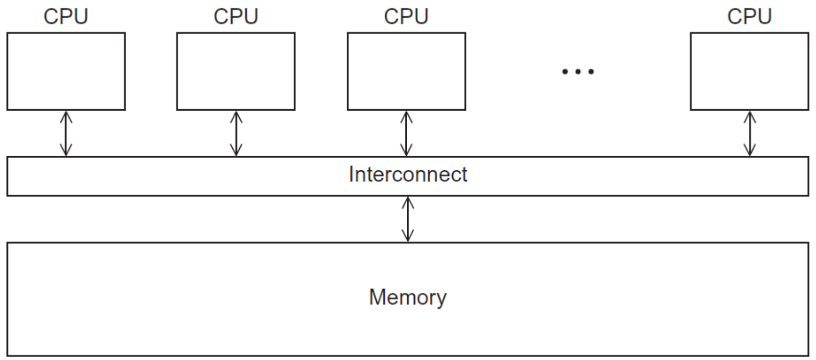
M4: Shared Memory Programming with OpenMP

- Shared Memory Programming with OpenMP: A Standard for Directive Based Parallel Programming. (6-Hrs)
  - Writing programs that use OpenMP.
  - Using OpenMP to parallelize many serial for loops with only small changes to the source code.
  - Task parallelism.
  - Explicit thread synchronization.
  - Standard problems in shared-memory programming

Writing programs that use OpenMP.

- An API for shared-memory parallel programming.
  - MP = multiprocessing
  - Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPU's, all of which have access to main memory.

•Shared Memory Systems
Conceptual model
Real Hardware model



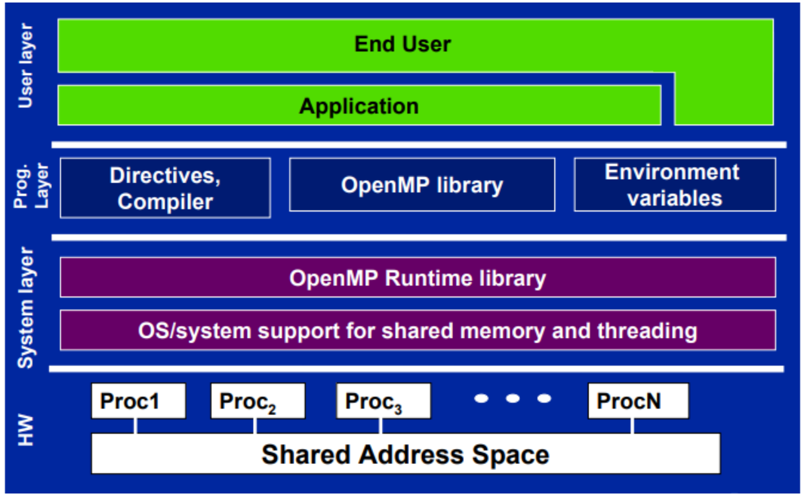
- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
  - Unintended sharing of data causes race conditions
  - To control race conditions: – Use synchronization to protect data conflicts.

OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.
  - #pragma omp construct [clause [clause]…]
- Function prototypes and types in the file: #include<omp.h>.
- constructs apply to a "structured block"

```
#pragma omp parallel num_threads(4)
```

Request a certain number of threads



OpenMp pragmas

# pragma omp parallel

Most basic parallel directive

# pragma omp parallel num_threads (thread_count)

It allows the programmer to specify the number of threads that should execute the following block

Implicit barrier : thread that has completed the block of code will wait for all the other threads in the team to complete the block

Pragmas

- **Pragmas**
  - Special preprocessor instructions.
  - Typically added to a system to allow behaviors that aren't part of the basic C specification.
  - Compilers that don't support the pragmas ignore them

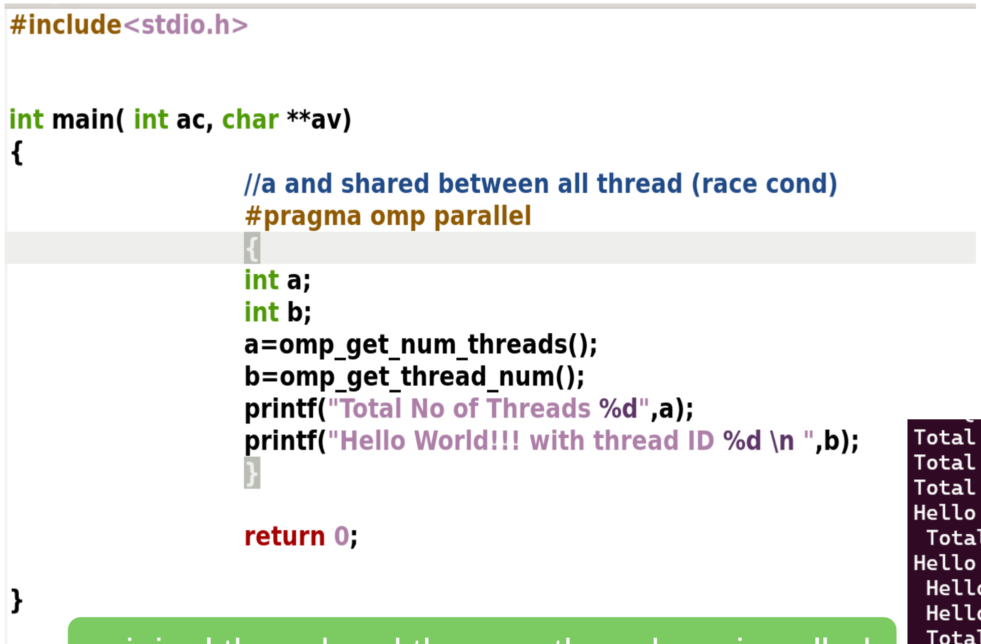
```
#pragma omp parallel
{
  //Parallel region code
}
```

```
#pragma
```

```
gcc −g −Wall −fopenmp −o omp_hello omp_hello . C
/ omp_hello 4
```

