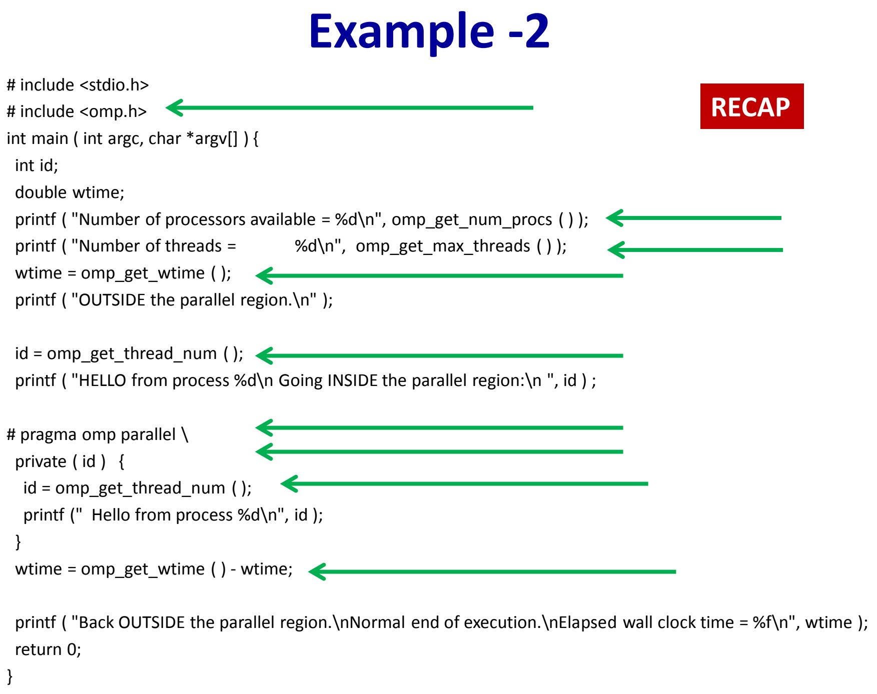
Lecture 05-06



Example -2

RECAP

```
# include <stdio.h>
# include <omp.h>
int main ( int argc, char *argv[] ) {
 int id;
 double wtime;
 printf ( "Number of processors available = %d\n", omp_get_num_procs ( ) );
 printf ( "Number of threads =          %d\n",  omp_get_max_threads ( ) );
 wtime = omp_get_wtime ( );
 printf ( "OUTSIDE the parallel region.\n" );

 id = omp_get_thread_num ( );
 printf ( "HELLO from process %d\n Going INSIDE the parallel region:\n ", id ) ;

# pragma omp parallel \
 private ( id )  {
  id = omp_get_thread_num ( );
  printf ("  Hello from process %d\n", id );
 }
 wtime = omp_get_wtime ( ) - wtime;

 printf ( "Back OUTSIDE the parallel region.\nNormal end of execution.\nElapsed wall clock time = %f\n", wtime );
 return 0;
}
```

Example -2

RECAP

Initialization:

```
export OMP_NUM_THREADS=16
```

Compilation:

```
g++  -fopenmp  example.c
```

Execution:

```
./a.out
```

Example -4

RECAP

```
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
int main()
{
    int i,j,n,m,temp,a[100][100];
    n=m=7;
    #pragma omp parallel
    {
        for(i=0;i<=n*m-1;i++) {
            temp=i/m+1;
            j=i%m+1;
            sleep(1);
            a[temp][j]=temp+100*(j-1);
        }
    }
    for(i=0;i<=n*m-1;i++) {
        temp=i/m+1;
        j=i%m+1;
        if(i%m==0) printf("\n");
        printf("%d\t",a[temp][j]);
    }
    printf("\n");
    return 0;
}
```

```
$ ./a.out

1    0    201  0    401  0    601
0    102  0    302  0    502  0
3    0    203  0    403  0    603
0    104  0    304  0    504  0
5    0    205  0    405  0    605
0    106  0    0    0    506  606
7    0    207  0    407  0    607
```

