



# Open Multi-Processing Shared Memory Parallel Programming in the Multi-Core



# LOGIN

- User name : student
- Pwd: student123
- ftp://172.16.15.100



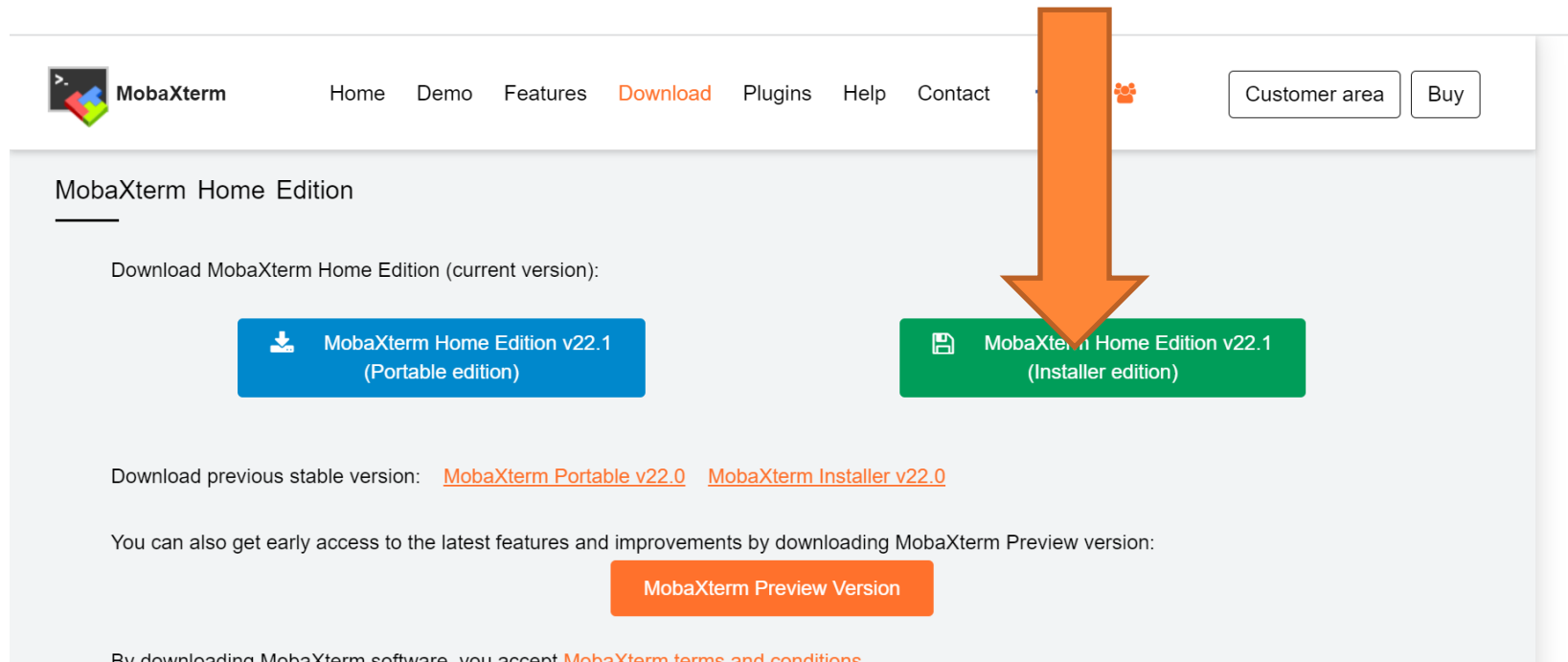
# MOBAXTERM

- MobaXterm is a **ultimate toolbox for remote computing**.
- In a single Windows application, it provides loads of functions that are tailored for **programmers, webmasters, IT administrators** and pretty much all users who need to handle their remote jobs in a more simple fashion.



# MOBAXTERM

- <https://mobaxterm.mobatek.net/download-home-edition.html>



The screenshot shows the MobaXterm website's download page. The navigation bar at the top includes links for Home, Demo, Features, Download (highlighted in orange), Plugins, Help, and Contact. There are also buttons for 'Customer area' and 'Buy'. The main content area is titled 'MobaXterm Home Edition' and instructs users to 'Download MobaXterm Home Edition (current version):'. Two buttons are presented: a blue button for the 'Portable edition' and a green button for the 'Installer edition'. A large orange arrow points directly to the green 'Installer edition' button. Below these, there are links for 'Download previous stable version: MobaXterm Portable v22.0' and 'MobaXterm Installer v22.0'. Further down, a message states 'You can also get early access to the latest features and improvements by downloading MobaXterm Preview version:' followed by an orange button labeled 'MobaXterm Preview Version'. At the bottom, a small line of text reads 'By downloading MobaXterm software, you accept MobaXterm terms and conditions'.


MobaXterm


Home Demo Features **Download** Plugins Help Contact

Customer area Buy

## MobaXterm Home Edition

Download MobaXterm Home Edition (current version):

 MobaXterm Home Edition v22.1  
(Portable edition)

 MobaXterm Home Edition v22.1  
(Installer edition)

Download previous stable version: [MobaXterm Portable v22.0](#) [MobaXterm Installer v22.0](#)

You can also get early access to the latest features and improvements by downloading MobaXterm Preview version:

[MobaXterm Preview Version](#)

By downloading MobaXterm software, you accept [MobaXterm terms and conditions](#)

