Multicore Computing
Lecture04 - OpenMP

SUNG KYUN KWAN UNIVERSITY

남 범 석
bnam@skku.edu

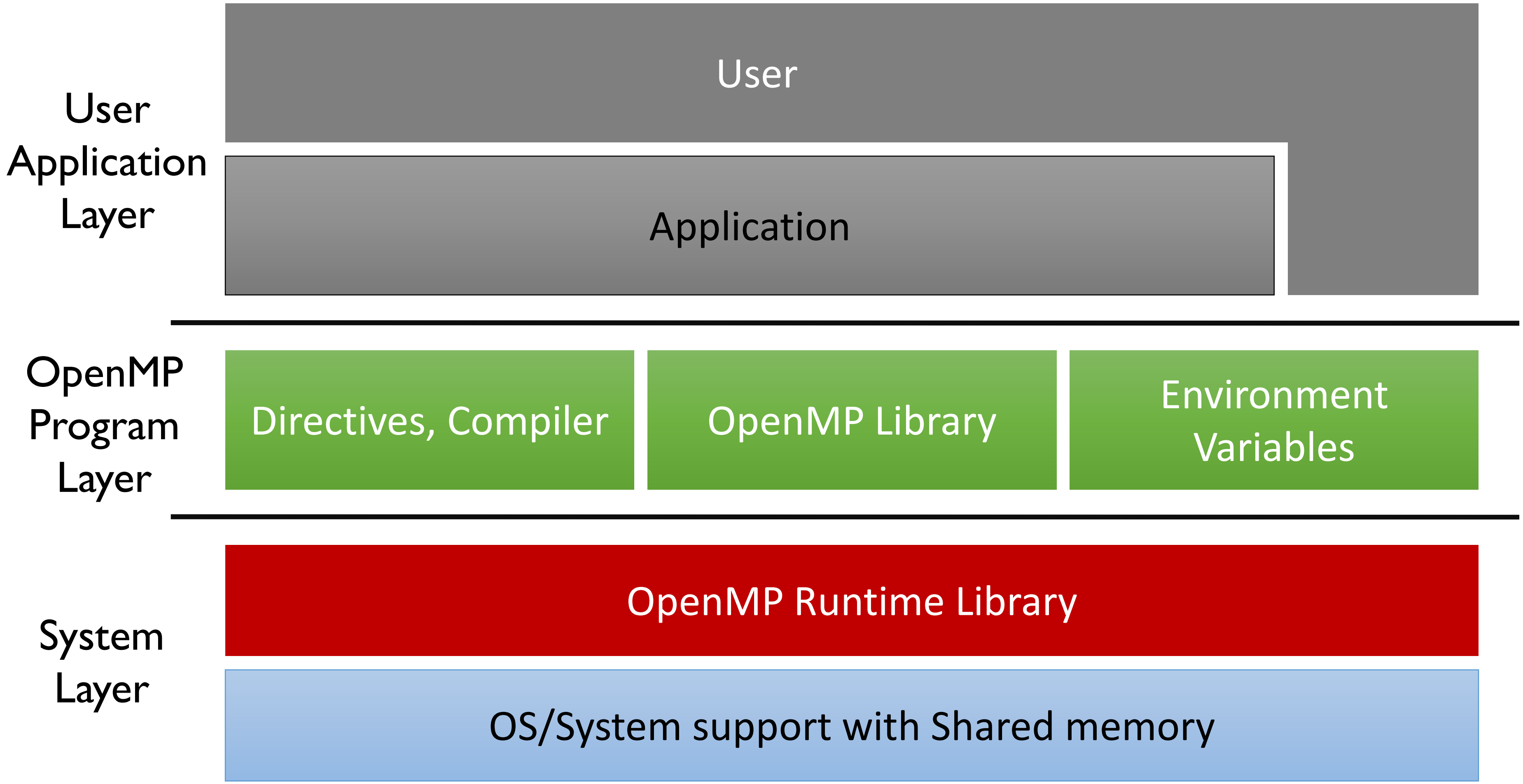
Topic Overview

- Introduction to OpenMP
- OpenMP directives
  - Concurrency control
    - parallel, for, sections
  - Synchronization
    - reduction, barrier, single, master, critical, atomic, ordered, …
  - Data handling
    - private, shared, firstprivate, lastprivate, threadprivate, …
- OpenMP library APIs
- Environment variables

OpenMP

- Open specifications for Multi Processing
- A standard for directive-based Parallel Programming
  - Shared-address space programming
  - FORTRAN, C, and C++
  - Support concurrency, synchronization, and data handling
  - Obviate the need for explicitly setting up mutexes, condition variables, data scope, and initialization

OpenMP Solution Stack



Parallel Programming Practice

- Current
  - Start with a parallel algorithm
  - Implement, keeping in mind
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - Debug ….

