# Introduction to Parallel Programming with OpenMP

*By Nick D'Imperio*

BROOKHAVEN NATIONAL LABORATORY
*a passion for discovery*

U.S. DEPARTMENT OF ENERGY | Office of Science

# What will be covered

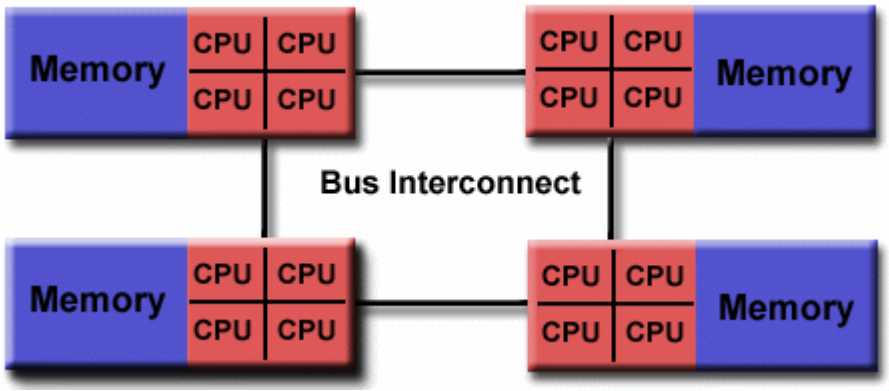
- What is OpenMP
- Getting Started with OpenMP
- Loop-level Parallelism
- Parallel Regions
- Synchronization
- Work Sharing

# Hands On Code Examples

- Hello World
- Map
- Saxpy
- Trapezoid Rule
- Monte Carlo
- Difference Eq.

# OpenMP Defined

OpenMP is a Parallel Programming Model for Shared memory and distributed shared memory multiprocessors.



# OpenMP Concepts

OpenMP is not a computer language

Works in conjuction with C/C++ or Fortran

Comprised of compiler directives and supporting library

```
#pragma omp parallel      (in C)
!$omp parallel            (in Fortran)
```

# Execution Model

Program begins execution as a single thread (master)

Master thread executes in serial until parallel construct encountered

Team of threads created which execute statements in parallel region

After parallel region, serial execution resumes with master thread

# OpenMP Directives

OpenMP directives are descriptive hints to the compiler

#pragmas in C/C++

Source code comments in Fortran

# Compiler Directive Syntax

In C

#pragma omp …

The *omp* keyword signals the pragma as OpenMP specific. Non OpenMP compilers will ignore.

In Fortran

!$omp …
c$omp …
*$omp ...

In fixed form, a line beginning with one of the above keywords and containing a space or zero in the sixth column will be treated as an OpenMP directive. It will be treated as a comment by non-OpenMP compilers.

# The Parallel Directive

The parallel directive defines a parallel region of code

In C:

```
… serial code ...

#pragma omp parallel
{
  … parallel code …
}

… serial code ...
```

In Fortran:

```
… serial code …

!$omp parallel
    … parallel code …
!$omp end parallel

… serial code ...
```

# Serial Hello World

In C:

```
#include <stdio.h>

int main()
{
  printf("hello world\n");
  return;
}
```

In Fortran:

```
PROGRAM HELLOWORLD
print *, "Hello World"
end
```

# Parallel Environment and Building Code

```
$>export CC=gcc

$>export FC=gfortran

$>export CFLAGS=-fopenmp

$>export FFLAGS=-fopenmp

$>export OMP_NUM_THREADS=4

$>gcc -fopenmp foo.c -o foo

$>gfortran -fopenmp foo.c -o foo
```

# Parallel Hello World
# Includes, Functions, and Directives

```
#include <omp.h>

#pragma omp parallel
{
   ...parallel code...
}

omp_get_thread_num()
```

```
USE OMP_LIB


!$omp parallel
    … parallel code …
!$omp end parallel

omp_get_thread_num()
```

Write a parallel Hello World that outputs "Hello World from Thread # [N]" using the above OpenMP Includes, Directives and functions.

# Parallel Hello World In C

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

int main()
{

#pragma omp parallel
  {
    int threadID = omp_get_thread_num();
    printf("%s %d\n", "hello parallel world from thread #",
threadID);
  }

  return 0;
}
```

# Parallel Hello World
# In Fortran

```
      PROGRAM HELLOWORLD
      USE OMP_LIB

!$omp parallel
      print *, "Hello Parallel World from thread #",
     &omp_get_thread_num()
!$omp end parallel
      end
```

# Mutual Exclusion and Synchronization

Threads communicate via shared variables

Access to shared variables must be controlled to avoid simultaneous writes. *Critical* directive provides exclusive thread access to variables.

Simplest form of synchronization done via *Barrier* directive. Defines a point where each thread waits for all other threads to arrive.

# Simple Loop Parallelization
# Parallel for/do directives

```
!   serial code

!$omp parallel do
   do I = 1, N
      !compute stuff
   enddo
!$omp end parallel do
```

