COSC 407
Intro to Parallel Computing

Topic 8 - Work Sharing (Parallel For, Single)

COSC 407: Intro to Parallel Computing

1

Outline (Asynchronous)

Previously:

- Mutual Exclusion (critical, atomic, locks)
- Variable scope (shared, private, firstprivate)
- Reduction, Synchronization (barriers, nowait)

Today

- Work-sharing: parallel for construct
- Data dependency
- Single, master constructs

2

Work-Sharing Constructs

- Within a parallel region, you can decide how work is distributed among the threads.

  for - The for construct is probably the most important construct in OpenMP.

  single - Assigning a task to a single thread

  sections - Dividing the tasks into sections. Each section is executed by one thread.

- **Implied barrier on exit** (unless **nowait** is used). No implied barrier on entry….
- As they used in parallel region, use existing threads (do not create new threads)

Parallel For

- Loop iterations are distributed across the thread team (at run time)
  - Loops must have an integer counter variable whose value is inc/dec by a fixed value
  - Reached a specified bound
- Loop variable is explicitly declared to be a private variable (each thread gets own copy)
  - Value at end of loop is undefined (unless set as lastprivate)
  - Recommended that programmers do not rely on OpenMP default rules
    - Explicitly set data sharing attributes
- Output/processing order is non-deterministic

Parallel For Loop

- Uses a team of threads to execute **a for loop block following the *for* directive**.
  - The system parallelizes the for loop by **dividing the iterations of the loop among the threads**.

Syntax (two options)

(1) `#pragma omp for [clause [clause]…]` //does not create new threads

```
for(i = start; i <OP> end, incr_expression)
```

The above block must be **placed within a parallel region**.

Use this syntax if your parallel region **includes a for loop and other statements**.

Parallel For Loop

(2) `#pragma omp parallel for [clause [clause]…]` //creates new threads

```
for(i = start; i <OP> end, incr_expression)
```

The above is for creating a parallel block that **ONLY includes** a for loop

Variable scope: the loop variable of the for-statement immediately following the for-directive is *private*

- *Recall recommend best-practice – declare as private*

WITHOUT the Parallel For Directive

```
#pragma omp parallel num_threads(4)
{
  int i, n = omp_get_thread_num();

  for(i=0; i<4; i++)
    printf("T%d: i=%d\n", n , i);
}
```

- **WITHOUT** the for directive, **EACH** thread runs a copy of the **WHOLE** for loop
  - On 4 threads (T0 to T3) we get 16 print-outs (since each thread executes 4 iterations concurrently with the other threads)
  - Would need to divide up work manually

```
Possible Output:
T0: i=0
T1: i=0
T0: i=1
T1: i=1
T0: i=2
T1: i=2
T0: i=3
T1: i=3
T2: i=0
T3: i=0
T2: i=1
T3: i=1
T2: i=2
T3: i=2
T2: i=3
T3: i=3
```


WITH the Parallel For Directive

```
#pragma omp parallel
{
  int i, n;
  #pragma omp for
  for (i = 0; i < 4; i++) {
        n = omp_get_thread_num();
        printf("T%d: i=%d\n", n, i);
  }
}
```

```
Possible Output:
T0: i=0
T3: i=3
T1: i=1
T2: i=2
```

WITH the for directive, iterations are divided among the threads

- On 4 threads (T0 to T3) we get 4 print-outs (since each thread executes one iteration concurrently with the other threads)
  - That is, regardless of the number of threads we will get the exact number of print-outs specified by the for loop.

Note: order is not preserved!

Who Does what?

```
#pragma omp parallel
{
  int i, n;
  #pragma omp for
  for (i = 0; i < 8; i++) {
        n = omp_get_thread_num();
        printf("T%d: i=%d\n", n, i);
  }
}
```

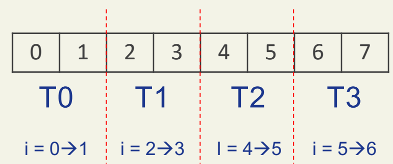
Possible Output:

```
T1: i=2
T0: i=0
T0: i=1
T2: i=4
T2: i=5
T1: i=3
T3: i=6
T3: i=7
```