Example -4

```
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
int main()
{
    int i,j,n,m,temp,a[100][100];
    n=m=7;
    #pragma omp parallel private (temp, j)
    {
            for(i=0;i<=n*m-1;i++) {
                temp=i/m+1;
                j=i%m+1;
                sleep(1);
                a[temp][j]=temp+100*(j-1);
            }
    }
    for(i=0;i<=n*m-1;i++) {
        temp=i/m+1;
        j=i%m+1;
        if(i%m==0) printf("\n");
        printf("%d\t",a[temp][j]);
    }
    printf("\n");
    return 0;
}
```

RECAP

```
$ ./a.out

1    101    201    301    401    501    601
2    102    202    302    402    502    602
3    103    203    303    403    503    603
4    104    204    304    404    504    604
5    105    205    305    405    505    605
6    106    206    306    406    506    606
7    107    207    307    407    507    607
```

Example -5

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int i,x=0, a[10],b[10];
    for(i=0;i<10;i++) {
        a[i]=i;   b[i]=10-i;
    }
    #pragma omp parallel
    {
        #pragma omp for reduction (+:x)
            for(i=0;i<10;i++) {
                x = x + a[i]*b[i];
            }
    }
    printf("%d\n",x);
    return 0;
}
```

RECAP

```
$ ./a.out
165
$
```

Work-sharing Constructs

- sections Construct
  - **#pragma omp sections** *[clause[[,] clause] ...] new-line*

```
{
   [#pragma omp section new-line
           structured-block]
   [#pragma omp section new-line
           structured-block ]
    ...
}
```

- Clause
  - **private(***variable-list***)**
  - **firstprivate(***variable-list***)**
  - **lastprivate(***variable-list***)**
  - **reduction(***operator***:** *variable-list***)**
  - **nowait**

Work-sharing Constructs

- single Construct
  - **#pragma omp single** *[clause[[,] clause] ...] new-line*

    structured-block

  - Clause
    - **private(***variable-list***)**
    - **firstprivate(***variable-list***)**
    - **copyprivate(***variable-list***)**
    - **nowait**

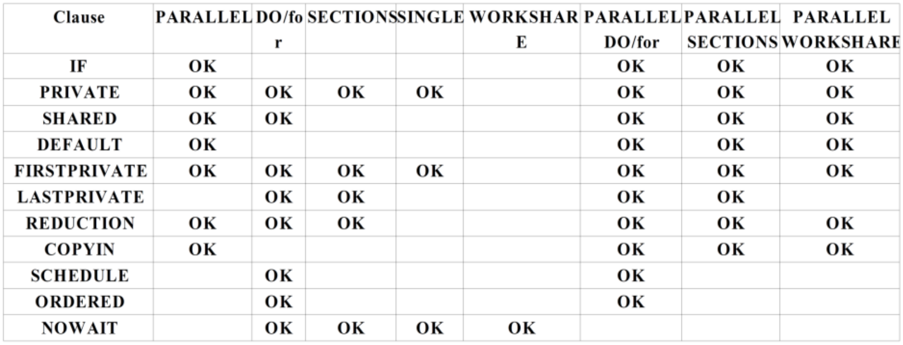
The SINGLE construct allows code that is serial in nature to be executed inside a parallel region. The thread executing the code will be the first to reach the directive in the code. It doesn't have to be the master thread. All other threads proceed to the end of the structured block where there is an implicit synchronization.

Master construct