```
/* serial code */

#pragma omp parallel for
   for(i = 0; i < N; i++)
      !compute stuff
```

# Mapping Code Example

Take a vector of real numbers and map them to exp(x^2) using *omp parallel for/do* directive.

[x1, x2, x3, ..., XN]
to
[exp(x1^2), exp(x2^2), ..., exp(xN^2)]

Use made up values for vector x, N=1000 and print the sum of the mapping on the screen. Write a serial and parallel version and compare.

# Simple Loop Parallelization MAPPING in C

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

int main()
{
  const int N = 1000;
  float sum = 0.0f;
  float x[N];
  float z[N]; /*the result*/
  int i;

  /* populate x */
  for(i = 0; i < N; i++)
    x[i] = (i+1)*.002;
```

```
  /* map */
#pragma omp parallel for
  for(i = 0; i < N; i++)
    {
      z[i] = exp(x[i]*x[i]);
    }

  /*do a sum*/
  for(i = 0; i < N; i++)
    {
      sum += z[i];
    }

  printf("%f\n", sum);

  return 0;
}
```

# Simple Loop Parallelization MAPPING in Fortran

```
    PROGRAM MADEXP
     USE OMP_LIB

     INTEGER N
     PARAMETER (N=1000)
     REAL SUM
     REAL X(N), Z(N)
     INTEGER I

! Populate X
     DO I = 1, N
       X(I) = I*.002;
     ENDDO
```

```
! Map
!$omp parallel do
     DO I = 1, N
       Z(I) = EXP(X(I)*X(I))
     ENDDO
!$omp end parallel do

! Sum up Z, store in SUM

     SUM = 0.0
     DO I = 1, N
       SUM = SUM + Z(I)
     ENDDO
```

```
     PRINT *, SUM

     END
```

# Simple Loop Parallelization (*saxpy*)

Single Precision a*x+y or *saxpy*

z(i) = a*x(i) + y(i),  (for i=1, n)

This loop has no dependences.  The result of one loop iteration does not depend on the result of any other iteration.  Iterations may be run simultaneously.

Write a code that implements SAXPY in serial and then parallel using the *parallel for/do* directive.  Use made up values to populate your vectors with a=.5 and N=1000.  Sum over the vector z and print the final sum on the screen.

# Simple Loop Parallelization (*saxpy*) in C

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

int main()
{
  const int N = 1000;
  const float a = .5f;
  float sum = 0.0f;
  float z[N], x[N], y[N];
  int i;

  for(i = 0; i < N; i++)
    {
      x[i] = (i+1)*.15;
      y[i] = (i+1)*.1;
    }
```

```
#pragma omp parallel for
  for(i = 0; i < N; i++)
    {
      z[i] = a*x[i] + y[i];
    }


  for(i = 0; i < N; i++)
    {
      sum += z[i];
    }

  printf("%f\n", sum);

  return 0;
}
```

# Simple Loop Parallelization (*saxpy*) in Fortran

```
      PROGRAM SAXPY
      USE OMP_LIB

      INTEGER N
      PARAMETER (N=1000)
      REAL A, SUM
      PARAMETER (A=.5)
      REAL Z(N), X(N), Y(N)

      INTEGER I

! put some numbers in the arrays
      DO I = 1, N
        X(I) = I*.15
        Y(I) = I*.1
      ENDDO
```

```
! saxby

!$omp parallel do
      DO I = 1, N
        Z(I) = A*X(I) + Y(I)
      ENDDO
!$omp end parallel do

! sum up Z, store in SUM

      SUM = 0.0

      DO I = 1, N
        SUM = SUM + Z(I)
      ENDDO

      PRINT *, SUM

      END
```

# Data Scoping in Simple Loop

```
  int i;
#pragma omp parallel for
  for(i = 0; i < N; i++)
    {
      z[i] = exp(x[i]*x[i]);
    }
```

X[i] is only read in the loop.
Z[i] is written but each iteration is independent.
What about the loop variable i?

Loop variable must be private to each thread.

This is the default for the *omp parallel for* directive.

The value of the loop variable is undefined after loop execution.

# Synchronization in Simple Loop

```
  int i;
#pragma omp parallel
for
  for(i = 0; i < N; i++)
    {
      z[i] = exp(x[i]*x[i]);
    }

/* omp implied barrier

  for(i = 0; i < N; i++)
    {
      Sum += z[i];
    }
```

Sum depends on all z values having completed writing at the end of the parallel loop.

OpenMP has an implied *barrier* call at the end of the *parallel for* directive.

At the end of the first loop, the parent thread waits for all child threads to complete. Parent thread resumes serial execution after the implied *barrier*.

# Shared and Private Clauses

```
#pragma omp parallel private (private_sum)
  {
    private_sum = 0.0;

#pragma omp for
    for(i = 0; i < N; i++)
      {
        private_sum += z[i];
      }
  }
```

```
      !$omp parallel private (private_sum)
        private_sum = 0.0

      !$omp do
        DO I = 0, N
           private_sum = private_sum + z(I)
        ENDDO
      !$omp end parallel
```

Directives may have clauses to define data scope of variables.

Shared scope clause specifies that the named variables are shared by all threads in the parallel construct. Variables are shared by default.

Private scope clause specifies that the named variables are private to each thread in the parallel construct. Private variables are undefined upon entry and exit from parallel construct.

*In examples to the left, private_sum is a private variable and z is shared.*

# Shared and Private Clauses cont. and the Critical Directive

