OpenMP Contd.

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

prime_count_omp.cpp

- Open the code!

```cpp
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <omp.h>

using namespace std;

//****************************************************************************
int prime_count ( int n )
//****************************************************************************
{
    int i, j, prime, total_prime=0;

    #pragma omp parallel for reduction(+:total_prime) \
                             schedule(static) private(i, j, prime)
    for ( i = 2; i <= n; i++ ) {
        prime = 1;
        for ( j = 2; j < i; j++ ) {
            if ( i % j == 0 ) {
                prime = 0;
                break;
            }
        }
        total_prime += prime;
    }
    return total_prime;
}

```

```cpp
//****************************************************************************80
int main (int argc, char *argv[])
//****************************************************************************80
{
    int n, num_procs, num_threads, total_prime=0;
    double start, end;

    // Returns the number of processors that are available to the program
    num_procs = omp_get_num_procs();

    // Returns the maximum value that can be returned by a call to the
    // OMP_GET_NUM_THREADS function
    num_threads = omp_get_max_threads();

    cout << "\n";
    cout << "SCHEDULE_OPENMP\n";
    cout << "  C++/OpenMP version\n";
    cout << "  Count the primes from 1 to N.\n";
    cout << "  This is an unbalanced work load, particular for two threads.\n";
    cout << "  Demonstrate static and dynamic scheduling.\n";
    cout << "\n";
    cout << "  Number of processors available = " << num_procs << "\n";
    cout << "  Number of threads =              " << num_threads << "\n\n";
    cout << "Type N and enter: ";
    cin >> n ;

    start = omp_get_wtime();  // start time check
    total_prime = prime_count(n);
    end = omp_get_wtime();    // end time check

    cout << "  " << setw(8) << "N"
         << "  " << setw(18) << "Count of primes"
         << "  " << setw(18) << "Numer of threads"
         << "  " << setw(30) << "Elapsed wall clock time (sec)" << "\n";

    cout << "  " << setw(8) << n
         << "  " << setw(18) << total_prime
         << "  " << setw(18) << num_threads
         << "  " << setw(30) << end-start << "\n";

    return 0;
}
```

Recap: Hands On: Let us test “schedule”

- **SCHEDULE**: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent. For a discussion on how one type of scheduling may be more optimal than others, see http://openmp.org/forum/viewtopic.php?f=3&t=83.

STATIC
Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC
Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED
Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.
The size of the initial block is proportional to: **`number_of_iterations / number_of_threads`**
Subsequent blocks are proportional to **`number_of_iterations_remaining / number_of_threads`**
The chunk parameter defines the minimum block size. The default chunk size is 1.
Note: compilers differ in how GUIDED is implemented as shown in the "Guided A" and "Guided B" examples below.

RUNTIME
The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

AUTO
The scheduling decision is delegated to the compiler and/or runtime system.

Combined parallel/worksharing construct

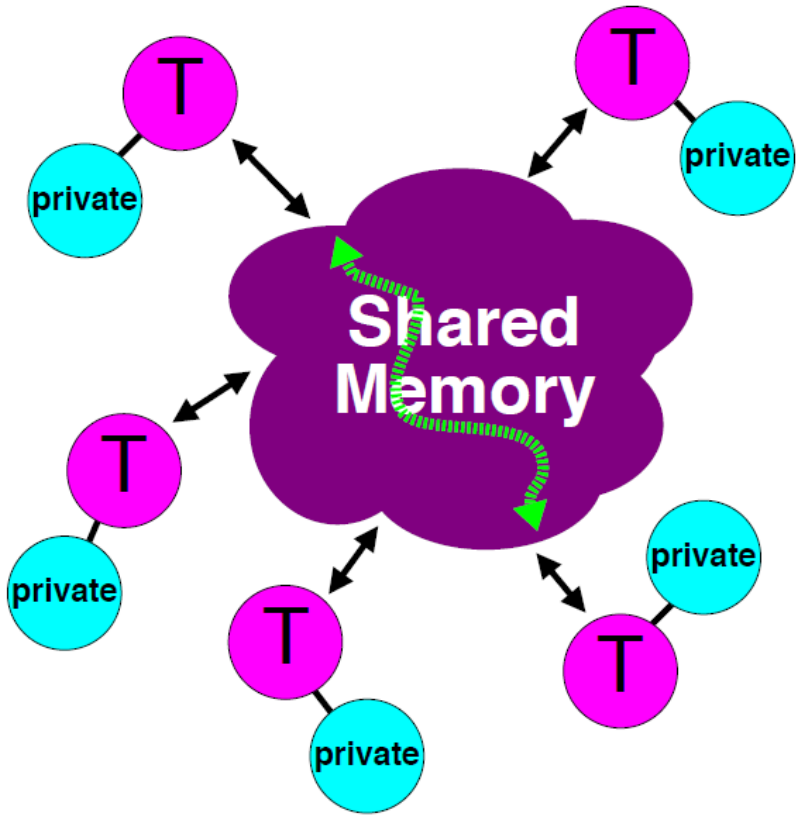
- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

```
#pragma omp parallel
{
#pragma omp for
   for (ii=0; ii<N; ii++)
      a[ii] = ...;
}
```

=

```
#pragma omp parallel for
   for (ii=0; ii<N; ii++)
      a[ii] = ...;
```

Data-sharing Attributes



- All threads have access to the same, *globally shared*, memory
- Data can be shared or private
- *Shared* data is accessible by all threads
- *Private* data can only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- *Synchronization* takes place, but it is *mostly implicit*

The PRIVATE and SHARED clauses

- Shared - There is only one instance of the data
  - All threads can read and write the data simultaneously, unless protected through a specific OpenMP construct
  - All changes made are visible to all threads
- Private - Each thread has a copy of the data
  - No other thread can access this data
  - Changes only visible to the thread owning the data

Hands On: Let us test “private”

- Delete “`private(i, j, prime)`”. Then, how is the result? Fast? Correct?
- How about just delete “`private(i, j)`? Then, how is the result? Fast? Correct?
- Declare the `i` and `j` inside of the for loop. Then, how is the result? Fast? Correct?