Download previous

You can also get ea

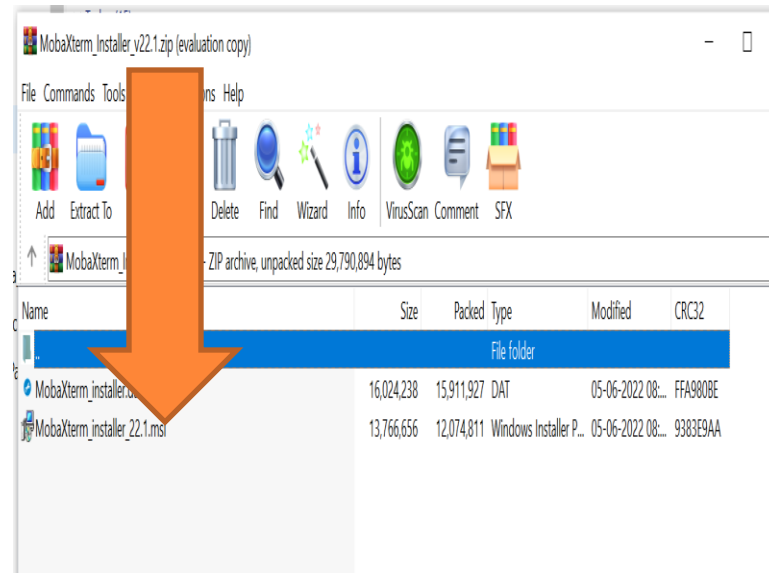
By downloading Mo

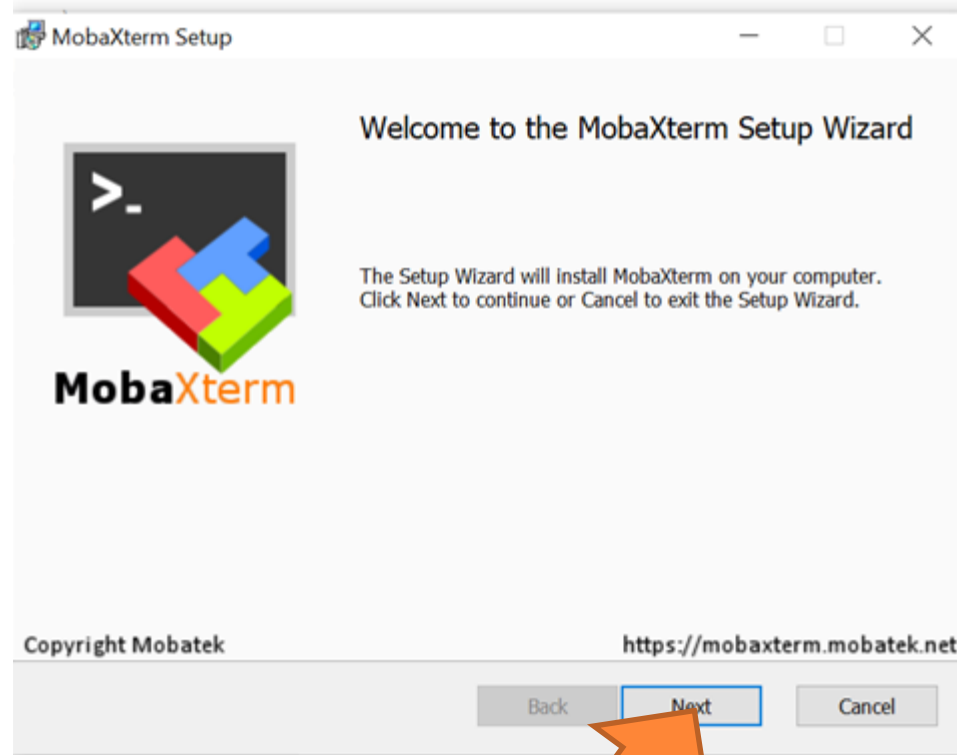
You can download t

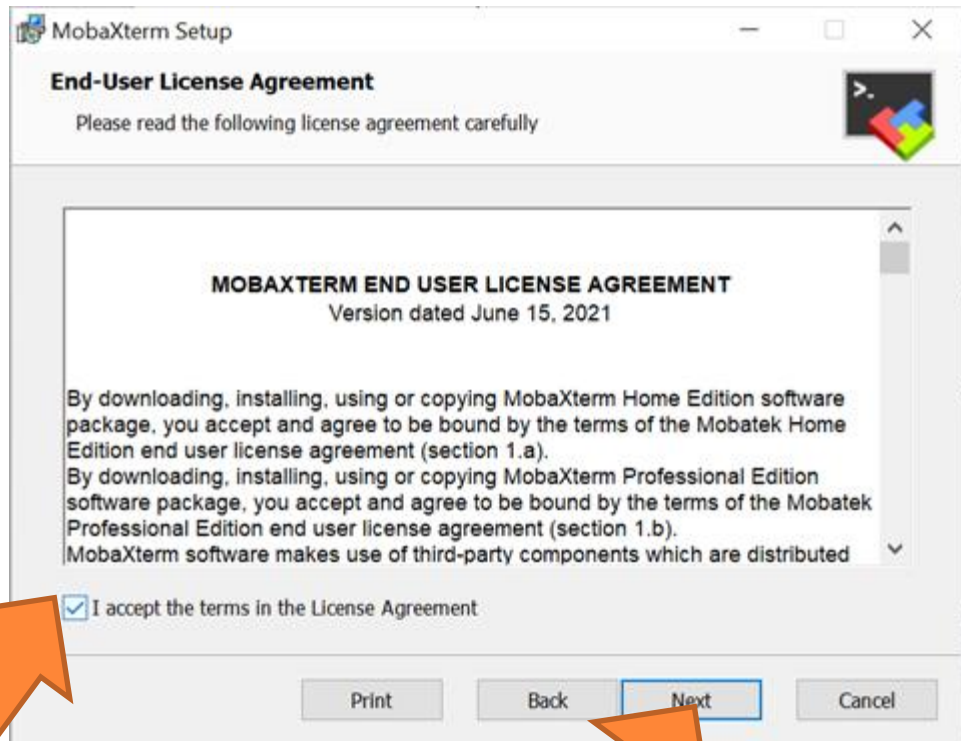


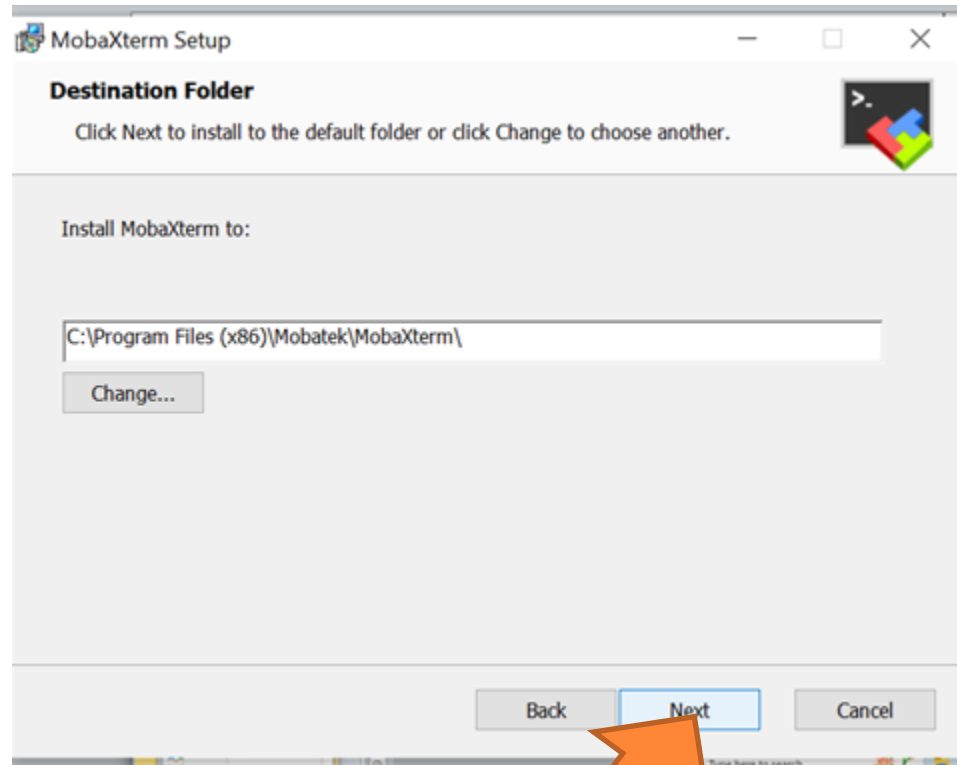
MobaXterm\_Install....zip

18.8/26.7 MB, 5 secs left

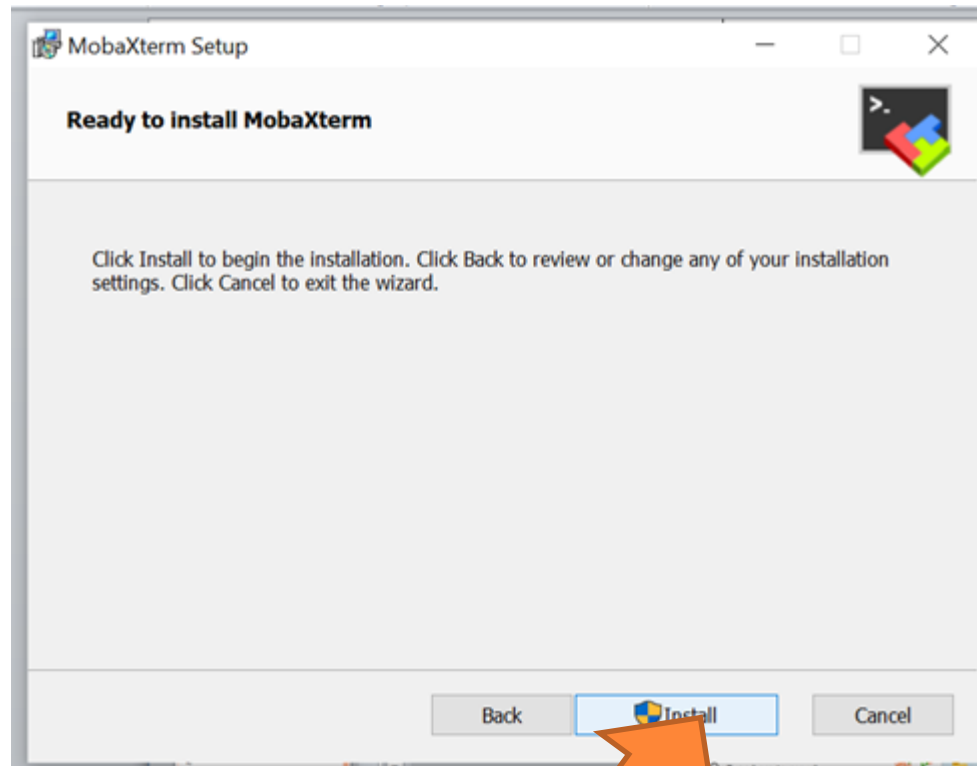


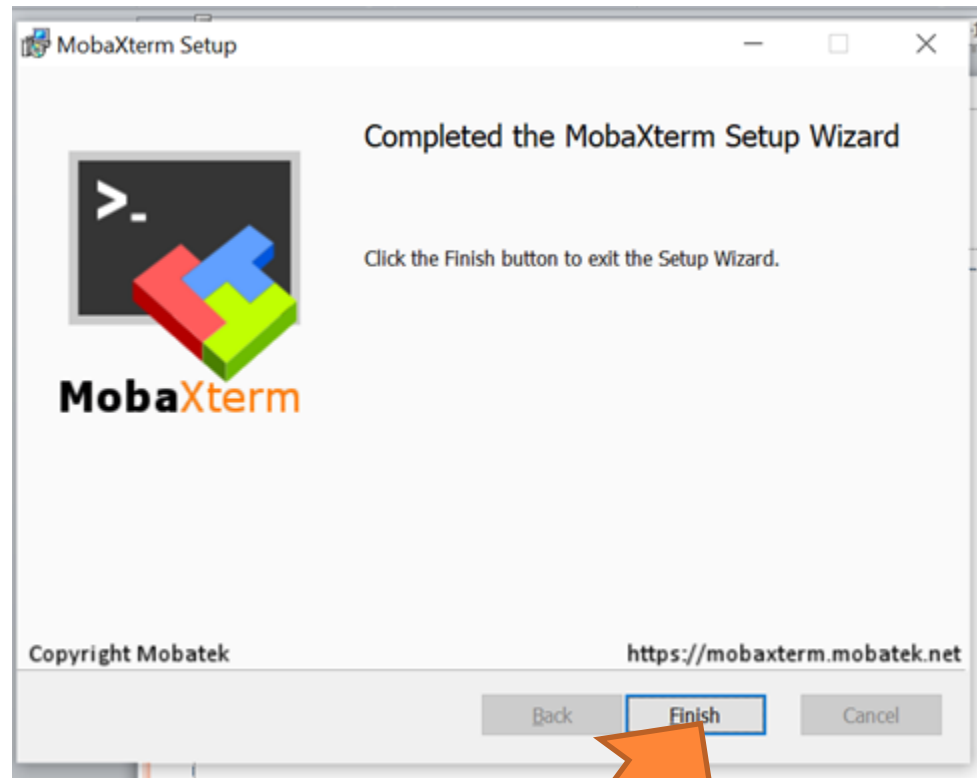


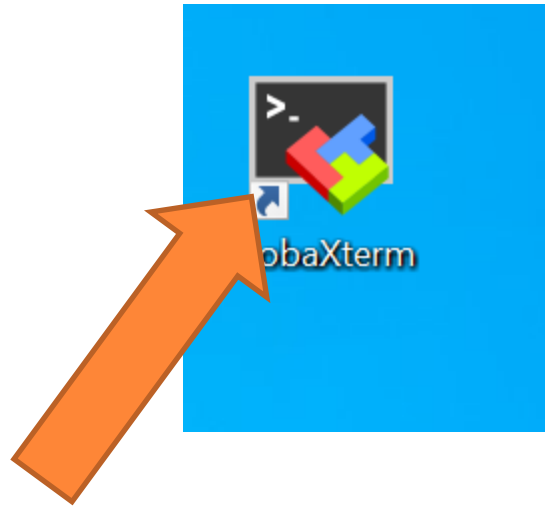


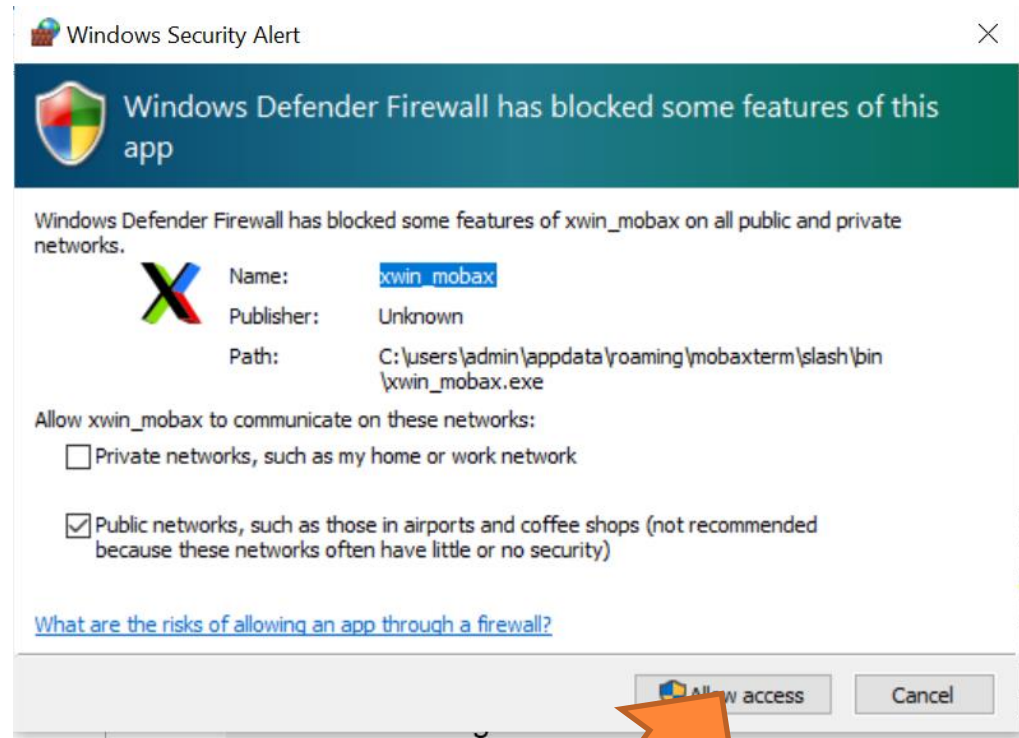
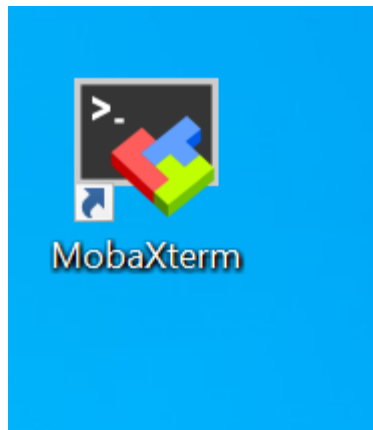




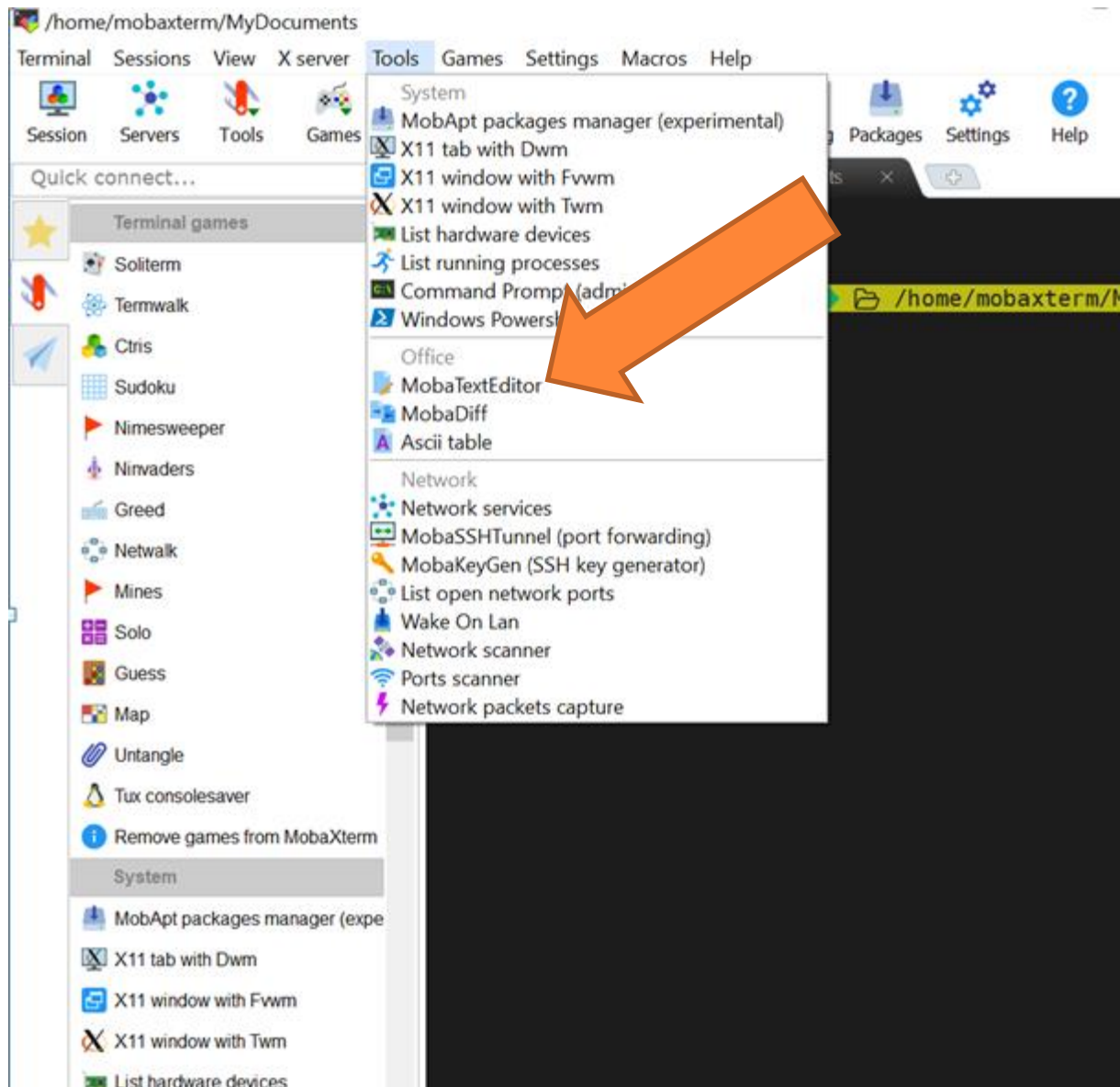


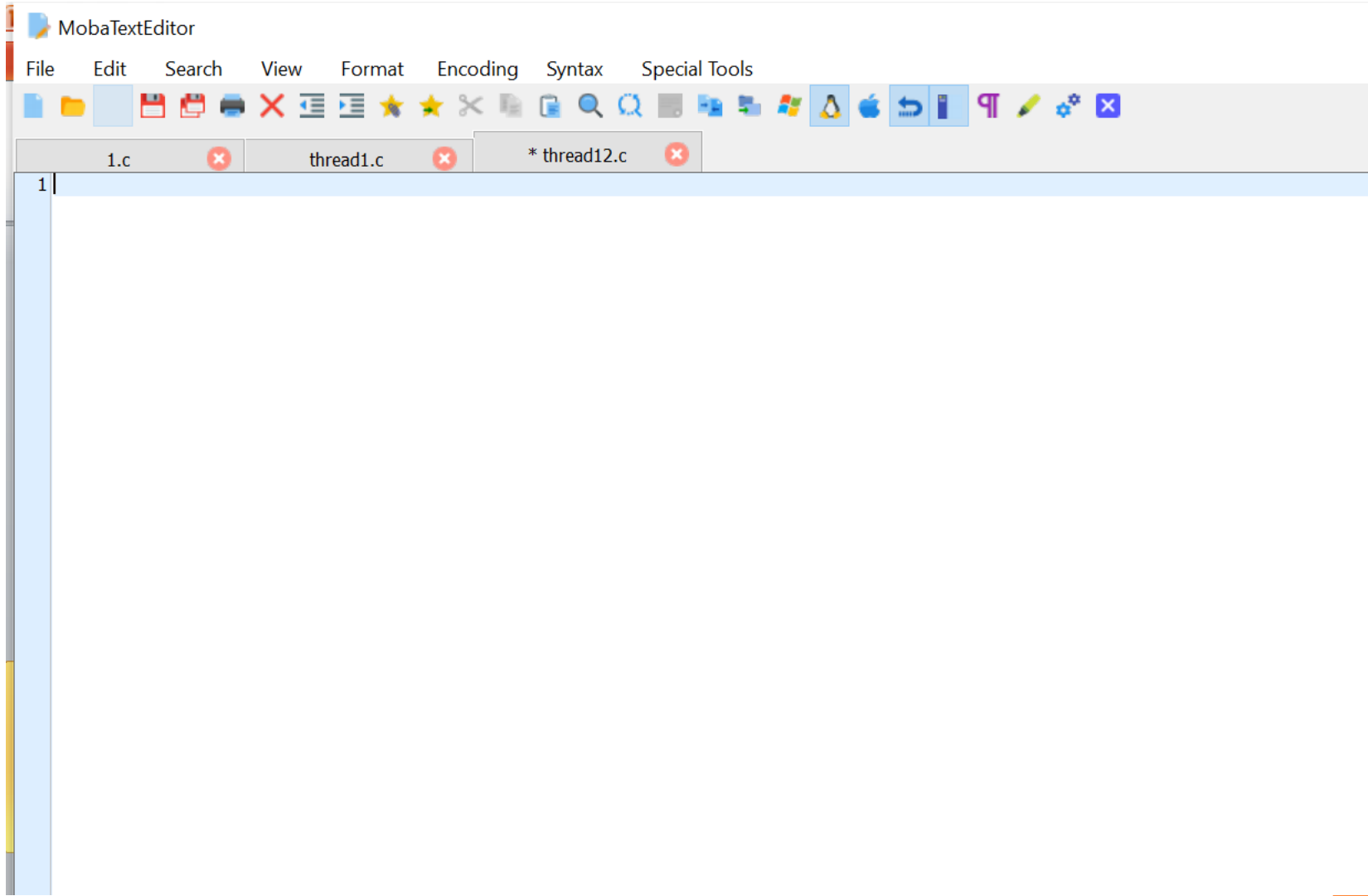






# MOBATEXTEDITOR



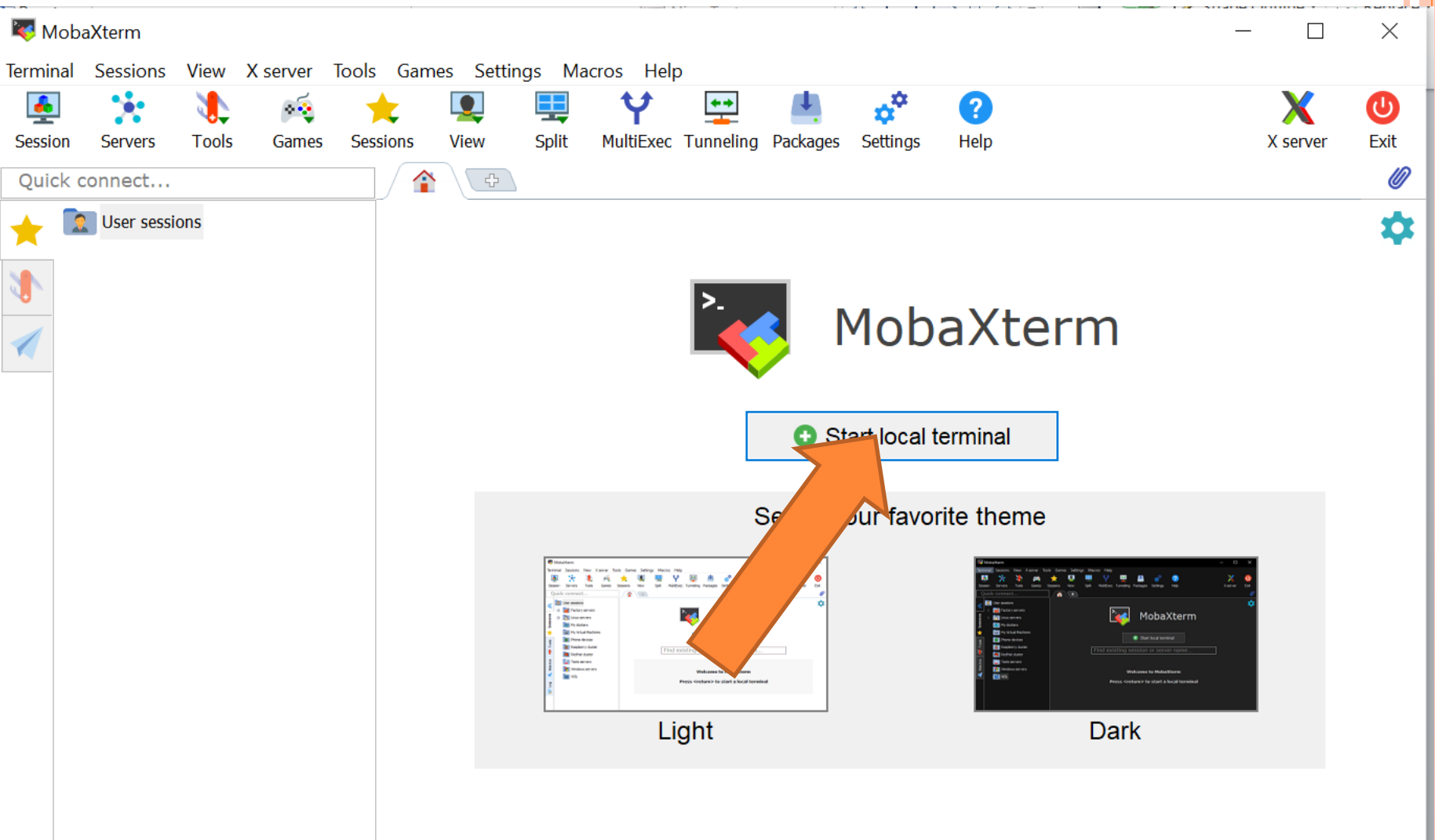


# Simple C programs

- ▶ Program to print “Welcome to VIT”

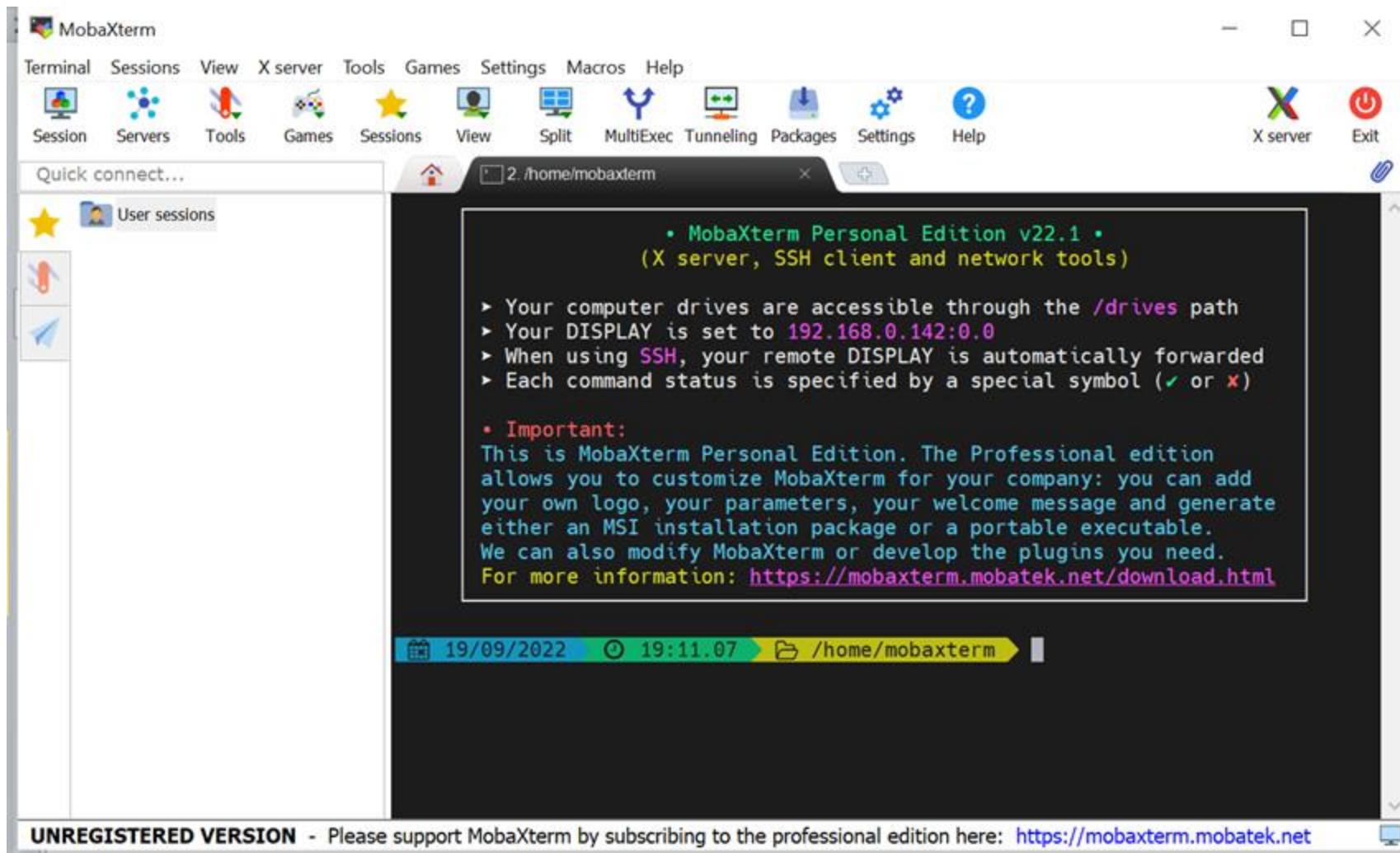
```
#include<stdio.h>
int main()
{
printf(“Welcome to VIT”);
}
```