```
  float sum = 0.0;
#pragma omp parallel private (private_sum) shared (sum)
  {
    private_sum = 0.0;

#pragma omp for
    for(i = 0; i < N; i++)
      {
        private_sum += z[i];
      }

#pragma critical
    {
      sum = sum + private_sum;
    }

  }
```

Parallel Reduction example.

*critical* directive restricts execution of block to one thread at a time.

# Shared and Private Clauses cont. and the Critical Directive

```
  REAL sum = 0.0
!$omp parallel private (private_sum) shared (sum)
  private_sum = 0.0

!$omp do
  DO I = 0, N
     private_sum = private_sum + z(I)
  ENDDO

!$omp critical
  sum =  sum + private_sum
!$omp end critical
!$omp end parallel
```

Parallel Reduction example.

*critical* directive restricts execution of block to one thread at a time.

# Firstprivate and Lastprivate Clauses

```
    float private_sum = 0.0;

#pragma omp parallel for firstprivate (private_sum) lastprivate (private_sum)
    for(i = 0; i < N; i++)
      {
        private_sum += z[i];
      }
```

*firstprivate* clause initializes the private variable with the value of the master thread's copy upon entry.

*lastprivate* clause saves the last iteration value of the variable to the master thread's copy upon exit.

# Firstprivate and Lastprivate Clauses

```
   private_sum = 0.0

!$omp parallel do firstprivate (private_sum)
lastprivate (private_sum)
  DO I = 0, N
     private_sum = private_sum + z(I)
  ENDDO

!$omp end parallel do
```

*firstprivate* clause initializes the private variable with the value of the master thread's copy upon entry.

*lastprivate* clause saves the last iteration value of the variable to the master thread's copy upon exit.

# Caveats on Parallel loops

```
    float sum = 0.0;

#pragma omp parallel for
reduction (+:sum)
    for(i = 0; i < N; i++)
     {
        sum += z[i];
     }
```

```
     sum = 0.0

  !$omp parallel do reduction
  (+:sum)
    DO I = 0, N
      sum = private_sum + z(I)
    ENDDO

  !$omp end parallel do
```

Parallel do/for loops must be followed immediately by a do/for loop.

In fortran, it must be index controlled (do-while is not allowed).

In C, the for loop must be in standard form and the start and end values of the loop must not change during iteration.

All iterations of the loop must complete. No goto or break statements **out** of the loop are allowed.

# OpenMP Runtime Library

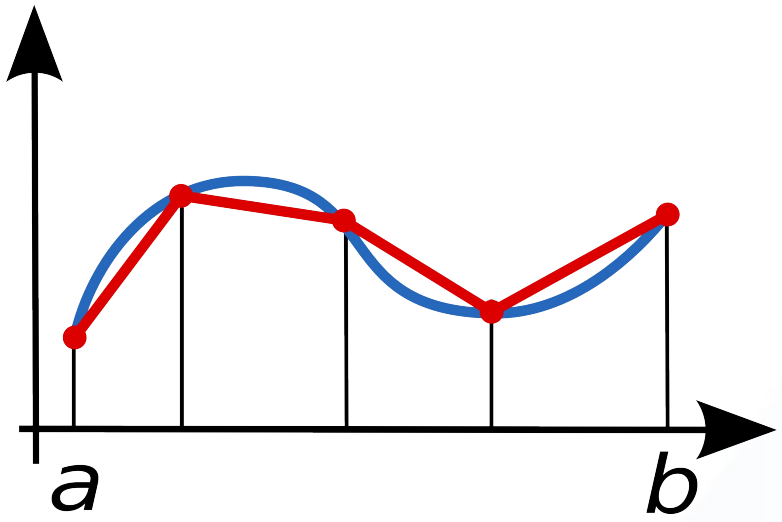
*omp_get_num_threads* returns the number of threads executing in the parallel region.

*omp_get_thread_num* returns the thread ID of calling thread. Master thread has ID=0.

*omp_set_num_threads(int)* sets the number of threads to use. Must be called from a serial portion of the code.

*omp_get_max_threads* returns the maximum number of threads available to parallel regions.

# Trapezoid Rule




$$I = h*[f(x_0)/2 + f(x_n)/2 + f(x_1) + \ldots f(x_{n-1})]$$

# Trapezoid Rule Serial Code

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

double f(double x)
{
  return exp(x*x);
}

int main()
{
  double integral;     /*definite integral*/
  const double a=0.0;  /*left end point*/
  const double b=1.0;  /*right end point*/
  const int N=100000; /*subdivisions*/
  double h; /*base width of subdivision*/
  double x;
  int i;
```

```
  h = (b-a)/N;
  integral = (f(a)+f(b))/2.0;
  x = a;

  for(i = 1; i <= N-1; i++)
   {
     x = x+h;
     integral = integral + f(x);
   }

  integral = integral*h;

  printf("%s%d%s%f\n", "WITH N=", N,
         " TRAPEZOIDS, INTEGRAL=",
         integral);

  return 0;
}
```