- Ideal way
  - Start with some algorithm
  - Implement serially, ignoring
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - Auto-magically parallelize

Implementation on Shared Memory

- Thread Library
  - Library calls
  - Low level programming
    - Explicit thread creation & work assignment
    - Explicit handling of synchronization
  - Parallelism expression
    - Task: create/join thread
    - Data: detailed programming
  - Design concurrent version from the start

- OpenMP
  - Compiler directives
  - Higher abstraction
    - Compilers convert code to use OpenMP library, which is actually implemented with thread APIs
  - Parallelism expression
    - Task: task/taskwait, parallel sections
    - Data: parallel for
  - Incremental development
    - Start with sequential version
    - Insert necessary directives

OpenMP Example

- Pragmas (compiler directives)

```
int count3s()
{
  int i, count_p;
  count=0;
  #pragma omp parallel shared(array, count, length)\
    private(count_p)
  {
    count_p=0;
    #pragma omp parallel for private(i)
    for(i=0; i<length; i++)
    {
      if(array[i]==3)
      {
        count_p++;
      }
    }
    #pragma omp critical
    {
      count+=count_p;
    }
  }
  return count;
}
```

Fork a set of threads

Parallel section executed by all threads

Join threads and resume serial code

OpenMP Example

- Threaded functions
  - Exploit data parallelism

```
node  A[N], B[N];

main()  {
    for (i=0;  i<nproc;  i++)
       thread_create(par_distance);
     for (i=0;  i<nproc;  i++)
       thread_join();
}
void par_distance() {
   tid = thread_id();     n = ceiling(N/nproc);
   s = tid * n;           e = MIN((tid+1)*n, N);
   for (i=s;  i<e;  i++)
       for (j=0;  j<N;  j++)
            C[i][j] = distance(A[i], B[j]);
}
```

- Parallel loops
  - Exploit data parallelism

```
node  A[N], B[N];

#pragma omp parallel for
for (i=0;  i<N;  i++)
     for (j=0;  j<N;  j++)
          C[i][j] = distance(A[i], B[j]);
```

Compiler Directives

- Appear as comments in your source code
  - Ignored by compilers unless you tell them otherwise

- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads

- Syntax:

```
#pragma omp <specifications>
```

Runtime Library Routines

- Routines are used for a variety of purposes
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution

- Example

```
int omp_get_num_threads(void)
```

Environment Variables

- Environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy

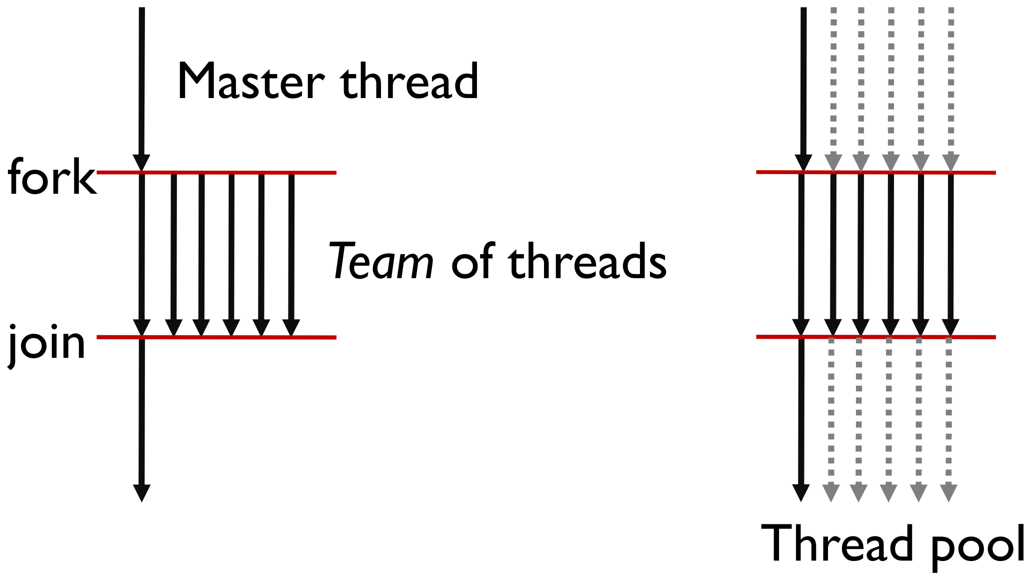
- Example

```
export OMP_NUM_THREADS=8
```