**UNREGISTERED VERSION** - Please support MobaXterm by subscribing to the professional edition here: <https://mobaxterm.mobatek.net>





- ▶ compile the code using
- ▶ **gcc -o <outputfilename> <c file>**
- ▶ (<outputfilename> – (Dont use .c file name for output file ))
  - **gcc -o hello1 hello.c**
- ▶ **To execute the program**
  - **./hello1**
- ▶ (gcc– GNU Compiler Collection)
- ▶ -o option specifies the file name of the output object program



- `apt -get install gcc-core`



□ gcc -o hello1 hello.c

□ ./hello1



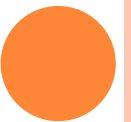
▶Average of five subject



```
#include <stdio.h>
main()
{
int m1,m2,m3,m4,m5,tot,average;
printf("enter marks scored");
scanf("%d%d%d%d%d",&m1,&m2, &m3, &m4,
&m5);
tot=m1+m2+m3+m4+m5;
average= tot/5;
printf("average is %d", average);
}
```

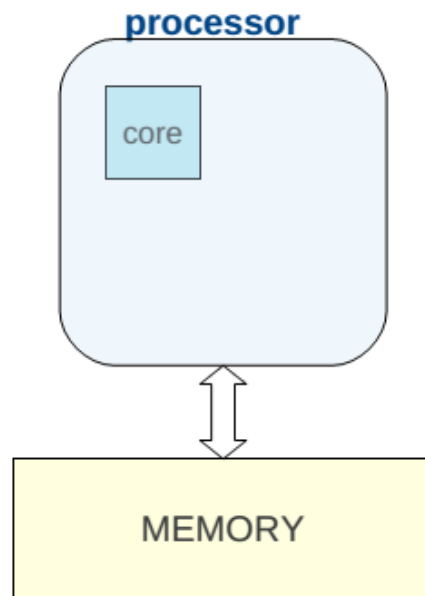


# Introduction

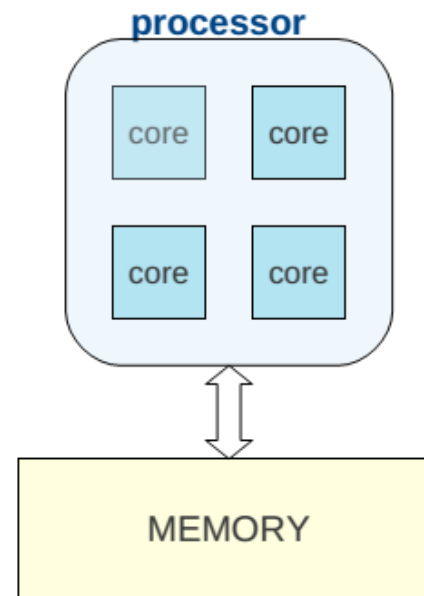


# Basic Architecture

Older processor had only one  
cpu core to execute instructions



Modern processors have 4 or more  
independent cpu cores to execute instructions





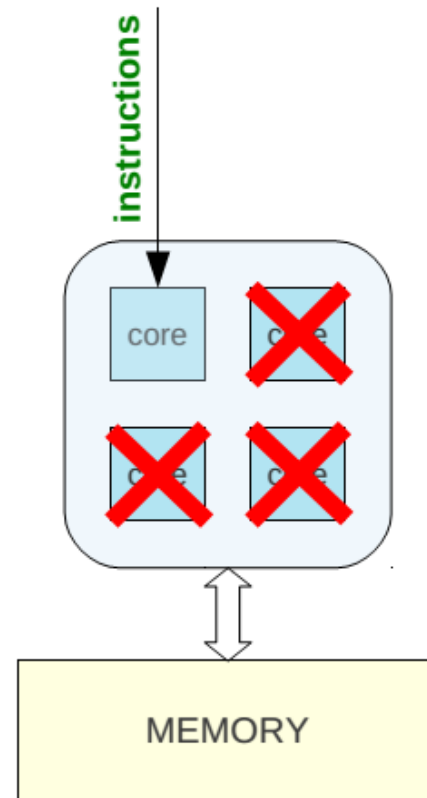
# Sequential Program

## When you run sequential program

- Instructions executed on 1 core
- Other cores are idle

**Waste of available resources. We want all cores to be used to execute program.**

**HOW?**



# OpenMP

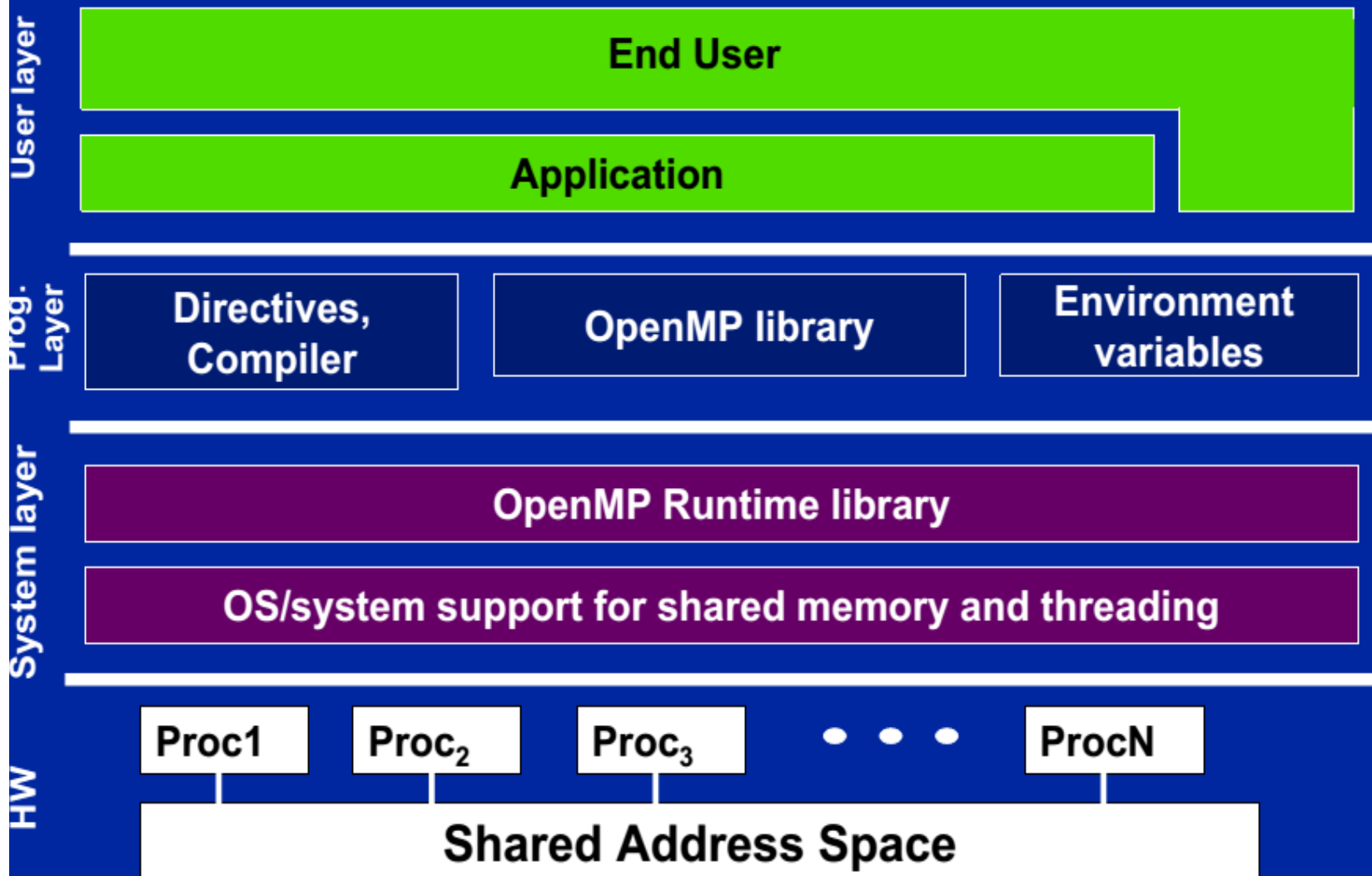
The collection of

- **compiler directives,**
- **library routines, and**
- **Environment variables**

collectively define the specification of the OpenMP Application Program Interface (OpenMP API) for **shared-memory parallelism** in C, C++ and Fortran programs



# OpenMP Basic Defs: Solution Stack



# How is OpenMP typically used?

- ▶ OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

```
void main()
{
    Sequential Program
    int i, k, N=1000;
    double A[N], B[N], C[N];
    for (i=0; i<N; i++) {
        A[i] = B[i] + k*C[i]
    }
}
```

```
#include "omp.h"
void main()
{
    int i, k, N=1000;
    double A[N], B[N], C[N];
    #pragma omp parallel for
    for (i=0; i<N; i++) {
        A[i] = B[i] + k*C[i];
    }
}
```

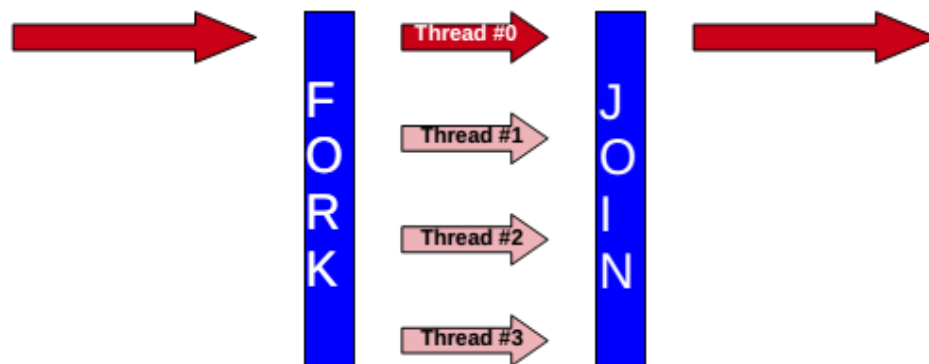
# Fork/Join Model

OpenMP follows the fork/join model:

- ◆ OpenMP programs start with a single thread; the master thread
- ◆ At start of parallel region master creates team of parallel "worker" threads (FORK)
- ◆ Statements in parallel block are executed in parallel by every thread
- ◆ At end of parallel region, all threads synchronize, and join master thread (JOIN)



Implicit barrier. Will discuss  
synchronization later

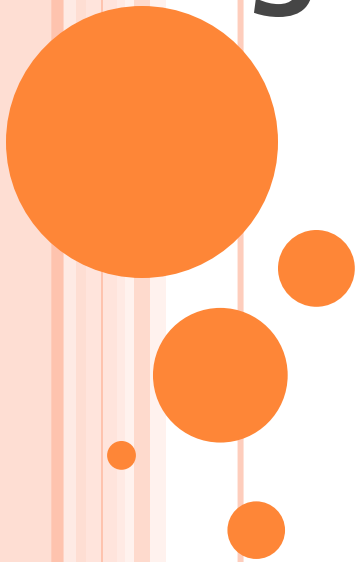


# Basic OpenMP Functions

- ▶
- ▶ 1. **omp\_get\_thread\_num()** – get the thread rank in a parallel region (0– **omp\_get\_num\_threads()** –1)
- ▶
- ▶ 2. **omp\_set\_num\_threads(nthreads)** – set the number of threads used in a parallel region
- ▶
- ▶ 3. **omp\_get \_num\_threads()** – get the number of threads used in a parallel region



# Simple OpenMP Programs



# C Directive for OpenMP

## ►Pragma

- `#pragma omp construct`





# #pragma

- ▶ #pragma directive to control the actions of the compiler in a particular portion of a program without affecting the program as a whole.
- ▶ The pragma directive is used to access compiler-specific preprocessor extensions.
- ▶ A common use of #pragma is the #pragma once directive, which asks the compiler to include a header file only a single time, no matter how many times it has been imported:



Most of the constructs in OpenMP are compiler directives or pragmas.  
For C and C++, the pragmas take the form:

`#pragma omp construct [clause [clause]...]`

*Various **construct** are....*



# OpenMP Constructs

## OpenMP's constructs:

- Parallel Regions
- Worksharing (for/DO, sections, ...)
- Data Environment (shared, private, ...)
- Synchronization (barrier, flush, ...)
- Runtime functions/environment variables (omp\_get\_num\_threads(), ...)



# Constructs

- ▶Parallel
- ▶Work-sharing
  - for directive
  - sections directive
  - single directive
  - parallel work-sharing
    - parallel for
    - parallel sections
- ▶Task
- ▶Synchronization
  - Master directive
  - Critical
  - Barrier
  - Taskwait directive
  - Atomic directive
  - Flush directive
  - Ordered Directive



# Parallel Region Construct

- ▶ A parallel region is a block of code that will be executed by multiple threads.
- ▶ This is the fundamental OpenMP parallel construct.



▶ #include “omp.h”



# Print Thread Number

```
#include "omp.h"
#include "stdio.h"
main()
{
#pragma omp parallel
printf("hello %d \b",omp_get_thread_num());
}
```



2. /home/mobaxterm

17/07/2020 12:44.50 /home/mobaxterm gcc -o hello -fopenmp hello.c  
hello.c:3:1: warning: return type defaults to 'int' [-Wimplicit-int]  
main()  
^~

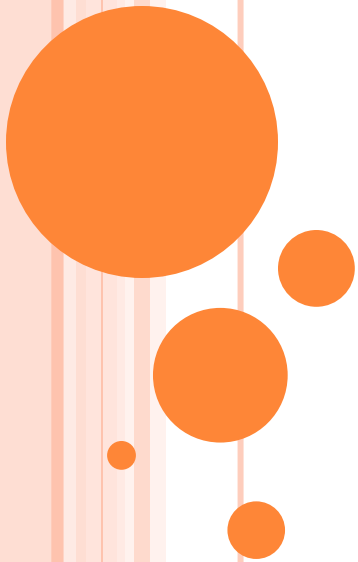
17/07/2020 12:45.26 /home/mobaxterm ls  
Desktop LauncherFolder MyDocuments hello.c hello.exe

17/07/2020 12:45.28 /home/mobaxterm ./hello.exe  
Hello  
Hello  
Hello  
Hello

17/07/2020 12:47.10 /home/mobaxterm



```
omp_get_thread_num());  
Omp_get_num_threads());
```



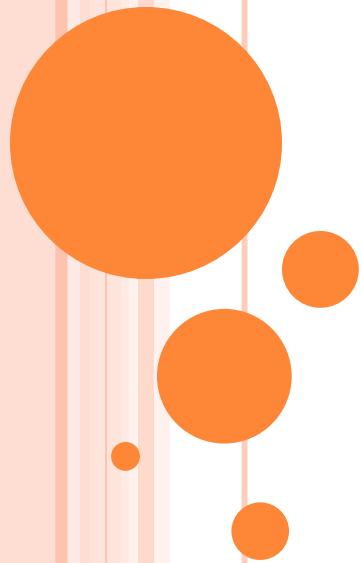
## □ Number of thread Printing

```
#include "omp.h"
#include "stdio.h"
main()
{
#pragma omp parallel
printf("Number of threads is %d", omp_get_num_threads());
}
```



- ▶ **gcc -o <outputfilename> -fopenmp <c file>**
- ▶ (**<outputfilename>** – (Dont use .c file name for output file ))
  - **gcc -o hello1 -fopenmp hello.c**
- ▶ **To execute the program**
  - **./hello1**
- ▶ (gcc– GNU Compiler Collection)
- ▶ **-o** option specifies the file name of the output program





```
gcc -o hello1 -fopenmp  
hello.c
```

- ▶ Hello 0
- ▶ Hello 1
- ▶ Hello 2



# Continued....

```
#pragma omp parallel [clause ...] newline  
    if (scalar_expression)  
    private (list)  
    shared (list)  
    default (shared | none)  
    firstprivate (list)  
    reduction (operator: list)  
    copyin (list)  
    num_threads (integer-expression)  
  
structured_block
```



# Continued...

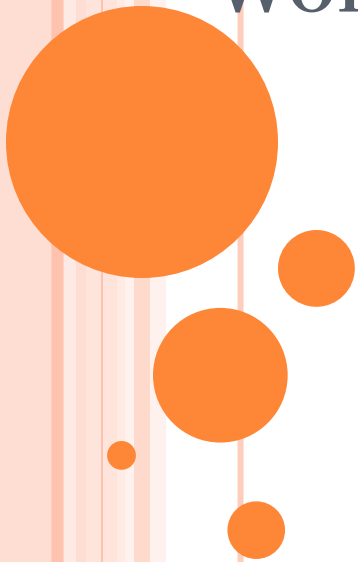
- ▶ When code reaches 'Parallel',
  - It creates a team of threads and becomes the master of the team
  - The master is also a member of the team and has thread number 0.