```
#pragma omp master
        structured-block
```

Same as *single nowait* but only for master thread

Allowed Combinations



Clause	PARALLEL	DO/for	SECTIONS	SINGLE	WORKSHARE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE
IF	OK					OK	OK	OK
PRIVATE	OK	OK	OK	OK		OK	OK	OK
SHARED	OK	OK				OK	OK	OK
DEFAULT	OK					OK	OK	OK
FIRSTPRIVATE	OK	OK	OK	OK		OK	OK	OK
LASTPRIVATE		OK	OK			OK	OK	
REDUCTION	OK	OK	OK			OK	OK	OK
COPYIN	OK					OK	OK	OK
SCHEDULE		OK				OK		
ORDERED		OK				OK		
NOWAIT		OK	OK	OK	OK			

Synchronization

- CRITICAL: Mutual Exclusion
- ATOMIC: Atomic Update
- BARRIER: Barrier Synchronization

Critical construct

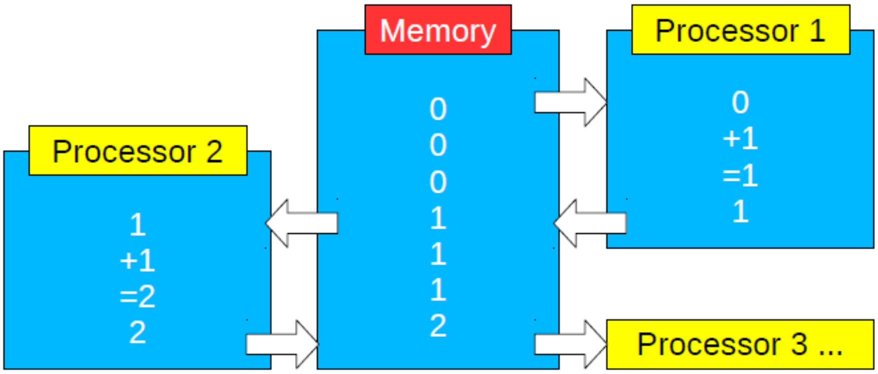
```
#pragma omp critical [name]
        structured-block
```

Example

```
        #pragma omp parallel
        {
                #pragma omp critical(long_critical_name)
                        doSomeCriticalWork_1();
                #pragma omp critical
                        doSomeCriticalWork_2();
                #pragma omp critical
                        doSomeCriticalWork_3();
        }
```

Critical construct

- Only one thread at time in the critical section



Example -6

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

int main()
{
        int x=0, size=12;

        x=x+1;
        printf("%d\n",x);
        return 0;
}
```

```
$ gcc -Wall example2.c
$ ./a.out
1
$
```

Example -6

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

int main()
{
    int x=0, size=12;

   omp_set_num_threads(size);
   #pragma omp parallel shared(x)
   {
        x=x+1;
    }
    printf("%d\n",x);
    return 0;
}
```

```
$ gcc -Wall -fopenmp example2_openMP.c
$ ./a.out
10
$ ./a.out
12
$ ./a.out
12
$ ./a.out
12
$ ./a.out
11
$
```

Example -6

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int x=0, size=12;
    omp_set_num_threads(size);
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        {
         x=x+1;
        }
    }
    printf("%d\n",x);
    return 0;
}
```

```
$ gcc -Wall -fopenmp example2_openMP2.c
$ ./a.out
12
$ ./a.out
12
$ ./a.out
12
$ ./a.out
12
$ ./a.out
12
$
```

Important Notes

- Loop indexes are automatically `PRIVATE`
- Everything “local or temporary” should be `PRIVATE` (or `FIRSTPRIVATE` or `LASTPRIVATE` if its *value* is used *outside* the loop, before or after respectively)
- Everything “persistent” and/or used for different values of the loop index should be `SHARED`
- a `SHARED` variable that is *not* an array accessed with the loop indexes, should be written only in a `CRITICAL` region (serialize, so it's slow)
- If you are using `CRITICAL`,see if `REDUCTION` is an option (maybe changing the math a little bit)

Barrier construct

#pragma omp barrier //Threads wait until all threads reach this point

Example (waiting for the master to come)

```
int counter = 0;
#pragma omp parallel
{
        #pragma omp master
                counter = 1;
        #pragma omp barrier
                printf("%d\n", counter);
}
```

Be careful not to cause deadlock:
No barrier inside of *critical, master, sections, single*!