Overriding storage attributes

- private:
  - A copy of the variable is created for each thread
  - There is no connection between the original variable and the private copies

    private variables are not initialized
  - Can achieve the same using variables inside { }
- firstprivate:
  - Same, but the initial value of the variable is copied **from** the main copy
  - All variables in the list are initialized with the value the original object had before entering the parallel construct
- lastprivate:
  - Same, but the last value of the variable is copied **to** the main copy
  - The thread that executes the sequentially last iteration or section updates the value of the objects in the list

firstprivate and lastprivate example

- Here I will consider firstprivate and lastprivate.
- Recall one of the earlier entries about private variables. When a variable is declared as private, each thread gets a unique memory address of where to store values for that variable while in the parallel region.
- When the parallel region ends, the memory is freed and these variables no longer exist. Consider the following bit of code as an example:

firstprivate and lastprivate example

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

using namespace std;

//*********************************************************************
int main ( int argc, char *argv[] )
//*********************************************************************
{
    int x(100);

    #pragma omp parallel for private(x)
    for(int i=0; i<10; i++){
        x=i;
        printf("Thread number: %d      x: %d\n",omp_get_thread_num(),x);
    }
    printf("x is %d\n", x);

    return 0;
}
```

firstprivate and lastprivate example

```
ai@ubuntu-20-04:~/Lab/OpenMP$ export OMP_NUM_THREADS=4
ai@ubuntu-20-04:~/Lab/OpenMP$ ./first_last_private_omp
Thread number: 0      x: 0
Thread number: 0      x: 1
Thread number: 0      x: 2
Thread number: 3      x: 8
Thread number: 3      x: 9
Thread number: 2      x: 6
Thread number: 2      x: 7
Thread number: 1      x: 3
Thread number: 1      x: 4
Thread number: 1      x: 5
x is 100
```

- You'll notice that `x` is exactly the value it was before the parallel region.

firstprivate and lastprivate example

- Suppose we wanted to keep the last value of x after the parallel region. This can be achieved with lastprivate. Replace `private(x)` with `lastprivate(x)` and this is the result:

```
ai@ubuntu-20-04:~/Lab/OpenMP$ ./first_last_private_omp
Thread number: 0      x: 0
Thread number: 0      x: 1
Thread number: 0      x: 2
Thread number: 2      x: 6
Thread number: 2      x: 7
Thread number: 3      x: 8
Thread number: 3      x: 9
Thread number: 1      x: 3
Thread number: 1      x: 4
Thread number: 1      x: 5
x is 9
```

- Notice that it is 9 and not 5. That is to say, it is the last iteration which is kept, not the last operation.

firstprivate and lastprivate example

- Now what if we replace lastprivate(x) with firstprivate(x). What do you think it will do?

This:

```
ai@ubuntu-20-04:~/Lab/OpenMP$ ./first_last_private_omp
Thread number: 0      x: 0
Thread number: 0      x: 1
Thread number: 0      x: 2
Thread number: 2      x: 6
Thread number: 2      x: 7
Thread number: 3      x: 8
Thread number: 3      x: 9
Thread number: 1      x: 3
Thread number: 1      x: 4
Thread number: 1      x: 5
x is 100
```

- Did you expect to get the value 0 i.e. the value of x on the first iteration? **NO**

firstprivate and lastprivate example

- Now what if we replace lastprivate(x) with firstprivate(x). What do you think it will do? This:

```
ai@ubuntu-20-04:~/Lab/OpenMP$ ./first_last_private_omp
Thread number: 0     x: 0
Thread number: 0     x: 1
Thread number: 0     x: 2
Thread number: 2     x: 6
Thread number: 2     x: 7
Thread number: 3     x: 8
Thread number: 3     x: 9
Thread number: 1     x: 3
Thread number: 1     x: 4
Thread number: 1     x: 5
x is 100
```

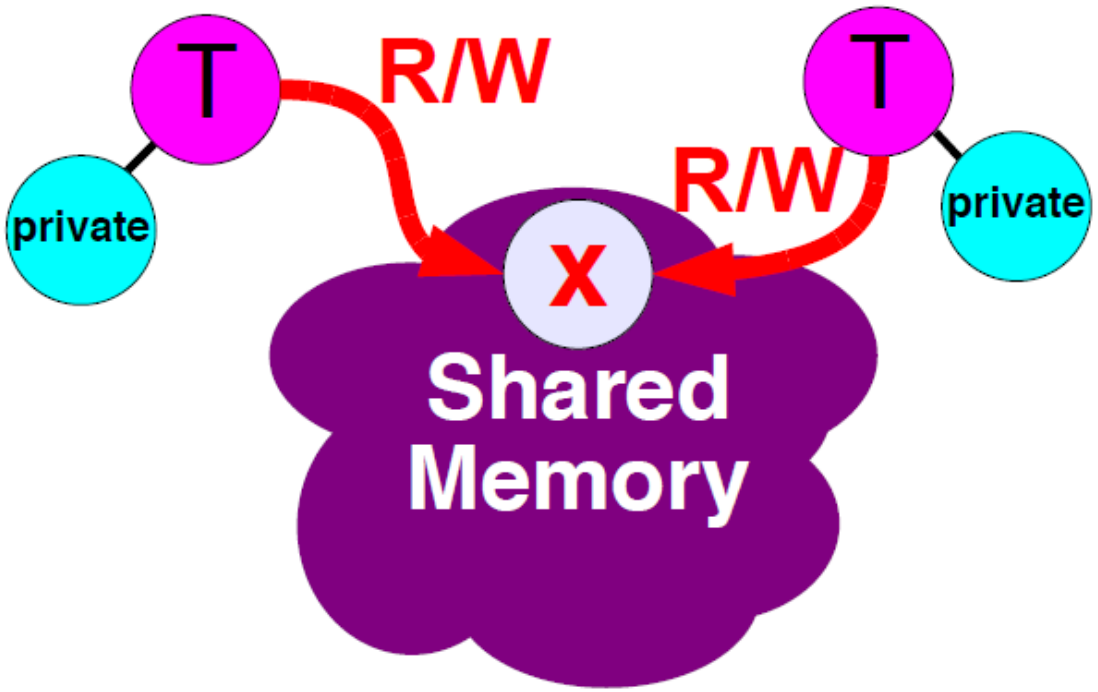
- Did you expect to get the value 0 i.e. the value of x on the first iteration? **NO**
- firstprivate Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.
- That is, every thread gets its own instance of x and that instance equals 100.

What is a Data Race?

- Two different threads in a multi-threaded shared memory program
- Loosely described, a data race means that the update of a shared variable is not well protected
- Access the same (=shared) memory location
  - Asynchronously
  - Without holding any common exclusive locks
  - At least one of the accesses is a write/store

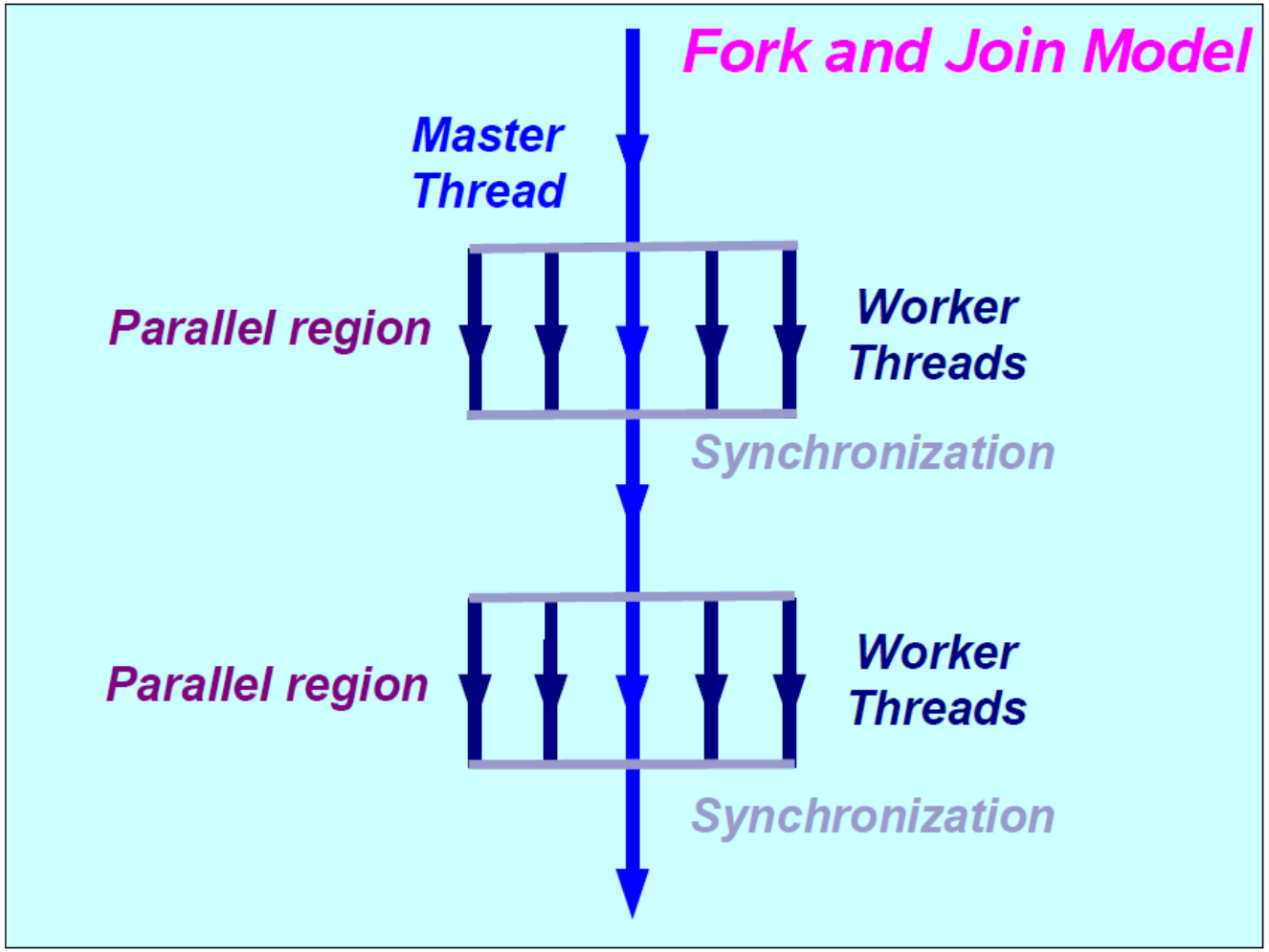
Example of a Data Race