# Trapezoid Rule Serial Code

```
PROGRAM TRAP
DOUBLE PRECISION INTEG
DOUBLE PRECISION A, B     !END POINTS
PARAMETER (A=0.0, B=1.0)  !LIMITS

INTEGER N   !NUMBER OF SUBDIVISION
PARAMETER (N=50000000)

!BASE WIDTH OF SUBDIVISION
DOUBLE PRECISION H
DOUBLE PRECISION X
INTEGER I

!FUNCTION TO INTEGRATE
DOUBLE PRECISION F

H = (B-A)/N
INTEG = (F(A)+F(B))/2.0
X = A
```

```
    DO 10 I=1,N-1,1
      X=X+H
      INTEG = INTEG + F(X)
10  CONTINUE

    INTEG = INTEG*H

    PRINT *, "WITH N=", N,
   &"TRAPEZOIDS, INTEGRAL=",
   &INTEG

    END




    FUNCTION F(X)
    DOUBLE PRECISION X,F
    F = EXP(X*X)
    END
```

# Trapezoid Rule Parallel Code in C

```
#include <stdio.h>
#include <math.h>
#include <omp.h>   /*openmp api*/

double f(double x)
{
  return exp(x*x);
}

int main()
{
  double integral, integral_priv;
  const double a=0.0;  /*left end point*/
  const double b=1.0;  /*right end point*/
  const int N=10; /*subdivisions*/
  double h;  /*width of subdivision*/
  double x;
  int i;
```

```
h = (b-a)/N;
integral = 0.0;
integral_priv = 0.0;
```

# Trapezoid Rule Parallel Code in C

```
#pragma omp parallel firstprivate(x, integral_priv) shared(integral)
  {
#pragma omp for
    for(i = 1; i <= N-1; i++)
      {
        x = a+i*h;
        integral_priv = integral_priv + f(x);
      }

#pragma omp critical
    integral = integral+integral_priv;
  }

  integral = (integral+(f(a)+f(b))/2.0)*h;

  printf("%s%d%s%f\n", "WITH N=", N,
         " TRAPEZOIDS, INTEGRAL=",
         integral);

  return 0;
}
```

# Trapezoid Rule Parallel Code in Fortran

```
PROGRAM TRAP
USE OMP_LIB

DOUBLE PRECISION INTEG, TMPINT  !DEFINITE INTEGRAL RESULT
DOUBLE PRECISION A, B      !END POINTS
PARAMETER (A=0.0, B=1.0)  !LIMITS

INTEGER N      !NUMBER OF SUBDIVISION
PARAMETER (N=10)

DOUBLE PRECISION H        !BASE WIDTH OF SUBDIVISION
DOUBLE PRECISION X
INTEGER I

DOUBLE PRECISION F        !FUNCTION TO INTEGRATE
```

# Trapezoid Rule Parallel Code in Fortran

```
      H = (B-A)/N
      INTEG = 0.0
      TMPINT = 0.0

!$omp parallel firstprivate(X, TMPINT) shared(INTEG)

!$omp do
      DO 10 I=1,N-1,1
         X=A+I*H
         TMPINT = TMPINT + F(X)
 10   CONTINUE
!$omp end do

!$omp critical
     INTEG = INTEG + TMPINT
!$omp end critical

!$omp end parallel
```

# Trapezoid Rule Parallel Code in Fortran

```
      INTEG = (INTEG+(F(A)+F(B))/2.0)*H

      PRINT *, "WITH N=", N, "TRAPEZOIDS, INTEGRAL=", INTEG

      END



      FUNCTION F(X)
      DOUBLE PRECISION X
      F = EXP(X*X)
      END
```

# Reduction Clause

```
   float sum = 0.0;

#pragma omp parallel for reduction (+:sum)
   for(i = 0; i < N; i++)
     {
        sum += z[i];
     }
```

```
  sum = 0.0

!$omp parallel do reduction (+:sum)
  DO I = 1, N
     sum = private_sum + z(I)
  ENDDO

!$omp end parallel do
```

*reduction* clause parallelizes reductions using a commutative-associative operator.

The syntax is

*reduction (*red_op : var_list)

Operators in C include +,-,*,&&,||

Operators in Fortran include +,-,*,.AND.,.OR.,MIN,MAX

# Trapezoid Rule Parallel Code in C Using Reduction Clause

```
#include <stdio.h>
#include <math.h>
#include <omp.h>   /*openmp api*/

double f(double x)
{
  return exp(x*x);
}

int main()
{
  double integral;     /*definite integral result*/
  const double a=0.0;  /*left end point*/
  const double b=1.0;  /*right end point*/
  const int N=10; /*number of subdivisions*/
  double h;            /*base width of subdivision*/
  double x;
  int i;
```

# Trapezoid Rule Parallel Code in C Using Reduction Clause

```
  h = (b-a)/N;
  integral = 0.0;

#pragma omp parallel for private(x) reduction(+:integral)
  for(i = 1; i <= N-1; i++)
    {
      x = a+i*h;
      integral = integral + f(x);
    }

  integral = (integral+(f(a)+f(b))/2.0)*h;

  printf("%s%d%s%f\n", "WITH N=", N, " TRAPEZOIDS, INTEGRAL=", integral);

  return 0;
}
```