# PRIVATE , LAST PRIVATE, FIRST PRIVATE WORK CONSTRUCT



# For Loop

```
#pragma omp for[clause clause[]...]
for loop
```

where each clause is one of

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `reduction(operator: list)`
- `ordered`
- `schedule(kind, [chunk_size])`
- `nowait`

# PRIVATE

```
#include <omp.h>
int main(void){
    int i;
    int x;
    x=44;
    #pragma omp parallel for private(x)
    for(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); }
```



- Thread number: 0      x: 0
- Thread number: 0      x: 1
- Thread number: 0      x: 2
- Thread number: 3      x: 9
- Thread number: 3      x: 10
- Thread number: 2      x: 6
- Thread number: 2      x: 7
- Thread number: 2      x: 8
- Thread number: 1      x: 3
- Thread number: 1      x: 4
- Thread number: 1      x: 5
- x is 44

**Notice that x is exactly the value it was before the parallel region.**



- Do you want to keep the last value of  $x$  after the parallel region.???

Replace `private(x)` with `lastprivate(x)` and this is the result:



- Thread number: 3      x: 9
- Thread number: 3      x: 10
- Thread number: 1      x: 3
- Thread number: 1      x: 4
- Thread number: 1      x: 5
- 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: 2      x: 8
- **x is 10**



- Notice that it is 10 and not 8. That is to say, it is the last iteration which is kept, not the last operation..



- `lastprivate(x)` with `firstprivate(x)`.





- Thread number: 3     x: 9
- Thread number: 3     x: 10
- Thread number: 1     x: 3
- Thread number: 1     x: 4
- Thread number: 1     x: 5
- 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: 2     x: 8
- x is 44

- every thread gets its own instance of x and that instance equals 44.



# FIRSTPRIVATE

- `#pragma omp parallel for firstprivate(list)`

Firstprivate is the superset of the functionality provided by private clause. All the members of list are private(local to each thread) and initialized to the value in the preceding serial code.



# LASTPRIVATE

`#pragma omp parallel for lastprivate(list)`

- Like private within the parallel construct - each thread has its own copy.
- The value corresponding to the last iteration of the loop(in serial mode) is saved following the parallel construct.



- Write an OpenMP code using C , create threads and without using temporary variable swap two values .
- Use lastprivate and first private to differentiate the value before swapping and after swapping.





- // Code to swap 'x' and 'y'
- $x = x + y$ ; // x now becomes 15
- $y = x - y$ ; // y becomes 10
- $x = x - y$ ; // x becomes 5
- 
- 



- Write an OpenMP code using C , create threads calculate factorial of “N” values
- Use lastprivate and first private to differentiate the value .
- Initialize a=1;
- 



ORDERED





```
#include <stdio.h>
#include <omp.h>
static float a[1000], b[1000], c[1000];
void test2(int iter)
{
    printf("test2() iteration %d\n", iter);
}
int main( )
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for ordered
        for (i = 0 ; i < 5 ; i++)
            test2(i);
    } }
```

output

**test2() iteration 0**

**test2() iteration 1**

**test2() iteration 2**

**test2() iteration 3**

**test2() iteration 4**

## EXERCISE

- Using OpenMP, write a program to print Odd and even numbers . Print all even numbers in orderly way using appropriate construct and print all odd numbers in unordered fashion.
- Note : get “n” as input



# OpenMP language extensions

Parallel control structures

Work Sharing

Data Environment

Synchronization

Runtime functions, Env. variables

Governs flow of control in the program

**parallel** directive

Distributes work amongst threads

**do/parallel do** and **Section** directives

Scopes variables

**shared** and **private** clauses

Coordinates thread execution

**critical** and **atomic** directives  
**barrier** directives

Runtime environments

**omp\_set\_num\_threads()**  
**omp\_get\_thread\_num()**  
**OMP\_NUM\_THREADS**  
**OMP\_SCHEDULE**

# Work sharing

used to specify how to assign independent work to one or all of the threads.

## i.e Organization of parallel work

- *omp for* or *omp do*: used to split up loop iterations among the threads
- *sections*: assigning consecutive but independent code blocks to different threads
- *single*: specifying a code block that is executed by only one thread, a barrier is implied in the end
- *master*: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

# Work Sharing

Suppose:  $N=100$  and  $\text{num}=4 \rightarrow N/\text{num}=25$

<u>Thread 0</u>	<u>Thread 1</u>	<u>Thread 2</u>	<u>Thread 3</u>
$f=0*25+1 = 1$ $l=1*25 = 25$	$f=1*25+1=26$ $l=2*25 = 50$	$f=2*25+1=51$ $l=3*25 = 75$	$f=3*25+1=76$ $l=4*25 = 100$

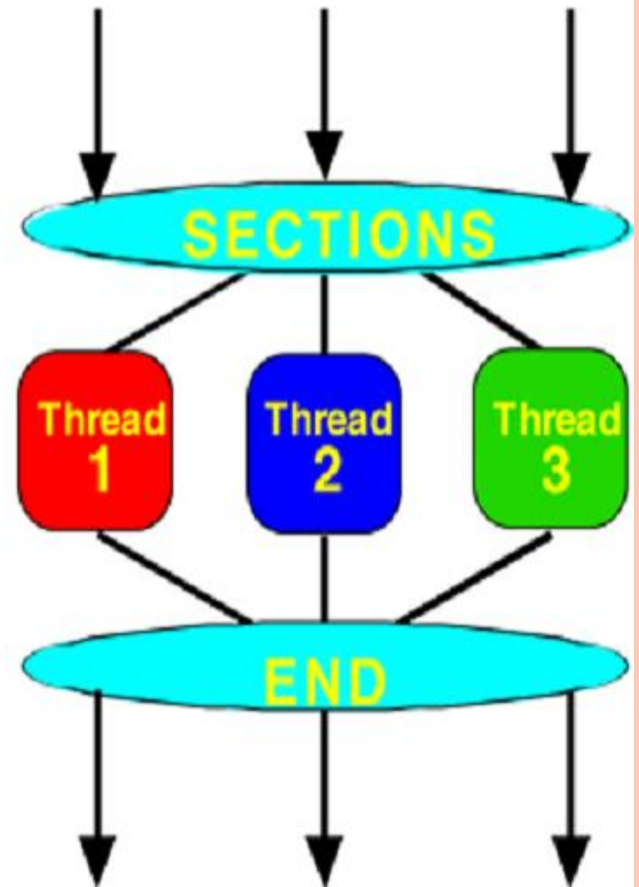
Thread 0 computes elements from index 1 to 25, Thread 1 computes from index 26 to 50, etc.

```
#pragma omp parallel  
{  
#pragma omp for  
for (i=0;i<100;++i) {  
    A(i) = A(i) + B  
}
```



# SECTION DIRECTIVES

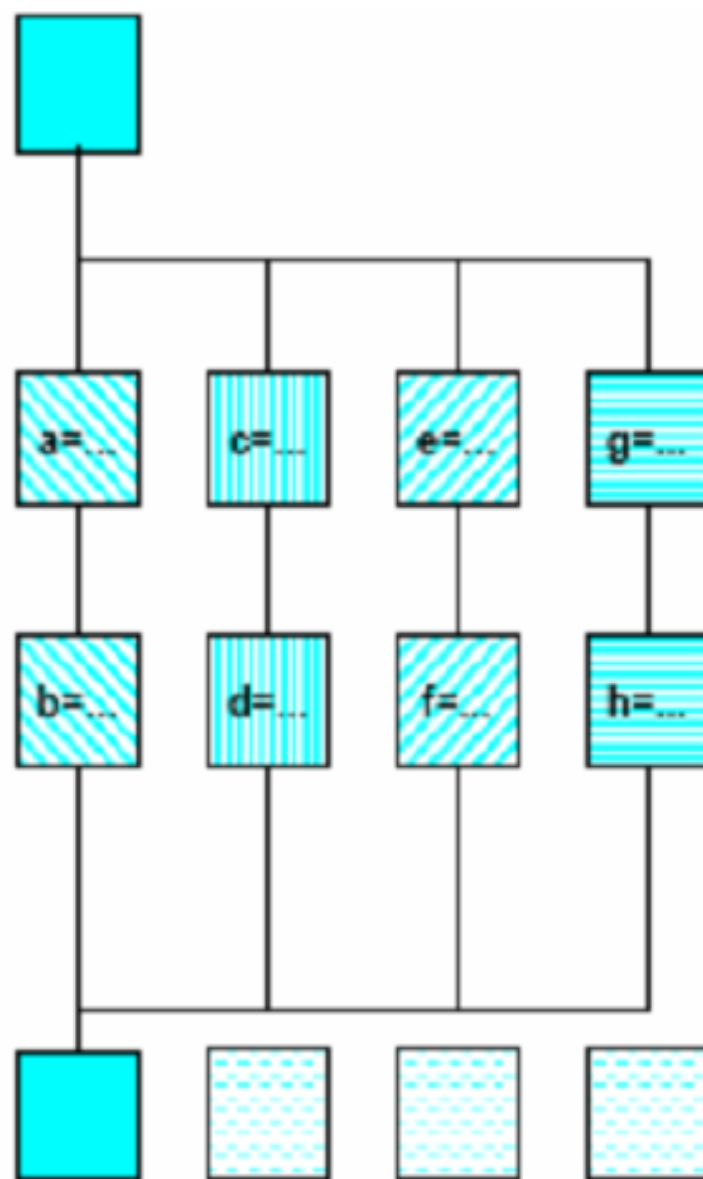
```
#pragma omp parallel
  #pragma omp sections nowait
  {
    #pragma omp section
      thread1_work();
    #pragma omp section
      thread2_work();
    #pragma omp section
      thread3_work();
  }
```



```

#pragma omp parallel
{
#pragma omp sections
    { { a=...;
      b=...; }
      #pragma omp section
      { c=...;
        d=...; }
      #pragma omp section
      { e=...;
        f=...; }
      #pragma omp section
      { g=...;
        h=...; }
    } /*omp end sections*/
} /*omp end parallel*/

```





**schedule** clause specifies how iterations of the loop are divided among the threads of the team.

OpenMP supports four scheduling classes:

- o static
- o dynamic
- o guided
- o runtime



**schedule(static [,chunk])**

Threads get a chunk of data to iterate over

**schedule(dynamic [,chunk])**

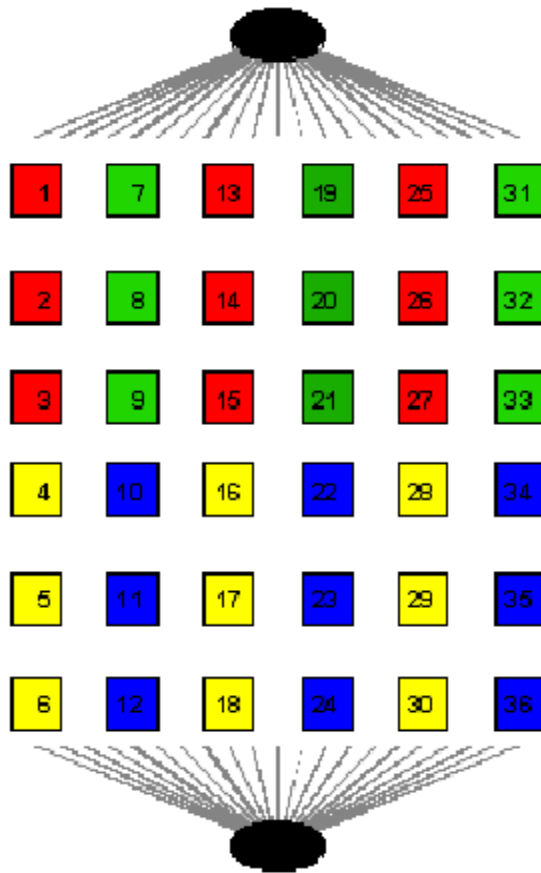
Threads grab chunk iterations off work queue until all work is exhausted

**schedule(guided [,chunk])**

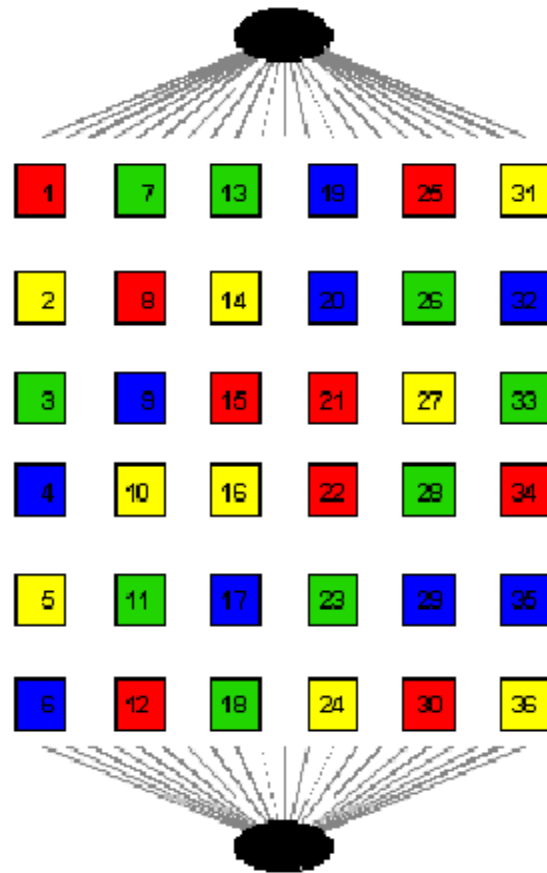
Threads grab large chunk sizes and decreases to specified chunk size as the computation progresses

**schedule(runtime)**

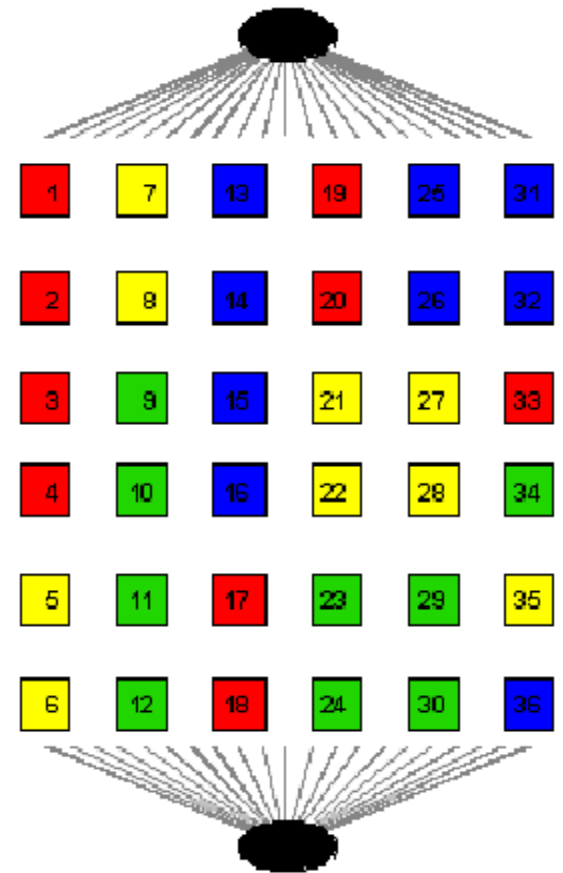
Use the schedule defined at runtime by the OMP\_SCHEDULE environment variable



Static (3)



Dynamic(1)



Guided(1)

# LOOP WORKSHARING CONSTRUCTS: THE SCHEDULE CLAUSE

- `schedule(static [,chunk])`
  - Deal-out blocks of iterations of size “chunk” to each thread



## EXAMPLE - STATIC

```
main()
```

```
{
```

```
int i;
```

```
#pragma omp parallel num_threads(3)
```

```
{
```

```
#pragma omp for schedule(static, 10)
```

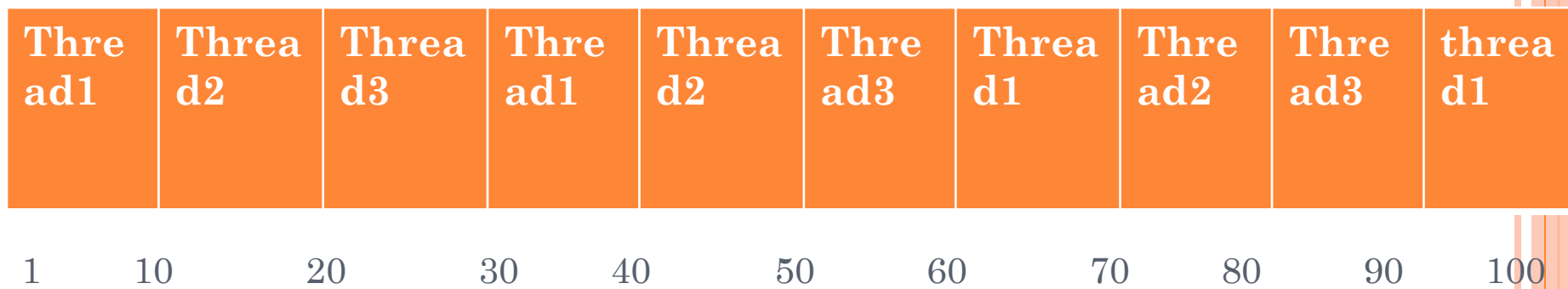
```
for (i=0; i<40;i++)
```

```
printf("iteration %d run by thread  
%d\n",i,omp_get_thread_num());
```

```
}
```

```
}
```





## DYNAMIC

- `schedule(dynamic [,chunk])`
- Each thread grabs “chunk” iterations off a queue until all iterations have been handled.



## EXAMPLE - DYNAMIC

```
main()
{
int i;
#pragma omp parallel num_threads(3)
{
#pragma omp for schedule(dynamic, 10)
for (int i=0; i<60;i++)
printf("iteration %d run by thread
%d\n",i,omp_get_thread_num());
}
}
```





## CONTD..

- In **dynamic scheduling**, using a "chunksize" of 10:
  - thread 1 is assigned to do iterations 1 to 10.
  - thread 2 is assigned to do iterations 11 to 20.
  - thread 3 is assigned to do iterations 21 to 30.
- The next chunk is iterations 31 to 40, and will be assigned to whichever thread finishes its current work first, and so on until all work is completed.



## GUIDED

- `schedule(guided[,chunk])`
- Threads dynamically grab blocks of iterations.  
The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.