OpenMP Programming Model

- Fork-join model
  - Thread pool
  - Implicit barrier

  - #pragma omp
    - parallel for
    - parallel sections



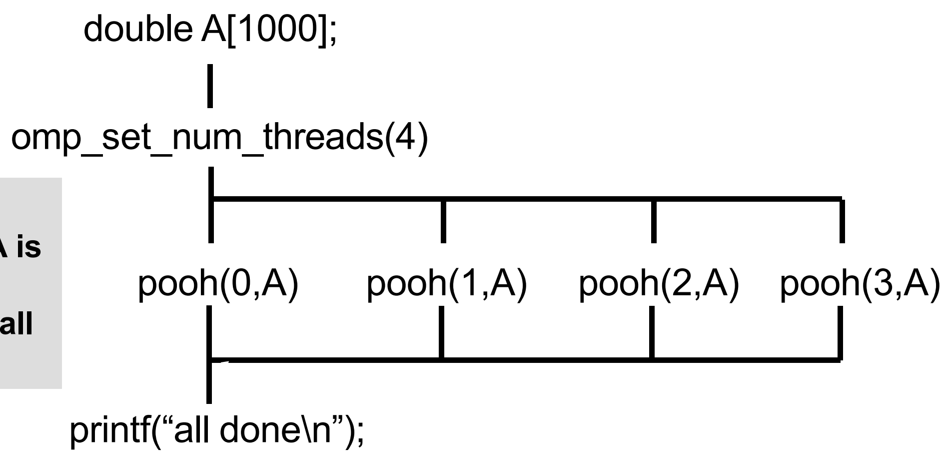
- Data scoping semantics are somewhat complicated
  - private, shared, copyin, firstprivate, lastprivate, copyprivate, threadprivate, …
  - Implicit rules,…

How to set number of threads?

- Setting of the NUM_THREADS clause
- Or, use of the omp_set_num_threads() library function
- Or, setting of the OMP_NUM_THREADS environment variable

```
double A[1000];

#pragma omp parallel num_threads(4)

{

   int ID = omp_get_thread_num();

   Pooh(ID,A);

}

printf(“all done\n”);
```

A single copy of A is shared between all threads.



Hello World in OpenMP

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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Hello World in OpenMP – con't

- Compile
  - #gcc –g –Wall –fopenmp –o omp_hello omp_hello.c
- Run
  - #./omp_hello 4

Possible outcomes

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

Pragmas

- Special compiler directives
  - **#pragma**
  - Provides extension to the basic C (or C++)
  - Compilers that don't support the pragmas **ignore** them
- OpenMP pragmas

```
#pragma omp directive [clause list]
/* structured block */
```

  - Directives specify actions OpenMP supports
  - Additional clauses follow the directive
  - Parallel directive

```
#pragma omp parallel [clause list]
```

    - Most basic parallel directive in OpenMP

parallel Directive

```
#pragma omp parallel [clause list]
```

- Possible clauses
  - Conditional Parallelization
    - `if (scalar expression)`
      - Determines whether to create threads or not
  - Degree of Concurrency
    - `num_threads (integer expression)`
      - Specifies the number of threads that are created.
  - Data Handling
    - `private (variable list)`
      - Variables local to each thread
    - `firstprivate (variable list)`
      - Variables are initialized to corresponding values before the directive
    - `lastprivate: (variable list)`
      - PRIVATE + copy from the last thread execution
    - `shared (variable list)`
      - Variables are shared across all the threads.
    - `default (shared|private|none)`
      - Default data handling specifier