# Trapezoid Rule Parallel Code in Fortran Using Reduction Clause

```
PROGRAM TRAP
USE OMP_LIB

DOUBLE PRECISION INTEG  !DEFINITE INTEGRAL RESULT
DOUBLE PRECISION A, B      !END POINTS
PARAMETER (A=0.0, B=1.0)  !LIMITS

INTEGER N      !NUMBER OF SUBDIVISION
PARAMETER (N=10)

DOUBLE PRECISION H        !BASE WIDTH OF SUBDIVISION
DOUBLE PRECISION X
INTEGER I

DOUBLE PRECISION F        !FUNCTION TO INTEGRATE

H = (B-A)/N
INTEG = 0.0
```

# Trapezoid Rule Parallel Code in Fortran Using Reduction Clause

```
!$omp parallel do private(X) reduction(+:INTEG)
      DO 10 I=1,N-1,1
         X=A+I*H
         INTEG = INTEG + F(X)
 10   CONTINUE
!$omp end parallel do

      INTEG = (INTEG+(F(A)+F(B))/2.0)*H

      PRINT *, "WITH N=", N, "TRAPEZOIDS, INTEGRAL=", INTEG

      END


     FUNCTION F(X)
     DOUBLE PRECISION X
     F = EXP(X*X)
     END
```

# Random Number Generator

```
#include <stdio.h>

unsigned int seed = 1;  /* random number seed */
const unsigned int rand_max = 32768;

double rannum()
{
  unsigned int rv;
  seed = seed * 1103515245 + 12345;
  rv = ((unsigned)(seed/65536) % rand_max);
  return (double)rv/rand_max;
}
```

```
int main()
{
  const int N = 10;
  int i;

  for(i = 0; i < N; i++)
    {
      printf("%g\n",rannum());
    }

  return;
}
```

Random number between 0 and 1

***seed must never be initialized to zero**

# Random Number Generator

```
    PROGRAM RANGEN
    INTEGER SEED   !RANDOM SEED
    COMMON /RAND/ SEED
    INTEGER N  !# of RANDOMS
    PARAMETER (N=10)
    DOUBLE PRECISION RANNUM
    INTEGER I !LOOP INDEX

    SEED = 1

    DO 10 I=1, N, 1
       PRINT *, RANNUM()
10   CONTINUE

    END
```

```
  DOUBLE PRECISION
&FUNCTION RANNUM()
  INTEGER SEED
  COMMON /RAND/ SEED
  SEED = SEED*65539
  IF(SEED .LT. 0) SEED =
&(SEED+1)+2147483647
  RANNUM = SEED * 0.4656613E-9
  END
```

Random number between 0 and 1

***seed must never be initialized to zero**

# Threadprivate Directive

The *threadprivate* directive identifies a global variable or common block as being private to each thread. In essence, it's similar to the *private* clause except it applies to the entire program and not just a parallel region.

# Threadprivate Directive cont.

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

unsigned int seed = 1;  /* random seed */
const unsigned int rand_max = 32768;

double rannum()
{
#pragma omp threadprivate(seed)
  unsigned int rv;
  seed = seed * 1103515245 + 12345;
  rv = ((unsigned)(seed/65536) % rand_max);

  return (double)rv/rand_max;
}
```

# Threadprivate Directive cont.

```
int main()
{
  const int N = 10;  /*# of random
numbers*/
  int i;

#pragma omp threadprivate(seed)

#pragma omp parallel
  {
    seed = omp_get_thread_num()+1;
```

```
#pragma omp for
    for(i = 0; i < N; i++)
      {
        printf("%d\t%g\n",
              omp_get_thread_num(),
              rannum());
      }
  }

  return;
}
```

# Threadprivate Directive cont.

```
      PROGRAM RANSIM
      USE OMP_LIB
      INTEGER SEED   !RANDOM SEED
      COMMON /RAND/ SEED
!$OMP THREADPRIVATE(/RAND/)
      INTEGER N  !NUMBER OF RANDOM
NUMBERS
      PARAMETER (N=10)
      DOUBLE PRECISION RANNUM
      INTEGER I !LOOP INDEX

      SEED = 1
!$OMP PARALLEL
!SEED CAN'T BE ZERO
      SEED = OMP_GET_THREAD_NUM()+1
```

# Threadprivate Directive cont.

```
!$OMP DO
      DO 10 I=1, N, 1
         PRINT *, OMP_GET_THREAD_NUM(), RANNUM()
 10   CONTINUE
!$OMP END DO
!$OMP END PARALLEL
      END