## EXAMPLE - GUIDED

```
main()
{
int i;
#pragma omp parallel num_threads(3)
{
#pragma omp for schedule(guided, 10)
for (int i=0; i<100;i++)
Printf("iteration %d run by thread
%d",i,omp_get_thread_num());
}
}
```



## EXERCISE

- For rainfall statistics recording, four sensors placed strategically across the city measures the rainfall in mm and store the values (one for each day for each sensor) in four different file. Calculate the maximum rainfall recorded in a particular day across the city.



- A state government have fixed the electricity tariff as shown in the following table. They would like to print the receipt for the consumer while paying the bill.
- Write a C program using OpenMP to print the bill containing the energy consumed, fixed charge, energy charge and total amount(fixed charge and consumed energy charge) to be paid by the customer. Use static schedule to print unit less than 200, dynamic schedule till 500 and guided for above 500.



S. No	Consumption Slab in Unit	Tariff Rate payable by Consumer	
		Fixed Charge per service	Energy Charge / Unit
1	0-100	Rs. 20	Rs. 1
2	0-200	Rs. 20	Rs. 1.5
3	For consumers who consume 201 units to 500 units		
	0-200	Rs. 30	Rs. 2
	201-500		Rs. 3
4	For consumers who consume 501 units and above		
	0-200	Rs. 40	Rs. 3
	201-500		Rs. 4
	501 and above		Rs. 5.75

# NO WAIT CLAUSE

- Overrides the barrier implicit in a directive.
- For example, if we have two parallel for loops

```
#pragma omp parallel for  
for(i=0;i<n;i++) { }
```

```
#pragma omp for  
for(i=0;i<n;i++) { }
```

There will be a barrier at the end of the for loop one thread will be waiting till all the other threads have finished working



**CONTD...**  
**#pragma omp for schedule(static, 3) num\_threads(3)**

**In static allocation, one thread will wait till other threads finish before next chunk is allotted. No overrides this barrier**





# CONTD..

- Applies to following directives
  - For
  - Sections
  - single



## EXAMPLE

```
#include"stdio.h"
#include"omp.h"
#pragma omp parallel {
#pragma omp for nowait
    for (i = 0; i < size; i++)
        b[i] = a[i] * a[i];
#pragma omp for nowait
    for (i = 0; i < size; i++)
        c[i] = a[i]/2;
}
```



- #include"stdio.h"
- #include"omp.h"
- main()
- {int size,a[5],b[5],c[5],i;
- scanf("%d",&size);
- #pragma omp parallel
- {     #pragma omp for nowait
- for (i = 0; i < size; i++)
- {scanf("%d",&a[i]);
- b[i] = a[i] \* a[i];
- printf("thread is %d",b[i]);
- }     #pragma omp for nowait
- for (i = 0; i < size; i++)
- {scanf("%d",&a[i]);
- c[i] = a[i]/2;
- printf("%d",c[i]);
- } } }



- Write an OPENMP program to calculate sum of “N” natural Numbers .
- $1+2+3+4+\dots N$
- **Using critical construct**  
**#pragma omp critical**



```
main()
```

```
{ int i,sum=0;
```

```
scanf("%d",&n);
```

```
#pragma omp critical
```

```
for(i=0;i<n;i++)
```

```
{
```

```
sum=sum+i;
```

```
}
```

```
printf("sum is %d",sum);
```

```
}
```





# LOCKS

## Protect resources with locks.

- lock variable must have type **omp\_lock\_t**.
- The **omp\_init\_lock** function initializes a simple lock.
- The **omp\_destroy\_lock** function removes a simple lock.
- The **omp\_set\_lock** function waits until a simple lock is available.
- The **omp\_unset\_lock** function releases a simple lock.



```
main()
```

```
EXAMPLE  
{int id,i;
```

```
omp_lock_t lck;
```

```
omp_init_lock(&lck);
```

```
#pragma omp parallel num_threads(3)
```

```
id= omp_get_thread_num();
```

```
#pragma omp for
```

```
for(i=0;i<7;i++)
```

```
{ omp_set_lock(&lck);
```

```
printf("\n%d  %d",id,i);
```

```
omp_unset_lock(&lck);
```

```
} } omp_destroy_lock(&lck);}
```





## EXERCISE

- A contest is being held for JUGAAD. Students can register, aftermath if they want, they can unregister. Registered students (registration numbers:9,3,2...)is stored in an array. Only one student can register or unregister at a time. But, they can view registered list without any constraint. Design a parallel program with the help of locks.



## EXAMPLE

```
#pragma omp parallel {  
#pragma omp for nowait  
  for (i = 0; i < size; i++)  
    b[i] = a[i] * a[i];  
#pragma omp for nowait  
  for (i = 0; i < size; i++)  
    c[i] = a[i]/2;  
}
```



# SINGLE DIRECTIVE

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
- May be useful when dealing with sections of code that are not thread safe (such as I/O)



# CONTD...

- Syntax

`#pragma omp single`



# MASTER CONSTRUCT

- 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



# CONTD..

- Syntax

`#pragma omp master`



# IF CLAUSE

- `#pragma omp parallel for if(n>5)`  
    `for(i=0;i<n;i++)`  
    `{`  
        `}`



# EXAMPLE

```
#include <omp.h>
#define N 1000
main ()
{ int i;
  float a[N], b[N], c[N], d[N];
  /* Some initializations */
  for (i=0; i < N; i++)
  {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
  }
  #pragma omp parallel shared(a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
      for (i=0; i < N; i++) c[i] = a[i] + b[i];
      #pragma omp section
      for (i=0; i < N; i++) d[i] = a[i] * b[i];
    } /* end of sections */
  } /* end of parallel section */
}
```





# OMP BARRIER- DIRECTIVE

**#pragma omp barrier**

- OMP BARRIER will enforce every thread to wait at the barrier until all threads have reached the barrier.
- OMP BARRIER is probably the most well known synchronization mechanism; explicitly or implicitly.



```
#include <stdio.h>
#include <omp.h>
int main(){
int x;
x = 2;
#pragma omp parallel num_threads(2) shared(x)
{
if (omp_get_thread_num() == 0) {
x = 5;
} else {
/* Print 1: the following read of x has a race */
printf("1: Thread# %d: x = %d\n",
omp_get_thread_num(),x );
}
}
```



```
#pragma omp barrier
if (omp_get_thread_num() == 0) {
/* Print 2 */
printf("2: Thread# %d: x = %d\n",
      omp_get_thread_num(),x );
} else {
/* Print 3 */
printf("3: Thread# %d: x = %d\n",
      omp_get_thread_num(),x );
}
}
return 0;
}
```



## EXERCISE

- A contest is being held for JUGAAD. Students can register, aftermath if they want, they can unregister. Registered students (registration numbers:9,3,2...)is stored in an array. Only one student can register or unregister at a time. But, they can view registered list without any constraint. Design a parallel program with the help of locks.



- Sample program for parallel – section, nowait, shared variable



# EXAMPLE

```
#include <omp.h>
#define N 10
main ()
{ int i;
  float a[N], b[N], c[N], d[N];
  /* Some initializations */
  for (i=0; i < N; i++)
  {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
  }
  #pragma omp parallel shared(a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
      for (i=0; i < N; i++) c[i] = a[i] + b[i];
      #pragma omp section
      for (i=0; i < N; i++) d[i] = a[i] * b[i];
    } /* end of sections */
  } /* end of parallel section */
}
```



# OMP BARRIER- DIRECTIVE

**#pragma omp barrier**

- OMP BARRIER will enforce every thread to wait at the barrier until all threads have reached the barrier.
- OMP BARRIER is probably the most well known synchronization mechanism; explicitly or implicitly.



```
#include <stdio.h>
#include <omp.h>
int main(){
int x;
x = 2;
#pragma omp parallel num_threads(2) shared(x)
{
if (omp_get_thread_num() == 0) {
x = 5;
} else {
/* Print 1: the following read of x has a race */
printf("1: Thread# %d: x = %d\n",
omp_get_thread_num(),x );
}
}
```





```
#pragma omp barrier
if (omp_get_thread_num() == 0) {
/* Print 2 */
printf("2: Thread# %d: x = %d\n",
      omp_get_thread_num(),x );
} else {
/* Print 3 */
printf("3: Thread# %d: x = %d\n",
      omp_get_thread_num(),x );
} } return 0; }
```



# BARRIER

- Addition of two array
- Enter value of n
- Using one for loop Take two array values

omp parallel for

```
for(i=0;i<n;i++)
```

```
{
```

```
c[i]=a[i]+b[i];
```

```
printf("\n c is %d",c[i]);
```

```
}
```



```
#pragma omp barrier
{
for(i=0;i<n;i++)
{
sum=sum+c[i];
}
printf(“%d”,sum);
}
}
```



# BARRIER

- Addition of two array
- Enter value of n
- Take two for loop to take two array values

omp parallel for

for(i=0;i<n;i++)

{

if(omp\_get\_thread\_num()==0)

for(k=0;k<50;k++)

c[i]=a[i]+b[i];

printf("\n waiting for  
syn...%d",c[i]);

}

```
#pragma omp  
barrier
```

```
{
```

```
for(j=0;j<n;j++)
```

```
{
```

```
sum=sum+c[j];
```

```
}
```

```
printf("%d",sum  
);
```

```
}
```

```
}
```



# OPENMP

- ENVIRONMENT – set thread, get thread, num thread  
....
- For, parallel, time , private and shared variable(first private, last private) , nowait, barrier , critical , scheduling (static, dynamic , guided), locks (variable), work sharing, section , atomic,



```
#pragma omp barrier
{
for(j=0;j<n;j++)
{
sum=sum+c[j];
}
printf(“%d”,sum);
}
}
```



# ATOMIC

This Directive is very similar to the  
OMP CRITICAL directive on the previous slide.

Difference is that

OMP ATOMIC is only used for the update of a  
memory location.

Sometimes OMP ATOMIC is also referred to as a  
mini critical section.



# ATOMIC SECTION

```
main()
```

```
{
```

```
double area, x;
```

```
int i, n;
```

```
area = 0.0;
```

```
# pragma omp parallel for private(x)
```

```
for (i=0; i<n; i++)
```

```
{
```

```
x=(i+0.5) / n;
```

```
#pragma omp atomic {
```

```
area += 4.0 / (1.0 +x*x);
```

```
printf(“%d\t%d\tArea:%f\n”,sched_getcpu(),omp_get_thread_n  
um(),area);
```

```
}
```

```
}
```

```
pi=area/n;
```

```
}
```





BARRIER



# REDUCTION

- A Local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
- Compiler finds standard reduction expressions containing “op” and uses them to update the local copy.
- Local copies are reduced into a single value and combined with the original global value



## “REDUCTION” CLAUSE EXAMPLE

- `#include <omp.h>`
- `main () {`
  - `int i, n, chunk;`
  - `float a[100], b[100], result;`
  - `n = 100 ; chunk = 10 ; result = 0.0 ;`
  - `for (i=0; i < n; i++) {`
    - `a[i] = i * 1.0 ; b[i] = i * 2.0;`
  - `}`
  - `#pragma omp parallel for default(shared)`  
`private(i) schedule(static,chunk)`  
`reduction(+:result)`
  - `for (i=0; i < n; i++)`
    - `result = result + (a[i] * b[i]);`
  - `printf("Final result= %f\n",result);`
- `}`



# OMP\_GET\_WTIME FUNCTION

- The actual "time in the past" .- WALL TIME
- The `omp_get_wtime` function returns a double-precision floating point
  - `double start;`
  - `double end;`
  - `start = omp_get_wtime();`
  - ... work to be timed ...
  - `end = omp_get_wtime();`
  - `printf("Work took %f sec. time.\n", end-start);`



```
start = omp_get_wtime();  
#pragma omp parallel for  
for (i=0;i<5;i++)  
{  
    b[i]=i;  
    c[i]=i;  
    a[i] = b[i] + c[i];  
    printf ("%d",a[i]);  
}  
end = omp_get_wtime();  
printf ("Parallel Loop took %.04f  
seconds\n",end-start);
```



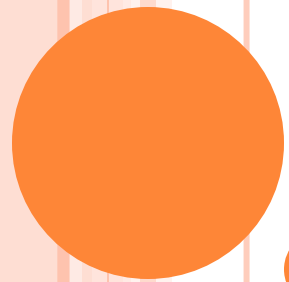
# USE O OMP\_GET\_WTIME WITHOUT PARALLEL THREAD.

○ .....



- Design a arithmetic and logical calculator using OpenMP. For even number of threads arithmetic calculation has to be executed , for odd number of threads logical calculation gets executed . Calculate the wall time for each operation.
- Schedule the operations using guided and static.
- Note : The output obtained from threads in Arithmetic operation will not wait for other outcomes whereas the logical operations has dependency of all threads outcome.
- Set maximum of 10 threads . Use critical section concepts.





## LAB 3 OPENMP



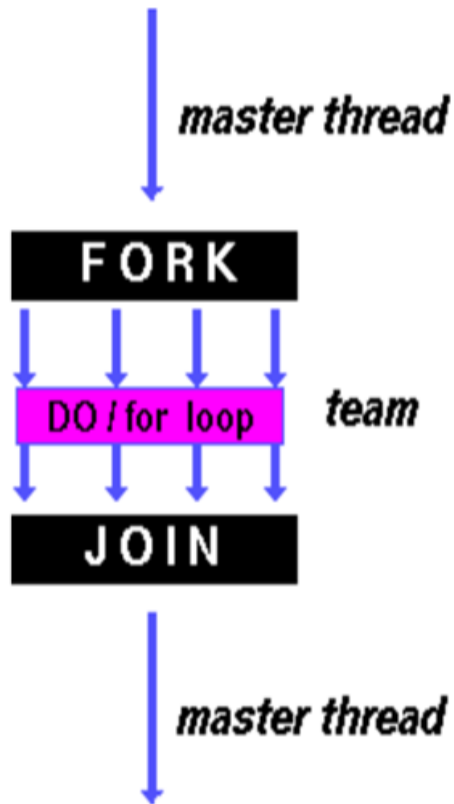
# SECTION DIRECTIVE

- The SECTIONS directive is a **non-iterative** work-sharing construct
- It specifies that the enclosed section(s) of code are to be **divided among the threads** in the team.



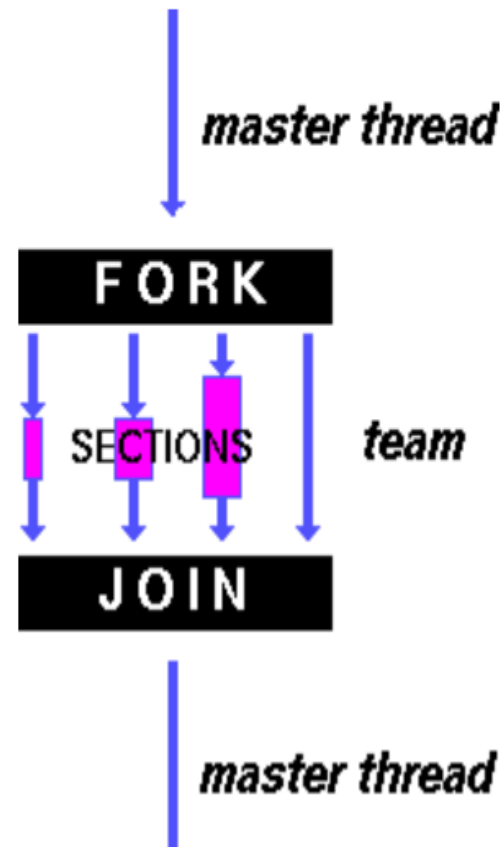
## DO / for

DO / for shares iterations of a loop across the team



## SECTIONS

SECTIONS breaks work into separate, discrete sections.



# SYNTAX

**#pragma omp parallel sections**

**{**

**#pragma omp section**

**.....**

**#pragma omp section**

**.....**

**}**



```

#include <omp.h>
#include <stdio.h>
int main ()
{
int N=1;
int i;
float a[N], b[N], c[N], d[N], e[N];

/* Some initializations */
for (i=0; i < N; i++) {
    a[i] = i * 2.0;
    b[i] = i + 10.0;
}

```

```

#pragma omp parallel
{
    for (i=0; i < N; i++)
    {
        c[i] = a[i] + b[i];
        printf("Thread is %d ADD\n", omp_get_thread_num(), c[i]);
    }
    for (i=0; i < N; i++)
    {
        d[i] = a[i] * b[i];
        printf("Thread is %d MUL\n", omp_get_thread_num(), d[i]);
    }
    for (i=0; i < N; i++)
    {
        e[i] = a[i] - b[i];
        printf("Thread is %d SUB\n", omp_get_thread_num(), e[i]);
    }

} /* end of parallel section */

```

```
#pragma omp parallel
{
  for (i=0; i < N; i++)
  {
    c[i] = a[i] + b[i];
    printf("Thread is %d ADD\n", omp_get_thread_num(), c[i]);
  }
}
```



04/10/2022 19:19.10 /home/mobaxterm/MyDocuments gcc -o sec1234 -fopenmp nosec.c

04/10/2022 19:19.14 /home/mobaxterm/MyDocuments ./sec1234

```
Thread is 0 ADD 10.000000
Thread is 4 ADD 10.000000
Thread is 1 ADD 10.000000
Thread is 2 ADD 10.000000
Thread is 6 ADD 10.000000
Thread is 5 ADD 10.000000
Thread is 3 ADD 10.000000
Thread is 7 ADD 10.000000
Thread is 0 MUL 0.000000
Thread is 4 MUL 0.000000
Thread is 1 MUL 0.000000
Thread is 2 MUL 0.000000
Thread is 6 MUL 0.000000
Thread is 5 MUL 0.000000
Thread is 3 MUL 0.000000
Thread is 7 MUL 0.000000
Thread is 0 SUB -10.000000
Thread is 4 SUB -10.000000
Thread is 1 SUB -10.000000
Thread is 2 SUB -10.000000
Thread is 6 SUB -10.000000
Thread is 5 SUB -10.000000
Thread is 3 SUB -10.000000
Thread is 7 SUB -10.000000
```

```

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

int main ()
{

int i,N=10;
float a[N], b[N], c[N], d[N], e[N];
/* Some initializations */
for (i=0; i < N; i++) {
    a[i] = i * 2.0;
    b[i] = i + 10.0;
}

