Q.1 What are Condition Variables? Explain the use of Condition Variable with an example. [4]

Ans: Pthreads API can be informally grouped into three major classes:

1. Thread management
2. Mutexes
3. Condition variables: The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

In a critical section (i.e. where a mutex has been used), a thread can suspend itself on a condition variable if the state of the computation is not right for it to proceed.

– It will suspend by waiting on a condition variable.

– It will, however, release the critical section lock (mutex).

– When that condition variable is signaled, it will become ready again; it will attempt to re-acquire

that critical section lock and only then will be able proceed.

With Posix threads, a condition variable can be associated with only one mutex variable!

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

With locks, the global variable would need to be examined at frequent intervals (“polled”) within a critical section.

This is a very time-consuming and unproductive exercise. Can be overcome by introducing so-called condition variable!

Condition Variables

Main Thread

- o Declare and initialize global data/variables which require synchronization (such as "count")
- o Declare and initialize a condition variable object
- o Declare and initialize an associated mutex
- o Create threads A and B to do work

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. *Note that a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.*
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Thread B

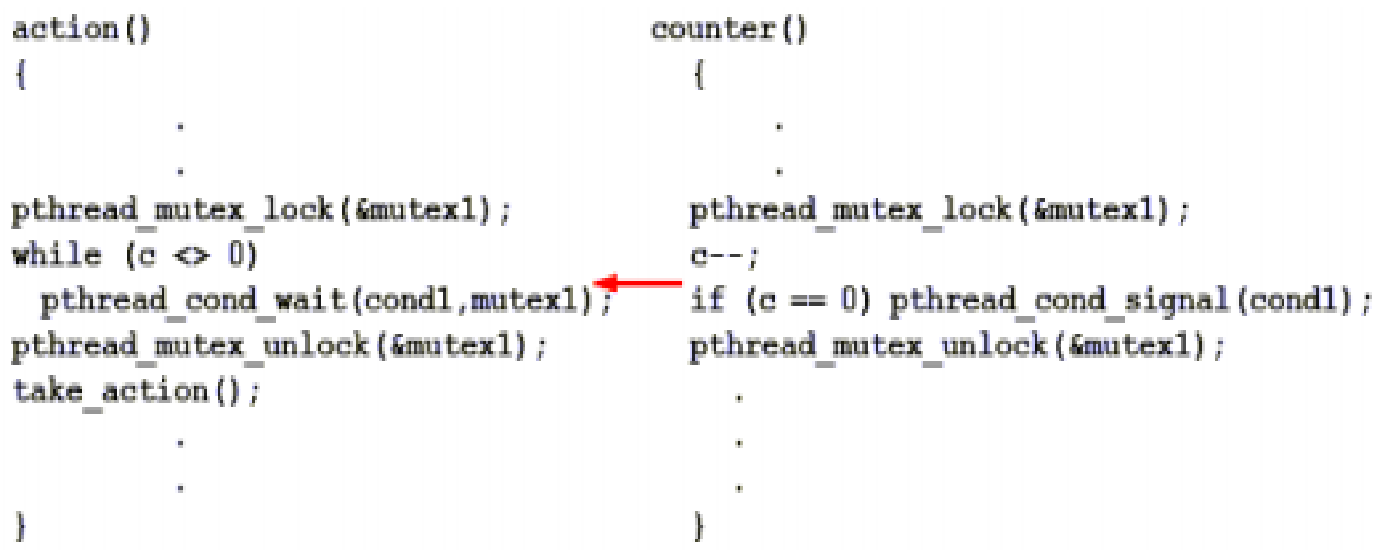
- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

Main Thread
Join / Continue

Associated with a specific mutex. Given declarations:

```
pthread_cond_t cond1;

pthread_mutex_t mutex1;

pthread_cond_init(&cond1, NULL);

pthread_mutex_init(&mutex1, NULL);
```

the Pthreads arrangement for signal and wait is as follows:



Signals are not remembered - threads must already be waiting for a signal to receive it.