      DOUBLE PRECISION FUNCTION RANNUM()
      INTEGER SEED
      COMMON /RAND/ SEED
      SEED = SEED*65539
      IF(SEED .LT. 0) SEED = (SEED+1)+2147483647
      RANNUM = SEED * 0.4656613E-9
      END
```

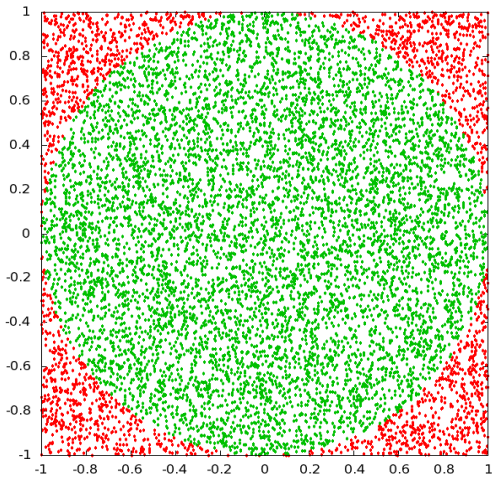
# Monte Carlo to Calculate Pi

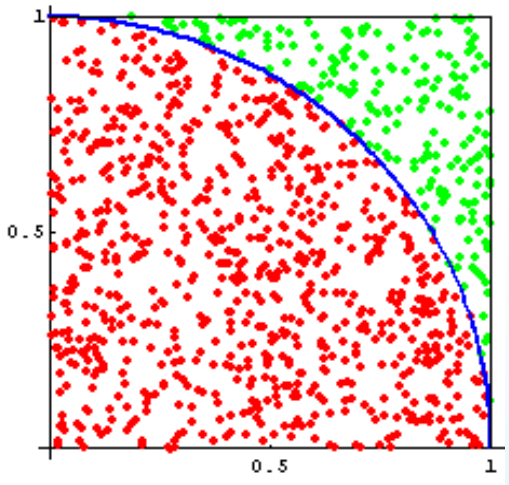
$$A_c / A_s = \pi * r^2 / (2 * r)^2$$

When r=1

$$A_c = A_s * \pi / 4$$

$$\pi = 4 * A_c / A_s$$




If we randomly assign points inside the unit square and take the ratio of points that fall inside the circle to the total number of points, we can calculate $\pi$ with the following formula: $\pi = 4 * N / M$

# Monte Carlo to Calculate Pi cont.

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

unsigned int seed = 1;  /* random number seed */
const unsigned int rand_max = 32768;

double rannum()
{
#pragma omp threadprivate(seed)
  unsigned int rv;
  seed = seed * 1103515245 + 12345;
  rv = ((unsigned)(seed/65536) % rand_max);

  return (double)rv/rand_max;
}
```

# Monte Carlo to Calculate Pi cont.

```
int main()
{
  const int N = 100000000;      /* number of
randoms */
  const double r = 1.0;  /* radius of unit circle */

  int i;

  double x, y;   /* function inputs */
  double sum = 0.0;
  double Q = 0.0;

#pragma omp threadprivate(seed)

#pragma omp parallel
  {
    seed = omp_get_thread_num()+1;
  }
```

# Monte Carlo to Calculate Pi cont.

```
#pragma omp parallel for private(x,y) reduction(+:sum)
  for(i = 0; i < N; i++)
    {
      /* random number, can't use library function, not thread safe */
      x = rannum();
      y = rannum();

      if((x*x + y*y) < r)
        {
          sum = sum+1.0;
        }
    }

  Q = 4.0*sum*1.0/N;

  printf("%.9g\n", Q);

  return;
}
```

# Monte Carlo to Calculate Pi cont.

```
      PROGRAM MONTECARLO
      USE OMP_LIB
      INTEGER SEED   !RANDOM NUMBER SEED
      COMMON /RAND/ SEED
!$OMP THREADPRIVATE(/RAND/)
      INTEGER N  !NUMBER OF RANDOM NUMBERS
      PARAMETER (N=100000000)
      !RANDOM NUMBUR GENERATOR
      DOUBLE PRECISION RANNUM
      DOUBLE PRECISION X,Y,SUM,Q
      DOUBLE PRECISION RAD !RADIUS
      PARAMETER (RAD=1.0)
      INTEGER I !LOOP INDEX
```

# Monte Carlo to Calculate Pi cont.

```
      SUM = 0.0
      Q = 0.0

      SEED = 1
!$OMP PARALLEL
      !SEED CAN'T BE ZERO
      SEED = OMP_GET_THREAD_NUM()+1

!$OMP DO PRIVATE(X,Y) REDUCTION(+:SUM)
      DO 10 I=1, N, 1
        X = RANNUM()
        Y = RANNUM()

        IF((X*X + Y*Y) .LT. RAD) THEN
          SUM = SUM+1.0
        ENDIF

 10   CONTINUE
!$OMP END DO
!$OMP END PARALLEL
```

