# COSC 407
# Intro to Parallel Computing

Topic 7 - Variable Scope, Reduction



## Outline

*Previously:*

- Synchronization (barriers, nowait)
- Mutual Exclusion (critical, atomic, locks)

***Today***

- **Variable scope (shared, private, firstprivate)**
- **Reduction**

# Variable Scope

- In **serial programming**, the scope of a variable consists of those parts of a program in which the variable can be used
- **In OpenMP**, the scope of a variable refers to the **set of threads that can access the variable in a parallel block**
- A **shared variable** exists in only **one memory** location and all threads in the team access this location
  - All variables **declared *BEFORE* a parallel block** are shared by default
  - `shared(x)`
    - x will refer to the same memory block for all threads



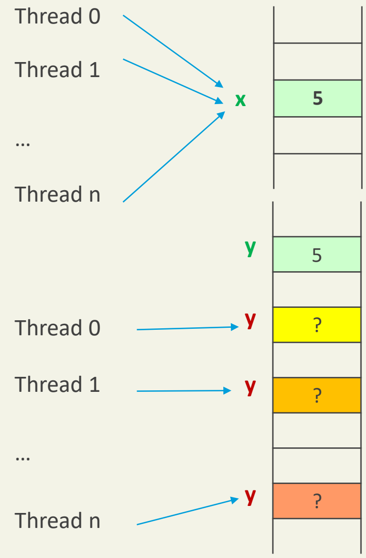
# Variable Scope

- A **private variable** can only be accessed by a single thread (**each thread has its own copy**).
  - variables **declared *WITHIN* a parallel block** are private by default
  - `private(y)`
    - y will refer to a different memory block for each thread. Each copy of y *is uninitialized*.
  - `firstprivate(z)` same as private, but each copy of z *is initialized* with the value that the original z has when the construct is encountered

# Variable Scope, *cont'd*

int x = 5;
#pragma omp parallel **shared** (x)

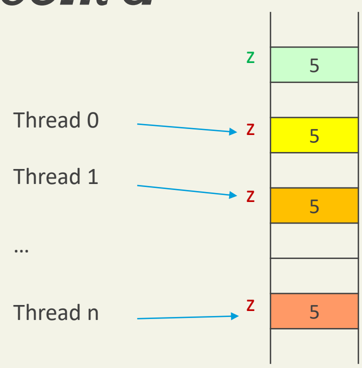
int y = 5;
#pragma omp parallel **private(y)**
//each thread creates a new copy of y, and these y's are ***uninitialized***






# Variable Scope, *cont'd*

int z = 5;
#pragma omp parallel firstprivate(**z**)
//each thread creates a new copy of z, and these z's are initialized with value of original z




- In some cases, the variables scope is predetermined and cannot be changed:
  – Variables declared inside the parallel region are private
  – A loop variable in a parallel loop is private
  – const variables are shared

# Variable Scope: Example

```
#include <omp.h>
#include <stdio.h>
int main() {
 int j=0, i;                              //i, j is shared by default
 #pragma omp parallel private(i)          //i is private in this block
 {
   printf("Started T%d\n", omp_get_thread_num());
   for (i = 0; i < 10000; i++)
      j++;
   printf("Finished T%d\n", omp_get_thread_num());
 }
 printf("%d\n", j);
 return 0;
}
```

Possible outputs with 3 threads

```
Started T0
Finished T0
Started T1
Finished T1
Started T2
Finished T2
30000
```

```
Started T0
Started T1
Finished T1
Started T2
Finished T2
Finished T0
24624
```

```
Started T2
Started T0
Finished T0
Finished T2
Started T1
Finished T1
19616
```

race condition (WHY?)



# default clause

- Sets the default scope.

  **default(shared | private | none)**

  **default(none)** forces the programmer to specify the scope of each variable in a block – i.e. the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

We don't have to mention z in #pragma as it is not used in the parallel region

```
int x = 0, y = 0, z = 0;
#pragma omp parallel num_threads(4) default(none) private(x) shared(y)
{
   x = omp_get_thread_num();
   #pragma omp atomic
   y = y + x;
}
printf("x:%d y:%d z:%d", x, y, z);
```

Try removing shared(y) or private(x) and notice what happens.

Output:
x:0 y:6 z:0

# Back to Sums…..

- Assume you want to find the sum of a function values in a range

```
double global_sum = 0;          // 1) create a shared global_sum
# pragma omp parallel num_threads(4)
{     int my_id = ..;
      int my_sum = f(my_id);  // 2) create a private my_sum and compute it
      #pragma omp critical
      global_sum += my_sum;    // 3) update global sum in critical section
 }
```

- Is there a better solution?