Q.2 What will be the output of the C/OpenMP program (assuming no syntax error) on dual core computer if [4]

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N     3
#define CHUNKSIZE  5

int main (int argc, char *argv[])
{
int i, chunk, tid; float a[N], b[N], c[N]; char first_time;

for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE; first_time = 'y';

#pragma omp parallel for shared(a,b,c,chunk)  private(i,tid) \
 schedule(static,chunk)  firstprivate(first_time)

 for (i=0, i < N; i++)
  {   if (first_time == 'y')
   {
   tid = omp_get_thread_num();  first_time = 'n';
   }
    c[i] = a[i] + b[i];
  printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
  }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N       50
#define CHUNKSIZE   5

int main (int argc, char *argv[])
{
  int i, chunk, tid;
  float a[N], b[N], c[N];
  char first_time;

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
```

```
first_time = 'y';

#pragma omp parallel for     \
  shared(a,b,c,chunk)        \
  private(i,tid)             \
  schedule(static,chunk)     \
  firstprivate(first_time)

  for (i=0; i < N; i++)
  {
    if (first_time == 'y')
    {
      tid = omp_get_thread_num();
      first_time = 'n';
    }
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
  }
}
```

The output is as expected:

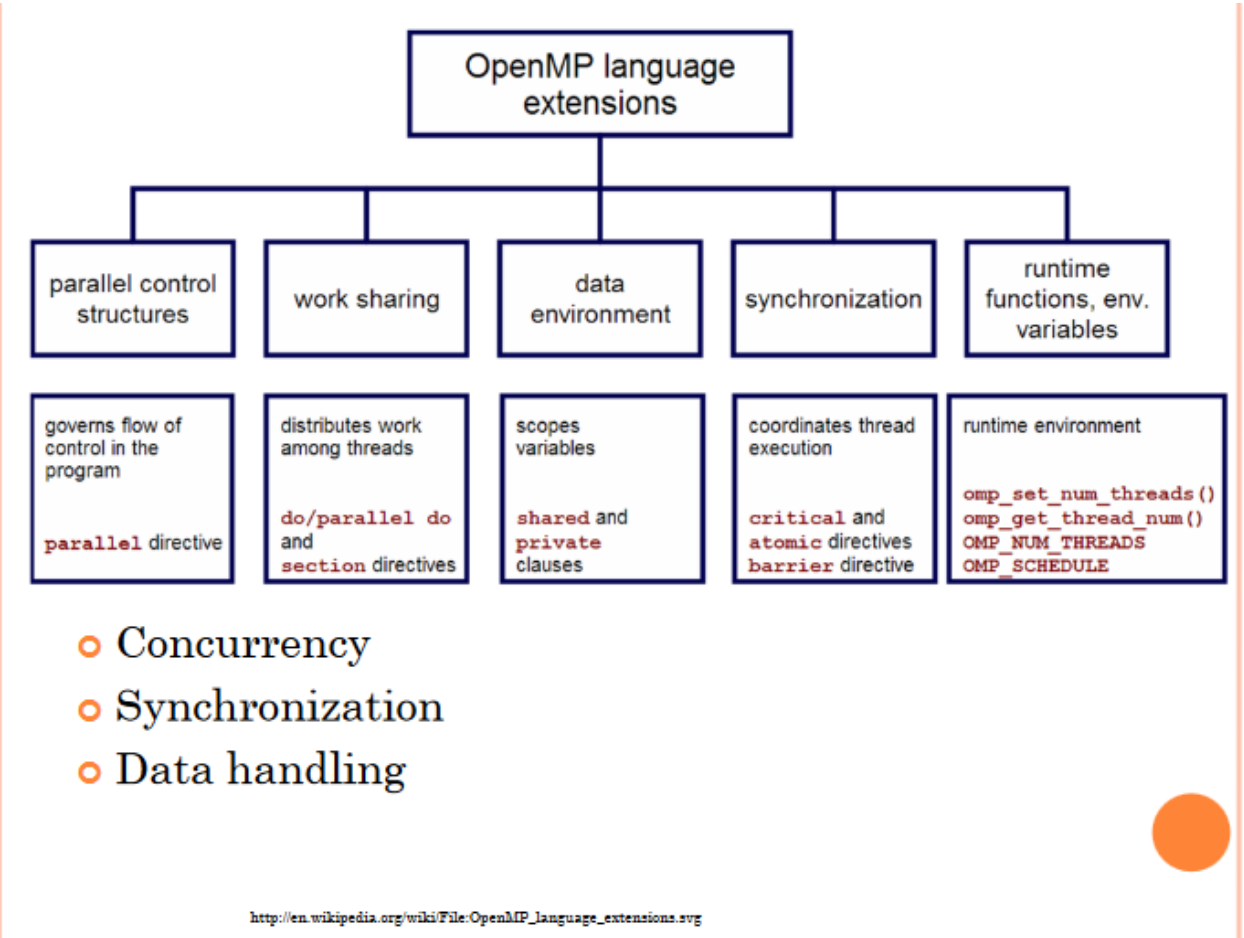
Hide Copy Code