Number iterations is divided equally among threads

This is called static scheduling

- T0 : 2 iterations (i= 0, 1)
- T1 : 2 iterations (i= 2, 3)
- T2 : 2 iterations (i= 4, 5)
- T3 : 2 iterations (i= 6, 7)



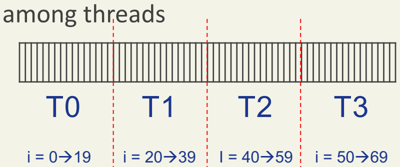
Each thread gets its own loop counter

Who Does What? *cont'd*

```
#pragma omp parallel
{
  int i, n;
  #pragma omp for
  for (i = 0; i < 80; i++) {
        //some work
  }
}
```

Number of iterations is divided equally among threads

- T0 : 20 iterations (i= 0…19)
- T1 : 20 iterations (i= 20…39)
- T2 : 20 iterations (i= 40…59)
- T3 : 20 iterations (i= 60…79)



Each thread gets its own loop counter

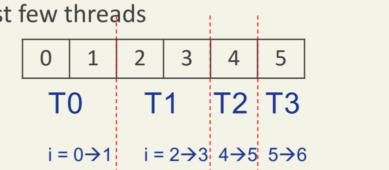
Who Does What? *cont'd*

```
//assuming 4 threads
#pragma omp parallel
{
  int i, n;
  #pragma omp for
  for (i = 0; i < 6; i++) {
        n = omp_get_thread_num();
        printf("T%d: i=%d\n", n, i);
  }
}
```

```
Possible Output:
T0: i=0
T1: i=2
T0: i=1
T1: i=3
T3: i=5
T2: i=4
```

Note: extra iterations are assigned to first few threads

- T0 took 2 iterations (i= 0, 1)
- T1 took 2 iterations (i= 2, 3)
- T2 took 1 iteration (i= 4)
- T3 took 1 iteration (i= 5)



Each thread gets its own loop counter

11

More Practical Examples

- **Image processing:** Converting an RGB image to Grayscale.
  - Assume the luminosity of each grayscale pixel is equal to 0.21 * Red + 0.72 * Green + .07 * Blue. Then:

```
#pragma omp parallel for
for(i=0; i < numPixels; i++)
        gray[i]= (unsigned char)(.21*red[i]+.72*green[i]+.07*blue[i]);
```

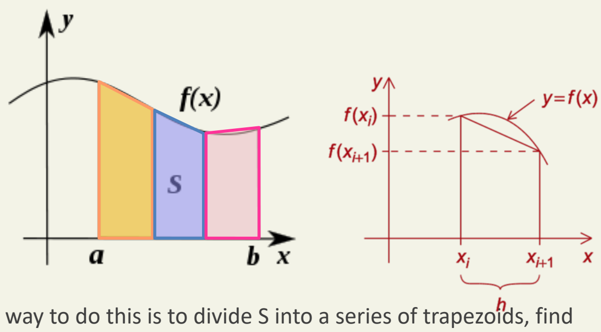
- **3D rendering**:
  - Divide image into blocks and iterate over them.
- **Matrix / array operations:**
  - Add two matrixes, find the transpose, etc.
- **Many other applications** that require applying the **same action** to iteratively in a loop
  - Refer to introduction to parallel computing notes

12

Area Under the Curve

Revisited

The aim is to compute the area S (from a to b) under a curve defined by f(x).



- One way to do this is to divide S into a series of trapezoids, find the area of these trapezoids, and then sum.
- The more trapezoids, the better approximation.

13

The Serial Algorithm

Revisited

$$h = \frac{b-a}{n}$$

$$Sum\ of\ trapezoid\ areas = h\left(\frac{f(a)+f(b)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1})\right)$$

$$x_i = a + i * h$$

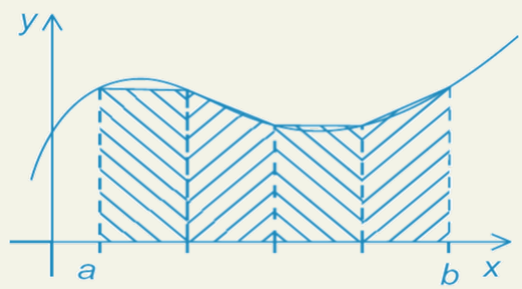
```
h = (b - a) / n;                    //calculate h only once
approx = (f(a) + f(b)) / 2.0;       //first and last terms
for (i = 1; i <= n-1; i++){         //remaining terms
        xi = a + i * h;
        approx += f(xi);
}
approx = h * approx;                //approximate area
```