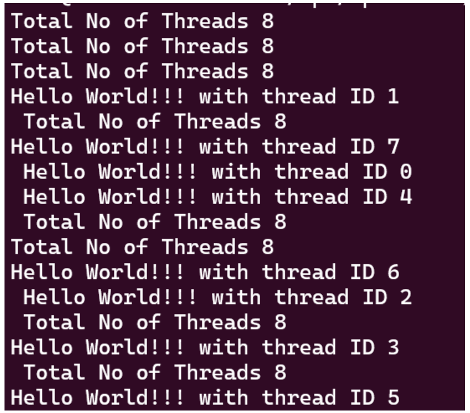
```
export OMP_NUM_THREADS=5
```

Exercise 1:

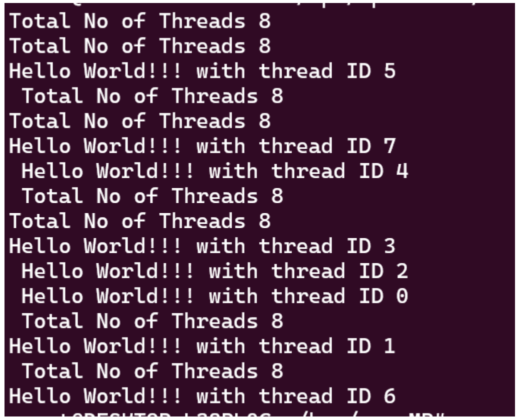


```c
#include<stdio.h>


int main( int ac, char **av)
{
                    //a and shared between all thread (race cond)
                    #pragma omp parallel
                    {
                    int a;
                    int b;
                    a=omp_get_num_threads();
                    b=omp_get_thread_num();
                    printf("Total No of Threads %d",a);
                    printf("Hello World!!! with thread ID %d \n ",b);
                    }

                    return 0;

}
```

original thread and the new threads — is called a team, the original thread is called the master, and the additional threads ar called slaves.



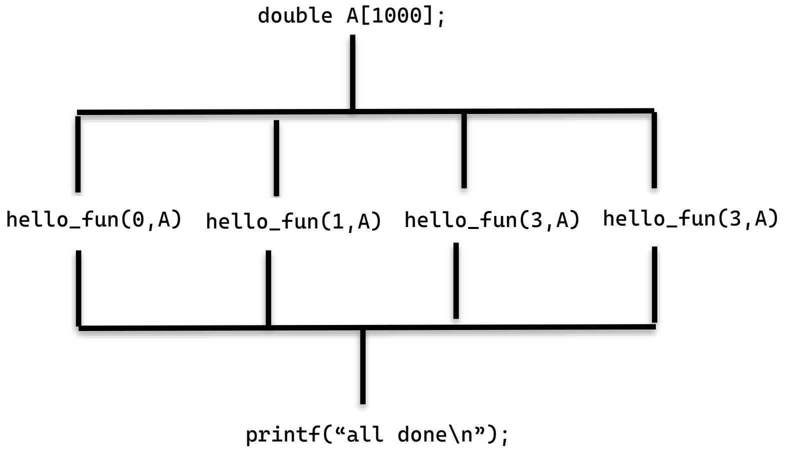
```
Total No of Threads 8
Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 1
 Total No of Threads 8
Hello World!!! with thread ID 7
 Hello World!!! with thread ID 0
 Hello World!!! with thread ID 4
 Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 6
 Hello World!!! with thread ID 2
 Total No of Threads 8
Hello World!!! with thread ID 3
 Total No of Threads 8
Hello World!!! with thread ID 5
```



```
Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 5
 Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 7
 Hello World!!! with thread ID 4
 Total No of Threads 8
Total No of Threads 8
Hello World!!! with thread ID 3
 Hello World!!! with thread ID 2
 Hello World!!! with thread ID 0
 Total No of Threads 8
Hello World!!! with thread ID 1
 Total No of Threads 8
Hello World!!! with thread ID 6
```

Thread Creation: Parallel Regions example

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{ int ID = omp_get_thread_num();
 hello_fun(ID, A);
 }
printf(“all done\n”);
```



1. passing the number of threads at the command line

Parallel construct