```
#pragma omp parallel shared(x)
{x = x + 1;}
```



How to control race conditions

- Use **synchronization** to protect data conflicts



Implicit Barrier

Implicit Barrier

- beginning and end of `parallel` constructs
- end of all other control constructs

```
#pragma omp parallel [clause…]
   structured_block
```

```
#pragma omp parallel
{
  printf("Hello!\n");
} // implicit barrier
```

```
Hello!
Hello!
Hello!
Hello!
```

The if/private/shared clauses

if (scalar expression)

- Only execute in parallel if expression evaluates to true
- Otherwise, execute serially

private (list)

- No storage association with original object
- All references are to the local object
- Values are undefined on entry and exit

shared (list)

- Data is accessible by all threads in the team
- All threads access the same address space

```
#pragma omp parallel if (n > threshold) \
shared(n,x,y) private(i)
{
  #pragma omp for
  for (i=0; i<n; i++)
  x[i] = y[i] + 10;
} /*-- End of parallel region --*/
```

The if clause

if (scalar expression)

- Only execute in parallel if expression evaluates to true
- Otherwise, execute serially

```
#pragma omp parallel if (n > some_threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

Barrier

- Suppose we run each of these two loops in parallel over i:

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

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

This may give us a wrong answer. Why ?

Barrier

- Suppose we run each of these two loops in parallel over i:

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

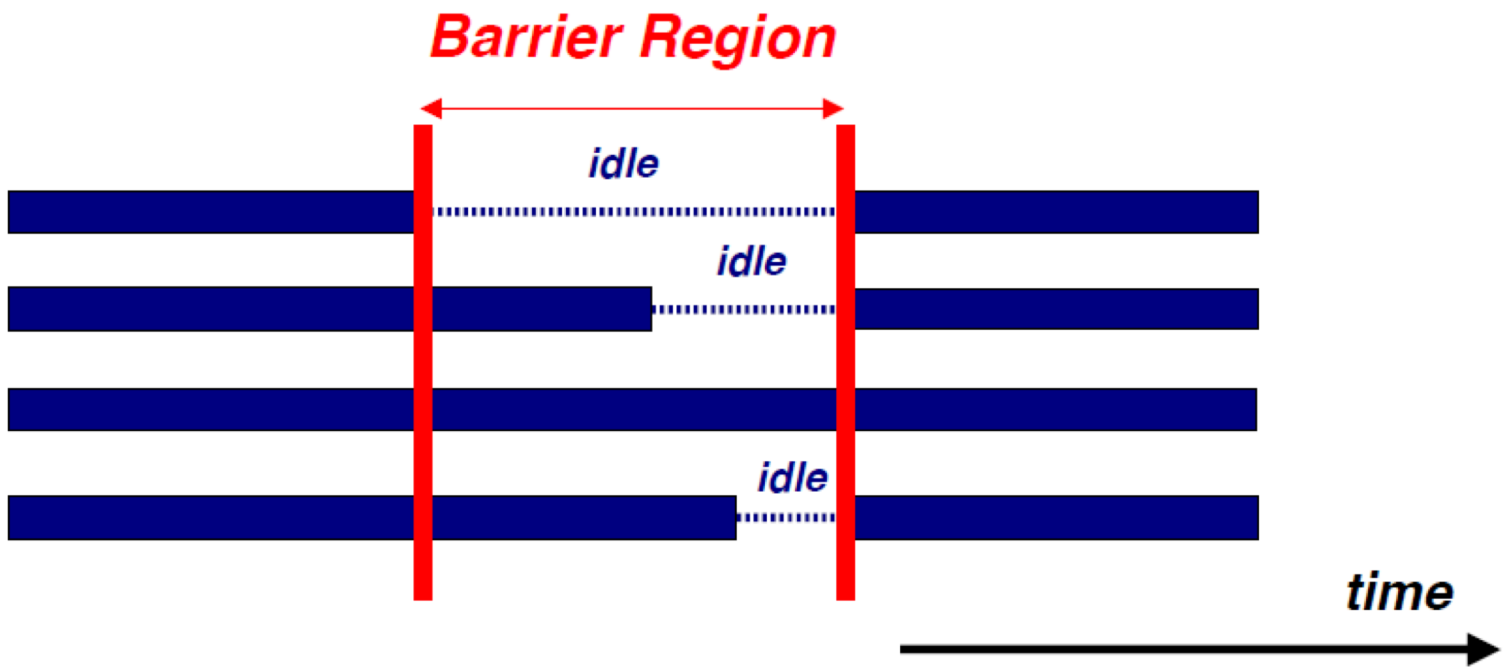
wait

Barrier

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

All threads wait at the barrier point and only continue when all threads have reached the barrier point

#pragma omp barrier



Barrier syntax in OpenMP:

```
#pragma omp barrier
```

```
!$omp barrier
```

#pragma omp for nowait

- `nowait` – Removes implicit barrier from end of block
  - To minimize synchronization, some OpenMP directives/pragmas support the optional `nowait` clause
  - If present, threads do not synchronize/wait at the end of that particular construct
  - In Fortran the nowait clause is appended at the closing part of the construct
  - In C, it is one of the clauses on the pragma

```
#pragma omp for nowait
{
        :
}
```

```
!$omp do
        :
        :
!$omp end do nowait
```

CRITICAL directive

If sum is a shared variable, this loop can not run in parallel

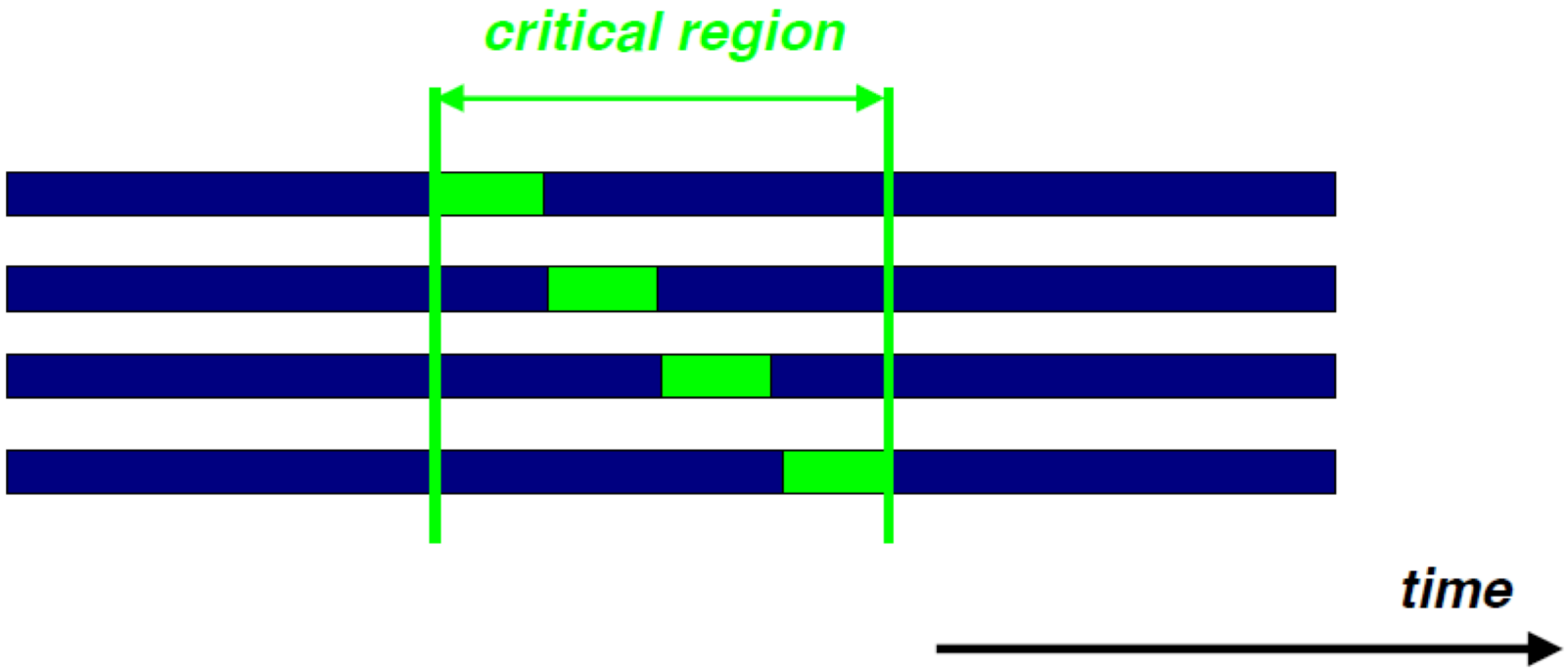
```
for (i=0; i < n; i++){
      .....
   sum += a[i];
      .....
}
```

We can use a critical region for this:

```
for (i=0; i < n; i++){
      .....                    one at a time can proceed
   sum += a[i];
      .....                    next in line, please
}
```

CRITICAL region

- Useful to avoid a race condition, or to perform I/O (but that still has random order)
- Be aware that there is a cost associated with a critical region



CRITICAL example

```
#pragma omp critical
```

```
#pragma omp parallel shared(x)
{
  #pragma omp critical
  {
    // only one thread in here
  }
} // implicit barrier
```

CRITICAL and ATOMIC Constructs

Critical: All threads execute the code, but only one at a time:

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
        <code-block>
!$omp end critical [(name)]
```

There is no implied barrier on entry or exit !

Atomic: only the loads and store are atomic ....