14

Area Calculation

Given the serial code in previous slide, provide the parallel version

- In this version, use the **parallel for** directive
- **Note that when you use parallel-for, it's NOT necessary for n to be evenly divisible by thread_count**



The Parallel Algorithm

v.5

$$h = \frac{b-a}{n}$$

$$Sum\ of\ trapezoid\ areas = h\left(\frac{f(a)+f(b)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1})\right)$$

$$x_i = a + i * h$$

```
h = (b - a) / n;                    //calculate h only once
approx = (f(a) + f(b)) / 2.0;       //first and last terms
#pragma omp parallel for reduction(+: approx)
for (i = 1; i <= n-1; i++){         //remaining terms
       xi = a + i * h;
       approx += f(xi);
}
approx = h * approx;                //approximate area
```

The Complete Program

```
main() {
   double  a = 1, b = 2;
   int n = 10, thread_count = 4;
   double global_result = Trap(a, b, n, thread_count);
   printf("Approximate area: %f\n", global_result); return 0;
}

double Trap(double a, double b, int n, int thread_count) {
   double h = (b-a)/n;
   double approx = (f(a) + f(b))/2.0;
   int  i;
#pragma omp parallel for num_threads(thread_count) reduction(+: approx)
   for (i = 1; i <= n-1; i++)
     approx += f(a + i*h);
   return h * approx;
}
```


Previous Version without Parallel For

```
main() {
   double global_result = 0, a=1, b=2;    //global_result is shared
   int n = 12;
   # pragma omp parallel num_threads(4) reduction(+:global_result)
        global_result += Local_trap(...);
   printf("Approximate area: %f\n", global_result);
}
double Local_trap(double a, double b, int n) {
   double x, my_approx, my_a, my_b;
   int i, my_n, my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   my_n = n / thread_count;
   my_a = a + my_rank * my_n * h;
   my_b = my_a + my_n * h;
   double h = (b - a) / n;
   my_approx = (f(my_a) + f(my_b)) / 2.0;
   for (i = 1; i <= my_n - 1; i++)
        my_approx += f(my_a + i * h);
   return h * my_approx;  //instead of adding it to global_result
}
```

- we need to define range for each thread
- n must be divisible by num_threads

Nested Loops

```
int i, j;
#pragma omp for
for(i = 0; i<5; i++)     //this loop is a parallelized
    for(j=0; j<10; j++)  //but this one is not
            …
```

Nested for loops: (In the above example)

- **outer for loop** is parallelized
  - Iterations are divided among threads
- **inner for loop** is not parallelized
  - Each thread will execute all iterations

Variable scope:

- The loop counter of the for-statement **immediately following** the for-directive is private for each thread.
  - Hence **(i) in the above** code is private
  - However, **(j) is not private** unless it is declared right after the outer for loop, or it is explicitly defined private

Caution: How and When to Use Parallel For?

Only the following form is legal for parallelized for statement

```
                                              index++
                                              ++index
                             index < end      index--
                             index <= end     --index
for ( index = start   ;     index >= end  ;  index += incr )
                             index > end      index -= incr
                                              index = index + incr
                                              index = incr + index
                                              index = index - incr
```

<u>**Limitations:**</u>

`index`

- Must have **integer** or **pointer type** (e.g., it can't be float)
- Can only be modified by the "increment expression" <u>**in**</u> the for statement

`start`, **`end`**, and **`incr`** must:

- Have a compatible type (e.g., if index is a pointer, then incr must be **int**)
- Not change during execution of the loop

Caution, cont'd

Cannot parallelize:

- `while` or `do-while` loops
- `for (while ; ; ) {…}`   //infinite loop
- `for (i=0; i<n; i++){`   `//cannot determine the`
  `//number of iterations`

```
    if (…) break
  }
```