Atomic construct

```
#pragma omp atomic \
        [read | write | update | capture]
        expression-stmt

#pragma omp atomic capture
        structured-block
```

Work out - 1

- Read an integer number by master thread only
- Fork 8 threads
- Generate i^{th} prime number (i is the thread number)
- Multiply with the input number
- Output the result as ordered by the thread number (thread 0, 1, …)

Combined Parallel Work-sharing Constructs

- **parallel for Construct**
  - **#pragma omp parallel for** *[clause[[,] clause] ...] new-line*
    - *for-loop*

Combined Parallel Work-sharing Constructs

- **parallel sections Construct**

```
#pragma omp parallel sections [clause[[,] clause] ...] new-line
{
        [#pragma omp section new-line]
            structured-block
        [#pragma omp section new-line
            structured-block ]
...
}
```

Loop construct: Scheduling

```
#pragma omp for schedule(kind[, chunk_size])
        for-loops
```

Static

- iterations are divided into chunks of size chunk_size
- the chunks are assigned to the threads in a round-robin fashion
- must be reproducible within the same parallel region

Dynamic

- iterations are divided into chunks of size chunk_size
- the chunks are assigned to the threads as they request them
- the default chunk_size is 1

Guided

- iterations are divided into chunks of decreasing size
- the chunks are assigned to the threads as they request them
- chunk_size controls the minimum size of the chunks

Run-time

- controlled by environment variables

Work out - 2

- Write a serial program to output the prime numbers occurring between 1 and 131072. Report the time required to compute the *i*th prime.
- Convert it to a OpenMP code. Report the percentage of improvement for each prime number over serial program using 4 and 16 cores.

Example 8: Matrix Multiplication

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

#define NRA 62              /* number of rows in matrix A */
#define NCA 15              /* number of columns in matrix A */
#define NCB 7              /* number of columns in matrix B */

int main (int argc, char *argv[])
{
int    tid, nthreads, i, j, k, chunk;
double  a[NRA][NCA],
      b[NCA][NCB],
      c[NRA][NCB];        /* result matrix*/

chunk = 10;                /* set loop iteration chunk size */
```

Example 8: Matrix Multiplication

```
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
 {
 tid = omp_get_thread_num();
 if (tid == 0)    {
  nthreads = omp_get_num_threads();
  printf("Starting matrix multiple example with %d threads\n",nthreads);
  printf("Initializing matrices...\n");
  }

#pragma omp for schedule (static, chunk)
 for (i=0; i<NRA; i++)
  for (j=0; j<NCA; j++)
   a[i][j]= i+j;
 #pragma omp for schedule (static, chunk)
 for (i=0; i<NCA; i++)
  for (j=0; j<NCB; j++)
   b[i][j]= i*j;
 #pragma omp for schedule (static, chunk)
 for (i=0; i<NRA; i++)
  for (j=0; j<NCB; j++)
   c[i][j]= 0;
```

Example 8: Matrix Multiplication

```
/*** Do matrix multiply sharing iterations on outer loop ***/
 /*** Display who does which iterations for demonstration purposes ***/
 printf("Thread %d starting matrix multiply...\n",tid);
 #pragma omp for schedule (static, chunk)
 for (i=0; i<NRA; i++)  {
   printf("Thread=%d did row=%d\n",tid,i);
   for(j=0; j<NCB; j++)
     for (k=0; k<NCA; k++)
       c[i][j] += a[i][k] * b[k][j];
   }
 }  /*** End of parallel region ***/

/*** Print results ***/
printf("******************************************************\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)  {
 for (j=0; j<NCB; j++)
   printf("%6.2f   ", c[i][j]);
 printf("\n");
 }
printf("******************************************************\n");
printf ("Done.\n");
}
```

Question???

- Is the static kind optimum?

Loop construct: Nested Loops

```
#pragma omp parallel
{
        #pragma omp for
                for(int ii = 0; ii < n; ii++) {
                        #pragma omp for
                                for(int jj = 0; jj < m; jj ++) {
                                        A[ii][jj] = ii*m + jj;
                                }
                }
}
```

Collapse clause

- Available in OpenMP 3.0 and later
- Clause for the PARALLEL DO directive
- Cause an automatic “collapse” (merge) of the loops, and thus automatic parallelization of inner loops too
- Most things taken care automatically, but user have to be careful

Loop construct: Nested Loops