```
tid= 0 i= 0 c[i]= 0.000000
tid= 1 i= 5 c[i]= 10.000000
tid= 0 i= 1 c[i]= 2.000000
tid= 1 i= 6 c[i]= 12.000000
tid= 0 i= 2 c[i]= 4.000000
tid= 1 i= 7 c[i]= 14.000000
tid= 0 i= 3 c[i]= 6.000000
tid= 1 i= 8 c[i]= 16.000000
tid= 0 i= 4 c[i]= 8.000000
tid= 1 i= 9 c[i]= 18.000000

.  . .   .  .  and so on

tid= 1 i= 47 c[i]= 94.000000
tid= 0 i= 43 c[i]= 86.000000
tid= 1 i= 48 c[i]= 96.000000
tid= 0 i= 44 c[i]= 88.000000
tid= 1 i= 49 c[i]= 98.000000
```

Source: https://www.codeproject.com/Articles/60176/WebControls/

Q.3 What is work sharing construct how do you used them in OpenMP programming? [4]

Ans 3)



Main categories of OpenMP's constructs:

- Directives
- Parallel Regions
- Work-sharing
- Data Environment
- Synchronization

OpenMP: Work-sharing Construct

It distributes the execution of the associated statement among the members of the team that encounter it

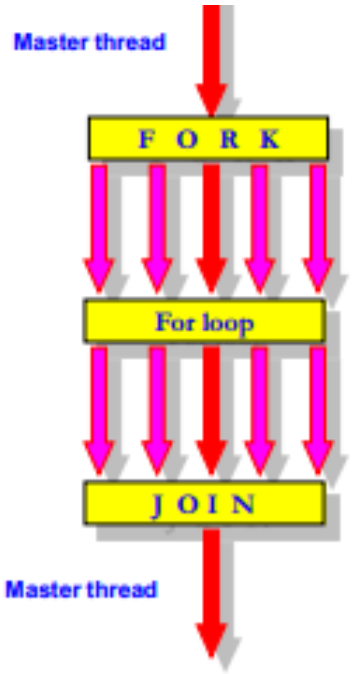
- Work sharing construct do not launch new threads
- There is no barrier upon entry to work-sharing construct.
- There is an implied barrier at the end of a work sharing construct

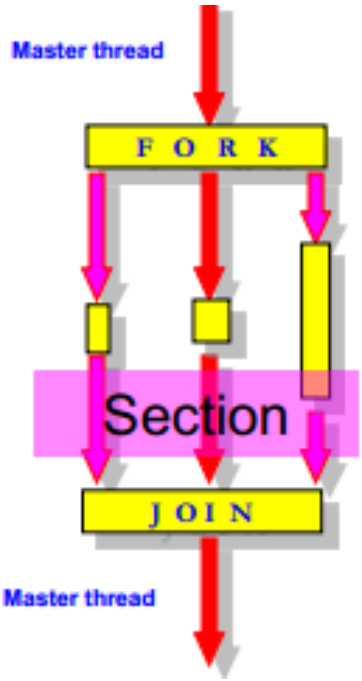
Restrictions

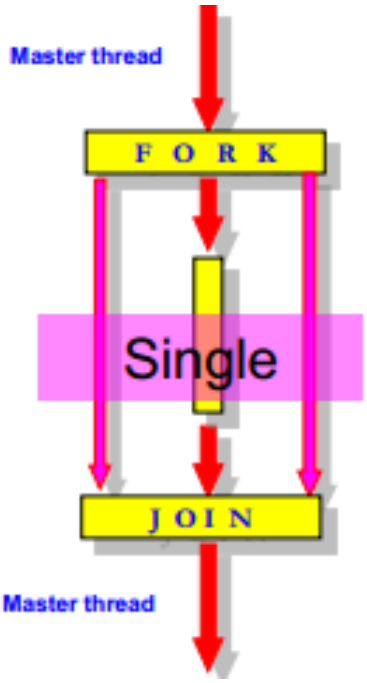
- Must be enclosed in the parallel region for parallel execution.
- Must be encountered by all the members of the team or none of them.

OpenMP defines the following work-sharing constructs.

- for directive
- sections directive
- single directive







For directive (Represents a type of “data parallelism”)

For directive identifies the iterative work-sharing construct.

#pragma omp for [clause[[,]clause]…] new-line

for-loop Clause is one of the following:

scheduled (type [,chunk])

private(variable list)

firstprivate (variable list)

lastprivate (variable list)

reduction (variable list)

ordered, nowait

For directive

The “for” Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
#pragma omp for
for (I=0; I<N; I++) {
        NEAT_STUFF(I);
}
```

Note: By default, there is a barrier at the end of the “omp for”.