```

```

#pragma omp parallel sections {
#pragma omp section
    for (i=0; i < N; i++)
    {
        c[i] = a[i] + b[i];
        printf("Thread is %d ADD\n",omp_get_thread_num(), c[i]);
    }
#pragma omp section
    for (i=0; i < N; i++)
    {
        d[i] = a[i] * b[i];
        printf("Thread is %d MUL\n",omp_get_thread_num(), d[i]);
    }
#pragma omp section
    for (i=0; i < N; i++)
    {
        e[i] = a[i] - b[i];
        printf("Thread is %d SUB\n",omp_get_thread_num(), e[i]);
    }
} /* end of sections */
} /* end of parallel section */

```

```
#pragma omp section
```

```
for (i=0; i < N; i++)
```

```
{
```

```
    e[i] = a[i] - b[i];
```

```
    printf("Thread is %d SUB %f\n",omp_get_thread_num(), e[i]);
```

```
}
```

```
} /* end of sections */
```

```
} /* end of parallel section */
```





```
#pragma omp parallel sections      {  
#pragma omp section  
    for (i=0; i < N; i++)  
        {  
            c[i] = a[i] + b[i];  
            printf("Thread is %d ADD  
%f\n",omp_get_thread_num(), c[i]);  
        }  
#pragma omp section  
    for (i=0; i < N; i++)  
        {  
            d[i] = a[i] * b[i];  
            printf("Thread is %d MUL  
%f\n",omp_get_thread_num(), d[i]);  
        }
```



```
#pragma omp parallel sections
```

```
{
```

```
#pragma omp section
```

```
for (i=0; i < N; i++)
```

```
{
```

```
c[i] = a[i] + b[i];
```

```
printf("Thread is %d ADD %f\n",omp_get_thread_num(), c[i]);
```

```
}
```

```
#pragma omp section
```

```
for (i=0; i < N; i++)
```

```
{
```

```
d[i] = a[i] * b[i];
```

```
printf("Thread is %d MUL %f\n",omp_get_thread_num(), d[i]);
```

```
}
```

```
#pragma omp section
```

```
for (i=0; i < N; i++)
```

```
{
```

```
e[i] = a[i] - b[i];
```

```
printf("Thread is %d SUB %f\n",omp_get_thread_num(), e[i]);
```

```
}
```

```
} /* end of sections */
```



04/10/2022 19:19.20 /home/mobaxterm/MyDocuments gcc -o sec12 -fopenmp secarith.c

04/10/2022 19:20.20 /home/mobaxterm/MyDocuments ./sec12

```
Thread is 4 MUL 0.000000
Thread is 2 SUB -10.000000
Thread is 1 ADD 10.000000
Thread is 4 MUL 22.000000
Thread is 2 SUB -8.000000
Thread is 1 ADD 19.000000
Thread is 4 MUL 112.000000
Thread is 2 SUB -5.000000
Thread is 1 ADD 28.000000
Thread is 4 MUL 238.000000
Thread is 2 SUB -2.000000
Thread is 1 ADD 37.000000
```



- One thread  $\rightarrow$  \*
- 2 thread  $\rightarrow$  +
- 3 thread  $\rightarrow$



## EXERCISE 1- LAB 3

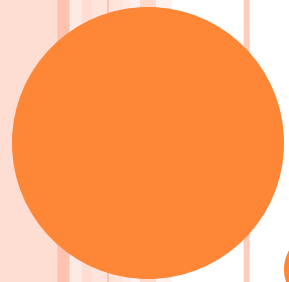
- The Election Commission has decided to organize a special camp to include young people (*age greater than or equal to 16 and less than 18*) in the electoral role. Help the officials to identify the eligible people. Use **thread “1”** to print eligible people and thread **“0”** to not eligible candidate. Get minimum 10 people data.



## EXERCISE 2 LAB 3

- For the same Election Commission program using **sections** calculate
  - (i) the total number of eligible candidates and
  - (ii) total number of not eligible candidates





## LAB 4 –OPENMP

# Computation of Execution Time Using OpenMP clock / windows clock





library function `char *ctime(const time_t *timer)`

returns a **string** representing the local time based on the argument timer.

The return format: **www mmm dd  
hh:mm:ss yyyy,**

where **www** is the weekday, **mmm** the month in letters, **dd** the day of the month, **hh:mm:ss** the time, and **yyyy** the year.

- Write a program to print thread ID along with the current date and time from the master thread



```
#include "omp.h"
#include "stdio.h"
#include "time.h"
main(){
int tid;
time_t t;
#pragma omp parallel private(tid)
tid=omp_get_thread_num();
printf("thread %d\n",tid);
time(&t);
printf("Current Day and time %s\n",ctime(&t));}
```



# OMP\_GET\_WTIME

- The `omp_get_wtime` routine returns elapsed wall clock time in seconds.
- **`double omp_get_wtime(void);`**



- **double start;**  
**double end;**  
**start = omp\_get\_wtime();**  
... work to be timed ...  
**end = omp\_get\_wtime();**  
**etime= end-start;**  
**printf("Work took %f seconds\n", etime);**



1. Consider you have to write a program for VIT placement cell where 10 students are placed in 4 companies namely, Amazon, Google, Shell, and Intel. Assume no student is offered more than one placement offer. The program has to do the following tasks in parallel and display the result with thread id. Use separate sections to perform each operations
  - Get as input the name, register number, the pay package of students selected for jobs in the particular organization
  - Display the total number of students selected in each company.
  - Display the average pay package of the 10 students

Calculate the execution time of each of the above process using wtime.











# LAB 5 OPENMP

Dr.Maheswari.R

# SHARED AND PRIVATE VARIABLES

- Variables declared **before a parallel block** can be *shared or private*
- Shared variables are shared among **all the threads**
- By default,
  - all variables declared **outside** a parallel block are **shared**
  - except the **loop index variable**, which is **private**



# PRIVATE VARIABLES

- Private variables **vary** independently within threads
  - **On entry**, values of private variables are **undefined**.
  - **On exit**, values of private variables are **undefined**
- Variables **declared in a parallel block** are always **private**
- Variables can be **explicitly** declared shared or private.



# SHARED OR PRIVATE VARIABLES???

A simple example:

```
int x[10],y[10],n,i;  
#pragma omp parallel for  
for (i = 0; i < n; i++)  
x[i] = x[i] + y[i];
```

Here **x, y, and n** are **shared** and **i** is **private** in the parallel loop.



# SYNTAX

- #pragma omp parallel for **shared(x, y, n)**  
**private(i)**

for (i = 0; i < n; i++)

x[i] = x[i] + y[i];

or

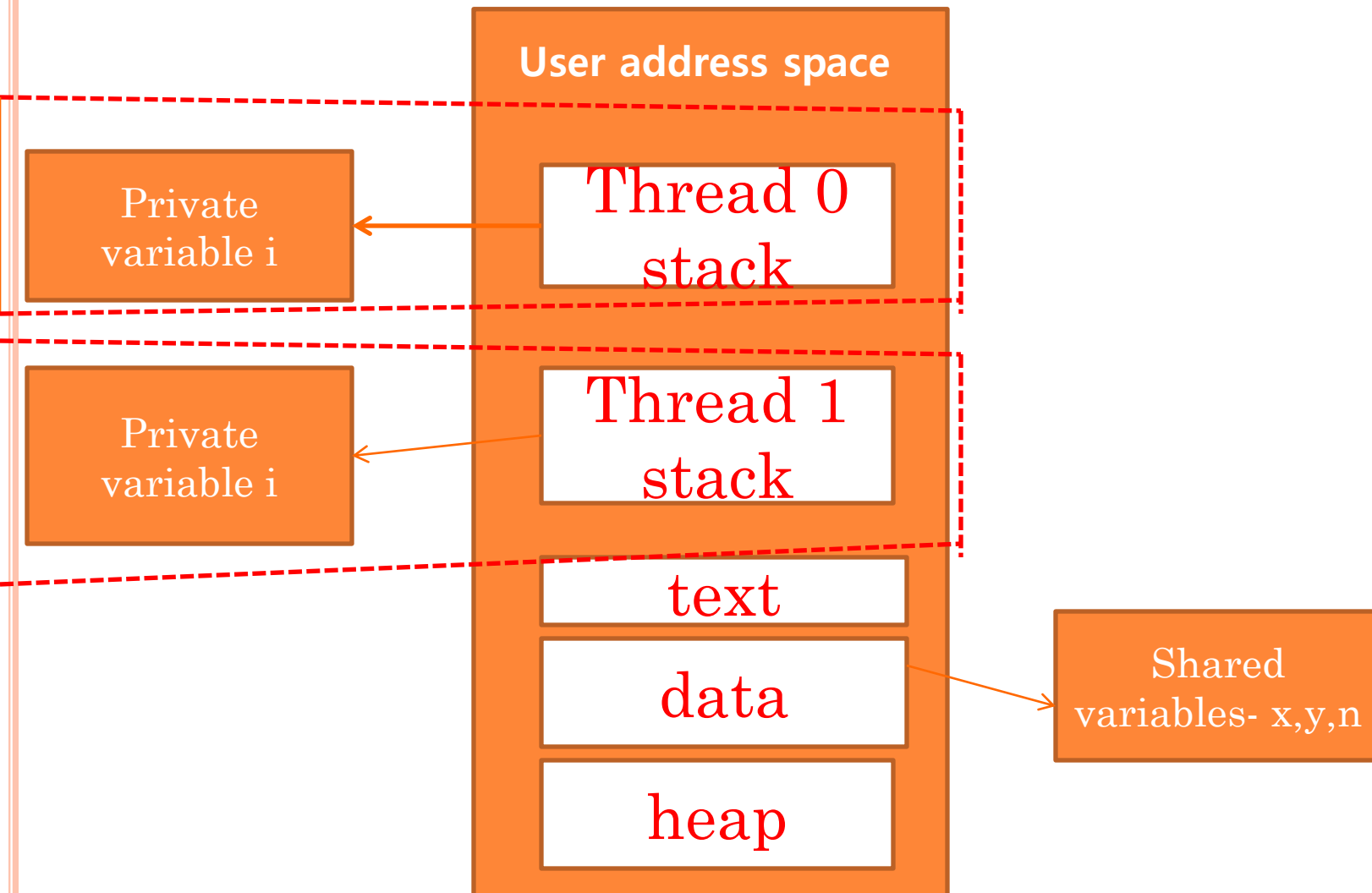
- #pragma omp parallel **for default(shared)**  
**private(i)**

for (i = 0; i < n; i++)

x[i] = x[i] + y[i];



# THREADS



# PRIVATE VARIABLE

- 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.**



```
#include<omp.h>
#include <stdio.h>
int main (){
int x=9;
#pragma omp parallel for private(x)
for(int i=0;i<5;i++)
{
x=i;
printf ("x value inside parallel %d of thread
number %d\n",x,omp_get_thread_num());
} //end of parallel block
printf ("x value outside parallel %d",x); } //end
of main
```





```
09/10/2022 20:38.16 /home/mobaxterm/MyDocuments ./t1
x value inside parallel 0 of thread number 0
x value inside parallel 1 of thread number 1
x value inside parallel 3 of thread number 3
x value inside parallel 2 of thread number 2
x value inside parallel 4 of thread number 4
x value outside parallel 9
```

## Output

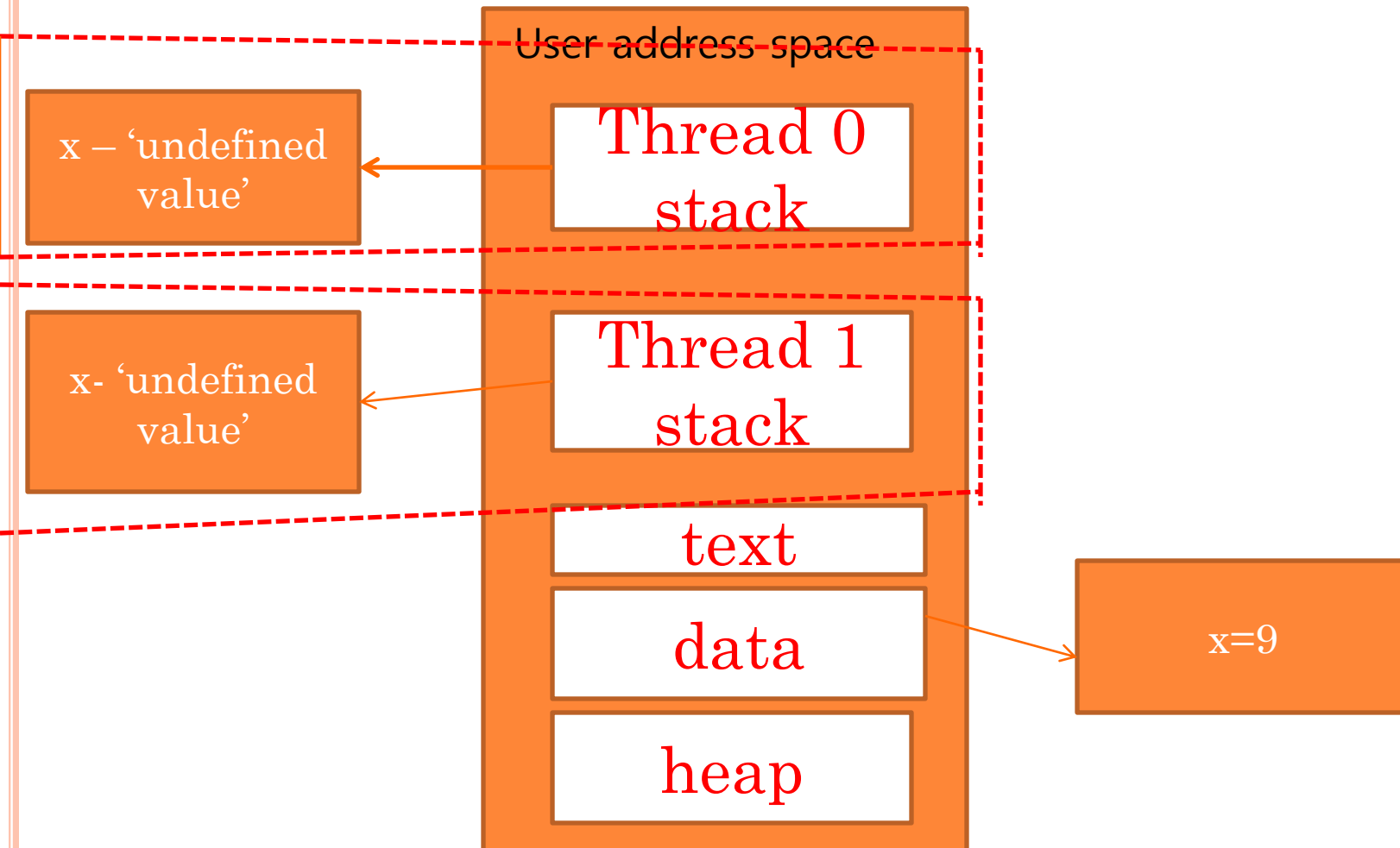
**X value 9**

**Notice : x is initialized as 9 outside parallel block.**

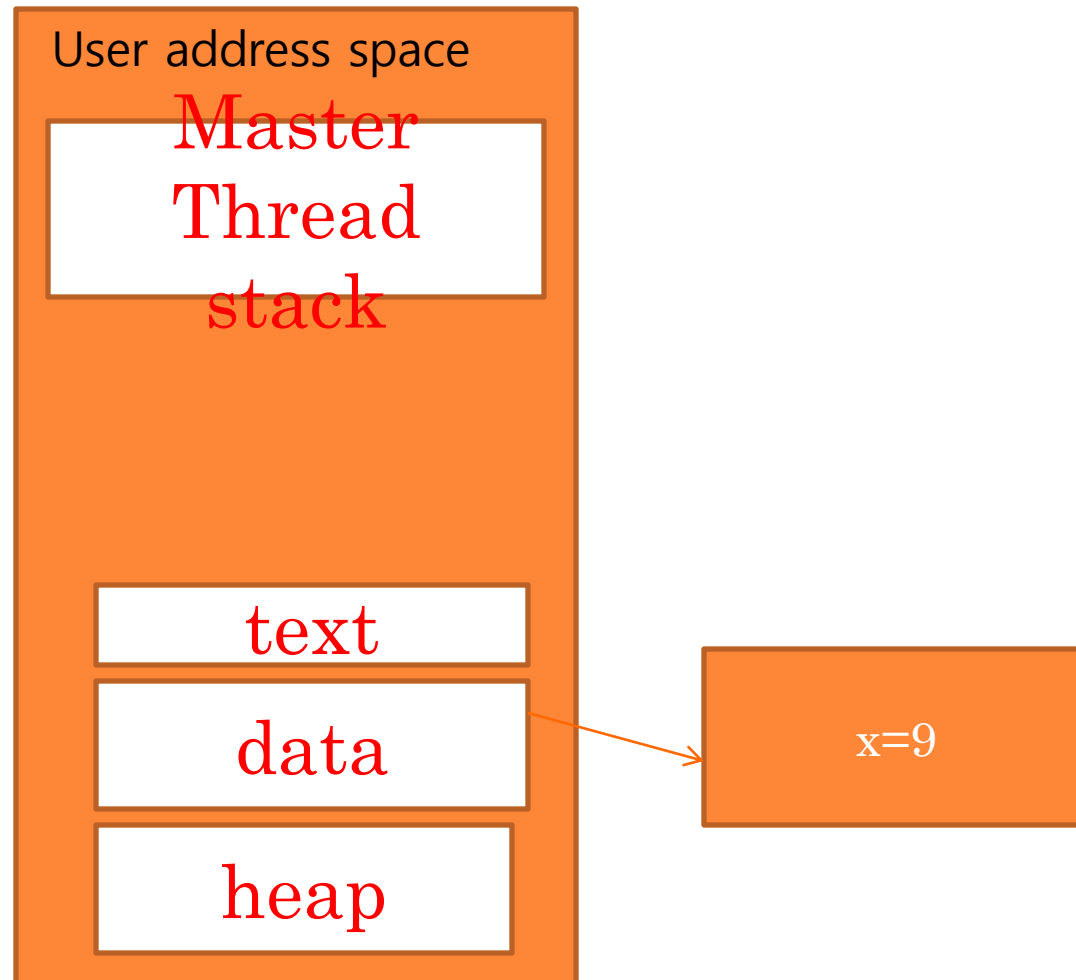
**‘x’ is made private in parallel block means a local copy of x is created in each thread. After the parallel block these local copy no longer exists.**