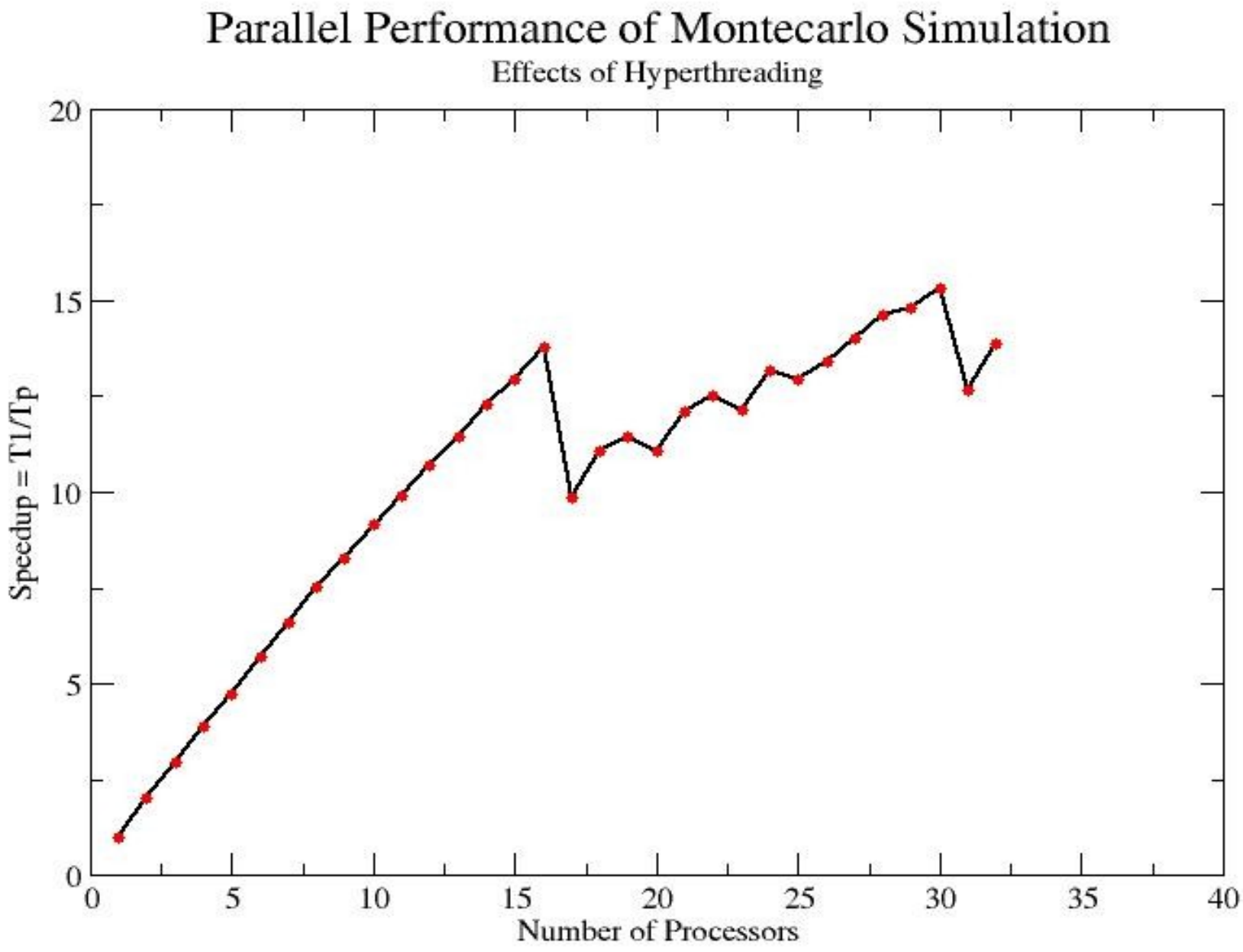
# Monte Carlo to Calculate Pi cont.

```
Q = 4.0*SUM*1.0/N

PRINT *, Q

END

DOUBLE PRECISION FUNCTION RANNUM()
INTEGER SEED
COMMON /RAND/ SEED
SEED = SEED*65539
IF(SEED .LT. 0) SEED = (SEED+1)+2147483647
RANNUM = SEED * 0.4656613E-9
END
```


Parallel Performance of Montecarlo Simulation
Effects of Hyperthreading
Speedup = T1/Tp
Number of Processors
0
5
10
15
20
25
30
35
40

# Data Dependencies, Recurrences

```
for(i = 0; i < N-1; i++)
  {
    y[i] = y[i+1] – y[i];
  }
```

For N=4 and 2 Threads …

| Iteration | Thread 1 | Thread2 |
|---|---|---|
| 0 | y[0]=y[1]-y[0] | y[2]=y[3]-y[2] |
| 1 | y[1]=y[2]-y[1] | no-op |

In Iteration 1, Thread 1 reads y[2] which has already been written by Thread 2.

# Forward Difference, 1st Derivative C

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

int main()
{
  const int N = 1000000;
  int i;
  const double h = 0.00001;
  double y[N];
  const int prune = 1000;

  for(i = 0; i < N; i++)
    y[i] = sin(i*h);

  for(i = 0; i < N; i++)
    {
      if(i%prune == 0)
       printf("%g\t%g\n", i*h, y[i]);
    }

  for(i = 0; i < N - 1; i++)
    {
      y[i] = (y[i+1]-y[i])/h;
    }

  y[N-1] = y[N-2];

  printf("\n\n");

  for(i = 0; i < N; i++)
    {
      if(i%prune == 0)
       printf("%g\t%g\n", i*h, y[i]);
    }

  return 0;
}
```

# Forward Difference, 1st Derivative Fortran

```
      PROGRAM FDIFF
      INTEGER N
      PARAMETER (N=1000000)
      DOUBLE PRECISION H
      PARAMETER (H=0.00001)
      DOUBLE PRECISION Y(N)
      INTEGER PRUNE
      PARAMETER (PRUNE=1000)
      INTEGER I !LOOP INDEX

      DO 10 I=1, N, 1
        Y(I) = SIN(i*H)
10 CONTINUE

      DO 20 I=1, N, 1
        IF(MOD(I,PRUNE) .EQ. 0) PRINT *, I*H, Y(I)
20 CONTINUE
```

# Forward Difference, 1st Derivative Fortran