```
#pragma omp parallel [clause [, clause] ...]
  structured-block
```

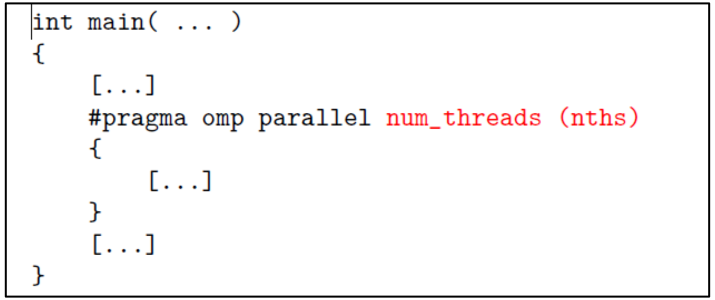
The following clauses apply:

- `if`
- `num_threads`
- `shared, private, firstprivate, default`
- `reduction`
- `copyin`
- `proc_bind`

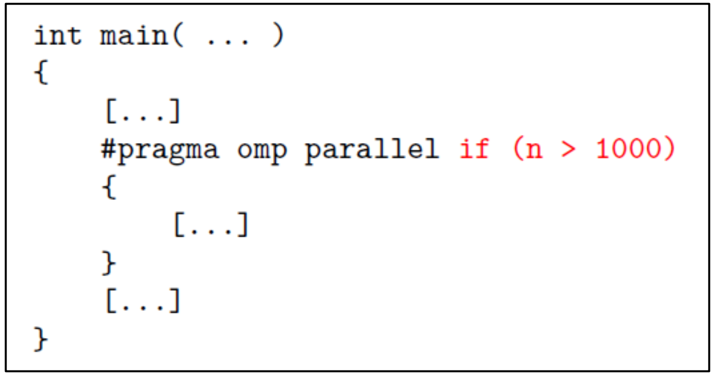
Parallel construct

- num_threads clause
  - Specifies how many threads should execute the region
  - Syntax: “num_threads (*scalar-logical-expression*)”
- if clause
  - Conditional parallel execution
  - Avoid parallelization overhead if little work to be parallelized
  - Syntax: “if (scalar-logical-expression)”
  - If the logical expression evaluates to true: execute in parallel

Example:



```
int main( ... )
{
    [...]
    #pragma omp parallel num_threads (nths)
    {
        [...]
    }
    [...]
}
```



```
int main( ... )
{
    [...]
    #pragma omp parallel if (n > 1000)
    {
        [...]
    }
    [...]
}
```

Thread synchronization

- High level synchronization: –
  - Critical
  - atomic
  - barrier
  - Ordered
- Low level synchronization
  - – flush
  - – locks

Explicit thread synchronization.

- High level synchronization: –
  - Critical
  - atomic
  - barrier
  - Ordered
- Low level synchronization
  - – flush
  - – locks

Synchronization

- Critical : Mutual exclusion: Only one thread at a time can enter a critical region

```
float res;
#pragma omp parallel
{
 float B; int i, id, nthrds;
 id = omp_get_thread_num();
 nthrds = omp_get_num_threads();
 for(i=id;i<niters;i+nthrds)
 {
 B = big_job(i);
 #pragma omp critical
   consume (B, res);
 }
}
```

Synchronization

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
{
double tmp, B;
B = func1();
tmp = func2(B);
#pragma omp atomic
X += func2(B);
}
```

Atomic only protects the read/update of X

- The number of threads in a parallel region is determined by the following factors, in order of precedence.
  - Evaluation of the if clause
  - Setting of the num_threads() clause
  - Use of the omp_set_num_threads() library function
  - Setting of the OMP_NUM_THREAD environment variable

Create threads with the parallel construct

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"
int main()
{
 int nthreads, tid;
 #pragma omp parallel num_threads(4) private(tid)
      {
      tid = omp_get_thread_num();
      printf("Hello world from (%d)\n", tid);
      if(tid == 0)
      {
      nthreads = omp_get_num_threads();
      printf(“number of threads = %d\n”, nthreads);
      }
 }
```

Write openMP program to fork a group of threads with each thread having a private variable to store thread number

```
#include<stdlib.h>
#include <stdio.h>
#include "omp.h"
int main()
{  int nthreads, A[100] , tid; //
   omp_set_num_threads(4);
   #pragma omp parallel private (tid)
     { tid = omp_get_thread_num();
     Fun1(tid, A);
     }
}
```

Thread termination

- By default, worksharing for loops end with an implicOpenit barrier
  - **nowait:** If specified, threads do not synchronize at the end of the parallel loop

```
#pragma omp parallel nowait
```

  - **ordered:** specifies that the iteration of the loop must be executed as they would be in serial program.
  - **collapse**: specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause.
  - The sequential execution of the iteration in all associated loops determines the order of the iterations in the collapsed iteration spac

Barrier: each threadfor s waits till all threads arrive.

#pragma omp **barrier**

No implicit barrier due to nowait.

#pragma omp for **nowait**

OpenMP programming

- Parallel Construct
- Work-Sharing Constructs
  - Loop Construct
  - Sections Construct
  - Single Construct
- Synchronization constructs
  - Barrier Construct
  - Critical Construct
  - Atomic Construct

- Data clauses

  shared, private, Lastprivate, firstprivate, default

- Parallel construct
  - Used to specify the computations that should be executed in parallel.