parallel Directive

```
int count3s()
{
  int i, count_p;
  count=0;
  #pragma omp parallel shared(array, count, length)\
    private(count_p)
  {
    count_p=0;
    #pragma omp parallel for private(i)
    for(i=0; i<length; i++)
    {
      if(array[i]==3)
      {
        count_p++;
      }
    }
    #pragma omp critical
    {
      count+=count_p;
    }
  }
  return count;
}
```

parallel Directive

`#pragma omp parallel [clause list]`

- Possible clauses
  - Conditional Parallelization
    - `if (scalar expression)`
      - Determines whether to create threads or not
  - Degree of Concurrency
    - `num_threads (integer expression)`
      - Specifies the number of threads that are created.
  - Data Handling
    - `private (variable list)`
      - Variables local to each thread
    - `firstprivate (variable list)`
      - Variables are initialized to corresponding values before the directive
    - `lastprivate: (variable list)`
      - PRIVATE + copy from the last thread execution
    - `shared (variable list)`
      - Variables are shared across all the threads.
    - `default (shared|private|none)`
      - Default data handling specifier

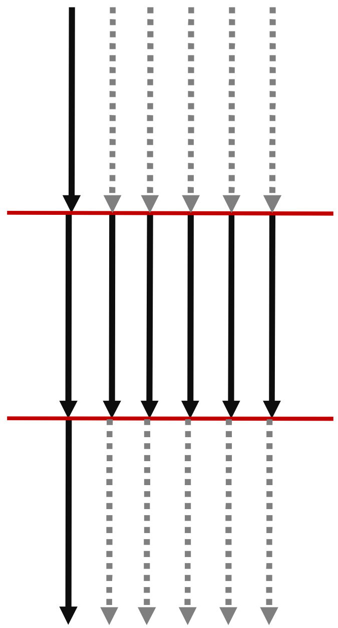
Example of **parallel** Directive

```
#pragma omp parallel if (is_parallel== 1) num_threads(8) \
  private (a) shared (b) firstprivate(c) default(none){
  /* structured block */
}
```

- `if (is_parallel==1) num_threads (8)`
  - If the value of the variable `is_parallel` equals one, eight threads are created.
- `private (a)`
  - Threads get private copy of variable a
- `firstprivate (c)`
  - private copy + initialization
  - The value of each copy of `c` is initialized to the value of `c` before the parallel directive.
- `shared (b)`
  - Threads share a single copy of variable `b`.
- `default (none)`
  - Default scope of variables are none
  - Compile error when not all variables are specified as **`shared`** or **`private`**

for Directive

- Split parallel iteration spaces (i.e., loop) across threads
- Implicit barrier at the end of a loop



- General form
  - #pragma omp for [clause list]
    - /* for loop */
- Possible clauses
  - private, firstprivate, lastprivate, reduction, schedule, nowait, and ordered.

`for` Directive example

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

```
double res[MAX]; int I;
#pragma omp parallel{
  #pragma omp for
  for (i=0;i<MAX;i++){
      res[i] = huge();
  }
}
```

```
double res[MAX]; int I;
#pragma omp parallel for
  for (i=0;i<MAX;i++){
      res[i] = huge();
  }
}
```

These are equivalent

Data Sharing: Private clause

- private(var) creates a new local copy of var for each thread
  - The value is uninitialized

```
void wrong() {
  int tmp = 0;
#pragma omp parallel for private(tmp)
  for (int j = 0; j < 1000; ++j)
     tmp += j;
  printf("%d\n", tmp);
}
```

Q: What is wrong in this code?

Data Sharing: FirstPrivate clause

- firstprivate(var) is a special case of private
  - Initializes each private copy with the corresponding value from the master thread

```
void useless() {
  int tmp = 0;
#pragma omp parallel for firstprivate(tmp)
  for (int j = 0; j < 1000; ++j)
      tmp += j;
  printf("%d\n", tmp);
}
```

Each thread gets its own tmp with an initial value of 0, but

Q: What is wrong in this code?

