# Synchronization Constructs

Bringing order to threads

So far, we've had little to no control of what order threads executed their tasks

- Barriers
- Critical Sections
- Atomic Sections
- Locks

## Synchronization, barriers

Defines a point where all active threads will stop and wait until all threads have arrived.

Look at the following code snippet:

```
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    x[mytid] = some_calculation();
    y[mytid] = x[mytid]+x[mytid+1];
}
```

## Synchronization, barriers

Defines a point where all active threads will stop and wait until all threads have arrived.

Look at the following code snippet:

```
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    x[mytid] = some_calculation();
#pragma omp barrier
    y[mytid] = x[mytid]+x[mytid+1];
}
```

# Synchronization, barriers

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

int main( )
{
    int a[5], i;

    #pragma omp parallel
    {
        // Perform some computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;
```

```

        // Print intermediate results.
         #pragma omp master
             for (i = 0; i < 5; i++)
                 printf_s("a[%d] = %d\n", i, a[i]);

         // Wait.
         #pragma omp barrier

         // Continue with the computation.
         #pragma omp for
         for (i = 1; i < 5; i++)
             a[i] += a[i-1];
    }
}
```

## Synchronization, implicit barriers

Worksharing constructs define an implicit barrier

```
#pragma omp parallel
{
    #pragma omp for
    for (int mytid=0; mytid<number_of_threads; mytid++)
        x[mytid] = some_calculation();

    #pragma omp for
    for (int mytid=0; mytid<number_of_threads-1; mytid++)
        y[mytid] = x[mytid]+x[mytid+1];
}
```

## Synchronization, critical section, critical pragma

Used when a specific region of code needs to be executed one thread at a time. (Not to be confused with `single` which allows you specify that a section of code should be executed on a single thread.)

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
    #pragma omp single
    a++;
    #pragma omp critical;
    b++;
}
printf("single: %d -- critical: %d\n", a, b);
```

## Synchronization, critical section, critical pragma

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

int main()
{
    int i;
    int max;
    int a[10];

    for (i = 0; i < 10; i++)
    {
        a[i] = rand();
        printf_s("%d\n", a[i]);
    }
```

```c
    max = a[0];
     #pragma omp parallel for num_threads(4)
         for (i = 1; i < 10; i++)
         {
             if (a[i] > max)
             {
                 #pragma omp critical
                 {
                     // compare a[i] and max again because max
                     // could have been changed by another thread after
                     // the comparison outside the critical section
                     if (a[i] > max)
                         max = a[i];
                 }
             }
         }

     printf_s("max = %d\n", max);
}
```

## Synchronization, critical section, critical pragma

- Easiest way to parallelize existing code
- But, not without some drawbacks
  - critical sections work by acquiring locks = overhead
  - sections are mutually exclusive

## Synchronization, critical section, atomic pragma

An atomic operation has much lower overhead. It relies on the hardware providing (say) an atomic increment operation; in that case there's no lock/unlock needed on entering/exiting the line of code, it just does the atomic increment which the hardware tells you can't be interfered with.

```
#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_s("Number of threads: %d\n", count);
}
```

## Synchronization, critical sections, atomic pragma

In a nutshell:

- **The region contains value whose memory location you want to protect against multiple writes.**

## Synchronization, critical section, critical pragma

```
int atomic_read(const int *x)
{
  int value;
  /* Ensure that the entire value of *x is read atomically. */
  /* No part of *x can change during the read operation. */
#pragma omp atomic read
  value = *x;
  return value;
}


void atomic_write(int *x, int value)
{
  /* Ensure that value is stored atomically into *x.   */
  /* No part of *x can change until after the entire write operation has completed. */
  #pragma omp atomic write
  *x = value;
}
```

## Synchronization, critical sections

Blocking is often the biggest factor keeping threaded applications from performing their best.

You can get a rough sense of how much blocking will slow your code: Look at the ratio of critical sections to the rest of your code. Large critical sections or high ratios of critical sections to the rest of your code are both signs of trouble.

**Bottomline: Critical sections have their place, but use them with care.**

## Reductions, a mode detailed look

- used when the threads in a team compute partial results which need to be combined.
- can be implemented having threads compute *partial* results or doing an `atomic` update, but much easier with a `reduction` clause
- Most common reduction operators:
  - Arithmetic reductions: +, ∗, −, max, min
  - Logical operator reductions: &, &&, |, ||, ˆ
  - Fortran additionally has max and min.

## Reductions, a mode detailed look

- Most commonly added to a `for loop`; the results of all iterations are then combined.
- But, can also be added to a `section` construct; the results of the individual section blocks are then combined.
- Also, can be added directly to the parallel directive; thus combining the results from the threads in the team that is created.

## Synchronization, critical section, critical pragma

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

#define NUM_THREADS 4
#define SUM_START   1
#define SUM_END     10
#define FUNC_RETS   {1, 1, 1, 1, 1}

int bRets[5] = FUNC_RETS;
int nSumCalc = ((SUM_START + SUM_END) * (SUM_END -
SUM_START + 1)) / 2;

int func1() {return bRets[0];}
int func2() {return bRets[1];}
int func3() {return bRets[2];}
int func4() {return bRets[3];}
int func5() {return bRets[4];}
```

```
int main()
{
    int nRet = 0, nCount = 0, nSum = 0, i,
    bSucceed = 1;

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel reduction(+ : nCount)
    {
        nCount += 1;

        #pragma omp for reduction(+ : nSum)
        for (i = SUM_START ; i <= SUM_END ; ++i)
            nSum += i;

        #pragma omp sections reduction(&& : bSucceed)
        {
            #pragma omp section
            { bSucceed = bSucceed && func1();}
```

# Synchronization, critical section, critical pragma

```
        #pragma omp section
         { bSucceed = bSucceed && func2();}

         #pragma omp section
         { bSucceed = bSucceed && func3();}

         #pragma omp section
         { bSucceed = bSucceed && func4();}

         #pragma omp section
         { bSucceed = bSucceed && func5( );}
     }
 }

 printf_s("The parallel section was executed %d times "
          "in parallel.\n", nCount);
 printf_s("The sum of the consecutive integers from "
          "%d to %d, is %d\n", 1, 10, nSum);
```

```
 if (bSucceed)
      printf("All of the the functions, func1 through "
               "func5 succeeded!\n");
 else
      printf("One or more of the the functions, func1 "
               "through func5 failed!\n");

 if (nCount != NUM_THREADS)
 {
      printf("ERROR: For %d threads, %d were counted!\n",
               NUM_THREADS, nCount);
      nRet |= 0x1;
 }

  if (nSum != nSumCalc)
  {
      printf("ERROR: The sum of %d through %d should be"
               "%d, but %d was reported!\n",
              SUM_START, SUM_END, nSumCalc, nSum);
      nRet |= 0x10;
  }
```

## Synchronization, critical section, critical pragma

```
    if (bSucceed != (bRets[0] && bRets[1] &&
                      bRets[2] && bRets[3] && bRets[4]))
     {
         printf("ERROR: The sum of %d through %d should be"
                  "%d, but %d was reported!\n",
                  SUM_START, SUM_END, nSumCalc, nSum);
         nRet |= 0x100;
     }
}
```

## Exercise 5, In Class Lab, Reduction

1. Build an array using rand()
2. using reduction,
   a. find the sum of the array elements
      i. find the sum of the even elements
      ii. find the sum of the odd
   b. find the product of the array elements
      i. find the product of the even elements
      ii. find the product of the off elements
   c. find the max and the min