```
#pragma omp atomic
   <statement>
```

```
!$omp atomic
   <statement>
```

This is a lightweight, special form of a critical section

```
#pragma omp atomic
    a[indx[i]] += b[i];
```

Atomic vs Critical

- Syntax

```
#pragma omp atomic
    expression
```

```
#pragma omp critical [(name)]
{
    code_block
}
```

Reduction?

- The REDUCTION clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in statements which have one of following forms:

Reduction

REDUCTION Clause

Purpose:

- The REDUCTION clause performs a reduction operation on the variables that appear in its list.
- A private copy for each list variable is created and initialized for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Format:

Fortran	`REDUCTION (operator: list)`
C/C++	`reduction (operator: list)`

Valid Operators and Initialization Values

Operation	Fortran	C/C++	Initialization
Addition	+	+	0
Multiplication	*	*	1
Subtraction	-	-	0
Logical AND	.and.	&&	0
Logical OR	.or.	||	.false. / 0
AND bitwise	iand	&	all bits on / 1
OR bitwise	ior	|	0
Exclusive OR bitwise	ieor	^	0
Equivalent	.eqv.		.true.
Not Equivalent	.neqv.		.false.
Maximum	max	max	Most negative #
Minimum	min	min	Largest positive #

prime_count_omp.cpp

- Open the code!

```cpp
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <omp.h>

using namespace std;

//****************************************************************************
int prime_count ( int n )
//****************************************************************************
{
    int i, j, prime, total_prime=0;

    #pragma omp parallel for reduction(+:total_prime) \
                             schedule(static) private(i, j, prime)
    for ( i = 2; i <= n; i++ ) {
        prime = 1;
        for ( j = 2; j < i; j++ ) {
            if ( i % j == 0 ) {
                prime = 0;
                break;
            }
        }
        total_prime += prime;
    }
    return total_prime;
}

```

```cpp
//****************************************************************************80
int main (int argc, char *argv[])
//****************************************************************************80
{
    int n, num_procs, num_threads, total_prime=0;
    double start, end;

    // Returns the number of processors that are available to the program
    num_procs = omp_get_num_procs();

    // Returns the maximum value that can be returned by a call to the
    // OMP_GET_NUM_THREADS function
    num_threads = omp_get_max_threads();

    cout << "\n";
    cout << "SCHEDULE_OPENMP\n";
    cout << "  C++/OpenMP version\n";
    cout << "  Count the primes from 1 to N.\n";
    cout << "  This is an unbalanced work load, particular for two threads.\n";
    cout << "  Demonstrate static and dynamic scheduling.\n";
    cout << "\n";
    cout << "  Number of processors available = " << num_procs << "\n";
    cout << "  Number of threads =              " << num_threads << "\n\n";
    cout << "Type N and enter: ";
    cin >> n ;

    start = omp_get_wtime();  // start time check
    total_prime = prime_count(n);
    end = omp_get_wtime();    // end time check

    cout << "  " << setw(8) << "N"
         << "  " << setw(18) << "Count of primes"
         << "  " << setw(18) << "Numer of threads"
         << "  " << setw(30) << "Elapsed wall clock time (sec)" << "\n";

    cout << "  " << setw(8) << n
         << "  " << setw(18) << total_prime
         << "  " << setw(18) << num_threads
         << "  " << setw(30) << end-start << "\n";

    return 0;
}
```

Hands On: Let us test “critical, atomic, and reduction”

- Delete reduction clause and add critical block to the `total_prime` line.
- Test the atomic.
- Test the reduction again.

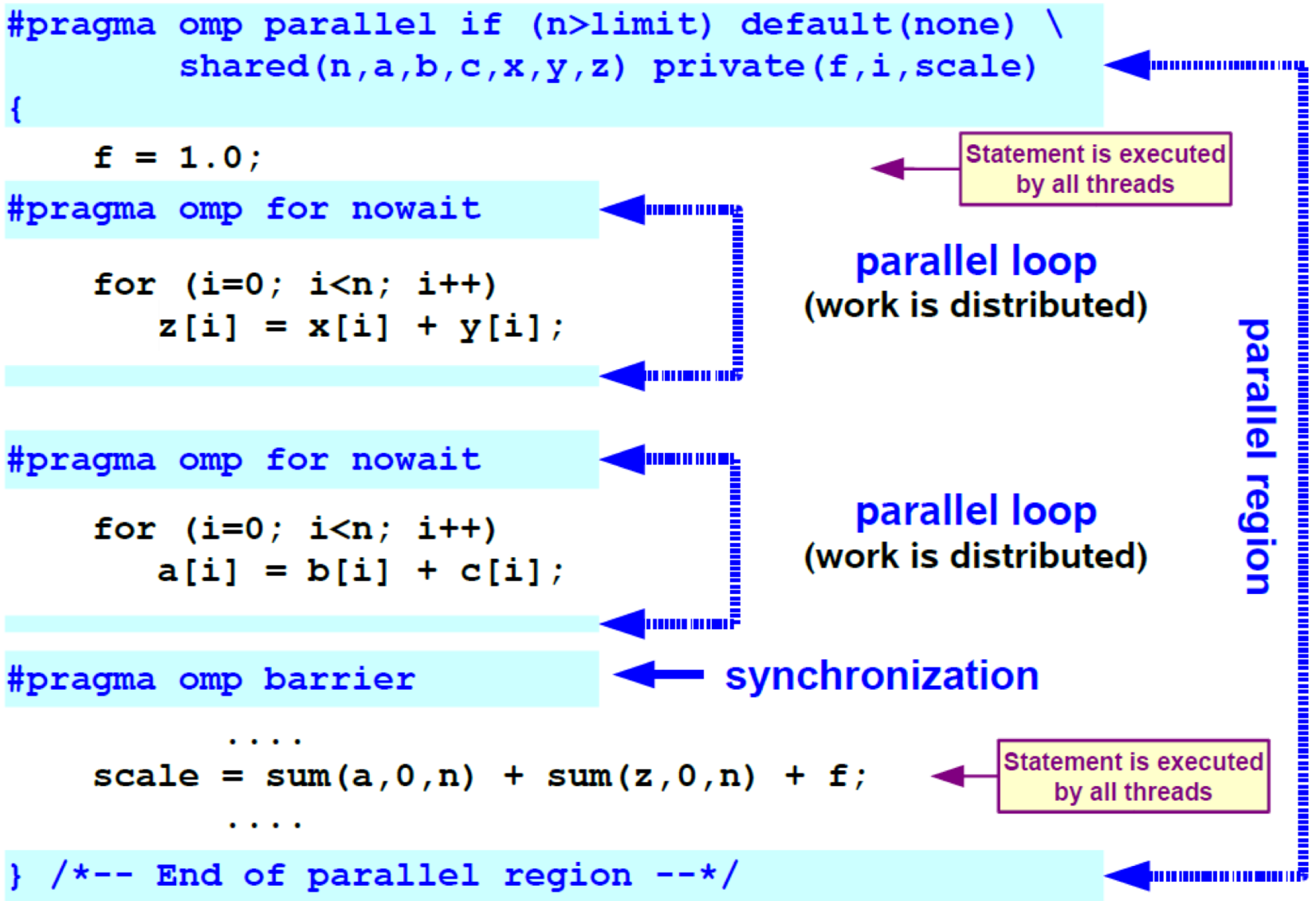
prime_count_omp.cpp

