing: shared Clause

Purpose:

- The **shared(list)** clause declares variables in its list to be shared among all threads in the team.

Implementation:

```
#pragma omp parallel for shared(n, a)
    for (int i = 0; i < n; i++) {
        int b = a + i;
        ...
}
```

Notes:

- **n** and **a** are shared variables.
- Shared variable exists in only one memory location and all threads can read or write to that address
- OpenMP does not put any restriction to prevent data races between shared variables. This is a **responsibility** of a programmer.

Data Sharing: default Clause

Purpose:

- The **default** clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region.

- **default(shared)** means that any variable in a parallel region will be treated as if it were specified with the **shared** clause.

- **default(none)** means that any variables used in a parallel region that aren't scoped with the **private**, **shared**, **reduction**, **firstprivate**, or **lastprivate** clause will cause a compiler error.

Notes:

- C/C++ only has default(shared) or default(none).

Data Sharing: default(shared)

Implementation:

```
int a, b, c, n;
...
#pragma omp parallel for default(shared)
      for (int i = 0; i < n; i++) {
      // using a, b, c
 }
```

- The **default(shared)** clause sets the data-sharing attributes of all the variables in the construct to "**shared**".
- **a**, **b**, **c** and **n** are shared variables.

Data Sharing: default(none)

Implementation:

```
int a = 0, b = 0;
#pragma omp parallel default(none) shared(b)
{
        b += a;
}
```

- The program above will cause compilation error.
- Because variable ‘a’ has not scope with any data-sharing clause.

Data Sharing: firstprivate Clause

- **firstprivate(list):** All variables in the list are initialised with the value the original object had before entering the parallel construct.

```
void test_firstprivate()
{
    int tmp = 0;
#pragma omp for firstprivate(tmp)
    for(int j = 0; j < 1000; ++j)
         tmp += j;
     printf("%d\n", tmp);
}
```

Each thread gets its own tmp with an initial value of 0

tmp: 0 in OpenMP 3.0, unspecified in OpenMP 2.5.

Exercise:
- change "firstprivate" to "private"
- print "tmp" within the for loop

Data Sharing: lastprivate Clause

- **lastprivate(list):** The thread that executes the sequentially last iteration or section updates the value of the objects in the list.

```
void test_firstprivate_lastprivate()
{
     int tmp = 0;
#pragma omp parallel for firstprivate(tmp) lastprivate(tmp)
     for (int j = 0; j < 1000; ++j)
          tmp += j;
     printf(“%d\n”, tmp);
}
```

Each thread gets its own tmp with an initial value of 0

tmp is defined as its value at the “last sequential” iteration (i.e., for j=999)

Data Sharing: threadprivate

Purpose:

- The **threadprivate** directive specifies that variables are replicated, with each thread having its own copy.
- Can be used to make global file scope variables and persistent to a thread through the execution of multiple parallel regions.

Notes:

- The directive must appear after the declaration of listed variables. Each thread then gets its own copy of the variable, so data written by one thread is not visible to other threads.
- On first entry to a parallel region, data in **threadprivate** variables should be assumed undefined, unless a **COPYIN** clause is specified in the PARALLEL directive
- **threadprivate** variables differ from private variables because:
    - with private global variables are masked.
    - **threadprivate** preserves global scope within each thread

Data Sharing: threadprivate Example

- Use **threadprivate** to create a counter for each thread:

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    Return counter;
}
```

Outline

- **Worksharing Constructs**
  - ✓ Loop worksharing construct
    - Ordered and schedule clauses
  - ✓ Reduction
- Data Sharing
  - ✓ Private
  - ✓ Firstprivate and lastptivate
  - ✓ Threadprivate
- Data Copying
- Other Constructs
  - ✓ Single worksharing construct
  - ✓ Sections worksharing construct
  - ✓ Master construct

Data Copying: copyin Clause

Purpose:

- copyin(list) allows to copy master thread's threadprivate variables to corresponding threadprivate variables of other threads.