```
#pragma omp parallel [clause[[,] clause]. . . ]
      ...... code block ......
```

  - The work of the region is replicated for every thread.

Work-Sharing Construct

- Work-sharing is to split up pathways through the code between threads within a team
  - Loop construct
    - #pragma omp for
  - Sections/section constructs
    - #pragma omp sections
  - Single construct
    - #pragma omp single
- Two directives to be studied
  - Do/for: concurrent loop iterations
    - Shares iterations of a loop across the group (data parallelism")
    - Partitions parallel iterations across threads
    - End of for loop: implicit barrier

```
#pragma omp for [clause list]
   /* for loop */
```

Work-Sharing Construct

- The parallel and work-sharing (except single) constructs can be combined
  - Loop construct #pragma omp parallel for
  - Sections construct #pragma omp parallel sections

- Distribute iterations in a parallel region
  - shared clause: All threads can read from and write to a shared variable.
  - private clause: Each thread has a local copy of a private variable

```
#pragma omp parallel for shared(n,a) private(i)
                for (i=0; i <n; i++)
                    a[i] = i + n;
```

  - The mapping between iterations and threads can be controlled by the schedule clause.

.

SPMD vs. worksharing

- A parallel construct by itself creates **a** Single Program Multiple Data.
  - each thread redundantly executes the same code.
- Work-sharing in OpenMP

```
for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

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

```
#pragma omp parallel for
```

OpenMP parallel region work-sharing for construct

```
#pragma omp parallel
#pragma omp for
for(i=0;I<N;i++)
 { a[i] = a[i] + b[i];}
```

Example Using for

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"
int main()
{
 int nthreads, tid;

 omp_set_num_threads(3);
 #pragma omp parallel private(tid)
   {
        int i;
         tid = omp_get_thread_num();
  printf("Hello world from (%d)\n", tid);
  #pragma omp for
      for(i = 0; i <=4; i++)
      {
      printf("Iteration %d by %d\n", i, tid)
      }
    }
}
```

Write openMP code to add two vectors using parallel for

1. • Two work-sharing loops in one parallel region

```
#pragma omp parallel shared(n,a,b) private(i)
{
#pragma omp for
for (i=0; i<n; i++) a[i] = i+1;
// there is an implied barrier
#pragma omp for
for (i=0; i<n; i++) b[i] = 2 * a[i];
}
```

schedule clause

- Defines schedules for OpenMP loops.
- a specification of how iterations of associated loops are divided into contiguous non-empty subsets
  - We call each of the contiguous non-empty subsets a chunk and how these chunks are distributed to threads of the team
- The size of a chunk, denoted as chunk_size must be a positive integer

Schedule Types

- static
- dynamic
- guided
- auto
- runtime

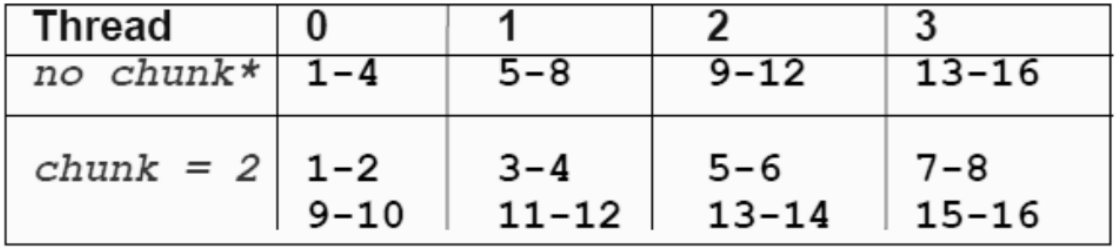
```
#pragma omp parallel for schedule([modifier [modifier]:]kind[,chunk_size])
```

provides a hint for how iterations of the corresponding OpenMP loop should be assigned to threads

schedule clause

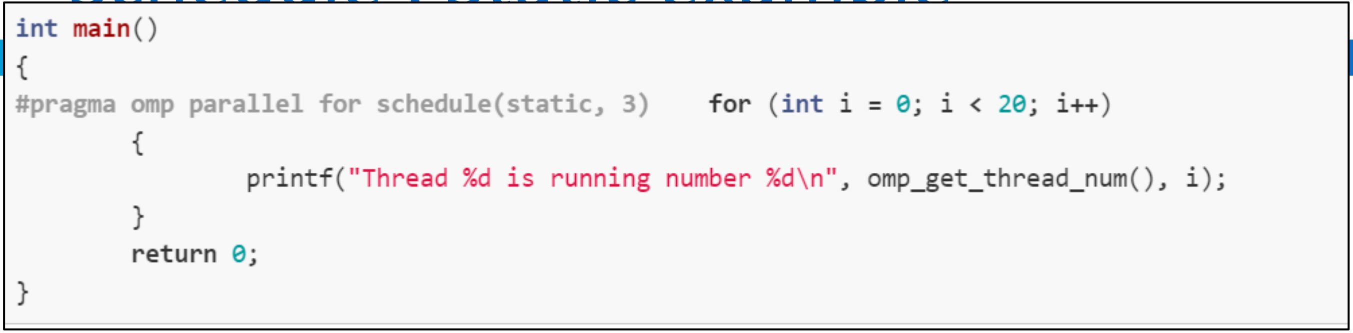
- Static
  - Loop iterations are divided into pieces of size chunk
  - If chunk is not specified, the iteration are evenly) divided contiguously among the threads

Static scheduling • 16 iterations, 4 threads

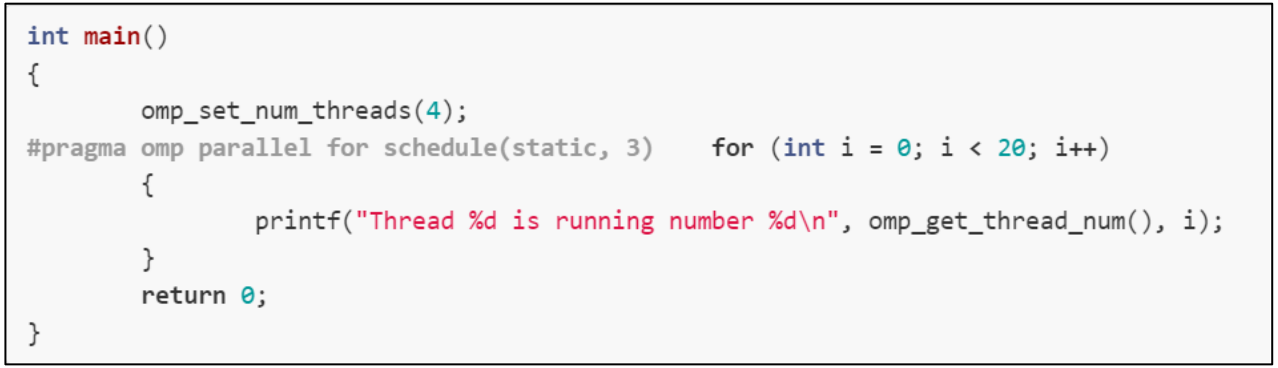


Thread	0	1	2	3
*no chunk**	1-4	5-8	9-12	13-16
chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

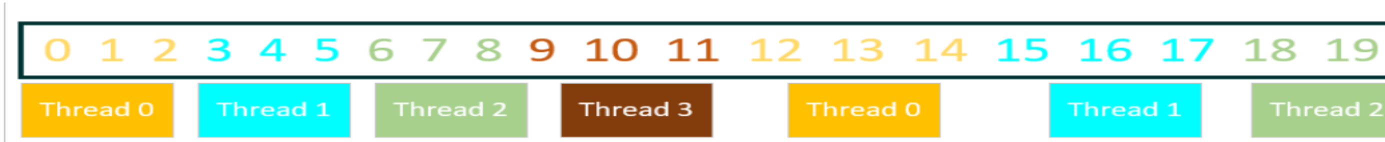
Schedule : Static example