# Reduction

- The reduction clause of parallel does the following:
  1. Provides a private copy of a the variable for each thread
     - The variable is scoped 'reduction' (private first then shared on exit)
       - i.e., **no need for critical clause**
  2. Combines the private results on exit

# Reduction

**Syntax**:

```
reduction (<op> : <variable list>)
```

- A reduction operator is a binary operation: +, *, -, &, |, ^, &&, ||
  - **No division** ( / )
  - op could also be **max** or **min**, (more later on this)
  - Assuming we have N threads and a variable x, on exit x's value is:

  $x = init\_value <op> x_0 <op> x_1 <op>... <op> x_{N-1}$

  Where init_value is x's value, and $x_0$, $x_1$, $x_2$, etc are its private copies.
- **Initial values** for the temporary private variables:

  **1** for (*, &&)   **0** for (+, -, |, ^, ||)   **~0** for (&)



# Reduction: Example 1

```
int main() {
   int x = 10;        //shared x
   #pragma omp parallel reduction(+:x)
   {
       x = omp_get_thread_num(); //private x
       printf("Private x = %d\n", x);
   } //on exit: shared x += all private x's

   printf("Shared x = %d\n", x);
   return 0;
}
```

**Possible output:**

```
Private x = 3
Private x = 1
Private x = 0
Private x = 2
Shared x = 16
```

# Reduction: Example 2

```
int main() {
   int x = 10, y = 10;       //shared x, y
   #pragma omp parallel reduction(+:x, y)
   {
       x = omp_get_thread_num();//private x,y
       y = 5;
       printf("Private: x=%d, y=%d\n", x, y);
   } //on exit: shared x += all private x's
     //   shared y += all private y's
   printf("Shared: x=%d,y=%d\n", x, y);
   return 0;
}
```

**Possible output:**

```
Private: x=1, y=5
Private: x=2, y=5
Private: x=0, y=5
Private: x=3, y=5
Shared: x=16,y=30
```



# Caution

- Reduction using (-) is same as (+), so if you want to subtract
  - i.e. both `reduction(-:x)` and `reduction(+:x)` have the same effect; i.e. shared_x = init_value + $x_0$ + $x_1$ + …
- To do reduction on subtraction, use this code:

```
x = init_value();
#pragma omp parallel reduction(-:x)
x -= f(…);
```

  - Why this works?
    - Initial x is 0, so value of private x is `-f()`
    - This means, final x is:

      `x = init_value() - f`$_0$`(…) - f`$_1$`(…) - etc`

      where `f`$_n$`()` is the function value computed by thread id n

# Back to the Sums…

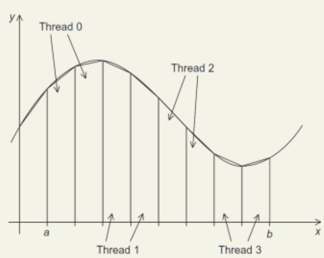
```
double global_sum = 0;                    // 1) create shared global_sum
# pragma omp parallel num_threads(4)
{      int my_id = ..;
       int my_sum = f(my_id);             // 2) create a private my_sum
       #pragma omp critical
       global_sum += my_sum;              // 3) update global sum in critical  section
 }
```

```
double global_sum = 0;               //global_sum is shared
# pragma omp parallel num_threads(4) reduction(+:global_sum)
{      int my_id = ..;
       global_sum = f(my_id); // each thread gets a private copy of global_sum
} //reduction is applied on exit
```



# Area Under a Curve

```
int main() {
   double global_result = 0.0;        // result stored in global_result
   double a = 1, b = 2;               // endpoints
   int thread_count = 4;              // should be = number of cores
   int n = 8;                         // Total number of trapezoids
                                      //   = multiple of thread_count
   # pragma omp parallel num_threads(thread_count)
       Trap(a, b, n, &global_result);
   printf("Approximate area: %f\n", global_result); return 0;
}
```



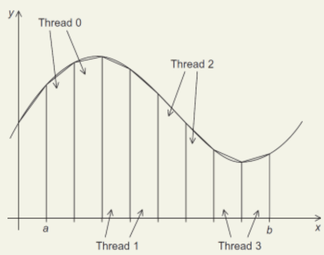
# Area Under a Curve

```
void Trap(double a, double b, int n, double* global_result_p) {
   double h, x, my_result, my_a, my_b; int i, my_n;
   int my_id = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b - a) / n;
   my_n = n / thread_count; // # of contiguous trapezoids per thread
   my_a = a + (my_id * my_n) * h;
   my_b = my_a + my_n * h;

   my_result = (f(my_a) + f(my_b)) / 2.0;
   for (i = 1; i <= my_n - 1; i++) {
         x = my_a + i * h;
         my_result += f(x);
   }
   my_result = my_result * h;

   # pragma omp critical
   *global_result_p += my_result;
}
```






# Area Calculation – v.2

## *small update without using reduction clause*

- For the Trap function, instead of:

**`void Trap(double a, double b, int n, double* global_result_p)`**

we would prefer the more attractive version pf

**`double Local_trap(double a, double b, int n)`**

which...