```
int global[100];
#pragma omp threadprivate(global)
int main()
{
    for(int i= 0; i<100; i++)
         global[i] = i+2; //initialise data
#pragma omp parallel copyin(global)
      {
        /// parallel region, each thread gets a copy of
        /// global, with initialised value
      }
}
```

Data Copying: copyin Clause (Cont'd)

THE UNIVERSITY OF WESTERN AUSTRALIA

Notes:

- List contains the names of variables to copy.

- The **master thread** variable is used as the copy source. The team threads are initialised with its value upon entry into the parallel construct.

Data Copying: copyprivate Clause

- **Purpose:**
  - The **copyprivate** clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.

Usage:

- The typical usage is to have one thread read or initialise private data that is subsequently used by the other threads as well.

- **After the "single" construct has ended**, but before the threads have left the associated barrier, the values of variables specified in the associated list are copied to the other threads.

- Do not use **copyprivate** in combination with the **nowait** clause!

Data Copying: copyprivate Example

Implementation:

```
#include "omp.h"
void input_parameters(int, int); // fetch values of input parameters

int main()
{
    int Nsize, choice;
#pragma omp parallel private(Nsize, choice)
    {
            #pragma omp single copyprivate (Nsize, choice)
                    input_parameters(Nsize,choice);

            do_work(Nsize, choice);
     }
}
```

Outline

- **Worksharing Constructs**
  - ✓ Loop worksharing construct
    - Ordered and schedule clauses
  - ✓ Reduction
- Data Sharing
  - ✓ Private
  - ✓ Firstprivate and lastptivate
  - ✓ Threadprivate
- Data Copying
- Other Constructs
  - ✓ Single worksharing construct
  - ✓ Sections worksharing construct
  - ✓ Master construct

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();
}
```