# BEGINNING OF PARALLEL BLOCK



# END OF PARALLEL BLOCK



# NUM\_THREAD()

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

int main (){
int x=9;
#pragma omp parallel for private(x) num_threads(3)
for(int i=0;i<5;i++)
{
x=i;
printf ("x value inside parallel %d of thread number %d\n",x,omp_get_thread_num());

//printf ("x value inside parallel %d\n",x);

} //end of parallel block

printf ("x value outside parallel %d",x); } //end of main
```



📅 09/10/2022 ⌚ 20:44.07 📁 /home/mobaxterm/MyDocuments ➡ ./t1

```
x value inside parallel 0 of thread number 0  
x value inside parallel 2 of thread number 1  
x value inside parallel 4 of thread number 2  
x value inside parallel 1 of thread number 0  
x value inside parallel 3 of thread number 1  
x value outside parallel 9
```



## LASTPRIVATE

- If we want to keep the last iteration value of 'x' after the parallel region, replace private with lastprivate

## LASTPRIVATE -EXAMPLE

```
#include<omp.h>
main()
{ int x=9;
#pragma omp parallel for lastprivate(x)
for(i=0;i<5;i++)
{ printf ("x value before assignment %d\n",x);
x=i;
printf ("x value inside parallel %d of thread number
%d\n",x,omp_get_thread_num());
} //end of parallel block
printf ("x value outside parallel %d",x); } //end of main
```

### Output

**X value 4**



09/10/2022 20:51.48 /home/mobaxterm/MyDocuments ./t1

```
x value before assignment 0
x value before assignment 0
x value before assignment 0
x value before assignment 0
x value before assignment 0
x value inside parallel 3 of thread number 3
x value inside parallel 2 of thread number 2
x value inside parallel 1 of thread number 1
x value inside parallel 0 of thread number 0
x value inside parallel 4 of thread number 4
x value outside parallel 4
```

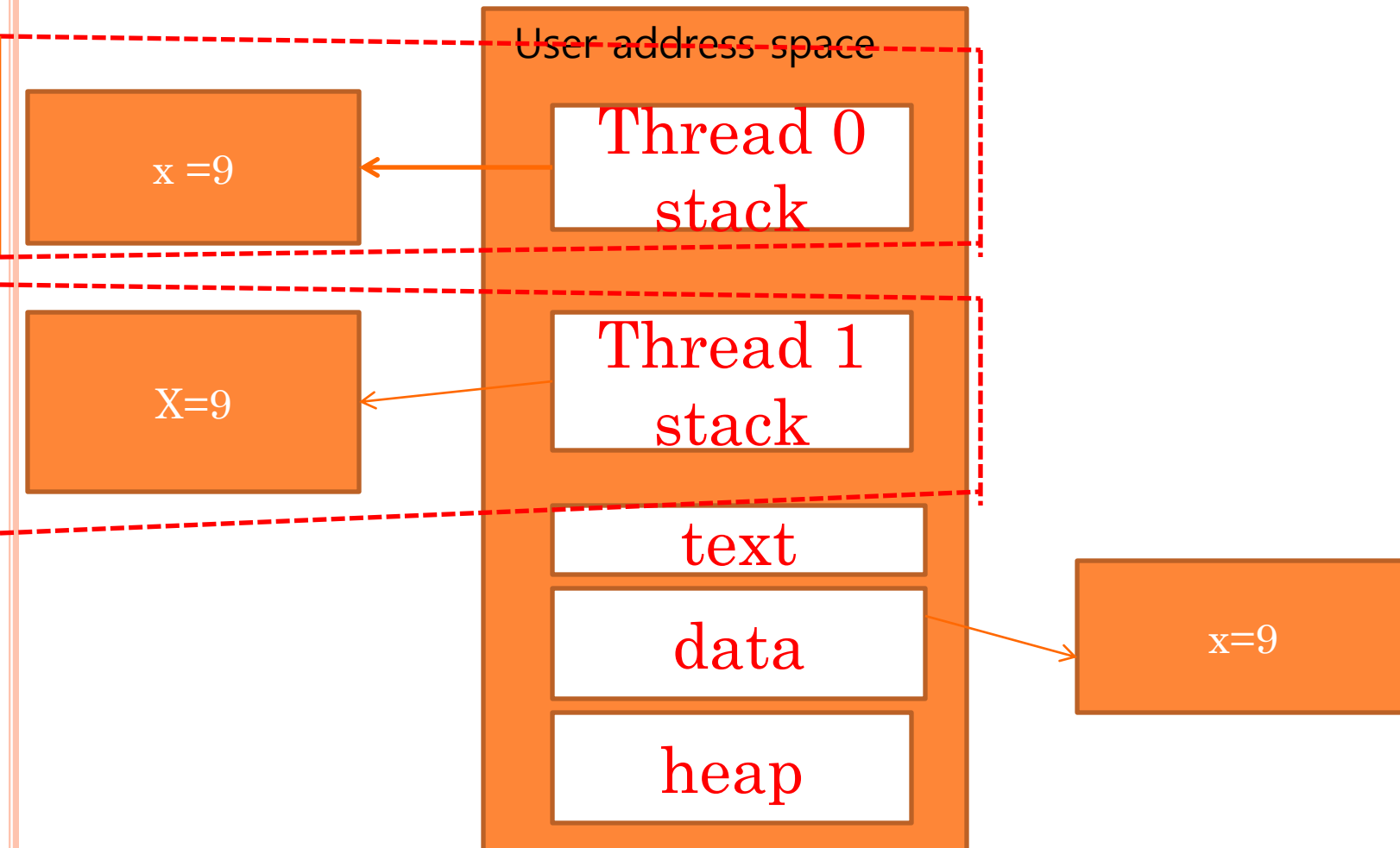




# FIRST PRIVATE

- 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.

# FIRSTPRIVATE



## FIRSTPRIVATE

```
#include<omp.h>
main()
{ int x=9;
#pragma omp parallel for firstprivate (x)
for(i=0;i<5;i++)
{ printf ("x value before assignment %d\n",x);
x=i;
printf ("x value inside parallel %d of thread number
%d\n",x,omp_get_thread_num());
} //end of parallel block
printf ("x value outside parallel %d",x); } //end of main
```

### Output

**X value 9**





09/10/2022



20:51.17



/home/mobaxterm/MyDocuments

./t1

```
x value before assignment 9
x value before assignment 9
x value before assignment 9
x value before assignment 9
x value before assignment 9
x value inside parallel 1 of thread number 1
x value inside parallel 3 of thread number 3
x value inside parallel 4 of thread number 4
x value inside parallel 2 of thread number 2
x value inside parallel 0 of thread number 0
x value outside parallel 9
```



# LIST OF CLAUSES SUPPORTED BY **SECTIONS CONSTRUCT**

- **First private**
- **Last private**
- **Private**
- **reduction**



## EXERCISE- LAB 5

- Design a math application:

It accepts an integer number as input and outputs whether it is a **rational number** or **perfect number** or a **prime number**.

Design a parallel program for the same.

Use sections for every operations

Incorporate **private for rational number**, **Last Private for perfect number**, **first Private for prime number**.

Limit the number of threads as 3.





## LAB 6 – OPENMP

# SCHEDULING IN OPENMP

- Scheduling is a method in OpenMP to distribute iterations to different threads in for loop.

The basic form of OpenMP scheduling is

- **#pragma omp parallel for**  
**schedule(scheduling-type) for(conditions){ do**  
**something }**





The basic form of OpenMP scheduling is

- **#pragma omp parallel for  
schedule(scheduling-type) for(conditions){ do  
something }**



# SCHEDULE

- **schedule** refers to the way in which loop indices are distributed among threads
- default is **static**, which we had seen in an earlier example
  - each thread is assigned a contiguous range of indices in order of thread number
    - called **round robin**
  - number of indices assigned to each thread is as equal as possible



## SCHEDULE (CONT'D)

### ○ static example

- loop runs from 1 to 51, 4 threads

thread	indices	no. indices
0	1-13	13
1	14-26	13
2	27-39	13
3	40-51	12



## SCHEDULE (3)

- number of indices doled out at a time to each thread is called the *chunk size*
- can be modified with the **SCHEDULE** clause

```
!$omp do schedule(static,5)
```

```
#pragma omp for schedule(static,5)
```



## SCHEDULE (4)

- example – same as previous example (loop runs from 1-51, 4 threads), but set chunk size to 5

thread	chunk 1 indices	chunk 2 indices	chunk 3 indices	no. indices
0	1-5	21-25	41-45	15
1	6-10	26-30	46-50	15
2	11-15	31-35	51	11
3	16-20	36-40	-	10



# LOOP WORKSHARING CONSTRUCTS: THE SCHEDULE CLAUSE

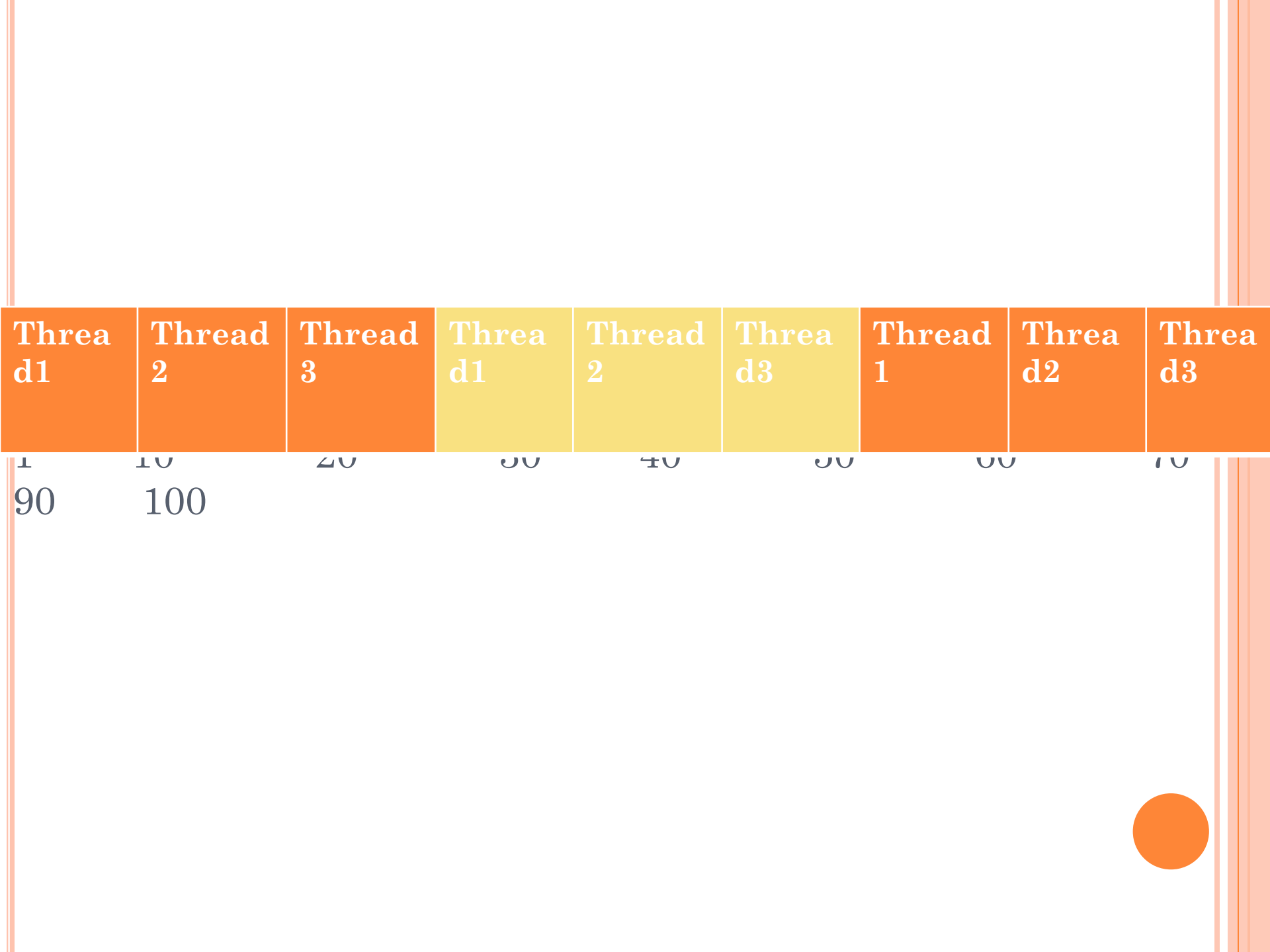
- `schedule(static [,chunk])`
  - Deal-out blocks of iterations of size “chunk” to each thread



## EXAMPLE - STATIC

```
main()
{
int i;
#pragma omp parallel num_threads(3)
{
#pragma omp for schedule(static, 10)
for (int i=0; i<100;i++)
Printf("iteration %d run by thread
%d",i,omp_get_thread_num());
}
}
```









18/10/2022 21:23.30 /home/mobaxterm/MyDocuments gcc -o o1 -fopenmp schedle\_stat.c

18/10/2022 21:24.56 /home/mobaxterm/MyDocuments ./o1

```
iteration 0 run by thread 0
iteration 10 run by thread 1
iteration 20 run by thread 2
iteration 1 run by thread 0
iteration 11 run by thread 1
iteration 21 run by thread 2
iteration 2 run by thread 0
iteration 12 run by thread 1
iteration 22 run by thread 2
iteration 3 run by thread 0
iteration 13 run by thread 1
iteration 23 run by thread 2
iteration 4 run by thread 0
iteration 14 run by thread 1
iteration 24 run by thread 2
iteration 5 run by thread 0
iteration 15 run by thread 1
iteration 25 run by thread 2
iteration 6 run by thread 0
iteration 16 run by thread 1
iteration 26 run by thread 2
iteration 7 run by thread 0
iteration 17 run by thread 1
iteration 27 run by thread 2
iteration 8 run by thread 0
iteration 18 run by thread 1
iteration 28 run by thread 2
iteration 9 run by thread 0
```

```
iteration 73 run by thread 1  
iteration 83 run by thread 2  
iteration 64 run by thread 0  
iteration 74 run by thread 1  
iteration 84 run by thread 2  
iteration 65 run by thread 0  
iteration 75 run by thread 1  
iteration 85 run by thread 2  
iteration 66 run by thread 0  
iteration 76 run by thread 1  
iteration 86 run by thread 2  
iteration 67 run by thread 0  
iteration 77 run by thread 1  
iteration 87 run by thread 2  
iteration 68 run by thread 0  
iteration 78 run by thread 1  
iteration 88 run by thread 2  
iteration 69 run by thread 0  
iteration 79 run by thread 1  
iteration 89 run by thread 2  
iteration 90 run by thread 0  
iteration 91 run by thread 0  
iteration 92 run by thread 0  
iteration 93 run by thread 0  
iteration 94 run by thread 0  
iteration 95 run by thread 0  
iteration 96 run by thread 0  
iteration 97 run by thread 0  
iteration 98 run by thread 0  
iteration 99 run by thread 0
```



## DYNAMIC

- `schedule(dynamic [,chunk])`
- Each thread grabs “chunk” iterations off a queue until all iterations have been handled.



## EXAMPLE - DYNAMIC

```
main()
{
int i;
#pragma omp parallel num_threads(3)
{
#pragma omp for schedule(dynamic, 10)
  for (int i=0; i<100;i++)
    Printf("iteration %d run by thread
           %d",i,omp_get_thread_num());
}
}
```



## DYNAMIC SCHEDULE

- `schedule(dynamic)` clause assigns chunks to threads dynamically as the threads become available for more work
- default chunk size is 1
- higher overhead than **STATIC**

```
!$omp do schedule(dynamic,5)
```

```
#pragma omp for  
schedule(dynamic,5)
```



CONTD..

- In **dynamic scheduling**, using a "chunksize" of 10:
  - thread 1 is assigned to do iterations 1 to 10.
  - thread 2 is assigned to do iterations 11 to 20.
  - thread 3 is assigned to do iterations 21 to 30.
- The **next chunk is iterations 31 to 40**, and will be assigned to **whichever thread finishes its current work first**, and so on until all work is completed.



# GUIDED

- `schedule(guided[,chunk])`
- Threads dynamically grab blocks of iterations.  
The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.





## GUIDED SCHEDULE

- `schedule(guided)` clause assigns chunks automatically, exponentially decreasing chunk size with each assignment
- specified chunk size is the *minimum* chunk size except for the last chunk, which can be of any size
- default chunk size is 1



## EXAMPLE - GUIDED

```
main()
{
int i;
#pragma omp parallel num_threads(3)
{
#pragma omp for schedule(guided, 10)
for (int i=0; i<100;i++)
Printf("iteration %d run by thread
%d",i,omp_get_thread_num());
}
}
```



## EXERCISE 1- LAB 6

- The quality checking unit in the toy modeling unit has incremental counter and counts the tested toy from 0 to 256. Once the counter reaches the max value all tested toy will transferred to dispatching unit in which this counter decrements from the maximum of 256 and reaches to zero. Use last private to get max value. Write a OpenMP program to perform the above scenario using all three scheduling concept.



# SCHEDULE: CONTROLLING WORK DISTRIBUTION

## ○ `schedule(static [, chunksize])`

- Default: chunks of approximately equivalent size, one to each thread
- If more chunks than threads: assigned in round-robin to the threads
- Why might want to use chunks of different size?

## ○ `schedule(dynamic [, chunksize])`

- Threads receive chunk assignments dynamically
- Default chunk size = 1

## ○ `schedule(guided [, chunksize])`

- Start with large chunks
- Threads receive chunks dynamically. Chunk size reduces exponentially, down to chunksize



# SCHEDULE: CONTROLLING WORK DISTRIBUTION

## ○ `schedule(static [, chunksize])`

- Default: chunks of approximately equivalent size, one to each thread
- If more chunks than threads: assigned in round-robin to the threads
- Why might want to use chunks of different size?

## ○ `schedule(dynamic [, chunksize])`

- Threads receive chunk assignments dynamically
- Default chunk size = 1

## ○ `schedule(guided [, chunksize])`

- Start with large chunks
- Threads receive chunks dynamically. Chunk size reduces exponentially, down to chunksize



## EXERCISE

- For rainfall statistics recording, four sensors placed strategically across the city measures the rainfall in mm and store the values (one for each day for each sensor) in four different file. Calculate the maximum rainfall recorded in a particular day across the city.





# OPENMP LAB 7

# ORDERED

- Specifies that code under a parallelized for loop should be executed like a sequential loop.
- The **ordered** directive supports no OpenMP clauses other than for loop.

- **Syntax**