Watch for loop-carried dependencies! *(discussed shortly)*

- Happen when the calculations in one iteration depends on the data written by other iterations. More details are discussed shortly.
- If they exist, you must do one of two things:
  - Re-write your algorithm to avoid loop-carried dependencies, OR
  - Do not use the parallel-for loops, OR
  - Order your iterations (poor performance)

21

Data Dependency

Sequential

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

Parallel

```
fibo[ 0 ] = fibo[ 1 ] = 1;
#pragma omp parallel for num_threads(8)
for (i = 2; i < n; i++)
      fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

- Output from the parallel code is sometimes correct

  1 1 2 3 5 8 13 21 34 55

  and sometimes it is wrong

  1 1 2 3 5 8 1861489712 -576187344 …

Q: Explain why!

22

Interdependency Among Iterations

- OpenMP **compilers don't check for dependences among iterations** in a loop that's being parallelized with `parallel for`.
- A loop in which the **results of one or more iterations depend on other iterations** cannot, *in general*, be correctly parallelized by OpenMP.

Rule: *In most cases*, don't use parallel for loops in which the computation of iterations depends on the data written by <u>other</u> iterations.

i.e. when one thread reads data written by another thread.

How to address this limitation (again)?

- redesign your algorithm!
- Use 'ordered' clause – has efficiency issues
- Don't use parallel for

23

Summary of Working With Loops

Basic approach:

- Identify computational intensive loops
- Use one of two techniques:
- **Technique 1**: using **`parallel for`** or **`for`** directive
  - Make sure the loops are **<u>iterations-independent</u>** (i.e., don't have loop-carried dependencies)
    - If not, rewrite the algorithm to avoid loop-carried dependencies.
  - Remember that the loop counter of the "for" statement immediately following the for directive is private by default.

24

Summary of Working With Loops

- **Technique 2**: using **`parallel`** directive and **manually divide** the range
  - Manually divide the range of iterations and use the thread id (rank) to determine the start, end, and number of iterations for each thread
  - Limitation: number of iterations **must be divisible** by number of threads
- Place appropriate OpenMP directives and test your code
  - Don’t forget to protect shared data
  - Use reduction clauses if they simplify the process

Assigning Work to a Single Thread

 # pragma omp single

Within a parallel region, you can execute a block of code just once by **any** thread in the team.

e.g., when reading in shared input variable or writing single output

There is an implicit barrier at **<u>the end</u>** of the “single” region

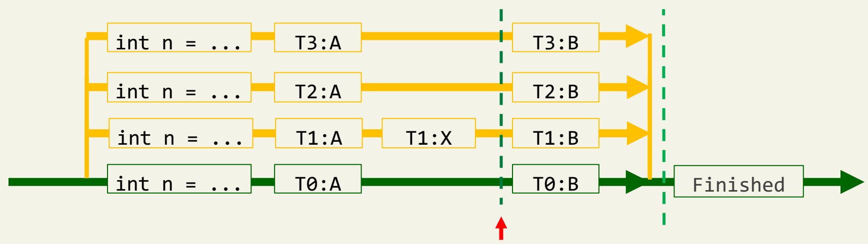
```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
   #pragma omp parallel
   {
         printf("Hi from T%d\n", omp_get_thread_num());
         #pragma omp single
          printf("One Hi from T%d\n", omp_get_thread_num());
   }
   return 0;
}
```

<u>Possible Output</u>

```
Hi from T2
Hi from T3
One Hi from T2
Hi from T0
Hi from T1
```

Assigning Work to a Single Thread, cont'd

```
#pragma omp parallel
{     int n = omp_get_thread_num();
      printf("T%d:A\n",n);
      #pragma omp single
       printf("T%d:X\n",n);
      
      printf("T%d:B\n",n);
}
printf("Finished");
```



Barrier

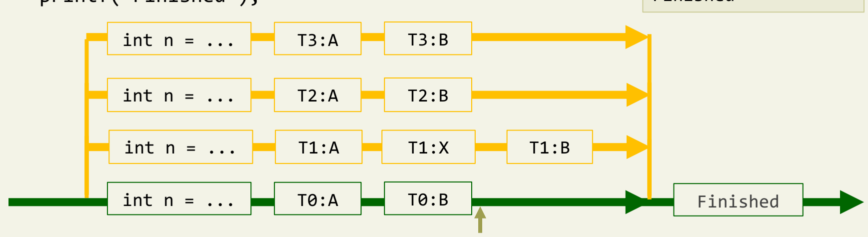
(Synchronization ***after*** 'single')

Assigning Work to a Single Thread, cont'd