```
      DO 30 I=1, N-1, 1
        Y(I) = (Y(I+1)-Y(I))/H
30 CONTINUE

      Y(N) = Y(N-1)

      PRINT *
      PRINT *

      DO 40 I=1, N, 1
        IF(MOD(I,PRUNE) .EQ. 0) PRINT *, I*H, Y(I)
40 CONTINUE

      END
```

# Forward Difference, 1st Derivative, C, OpenMP

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

int main()
{
  const int N = 1000000;
  int i;
  const double h = 0.00001;
  double y[N];
  const int prune = 1000;

#pragma omp parallel for
  for(i = 0; i < N; i++)
    y[i] = sin(i*h);

  for(i = 0; i < N; i++)
    {
      if(i%prune == 0)
       printf("%g\t%g\n", i*h, y[i]);
    }
```

```
#pragma omp parallel private(i)
  {
    int N_local;
    int index;
    double next;

    N_local = N/omp_get_num_threads();
    if(omp_get_thread_num() ==
       omp_get_num_threads()-1)
     {
        /*last thread takes extra*/
        N_local = N_local +
       N%omp_get_num_threads();
     }

    if(omp_get_thread_num() !=
       omp_get_num_threads()-1)
      index = omp_get_thread_num()*N_local;
    else
      index = omp_get_thread_num()*
                  (N/omp_get_num_threads());
```

# Forward Difference, 1st Derivative, C, OpenMP

```
    //last thread shouldn't over run array
    if(omp_get_thread_num() ==
       omp_get_num_threads()-1)
      N_local--;

    next = y[index+N_local];

#pragma omp barrier
    for(i = 0; i < N_local-1; i++)
      {
       y[index] = (y[index+1]-y[index])/h;
       index++;
      }

    y[index] = (next - y[index])/h;

    index++;
```

```
    if(omp_get_thread_num() ==
       omp_get_num_threads()-1)
      y[index] = y[index-1];
  }
  printf("\n\n");

for(i = 0; i < N; i++)
    {
      if(i%prune == 0)
        printf("%g\t%g\n", i*h, y[i]);
    }

  return 0;
}
```

# Forward Difference, 1st Derivative Fortran, OpenMP

```
PROGRAM FDIFF
USE OMP_LIB
INTEGER N
PARAMETER (N=1000000)
DOUBLE PRECISION H
PARAMETER (H=0.00001)
DOUBLE PRECISION Y(N)
INTEGER PRUNE
PARAMETER (PRUNE=1000)
INTEGER I !LOOP INDEX
INTEGER N_LOCAL
INTEGER INDEX
DOUBLE PRECISION NEXT
```

```
!$OMP PARALLEL DO
      DO 10 I=1, N, 1
         Y(I) = SIN((I-1)*H)
 10 CONTINUE
!$OMP END PARALLEL DO

     DO 20 I=1, N, 1
        IF (MOD(I,PRUNE) .EQ. 0) THEN
          PRINT *, (I-1)*H, Y(I)
        ENDIF
 20 CONTINUE
```

# Forward Difference, 1st Derivative Fortran, OpenMP

```
!$OMP PARALLEL PRIVATE (I, N_LOCAL, INDEX, NEXT)
      N_LOCAL = N/OMP_GET_NUM_THREADS()

      IF (OMP_GET_THREAD_NUM() .EQ. OMP_GET_NUM_THREADS()-1) THEN
        !Last thread takes extra
        N_LOCAL = N_LOCAL + MOD(N,OMP_GET_NUM_THREADS())
      ENDIF

      IF(OMP_GET_THREAD_NUM() .NE. OMP_GET_NUM_THREADS()-1) THEN
        INDEX = OMP_GET_THREAD_NUM() * N_LOCAL + 1
      ELSE
        INDEX = OMP_GET_THREAD_NUM()*(N/OMP_GET_NUM_THREADS())+1
      ENDIF

      !last thread shouldn't over run array
      IF (OMP_GET_THREAD_NUM() .EQ. OMP_GET_NUM_THREADS()-1) THEN
        N_LOCAL = N_LOCAL-1
      ENDIF

      NEXT = Y(INDEX+N_LOCAL)
```

# Forward Difference, 1st Derivative Fortran, OpenMP

```
!$OMP BARRIER
      DO 30 I=1, N_LOCAL-1, 1
         Y(INDEX) = (Y(INDEX+1)-Y(INDEX))/H
         INDEX = INDEX+1
 30 CONTINUE

      Y(INDEX) = (NEXT - Y(INDEX))/H
      INDEX = INDEX+1

      IF(OMP_GET_THREAD_NUM() .EQ. OMP_GET_NUM_THREADS()-1) THEN
        Y(INDEX) = Y(INDEX-1)
      ENDIF

!$OMP END PARALLEL
```

# Forward Difference, 1st Derivative Fort, OpenMP Cont.

```
      PRINT *
      PRINT *

      DO 40 I=1, N, 1
        IF(MOD(I,PRUNE) .EQ. 0) PRINT *, (I-1)*H, Y(I)
40 CONTINUE

      END
```