```
int main()
{
#pragma omp parallel for schedule(static, 3)    for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```



```
int main()
{
        omp_set_num_threads(4);
#pragma omp parallel for schedule(static, 3)    for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```



schedule clause

- OpenMP will still split task into iter_size/chunk_size chunks, but distribute trunks to threads dynamically without any specific order

#pragma omp parallel for schedule(dynamic, 1) is equivalent to #pragma omp parallel for schedule(dynamic)

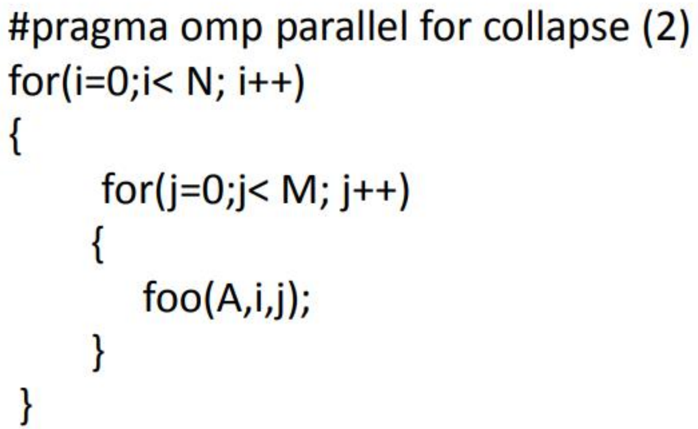
```
#pragma omp parallel for schedule(dynamic,chunk-size)
```

The dynamic scheduling type is appropriate when the iterations require different computational costs

schedule clause

- **Guided :** Chunk size is dynamic while using guided method, the size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Size of each successive chunks is decreasing
  - chunk size = max((num_of_iterations remaining / 2*num_of_threads), n)
- **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 made by the compiler and/or runtime system

- Compiler forms a single loop and then parallelizes this



```
#pragma omp parallel for collapse (2)
for(i=0;i< N; i++)
{
        for(j=0;j< M; j++)
       {
           foo(A,i,j);
       }
 }
```

Work sharing : Section Directive

- One thread executes one section.
  Each section is executed exactly once.

- #pragma omp single (Designated section is executed by single thread only.)
- #pragma omp master : Similar to single, but code block will be executed by the master thread only.
- #pragma omp critical:

```
#pragma omp parallel
 #pragma omp sections
  {
   #pragma omp section
     x_calculation();
   #pragma omp section
     y_calculation();
   #pragma omp section
     z_calculation();
  }
```

- #pragma omp parallel sections {
- #pragma omp section
- funcA();
- #pragma omp section
- funcB(); } /*-- End of parallel region --*/

Data Clauses

- Data clauses

  shared, private, Lastprivate, firstprivate, defaul

- **Shared clause :**
  - **Syntax: shared (item-list).**
  - Each thread can freely read and modify its value
- **Private Clause**
  - Specifies data that will be replicated so that each thread has a local copy
    - **private clause (*item-list*)**

firstprivate and **lastprivate** are just special cases of private. The first one results in bringing in values from the outside context into the parallel region while the second one transfers values from the parallel region to the outside context

Parallel construct

- Shared-memory programming model
- Variables are shared by default.
  - **Shared**:
    - All variables visible upon entry of the construct
    - Static variables
  - Private:
    - Variables declared within a parallel region
    - (Stack) variables in functions called from within a parallel region
- General rules for data-sharing clauses
  - Clauses default, **private, shared, firstprivate** allow changing the default behavior
  - Syntax : keyword and a comma-separated list. For instance: private(a,b).