Data Sharing: LastPrivate clause

- Lastprivate passes the value of a private from the last iteration to a global variable.

```
void closer() {
  int tmp = 0;
#pragma omp parallel for firstprivate(tmp) lastprivate(tmp)
  for (int j = 0; j < 1000; ++j)
     tmp += j;
  printf("%d\n", tmp);
}
```

Reduction

- How do we handle this case?

```
double  ave=0.0, A[MAX];
int i;
for (i=0;i< MAX; i++) {
    ave + = A[i];
}
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can’t be trivially removed
- This is a very common situation … it is called a “reduction”.
- Support for reduction operations is included in most parallel programming environments.

reduction Clause

- A reduction
  - #pragma omp parallel for **reduction(+: sum)** num_threads(8) {

```
#pragma omp parallel for reduction(+: sum) num_threads(8) {
    for (row = 0; row < Rows; row++) sum += val;
}
```

  - Applies the same reduction operator to a sequence of operands to get a single result
  - Reduction operators → +, *, -, &, |, ^, &&, ||
  - Commutative and associative operators can provide correct results
  - All of the intermediate results of the operation should be stored in the same variable: the reduction variable

- Reduction clause in OpenMP
  - reduction(<operator>: <variable list>)
  - The variables in the list are implicitly specified as being private to threads.

Example: estimating Pi

- Estimating Pi using Monte Carlo method

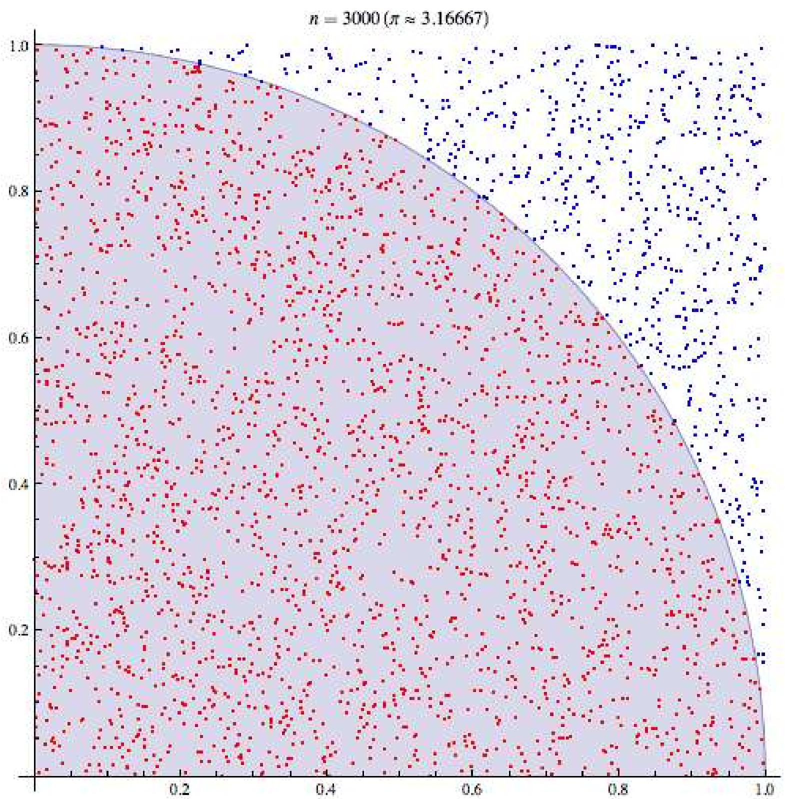


Image credit: wikipedia.org

Estimating Pi using OpenMP

```
/* ***********************************************************
An OpenMP version of a threaded program to compute PI.
******************************************************* */
#pragma omp parallel default(private) shared(npoints) \
    reduction(+: sum) num_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)rand_r(&seed)/RAND_MAX;
        rand_no_y =(double)rand_r(&seed)/RAND_MAX;
        if ( x * x + y * y < 1.0)
            sum ++;
    }
}
```

Estimating Pi using Pthreads (1)

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
    ...
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    ...
}
```

Estimating Pi using Pthreads (2)

```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double x, y;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        x =(double)rand_r(&seed)/RAND_MAX;
        y =(double)rand_r(&seed)/RAND_MAX;
        if ( x * x + y * y < 1.0 )
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Example: Estimating Pi using `for`

```
#pragma omp parallel default(private) \
  shared(npoints) reduction(+: sum) num_threads(8)
{
    sum = 0;
    #pragma omp for
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)rand_r(&seed)/RAND_MAX;
        rand_no_y =(double)rand_r(&seed)/RAND_MAX;
        if ( x * x + y * y < 1.0)
            sum ++;
    }
}
```