```
#pragma omp ordered  
    structured-block
```





```
#include <omp.h>
#include <stdio.h>
int main ()
{
    #pragma omp parallel for ordered
    for (int i = 1; i <= 8; i++)
    {
        #pragma omp ordered
        printf("Thread %d is executing iteration %d \n",
            omp_get_thread_num(), i);
    }
}
```



08/11/2022 11:52.45 /home/mobaxterm/MyDocuments gcc -o h1 -fopenmp ordered.c

08/11/2022 11:56.51 /home/mobaxterm/MyDocuments ./h1

Thread 0 is executing iteration 1  
Thread 1 is executing iteration 2  
Thread 2 is executing iteration 3  
Thread 3 is executing iteration 4  
Thread 4 is executing iteration 5  
Thread 5 is executing iteration 6  
Thread 6 is executing iteration 7  
Thread 7 is executing iteration 8



# WITHOUT ORDERED

```
#include <omp.h>
#include <stdio.h>
int main ()
{
    #pragma omp parallel for
    for (int i = 1; i <= 8; i++)
    {
        printf("Thread %d is executing iteration %d
\n", omp_get_thread_num(), i);
    }
}
```



```
08/11/2022 11:59.27 /home/mobaxterm/MyDocuments ./h1
Thread 6 is executing iteration 7
Thread 1 is executing iteration 2
Thread 7 is executing iteration 8
Thread 5 is executing iteration 6
Thread 4 is executing iteration 5
Thread 2 is executing iteration 3
Thread 3 is executing iteration 4
Thread 0 is executing iteration 1
```



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

int main ()
{

#pragma omp parallel for ordered
for (int i = 1; i <= 8; i++)
{
    printf("Thread %d is executing iteration %d Unordrered fashion \n",
omp_get_thread_num(), i);
    #pragma omp ordered
    printf("Thread %d is executing iteration %d Ordred fashion \n",
omp_get_thread_num(), i);
}}
```



📅 08/11/2022 🕒 12:21.26 📁 /home/mobaxterm/MyDocuments ➡ ./h1

```
Thread 0 is executing iteration 1 Unordrered fashion
Thread 1 is executing iteration 2 Unordrered fashion
Thread 2 is executing iteration 3 Unordrered fashion
Thread 5 is executing iteration 6 Unordrered fashion
Thread 6 is executing iteration 7 Unordrered fashion
Thread 7 is executing iteration 8 Unordrered fashion
Thread 4 is executing iteration 5 Unordrered fashion
Thread 3 is executing iteration 4 Unordrered fashion
Thread 0 is executing iteration 1 Ordrered fashion
Thread 1 is executing iteration 2 Ordrered fashion
Thread 2 is executing iteration 3 Ordrered fashion
Thread 3 is executing iteration 4 Ordrered fashion
Thread 4 is executing iteration 5 Ordrered fashion
Thread 5 is executing iteration 6 Ordrered fashion
Thread 6 is executing iteration 7 Ordrered fashion
Thread 7 is executing iteration 8 Ordrered fashion
```



## EXERCISE1-LAB 7

- Design a parallel program to print 'n' even and odd numbers in a sequential fashion of threads.



## EXAMPLE

```
main()
{
int n,sum=0,i;
printf("Enter n:");
scanf("%d",&n);
#pragma omp parallel num_threads(3)
{ #pragma omp for ordered
for(i=0;i<n;i++)
{ sum=sum+i;
#pragma omp ordered{
printf("\n%d run by thread %d",i,omp_get_thread_num());}
}
} printf("%d",sum);
}
```





# LOCKS

---



# LOCKS

- Locks offer additional **control of threads**
- A thread can take/release **ownership** of a lock over a specified region of code
- When **a lock is owned by a thread**, other threads **cannot** execute the locked region
- Can be used to perform useful work by one or more threads while another thread is engaged in a serial task



# LOCK ROUTINES

- Lock routines each have a single argument
  - argument type is a pointer to an integer to hold an address
  - OpenMP pre-defines required types

```
integer(omp_lock_kind) :: mylock
```

```
omp_nest_lock_t *mylock;
```



# LOCK ROUTINES

- **omp\_init\_lock(mylock)**
  - initializes *mylock*
  - must be called before *mylock* is used
- **omp\_set\_lock(mylock)**
  - gives ownership of *mylock* to calling thread
  - other threads cannot execute code following call until *mylock* is released

# LOCK ROUTINES

- `omp_test_lock(mylock)`
  - logical function
  - if *mylock* is presently owned by another thread, returns *false*
  - if *mylock* is available, returns *true* and sets lock, i.e., gives ownership to calling thread



# LOCK ROUTINES

- `omp_unset_lock(mylock)`
  - releases ownership of *mylock*
- `omp_destroy_lock(mylock)`
  - call when you're done with the lock
  - complement to `omp_init_lock(mylock)`



# LOCKS

- **Protect resources with locks.**
- lock variable must have **type `omp_lock_t`**.
- The **`omp_init_lock`** function initializes a simple lock.
- The **`omp_destroy_lock`** function removes a simple lock.
- The **`omp_set_lock`** function waits until a simple lock is available.
- The **`omp_unset_lock`** function releases a simple lock.



```
#include <omp.h>
#include <stdio.h>
int main ()
{ int id,i;
```





```
omp_lock_t lck;  
omp_init_lock(&lck);  
#pragma omp parallel  
{ id= omp_get_thread_num();  
#pragma omp for  
for(i=0;i<7;i++)  
{ omp_set_lock(&lck);  
printf("\n thread id %d  iteration %d",id,i);  
omp_unset_lock(&lck);  
}  
}  
omp_destroy_lock(&lck);  
}
```



 /home/mobaxterm/MyDocuments

 08/11/2022  12:49.29  ./h1

```
thread id 5    iteration 4
thread id 7    iteration 2
thread id 7    iteration 5
thread id 7    iteration 3
thread id 7    iteration 6
thread id 7    iteration 1
thread id 7    iteration 0
```



## EXAMPLE

```
main()
{
int id,i;
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel num_threads(3){
id= omp_get_thread_num();
#pragma omp for
for(i=0;i<7;i++)
{ omp_set_lock(&lck);
printf("\n%d  %d",id,i);
omp_unset_lock(&lck);
} }
omp_destroy_lock(&lck);
}
```



## EXERCISE2- LAB 7

- A contest is being held for TechnoVIT. Students can register, if they want, they can unregister. Registered students (registration numbers:9,3,2...)is stored in an array. Only one student can register or unregister at a time. But, they can view registered list without any constraint. Design a parallel program with the help of locks.



# CHANELLEGE



# ATOMIC

- **Atomic provides mutual exclusion but only applies to the update of a memory location**
- Syntax
  - `#pragma omp atomic`

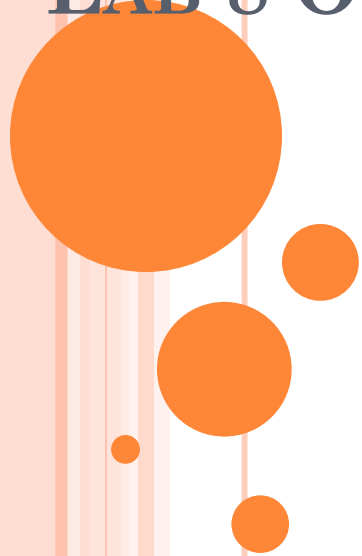


# BARRIER

- Each thread waits until all threads arrive.
- Syntax
  - `#pragma omp barrier`



# LAB 8 OPENMP





```
#include <stdio.h>
int main()
{
    char name[50];
    int marks, i, num;
    printf("Enter number of students:");
    scanf("%d", &num);
    FILE *fptr;

    fptr = (fopen("D:\\student.txt",
"w"));

    if(fptr == NULL)
    {
```

```
        printf("Error!");
    }
    for(i = 0; i < num; ++i)
    {
        printf("For
student%d\nEnter name: ",
i+1);
        scanf("%s", name);
        printf("Enter marks:");
        scanf("%d", &marks);
        fprintf(fptr, "\nName:
%s \nMarks=%d \n",
name, marks);
    }
    fclose(fptr);
```

# BARRIER

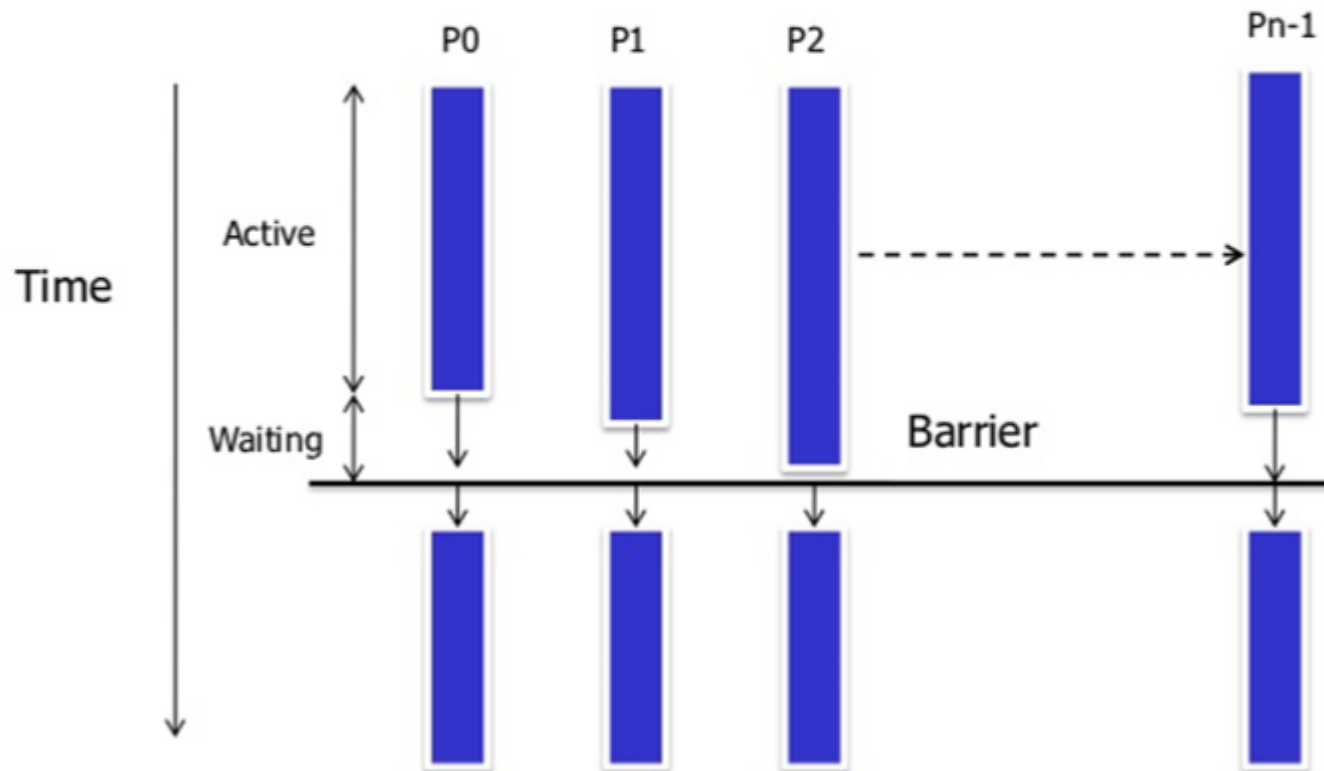
## **barrier**

- Synchronizes all threads at this point
- When a thread reaches a barrier it only continues after all threads have reached it
- Implicit barrier at: end of parallel do/for, single
- Restrictions:
  - Each barrier must be encountered by all threads in a team, or none at all
  - The sequence of work-sharing regions and barrier regions encountered must be same for all threads in team




# Barrier

`#pragma omp barrier`



```
#include <stdio.h>
#include <omp.h>
int main()
{
    int i;
    int
thread_id,num_threads;
FILE *fptr;
fptr =
(fopen("D:\\student.txt",
"w"));
if(fptr == NULL)
{
    printf("Error!");
}
```

```
#pragma omp parallel
{
    int thread_id =
omp_get_thread_num();
    if (thread_id == 0)
        num_threads =
omp_get_num_threads();
        #pragma omp
barrier
        fprintf(fptr,"Hello World
from thread %d of %d\n",
thread_id, num_threads);
}
    fclose(fptr);
}
```



```
#include <stdio.h>
#include <omp.h>
int main()
{
    int i;
    int thread_id,num_threads;
FILE *fptr;
fptr = (fopen("D:\\student.txt", "w"));
if(fptr == NULL)
{
    printf("Error!");
}
```



```
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    if (thread_id == 0)
        num_threads = omp_get_num_threads();
        #pragma omp barrier
        fprintf(fp_ptr, "Hello World from thread %d of %d\n",
thread_id, num_threads);
        fclose(fp_ptr);
}
```



- Only one thread read the total number of threads and all threads print that information



## LAB 8-EXCERISE

Write a parallel program using OpenMP to implement the following series,

$$1/2 + 1/4 + 1/8 + \dots$$

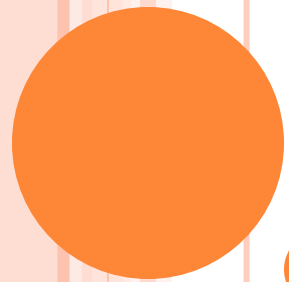
Find the sum of the series and print it along with the thread id and the last value in the series for given “N” value .

Incorporate barrier, scheduling, ordered constructs of OpenMP.

- Print the output in a file “series.txt”



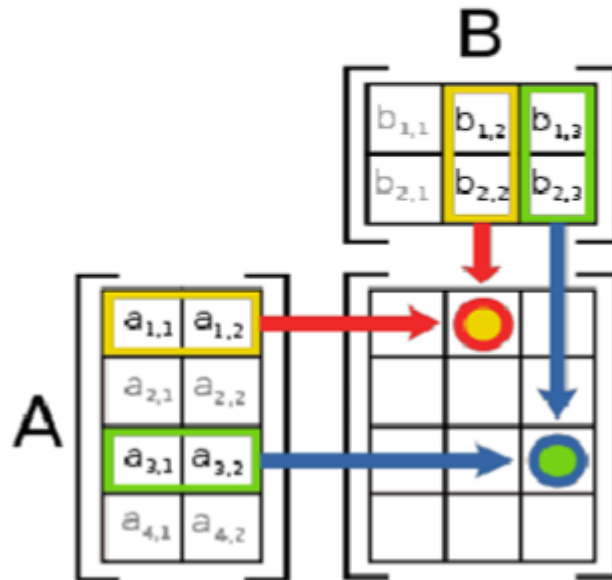




## LAB 9 OPENMP

# OPENMP – MATRIX- MATRIX MULTIPLICATION

$$(AB)_{i,j} = \sum_{k=1}^p A_{ik}B_{kj}$$



```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>
```

```
#define N 100
```

```
int A[N][N];
int B[N][N];
int C[N][N];
```



```
int main()
{
    int i,j,k;
    struct timeval tv1, tv2;
    struct timezone tz;
        double elapsed;
    omp_set_num_threads(omp_get_num_procs());
    for (i= 0; i< 3; i++)
        for (j= 0; j< 3; j++)
            {
                A[i][j] = 2;
                B[i][j] = 2;
            }
    gettimeofday(&tv1, &tz);
```



```
#pragma omp parallel for private(i,j,k)
shared(A,B,C)
```

```
    for (i = 0; i < 3; ++i) {
        for (j = 0; j < 3; ++j) {
            for (k = 0; k < 3; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
```

```
    gettimeofday(&tv2, &tz);
```

```
    elapsed = (double) (tv2.tv_sec-tv1.tv_sec) +
(double) (tv2.tv_usec-tv1.tv_usec) * 1.e-6;
```



```
printf("elapsed time = %f seconds.\n", elapsed);  
for (i= 0; i< 3; i++)  
    {  
        for (j= 0; j< 3; j++)  
        {  
            printf("%d\t",C[i][j]);  
        }  
        printf("\n");  
    }  
}
```



- create a parallel region
  - fork a ***team*** of threads (usually as many as cores)
- arrays A, B, C are *shared* among the threads
- the "iterators" are *private* to each threads



The structure is described though:

```
struct timeval {  
    long tv_sec; /*  
        seconds */  
    long tv_usec; /*  
        microseconds */ };
```

**long int tv\_sec**

**This represents the number of whole seconds of elapsed time.**

**long int tv\_usec**

**This is the rest of the elapsed time (a fraction of a second), represented as the number of microseconds. It is always less than one million**



```
elapsed = (double) (tv2.tv_sec-tv1.tv_sec) +  
(double) (tv2.tv_usec-tv1.tv_usec) * 1.e-6;
```

the total time is

$tv\_sec + (1.0/1000000) * tv\_usec$  seconds.

That's why when you need times under a second you set `tv_usec`,

when you need times over 1sec you set both (but usually end up setting only `tv_sec`)



```
22/11/2022 22:04.32 /home/mobaxterm/MyDocuments gcc -o h1 -fopenmp mat_mul.c

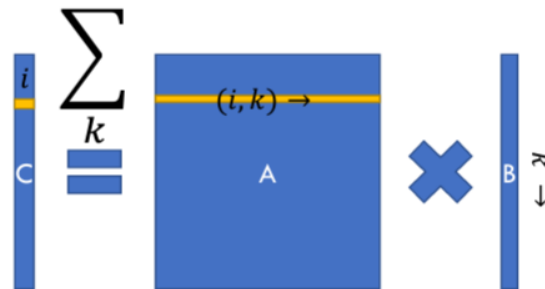
22/11/2022 22:06.55 /home/mobaxterm/MyDocuments ./h1
elapsed time = 0.004918 seconds.
12 12 12
12 12 12
12 12 12
```

```
22/11/2022 22:06.57 /home/mobaxterm/MyDocuments ./h1
elapsed time = 0.008234 seconds.
12 12 12
12 12 12
12 12 12
```



## LAB 9-EXCERISE

Write a parallel program using OpenMP to Matrix-Vector multiplication



- $(n \times n) \times (n \times 1) \Rightarrow (n \times 1)$  output vector
- Output = dot-products of rows from A and the vector B