```
int N = 10;
int main( void )
{
            double array[N];
            #pragma omp parallel
            {
            int i, myid;
            double thread_array[N];
            for ( i = 0; i < N; i++ )
            thread_array[i] = myid * array[i];
            function( thread_array );
            }
}
double function( double arg )
{
            static int cnt;
            double local_array[N];
            [...]
}
```

Firstprivate clause

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

int main (void)
{
    int i = 10;

    #pragma omp parallel private(i)
    {
        printf("thread %d: i = %d\n", omp_get_thread_num(), i);
        i = 1000 + omp_get_thread_num();
    }

    printf("i = %d\n", i);

    return 0;
}
```

```
thread 0: i = 0
thread 3: i = 32717
thread 1: i = 32717
thread 2: i = 1
i = 10

(another run of the same program)

thread 2: i = 1
thread 1: i = 1
thread 0: i = 0
thread 3: i = 32657
i = 10
```

When I is made firstprivate

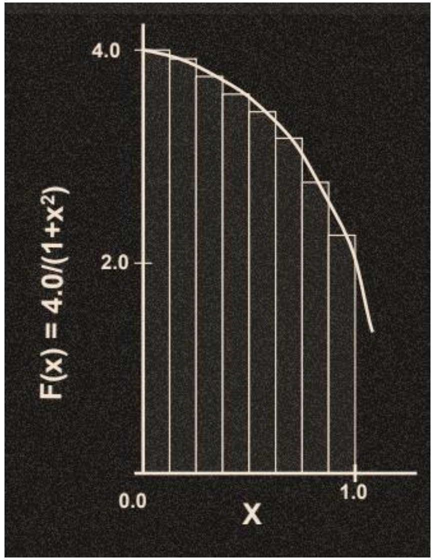
```
thread 2: i = 10
thread 0: i = 10
thread 3: i = 10
thread 1: i = 10
i = 10
```

Lastprivate clause

```
#pragma omp parallel for private(i) lastprivate(a)
for (i=0; i<n; i++) {
      a = i+1;
      printf("Thread %d has a value of a = %d for i = %d\n", omp_get_thread_num(),a,i);
}   /*-- End of parallel for --*/
printf("After parallel for: i = %d , a = %d\n", i, a);
```

Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$



$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Serial PI Program

```
static long num_steps = 100000;
double step;
void main ()
{
     int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++)
            {
             x= (i+0.5)*step; sum = sum + 4.0/(1.0+x*x);
            }
  pi = step * sum;
}
```

Numerical Integration Example

Homework

1. Create a parallel version of the pi program using a parallel construct. ???

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_STEPS 10000
int main( void )
{
int i;
double sum = 0.0, pi, x_i;
double step = 1.0/NUM_STEPS;
for ( i = 0; i < NUM_STEPS; i++ ) {
x_i = (i + 0.5) * step;
sum = sum + 4.0 / (1.0 + x_i * x_i);
}
pi = sum * step;
printf("Pi: %.15e\n", pi);
return 0;
}
```

Hints:

#pragma omp parallel

num_threads(...)

omp_set_num_threads(...)

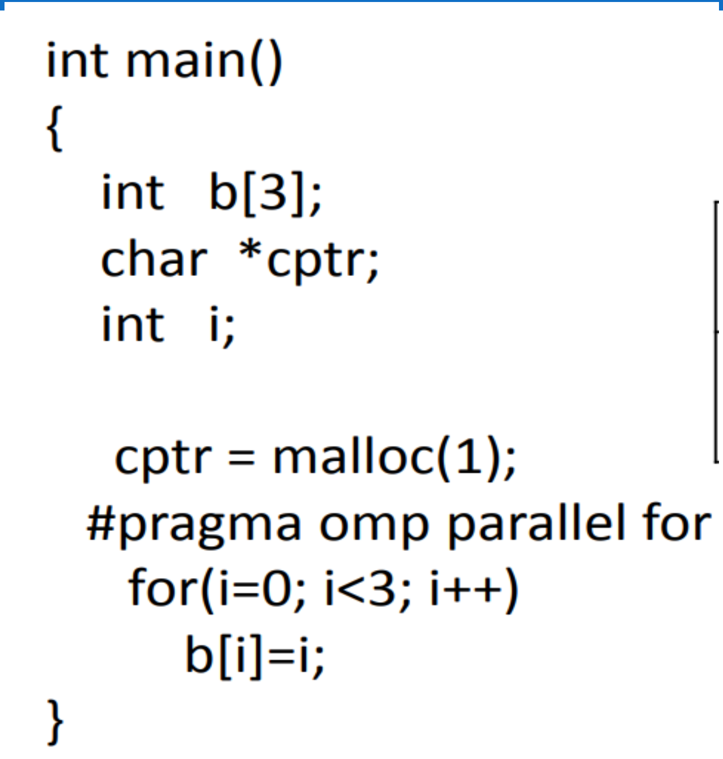
omp_get_num_threads(...)

omp_get_thread_num(...)

Challenges:

split iterations of the loop among threads

create an accumulator for each thread to hold partial sums, which can later be combined to generate the global sum