```
 8 //*****************************************************************************80
 9 int prime_count ( int n )
10 //*****************************************************************************80
11 {
12     int prime, total_prime=0;
13
14     #pragma omp parallel for schedule(static,100) private(prime)
15
16     for ( int i = 2; i <= n; i++ ) {
17         prime = 1;
18         for ( int j = 2; j < i; j++ ) {
19             if ( i % j == 0 ) {
20                 prime = 0;
21                 break;
22             }
23         }
24         #pragma omp critical
25         {
26             total_prime += prime;
27         }
28     }
29     return total_prime;
30 }
```

Synchronization Constructs

```
#pragma omp critical
#pragma omp atomic
#pragma omp barrier
#pragma omp master
```

A more elaborate example



MASTER directive

- The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code
- There is no implied barrier associated with this directive

```
#pragma omp master
{<code-block>}
```

```
!$omp master
        <code-block>
!$omp end master
```

OMP Master

- master singles out a block of code for execution by the master thread (thread with ID of 0) only.
- There is no implicit barrier at the end of the construct and also there is no requirement that all threads must encounter the master construct.
- master is basically a shorthand for

```
if (omp_get_thread_num() == 0) {
 ...
}
```

Can we use Master in this example?

```
 8 //*************************************************************************80
 9 int prime_count ( int n )
10 //*************************************************************************80
11 {
12     int prime, total_prime=0;
13
14     #pragma omp parallel for schedule(static,100) private(prime)
15
16     for ( int i = 2; i <= n; i++ ) {
17         prime = 1;
18         for ( int j = 2; j < i; j++ ) {
19             if ( i % j == 0 ) {
20                 prime = 0;
21                 break;
22             }
23         }
24         #pragma omp master
25         {
26             total_prime += prime;
27         }
28     }
29     return total_prime;
30 }
```

Can we use Master in this example? Nope!

```
#pragma omp parallel for schedule(static) private(prime)
for ( int i = 2; i <= n; i++ ) {
    prime = 1;
    for ( int j = 2; j < i; j++ ) {
        if ( i % j == 0 ) {
            prime = 0;
            break;
        }
    }
    // this shoud be not touched in multi-threads at same time
    #pragma omp master
    {
    total_prime += prime;
    }
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -Wall -fopenmp prime_count_omp.cpp -o prime_count_omp
prime_count_omp.cpp: In function ‘int prime_count(int)’:
prime_count_omp.cpp:24:10: error: ‘master’ region may not be closely nested inside of wo
rk-sharing, explicit ‘task’ or ‘taskloop’ region
   24 |   #pragma omp master
      |           ^~~
```

Two for loops!

- Nested loops!

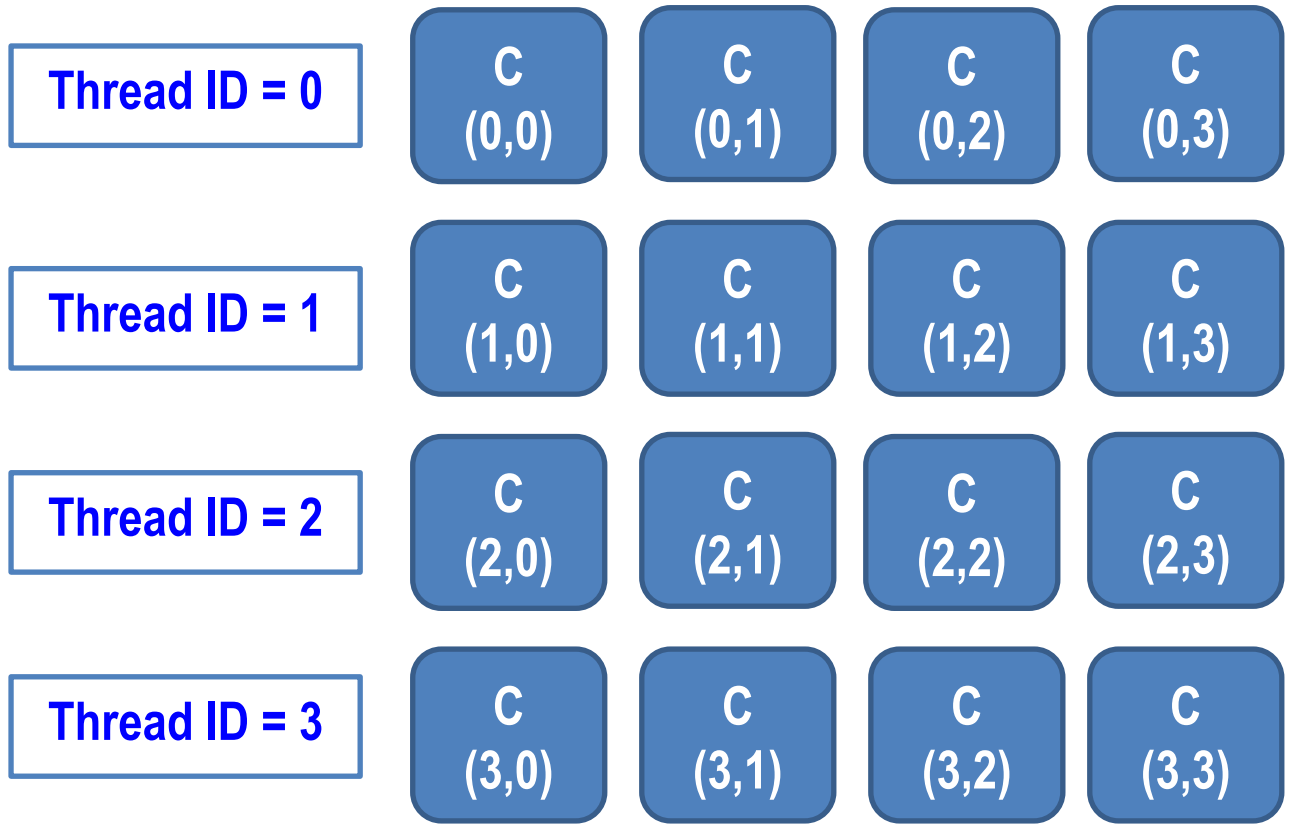
```
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <omp.h>

using namespace std;