Important: The for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main()
{
int i, chunk;  float a[N], b[N], c[N];
/* Some initializations */
```

```
for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)  {
#pragma omp for schedule(dynamic,chunk) nowait
for (i=0; i < N; i++)
c[i] = a[i] + b[i]; }/* end of || section */
}
```

sections directive

Can be used to implement a type of "functional parallelism".

sections directive gives different structured blocks to each thread.

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

- Independent SECTION directives are nested within a SECTIONS

directive.

- Each SECTION is executed once by a thread in the team. Different

sections will be executed by different threads.

Simple vector-add program

- The first n/2 iterations of the for loop will be distributed to the first thread, and the rest will be distributed to the second thread.
- When each thread finishes its block of iterations, it proceeds with whatever code comes next (NOWAIT)

```
#include <omp.h>
#define N 1000
main()
{
        int i;
        float a[N], b[N], c[N];

        for (i=0; i < N; i++)
                a[i] = b[i] = i * 1.0;
        #pragma omp parallel shared(a,b,c) private(i)
        {
                #pragma omp sections nowait
                {
                        #pragma omp section
```

```
                for (i=0; i < N/2; i++)
                        c[i] = a[i] + b[i];
                #pragma omp section
                for (i=N/2; i < N; i++)
                        c[i] = a[i] + b[i];
        }
    }
}
```

single directive

This identifies that the associated structured block is to be executed by only one thread in the team (It can be any thread including

master thread).

#pragma omp single [clause[[,] clause] ...]

new-line

structured-block

Restrictions

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all.
- Successive work-sharing constructs must be encountered in the same order by all members of a team.

Q.4 What is the basic difference between process and a thread? Explain the model OpenMP used for the parallel job execution. **[4]**

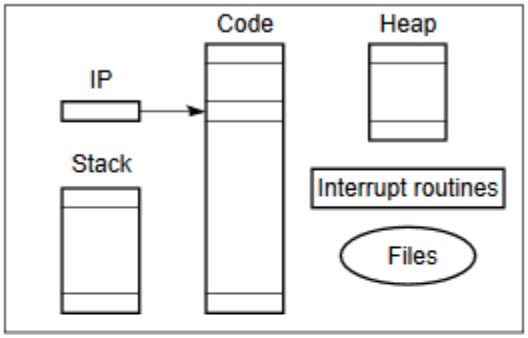
Ans:

slides8-10

Differences between a process and threads

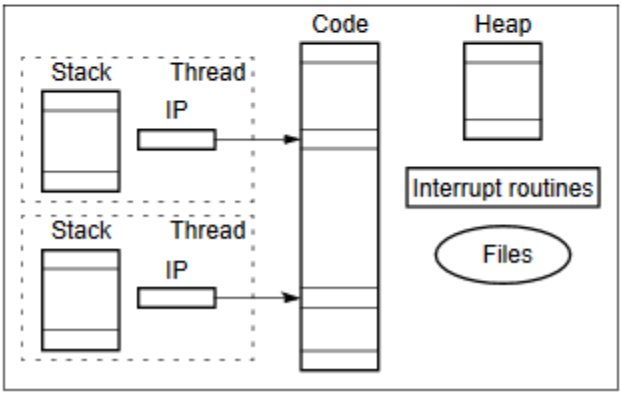
"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.

(a) Process



Threads - shares the same memory space and global variables between routines.

(b) Threads



	Process	Thread
Definition	An executing instance of a program is called a process.	A thread is a subset of the process.
Process	It has its own copy of the data segment of the parent process.	It has direct access to the data segment of its process.
Communication	Processes must use inter-process communication to communicate with sibling processes.	Threads can directly communicate with other threads of its process.
Overheads	Processes have considerable overhead.	Threads have almost no overhead.
Creation	New processes require duplication of the parent process.	New threads are easily created.

Control	Processes can only exercise control over child processes.	Threads can exercise considerable control over threads of the same process.
Changes	Any change in the parent process does not affect child processes.	Any change in the main thread may affect the behavior of the other threads of the process.
Memory	Run in separate memory spaces.	Run in shared memory spaces.