```
int main()
{
   int  b[3];
   char  *cptr;
   int  i;

    cptr = malloc(1);
  #pragma omp parallel for
    for(i=0; i<3; i++)
        b[i]=i;
}
```

1. List out the private and shared variable in following code

1. We want each thread's private copy of array element x[0] to inherit the value that the shared variable was assigned in the master thread

```
x[0] = 1.0;
for(i=0; i < n; i++){
    for(j=1; j<4; j++)
      x[j]=g(i, x[j-1]);
     answer[i]=x[1]-x[3];
}
```

#pragma omp parallel for private (j) firstprivate (x)

- How do we decide which variables should be shared and which private?
  - Loop indices - private –
  - Loop temporaries - private –
  - Read-only variables - shared –
  - Main arrays - shared

Example

- Write OpenMP program to calculate the average of an array of integers.
  - The program should efficiently distribute the workload among multiple threads, and you must handle critical sections appropriately to avoid data races and ensure accurate results.

Solution:

```
// Initialize the array with values
   for (int i = 0; i < ARRAY_SIZE; ++i) {
       array[i] = i + 1;
   }
```

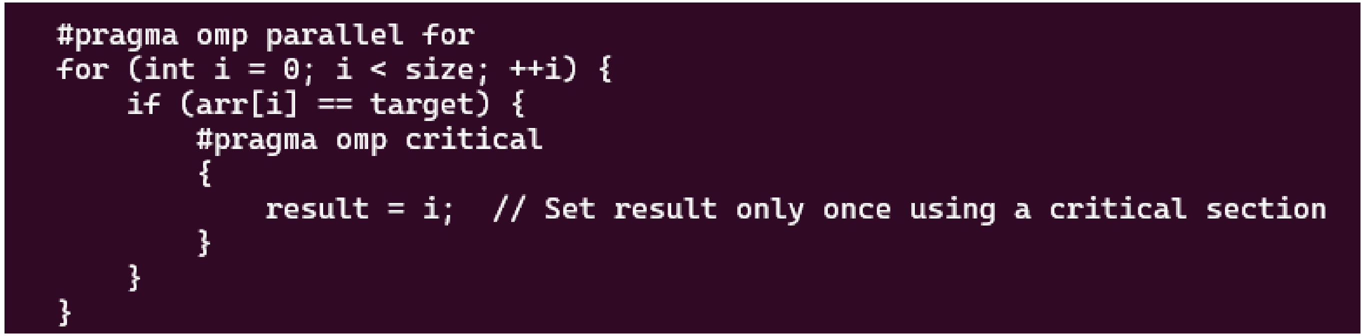
```
#pragma omp parallel num_threads(num_threads) shared(sum)
   {
       double local_sum = 0.0;
```

```
#pragma omp for
       for (int i = 0; i < ARRAY_SIZE; ++i) {
           local_sum += array[i];
       }
```

```
       #pragma omp critical
       {
           sum += local_sum;
       }
   }
```

```
 double average = sum /
ARRAY_SIZE;

    printf("Average: %f\n", average);
```

- Design a parallel program in C using OpenMP to perform a linear search on an array.
- The goal is to
  - efficiently search for a specific target
  - Handle critical sections appropriately to ensure the correct identification



```
#pragma omp parallel for
for (int i = 0; i < size; ++i) {
    if (arr[i] == target) {
        #pragma omp critical
        {
            result = i;  // Set result only once using a critical section
        }
    }
}
```

Reduction

- Reduction (operator: variable list):
  - Combining of local copies of a variable in different threads into a single copy at the master when threads exit.
  - Variables in variable list are implicitly private to threads.
- Operators: +, *, -, &, |, ^, &&, and || – Usage

```
#pragma omp parallel reduction(+: sums) num_threads(4)
{ /* compute local sums in each thread }
```

Example

```
sum = 0;
for (int  i = 0; i < 10; i++)
{
    sum += a[i]
}
```

#pragma omp parallel for shared(sum, a) **reduction(+: sum)**

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for ( int i = 0; i < 10; i++) {
sum += a[i] }
```

```
for( int i = 0; i < numSubdivisions; i++ )
{ double x = A + dx * (float) i;
 double f = Function( x );
sum += f;
 }
```

There are Three Ways to Make the Summing Work Correctly:

```
#pragma omp atomic
sum += f;
#pragma omp critical
sum += f;

#pragma omp parallel for shared(dx),reduction(+:sum)
```

Advantages and Disadvantages openMP

Advantages

- Shared address space provides user friendly programming
- Ease of programming
- Data sharing between threads is fast and uniform (low latency)
- Incremental parallelization of sequential code
- Leaves thread management to compiler
- Directly supported by compiler

Disadvantages

- Internal details are hidden
- Programmer is responsible for specifyingsynchronization, e.g. locks
- Cannot run across distributed memory
- Performance limited by memory architecture
- Lack of scalability between memory and CPUs
- Requires compiler which supports OpenMP