```
#pragma omp parallel
{
        #pragma omp for collapse(2)
                for(int ii = 0; ii < n; ii++) {
                        for(int jj = 0; jj < m; jj ++) {
                                A[ii][jj] = ii*m + jj;
                        }
                }
}
```

The collapsed loops must be perfectly nested.

Collapse Clause

- **Purpose**
  - Specifying the COLLAPSE clause allows you to parallelize multiple loops in a nest without introducing nested parallelism.
- **Rules**
  - Only one collapse clause is allowed on a work sharing DO or PARALLEL DO directive
  - The specified number of loops must be present lexically. That is, none of the loops can be in a called subroutine.
  - The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
  - If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
  - The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP directive between the loops which are collapsed.
  - The associated do-loops must be structured blocks. Their execution must not be terminated by an EXIT statement.
  - If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a CYCLE statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements except for the innermost associated loop.

Example-9

```
#pragma omp parallel shared(a,b,c)
 {
         for (i=0; i<M; i++)
                  for(j=0; j<N; j++)
                           for (k=0; k<P; k++)
                                   c[i][k] += a[i][j] * b[j][k];
 }
```

Only the outer loop (i) is parallel.
Loops on j,k are serial.
Possible solutions:
(i) change it to 1D array
(ii) nested parallelism

Example-9

```
#pragma omp parallel shared(a,b,c)
// i,j,k are automatically private
 {
     #pragma omp for collapse(3)
         for (i=0; i<M; i++)
                  for(j=0; j<N; j++)
                           for (k=0; k<P; k++)
                                   c[i][k] += a[i][j] * b[j][k];
 }
```

Now everything is parallel.
c is shared. Possibly will be modified by other threads!

Example-9

```
 #pragma omp parallel shared(a,b,c)
// i,j,k are automatically private
 {
      #pragma omp for collapse(2)
           for (i=0; i<M; i++)
                     for (k=0; k<P; k++)
                               for(j=0; j<N; j++)
                                         c[i][k] += a[i][j] * b[j][k];
 }
```

Parallelize as much as possible. Guarantee correctness and without using CRITICAL !!

Work Out - 3

- Can you rewrite the matrix multiplication using collapse?

Example 10: Random Number

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>
# include <time.h>
void monte_carlo ( int n, int *seed );
double random_value ( int *seed );
void timestamp ( void );

void main (  ) {
  int n;
  int seed;
  timestamp ( );
  printf ( "  Number of processors available = %d\n", omp_get_num_procs ( ) );
  printf ( "  Number of threads =            %d\n", omp_get_max_threads ( ) );

  n = 100;
  seed = 123456789;
  monte_carlo ( n, &seed );
  printf ( "RANDOM_OPENMP\n" );
  printf ( "  Normal end of execution.\n" );
  timestamp ( );
}
```

Example 10: Random Number

```
void monte_carlo ( int n, int *seed )
{
  int i, my_id,my_seed;
  double *x;
  x = ( double * ) malloc ( n * sizeof ( double ) );
# pragma omp master
{
   printf ( "  Thread   Seed  I   X(I)\n" );
}
# pragma omp parallel private ( i, my_id, my_seed ) shared ( n, x )
{
  my_id = omp_get_thread_num ( );
  my_seed = *seed + my_id;
  printf ( "  %6d  %12d\n", my_id, my_seed );

# pragma omp for
  for ( i = 0; i < n; i++ )  {
    x[i] = random_value ( &my_seed );
    printf ( "  %6d  %12d  %6d  %14.6g\n", my_id, my_seed, i, x[i] );
  }
}
  free ( x );
  return;
}
```

Example 10: Random Number

```
double random_value ( int *seed )
{
  double r;

  *seed = ( *seed % 65536 );
  *seed = ( ( 3125 * *seed ) % 65536 );
  r = ( double ) ( *seed ) / 65536.0;

  return r;
}
```

Example 10: Random Number

```
void timestamp ( void )
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  size_t len;
  time_t now;

  now = time ( NULL );
  tm = localtime ( &now );

  len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );

  printf ( "%s\n", time_buffer );

  return;
# undef TIME_SIZE
}
```

Lock Functions

- omp_init_lock
- omp_destroy_lock
- omp_set_lock
- omp_unset_lock
- omp_test_lock

- omp_init_nest_lock
- omp_destroy_nest_lock
- omp_set_nest_lock
- omp_unset_nest_lock
- omp_test_nest_lock