File descriptors	Most file descriptors are not shared.	It shares file descriptors.
File system	There is no sharing of file system context.	It shares file system context.
Signal	It does not share signal handling.	It shares signal handling.
Controlled by	Process is controlled by the operating system.	Threads are controlled by programmer in a program.
Dependence	Processes are independent.	Threads are dependent.

Fork – Join Model

- OpenMP uses fork and join model for parallel execution
- OpenMP programs begin with single process: master thread.
- FORK : Master thread creates a team of parallel threads
- JOIN: When the team threads complete the statements in parallel region, they synchronize and terminate leaving master thread.

- Parallelism is added incrementally conversion from a sequential to parallel is a little bit at a time
- Threads based parallelization

  Open MP is based on the existence of multiple threads in the shared memory programming paradigm
- Explicit parallelization

  It is an explicit (not automatic) programming model, and offers full control over parallelization to the programmer
- Compiler directive based

  All of OpenMP parallelization is supported through the use of compiler directives
- Nested parallelism support

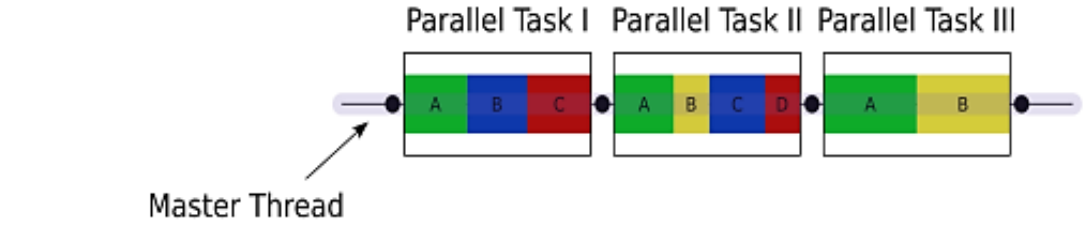
  The API support placement of parallel construct inside other parallel construct
- Dynamic threads

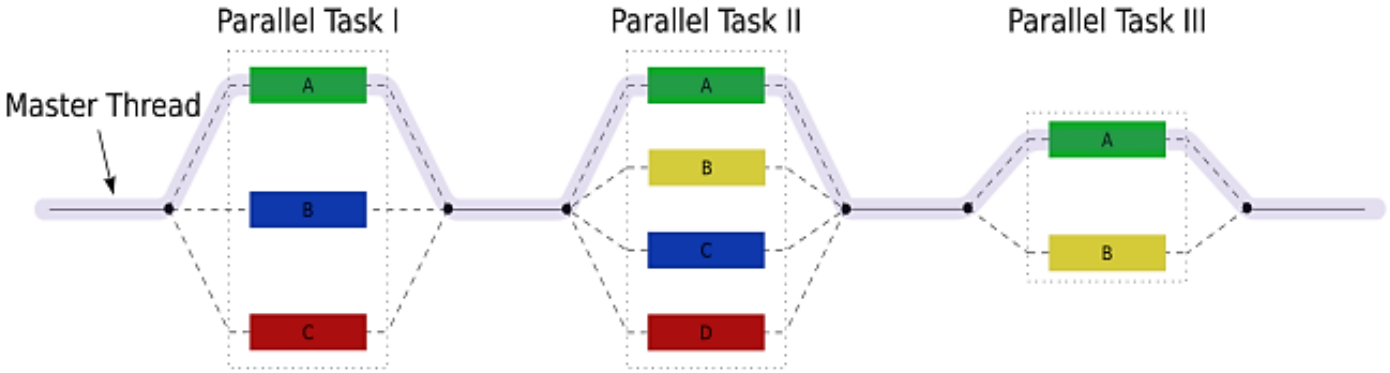
  The API provides dynamic altering of number of threads (Depends on the implementation)

How do threads interact?

OpenMP is shared memory model. Threads communicate by sharing variables

OpenMP Structure





C / C++ - General Code Structure

```
#include <omp.h>
main()
{
int var1, var2, var3;
Serial code...
Beginning of parallel section. Fork a team of threads.
Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
        {
        Parallel section executed by all threads...
        All threads join master thread and disband
        }
Resume serial code...
}
```

Q.5 Write a program (pthread or OpenMP) to calculate the value of pi using reduction clause. [4]

Reductions

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to parallel for pragma
- Specify reduction operation and reduction variable

• OpenMP takes care of storing partial results in private variables and combining partial results after the loop

• Local copies are reduced into a single global copy at the end of the construct.

reduction Clause

- The reduction clause has this syntax:
  reduction (<*op*> :<*variable*>)

- Operators

+	Sum
*	Product
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
&&	Logical and
||	Logical or

π-finding Code with Reduction Clause

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \
        private(x) reduction(+:area)
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Parallel Version

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

main(int argc, char *argv[])
{
    /* A Monte Carlo algorithm for calculating pi */
    int                 count;    /* points inside the unit quarter circle */
    unsigned short  xi[3];    /* random number seed */
    int                 i;          /* loop index */
    int                 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;      /* These statements 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);
}
```