Monte Carlo to estimate PI

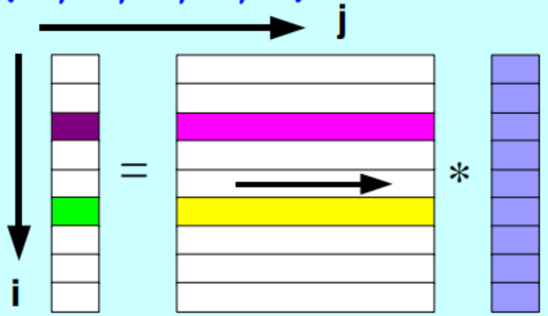
```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"
int main(int argc, char *argv[])
{
 long int i, count; // count points
inside unit circle
 long int samples; // number of samples
 double pi;
 unsigned short xi[3] = {1, 5, 177}; //
random number seed
 double x, y;
 samples = atoi(argv[1]);
 count = 0;
 for(i = 0; i < samples; i++)
 {
 x = erand48(xi);
 y = erand48(xi);
 if(x*x + y*y <= 1.0) count++;
 }
 pi = 4.0*count/samples;
 printf("Estimate of pi: %7.5f\n", pi);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include “omp.h”
main(int argc, char *argv[])
{
 int i, count; /* points inside the unit quarter
circle */
 unsigned short xi[3]; /* random number seed */
 int samples; /*samples Number of points to
Generate*/
 double x,y; /* Coordinates of points */
 double pi; /* Estimate of pi */
 samples = atoi(argv[1]);
 #pragma omp parallel
 {
 xi[0] = 1; /* set up the random seed */
 xi[1] = 1;
 xi[2] = omp_get_thread_num();
 count = 0;
 printf("I am thread %d\n", xi[2]);
 #pragma omp for firstprivate(xi) private(x,y)
reduction(+:count)
 for (i = 0; i < samples; i++)  {
       x = erand48(xi);
       y = erand48(xi);
       if (x*x + y*y <= 1.0) count++; }
 }
 pi = 4.0 * (double)count / (double)samples;

 printf("Count = %d, Samples = %d, Estimate of pi:
%7.5f\n", count, samples, pi);
}
```

Mid-QP

```
for (i=0; i<m; i++)
{
   a[i] = 0.0;
   for (j=0; j<n; j++)
      a[i] += b[i][j]*c[j];
}
```

```
#pragma omp parallel for default(none) \
        private(i,j) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   a[i] = 0.0;
   for (j=0; j<n; j++)
      a[i] += b[i][j]*c[j];
 }
```



Tutorial 0

1. You are given an array `shared_array` of size `ARRAY_SIZE`. Multiple threads need to concurrently increment each element of the array.
2. Design a parallel program using OpenMP to achieve this, ensuring that the increments are performed atomically to avoid race conditions.

Tutorial Exercise 1

- The SAXPY program is to add a scalar multiple of a real vector to another real vector:
- s = a*x + y.
- Provided a serial SAXPY code, parallelize it using OpenMP directives.
- Compare the performance between serial and OpenMP codes.

```
for { i = 0; i < n; i++ }
{
  y[i] = a * x[i] + y[i];
}
```

- Check total wall clock execution time versus thread numbers:
  - export OMP_NUM_THREADS=1
  - time ./saxpy
  - export OMP_NUM_THREADS=2
  - time ./saxpy
  - export OMP_NUM_THREADS=4
  - time ./saxpy
  - export OMP_NUM_THREADS=8
  - time ./saxpy

Tutorial Exercise 2

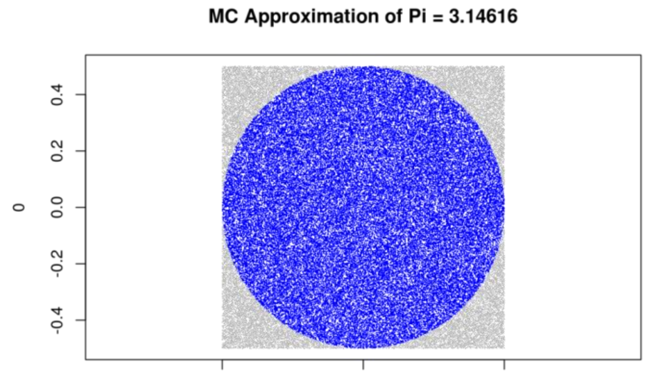
1. **Estimating the value of Pi using Monte Carlo**
   1. computational algorithms that rely on repeated random sampling to obtain numerical results

Estimation of Pi

We know that area of the square is 4r^2 unit sq while that of circle is pi*r^2.

The ratio of these two areas is as follows : pi/4

1. Generate a random point (x, y) inside a square of side 2 centered at the origin.
2. Determine whether the point falls inside the unit circle inscribed in the square by checking whether $x^2 + y^2 <= 1$.
3. Repeat steps 1 and 2 for a large number of points (e.g., 10^7).
4. Calculate the ratio of the number of points that fell inside the circle to the total number of points generated.
5. Multiply the ratio by 4 to estimate the value of pi.



Monte Carlo to estimate PI

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"
int main(int argc, char *argv[])
{
 long int i, count; // count points
inside unit circle
 long int samples; // number of samples
 double pi;
 unsigned short xi[3] = {1, 5, 177}; //
random number seed
 double x, y;
 samples = atoi(argv[1]);
 count = 0;
 for(i = 0; i < samples; i++)
 {
  x = erand48(xi);
  y = erand48(xi);
  if(x*x + y*y <= 1.0) count++;
  }
  pi = 4.0*count/samples;
  printf("Estimate of pi: %7.5f\n", pi);
 }
```

- Provided a serial code, parallelize it using OpenMP directives.
- Compare the performance between serial and OpenMP codes

References

1. http://openmp.org 2.
2. https://computing.llnl.gov/tutorials/openMP 3.
3. http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf
4. Michael J. Quinn, Parallel Programming in C with MPI and OpenMP, Mc Graw Hill, 2003.