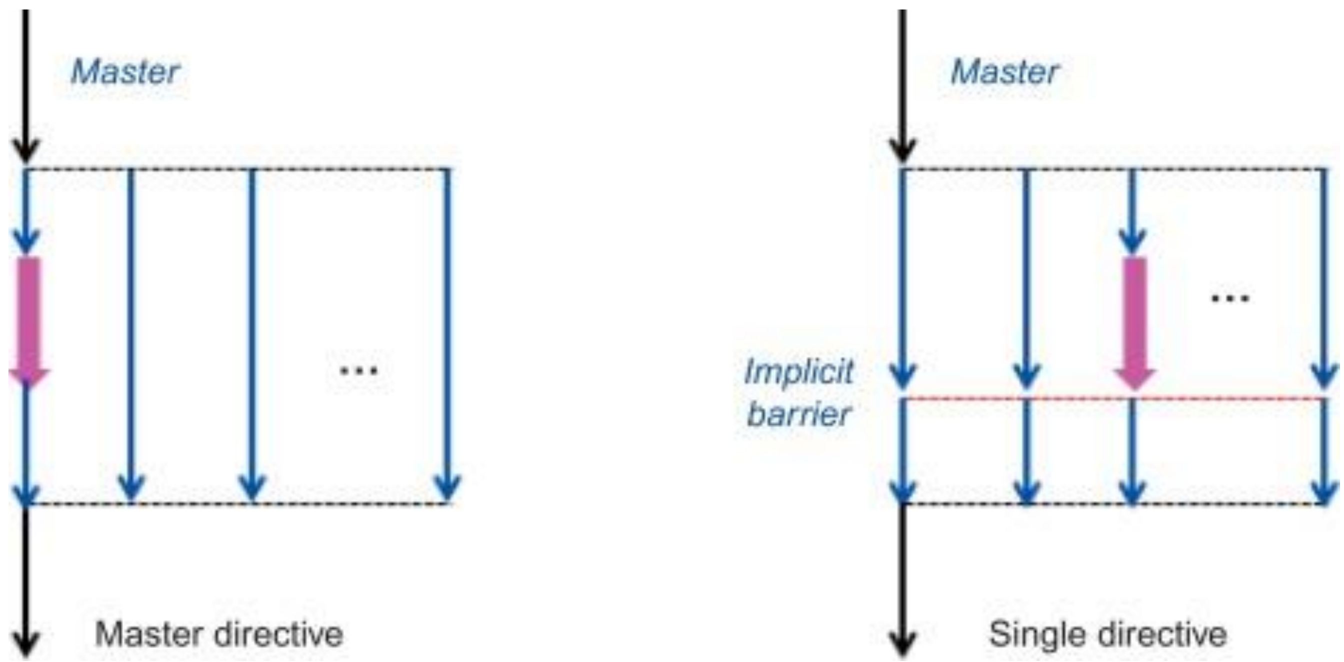
Single Worksharing Construct

- 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).

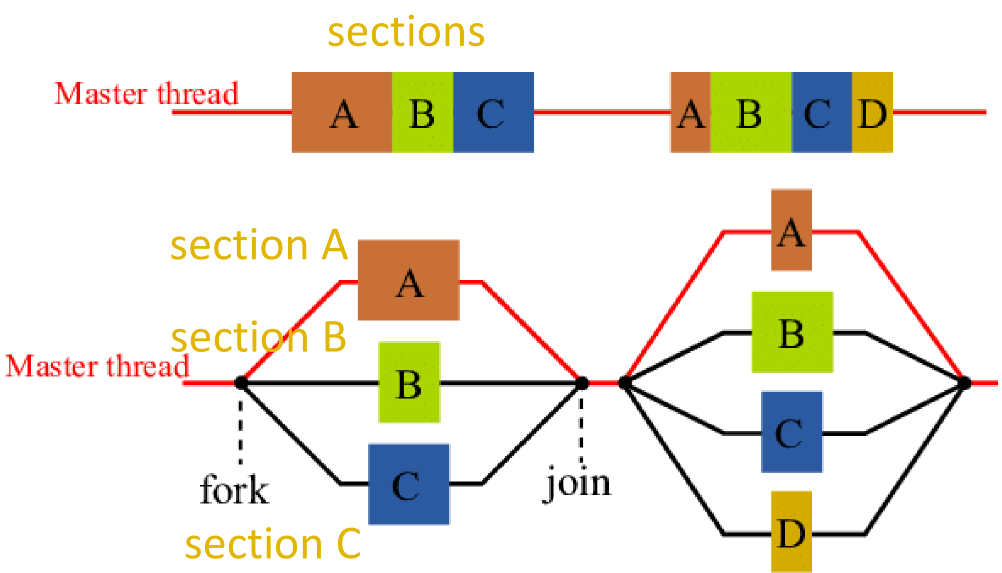
```
#pragma omp parallel
{
      do_many_things();

#pragma omp single
      {
        exchange_boundaries();
      }

      do_many_other_things();
}
```

Master vs. Single Constructs



Sections Worksharing Construct

- The **sections** construct is a non-iterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.
- Each structured block is executed once by one of the threads in the team in the context of its implicit task.
- There is **an implicit barrier** at the end of a sections construct unless a nowait clause is specified.



Sections Worksharing Construct (Cont'd)

THE UNIVERSITY OF WESTERN AUSTRALIA

- The syntax of the sections construct is as follows:

```
#pragma omp parallel
{
#pragma omp sections
    {
            #pragma omp section
                X_calculation();
            #pragma omp section
                Y_calculation();
            #pragma omp section
                Z_calculation();
    }
}
```

Barriers and Worksharing Constructs

- If the parallel region is worksharing construct, barriers will be automatically added after it;
- Worksharing construct includes **for**, **sections**, and **single**; pay special attention to that the **master** construct is not;
- **nowait** can remove barriers.

References

- Readings
  - Ordered Clause
  - OpenMP Data Sharing
  - Difference of Directive, Construct and Clause