//****************************************************************************
int prime_count ( int n )
//****************************************************************************
{
    int i, j, prime, total_prime=0;

    #pragma omp parallel for reduction(+:total_prime) \
                             schedule(static) private(i, j, prime)
    for ( i = 2; i <= n; i++ ) {
        prime = 1;
        for ( j = 2; j < i; j++ ) {
            if ( i % j == 0 ) {
                prime = 0;
                break;
            }
        }
        total_prime += prime;
    }
    return total_prime;
}

```

```
//****************************************************************************80
int main (int argc, char *argv[])
//****************************************************************************80
{
    int n, num_procs, num_threads, total_prime=0;
    double start, end;

    // Returns the number of processors that are available to the program
    num_procs = omp_get_num_procs();

    // Returns the maximum value that can be returned by a call to the
    // OMP_GET_NUM_THREADS function
    num_threads = omp_get_max_threads();

    cout << "\n";
    cout << "SCHEDULE_OPENMP\n";
    cout << "  C++/OpenMP version\n";
    cout << "  Count the primes from 1 to N.\n";
    cout << "  This is an unbalanced work load, particular for two threads.\n";
    cout << "  Demonstrate static and dynamic scheduling.\n";
    cout << "\n";
    cout << "  Number of processors available = " << num_procs << "\n";
    cout << "  Number of threads =              " << num_threads << "\n\n";
    cout << "Type N and enter: ";
    cin >> n ;

    start = omp_get_wtime();  // start time check
    total_prime = prime_count(n);
    end = omp_get_wtime();    // end time check

    cout << "  " << setw(8) << "N"
         << "  " << setw(18) << "Count of primes"
         << "  " << setw(18) << "Numer of threads"
         << "  " << setw(30) << "Elapsed wall clock time (sec)" << "\n";

    cout << "  " << setw(8) << n
         << "  " << setw(18) << total_prime
         << "  " << setw(18) << num_threads
         << "  " << setw(30) << end-start << "\n";

    return 0;
}
```

Parallelizing nested loops

- If we have nested for loops, it is often enough to simply **parallelize the outermost loop.**

```
#pragma omp parallel for
for (int i = 0; i < 4; ++i)
{
    for (int j = 0; j < 4; ++j)
    {
        c(i, j);
    }
}
```



Parallelizing nested loops

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

using namespace std;

void report(int i, int j)
{
    printf("Thread ID=%d: C(i,j)=C(%d,%d)\n",
        omp_get_thread_num(), i, j);
}

int main(int argc, char *argv[])
{
    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            printf("Thread ID=%d: Number of threads=%d\n",
                omp_get_thread_num(), omp_get_num_threads());
        }

        #pragma omp barrier

        #pragma omp for
        for (int i=0; i<4; i++)
        {
            for (int j=0; j<4; j++)
            {
                report(i, j);
            }
        }
    }

    return(0);
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ ./nested_forloops_omp
Thread ID=0: Number of threads=4
Thread ID=3: C(i,j)=C(3,0)
Thread ID=3: C(i,j)=C(3,1)
Thread ID=3: C(i,j)=C(3,2)
Thread ID=3: C(i,j)=C(3,3)
Thread ID=0: C(i,j)=C(0,0)
Thread ID=0: C(i,j)=C(0,1)
Thread ID=0: C(i,j)=C(0,2)
Thread ID=0: C(i,j)=C(0,3)
Thread ID=1: C(i,j)=C(1,0)
Thread ID=1: C(i,j)=C(1,1)
Thread ID=1: C(i,j)=C(1,2)
Thread ID=1: C(i,j)=C(1,3)
Thread ID=2: C(i,j)=C(2,0)
Thread ID=2: C(i,j)=C(2,1)
Thread ID=2: C(i,j)=C(2,2)
Thread ID=2: C(i,j)=C(2,3)
```

Parallelizing nested loops: Challenges

- Sometimes the outermost loop is so short that not all threads are utilized.

```
#pragma omp parallel for
for (int i = 0; i < 3; ++i)
{
    for (int j = 0; j < 6; ++j)
    {
        c(i, j);
    }
}
```

Thread ID = 0

Thread ID = 1

Thread ID = 2

Thread ID = 3

C (0,0)	C (0,1)	C (0,2)	C (0,3)	C (0,4)	C (0,5)
C (1,0)	C (1,1)	C (1,2)	C (1,3)	C (1,4)	C (1,5)
C (2,0)	C (2,1)	C (2,2)	C (2,3)	C (2,4)	C (2,5)

Parallelizing nested loops: Challenges

- Sometimes the outermost loop is so short that not all threads are utilized.