– is run by each thread to return a part of the calculations

– has no critical section but.....

```
double global_result = 0.0;  //global_result is shared
# pragma omp parallel num_threads(thread_count)
{
      # pragma omp critical
       global_result += Local_trap(a, b, n);
}
```

Why?

# Area Calculation – v.2

*(small update without using reduction clause)*

```
main() {
   double global_result = 0.0, a = 1, b = 2;
   int n = 12;
   # pragma omp parallel num_threads(4)
   {
         # pragma omp critical
         global_result += Local_trap(a, b, n); //global_result is shared
   }
   printf("Approximate area: %f\n", global_result); return 0;
}

double Local_trap(double a, double b, int n) {
   double h, x, my_result, local_a, local_b;
   int i, local_n, my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   h = (b - a) / n;
   local_n = n / thread_count;
   local_a = a + my_rank * local_n * h;
   local_b = local_a + local_n * h;
   my_result = (f(local_a) + f(local_b)) / 2.0;
   for (i = 1; i <= local_n - 1; i++) {
         x = local_a + i * h;
         my_result += f(x);
   }
   return h*my_result;  //instead of adding it to global_result
}
```

**Warning:** sequential execution!
**Q1) Explain! Q2) How to fix?**



# Area Calculation – v.3

*(still not using reduction() clause)*

- We can avoid the problem (of sequentially running the program) by declaring a private variable inside the parallel block and moving the critical section after the function call.

```
double global_result = 0.0;          //global_result is shared
# pragma omp parallel num_threads(thread_count)
{
    double my_result = Local_trap(a,b,n); //my_result is private
    # pragma omp critical
    global_result += my_result;
}
```

## Area Calculation – v.3

*(still not using `reduction()` clause)*

```
main() {
   double global_result = 0.0, a=1, b=2;    //global_result is shared
   int n = 12, thread_count = 4;
   # pragma omp parallel num_threads(thread_count)
   {
         double my_result = Local_trap(a, b, n); //my_result is private
         # pragma omp critical
            global_result += my_result;
    }
    printf("Approximate area: %f\n", global_result); return 0;
}

double Local_trap(double a, double b, int n) {
   double h, x, my_result, local_a, local_b;
   int i, local_n, my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   h = (b - a) / n;
   local_n = n / thread_count;
   local_a = a + my_rank * local_n * h;
   local_b = local_a + local_n * h;
   my_result = (f(local_a) + f(local_b)) / 2.0;
   for (i = 1; i <= local_n - 1; i++) {
         x = local_a + i * h;
         my_result += f(x);
   }
   return h * my_result;  //instead of adding it to global_result
}
```



## Area Calculation – v.4

(using *`reduction`*)

- Instead of using the private `my_result` and the shared `global_result`, the code can use reduction as following:

```
double global_result = 0;  //global_result is shared

# pragma omp parallel num_threads(4) reduction(+:global_result)
global_result += Local_trap(a, b, n); //the + is redundant
      // first, global_result will be private to each thread,
      // on exit, all private results are added(+) into it
```

# Area Calculation – v.4

(using *reduction*)

```
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(...); // or simply =
   printf("Approximate area: %f\n", global_result);
   return 0;
}

double Local_trap(double a, double b, int n) {
   double h, x, my_result, local_a, local_b;
   int i, local_n, my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   h = (b - a) / n;
   local_n = n / thread_count;
   local_a = a + my_rank * local_n * h;
   local_b = local_a + local_n * h;
   my_result = (f(local_a) + f(local_b)) / 2.0;
   for (i = 1; i <= local_n - 1; i++) {
         x = local_a + i * h;
         my_result += f(x);
   }
   return h * my_result;  //instead of adding it to global_result
}
```



# Remember!

1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

| Core | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| my_sum | 8 | 19 | 7 | 15 | 7 | 13 | 12 | 14 |

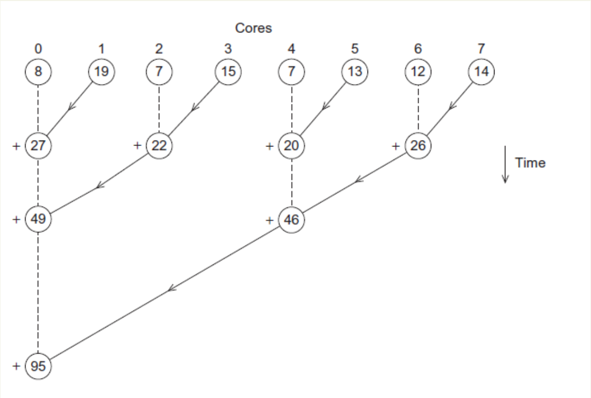
| Core | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| my_sum | 95 | 19 | 7 | 15 | 7 | 13 | 12 | 14 |

Global sum

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

# Multiple Cores Forming a Global Sum

- The reduction operator optimises the aggregation of results






# Conclusion/Up Next

- What we covered today (review key concepts):
  – Variable scope (shared, private, firstprivate)
  – Reduction
- Next:
  – Work Sharing