```
#pragma omp parallel
  {
        int n = omp_get_thread_num();
        printf("T%d:A\n",n);
        #pragma omp single nowait
         printf("T%d:X\n",n);
        printf("T%d:B\n",n);
  }
  printf("Finished");
```

Possible Output

```
T1:A
T0:A
T1:X
T0:B
T1:B
T3:A
T3:B
T2:A
T2:B
Finished
```



NO Barrier

(no synchronization ***after*** 'single' if nowait is used)

Assigning Work to the Master Thread

#pragma omp master

- Within a parallel region, executes block of code just once by **the master** thread in the team.
- Unlike "single directive", master directly ***does NOT have an implied barrier on exit***.

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

int main() {
   #pragma omp parallel
   {    printf("Hi from T%d\n", omp_get_thread_num());
        #pragma omp master
         printf("One Hi from T%d\n", omp_get_thread_num());
   }
   return 0;
}
```

Possible Output

```
Hi from T2
Hi from T3
Hi from T0
One Hi from T0
Hi from T1
```

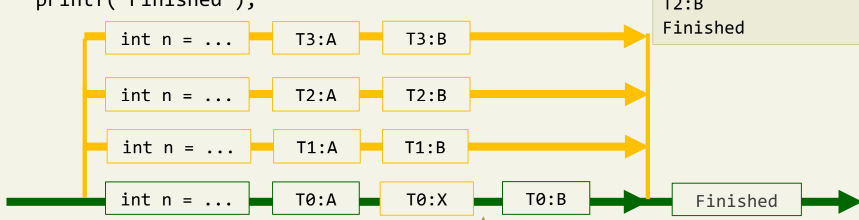

29

Assigning Work to Master Thread, cont'd

```
#pragma omp parallel
  {
        int n = omp_get_thread_num();
        printf("T%d:A\n",n);
        #pragma omp master
         printf("T%d:X\n",n);
        printf("T%d:B\n",n);
  }
  printf("Finished");
```

Possible Output

```
T1:A
T1:B
T3:A
T3:B
T0:A
T0:X
T0:B
T2:A
T2:B
Finished
```



NO Barrier

(no synchronization ***after*** 'master' even without nowait)

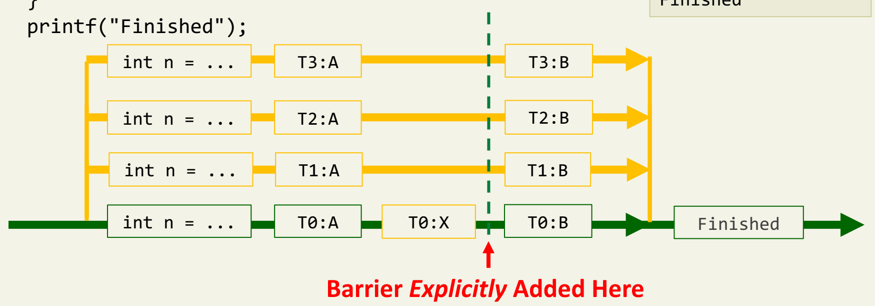
30

Assigning Work to Master Thread, cont'd

```
#pragma omp parallel
  {
        int n = omp_get_thread_num();
        printf("T%d:A\n",n);
        #pragma omp master
           printf("T%d:X\n",n);
        #pragma omp barrier
        printf("T%d:B\n",n);
  }
  printf("Finished");
```

Possible Output

```
T3:A
T2:A
T0:A
T0:X
T1:A
T3:B
T2:B
T0:B
T1:B
Finished
```



Conclusion/Up Next

- What we covered today (review key concepts):
  - Work-sharing: parallel for construct
  - Data dependency
  - Single, master constructs
- Next:
  - Sections
  - Scheduling Loops (static, dynamic, guided, auto)
  - Ordered Iterations
  - Nested Loops