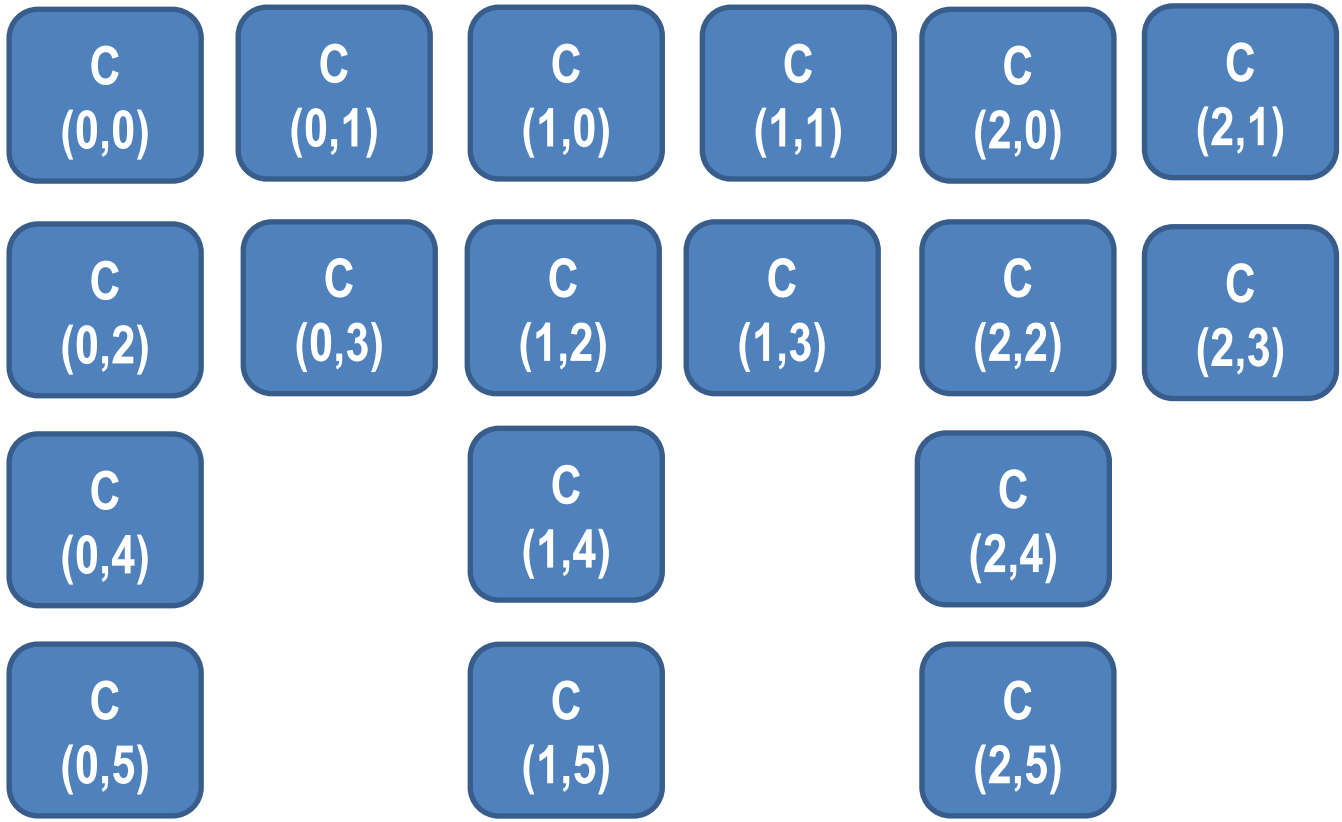
```
for (int i = 0; i < 3; ++i)
{
    #pragma omp parallel for
    for (int j = 0; j < 6; ++j)
    {
        c(i, j);
    }
}
```

Thread ID = 0

Thread ID = 1

Thread ID = 2

Thread ID = 3



Parallelizing nested loops: Challenges

- **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause.

```
#pragma omp parallel for collapse (2)
for (int i = 0; i < 3; ++i)
{
    for (int j = 0; j < 6; ++j)
    {
        c(i, j);
    }
}
```

Thread ID = 0

Thread ID = 1

Thread ID = 2

Thread ID = 3

C (0,0)	C (0,1)	C (0,2)	C (0,3)	C (0,4)
C (0,5)	C (1,0)	C (1,1)	C (1,2)	C (1,3)
C (1,4)	C (1,5)	C (2,0)	C (2,1)	
C (2,2)	C (2,3)	C (2,4)	C (2,5)	

Parallelizing nested loops

```
#pragma omp for collapse(2)
for (int i=0; i<3; i++)
{
    for (int j=0; j<6; j++)
    {
        report(i, j);
    }
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ ./nested_forloops_omp
Thread ID=0: Number of threads=4
Thread ID=0: C(i,j)=C(0,0)
Thread ID=2: C(i,j)=C(1,4)
Thread ID=3: C(i,j)=C(2,2)
Thread ID=1: C(i,j)=C(0,5)
Thread ID=0: C(i,j)=C(0,1)
Thread ID=0: C(i,j)=C(0,2)
Thread ID=3: C(i,j)=C(2,3)
Thread ID=3: C(i,j)=C(2,4)
Thread ID=3: C(i,j)=C(2,5)
Thread ID=2: C(i,j)=C(1,5)
Thread ID=2: C(i,j)=C(2,0)
Thread ID=2: C(i,j)=C(2,1)
Thread ID=0: C(i,j)=C(0,3)
Thread ID=0: C(i,j)=C(0,4)
Thread ID=1: C(i,j)=C(1,0)
Thread ID=1: C(i,j)=C(1,1)
Thread ID=1: C(i,j)=C(1,2)
Thread ID=1: C(i,j)=C(1,3)
```

omp single

- The three worksharing constructs cover three different general cases:
  - for (a.k.a. loop construct): distributes automatically the iterations of a loop among the threads - in most cases all threads get work to do;
  - sections: distributes a sequence of independent blocks of code among the threads - some threads get work to do. This is a generalization of the for construct as a loop with 100 iterations could be expressed as e.g. 10 sections of loops with 10 iterations each.
  - single: singles out a block of code for execution by one thread only, often the first one to encounter it (an implementation detail) - only one thread gets work. single is to a great extent equivalent to sections with a single section only.

omp critical and omp single

- single and critical belong to two completely different classes of OpenMP constructs.
- single is a worksharing construct, alongside for and sections.
- Worksharing constructs are used to distribute a certain amount of work among the threads.
- Such constructs are "collective" in the sense that in correct
- OpenMP programs all threads must encounter them while executing and moreover in the same sequential order, also including the barrier constructs.

omp critical and omp single

- single and critical are two very different things.
  - single specifies that a section of code should be executed by single thread (not necessarily the master thread)
  - critical specifies that code is executed by one thread at a time

```
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);
```

  - will print

    single: 1 -- critical: 4

omp master vs omp single

- master singles out a block of code for execution by the master thread (thread with ID of 0) only.
- Unlike single, there is no implicit barrier at the end of the construct and also there is no requirement that all threads must encounter the master construct.
- Also, the lack of implicit barrier means that master does not flush the shared memory view of the threads
- master is basically a shorthand for if (omp_get_thread_num() == 0) { ... }.

Nested Parallel Regions

- Make a new parallel region in a parallel region
  - Can make a recursive parallel algorithms

- We can turn-on/off with
  - omp_set_nested (1)
  - export OMP_NESTED=TRUE

Nested Parallel Regions

```c
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team - %d\n",
                level, omp_get_num_threads());
    }
}

int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(2)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ ./omp_nested_omp
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
ai@ubuntu-20-04:~/Lab/OpenMP$ export OMP_NESTED=TRUE
ai@ubuntu-20-04:~/Lab/OpenMP$ ./omp_nested_omp
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
ai@ubuntu-20-04:~/Lab/OpenMP$ export OMP_NESTED=FALSE
ai@ubuntu-20-04:~/Lab/OpenMP$ ./omp_nested_omp
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
```