Lecture 10: Introduction to OpenMP (Part 1)

What is OpenMP

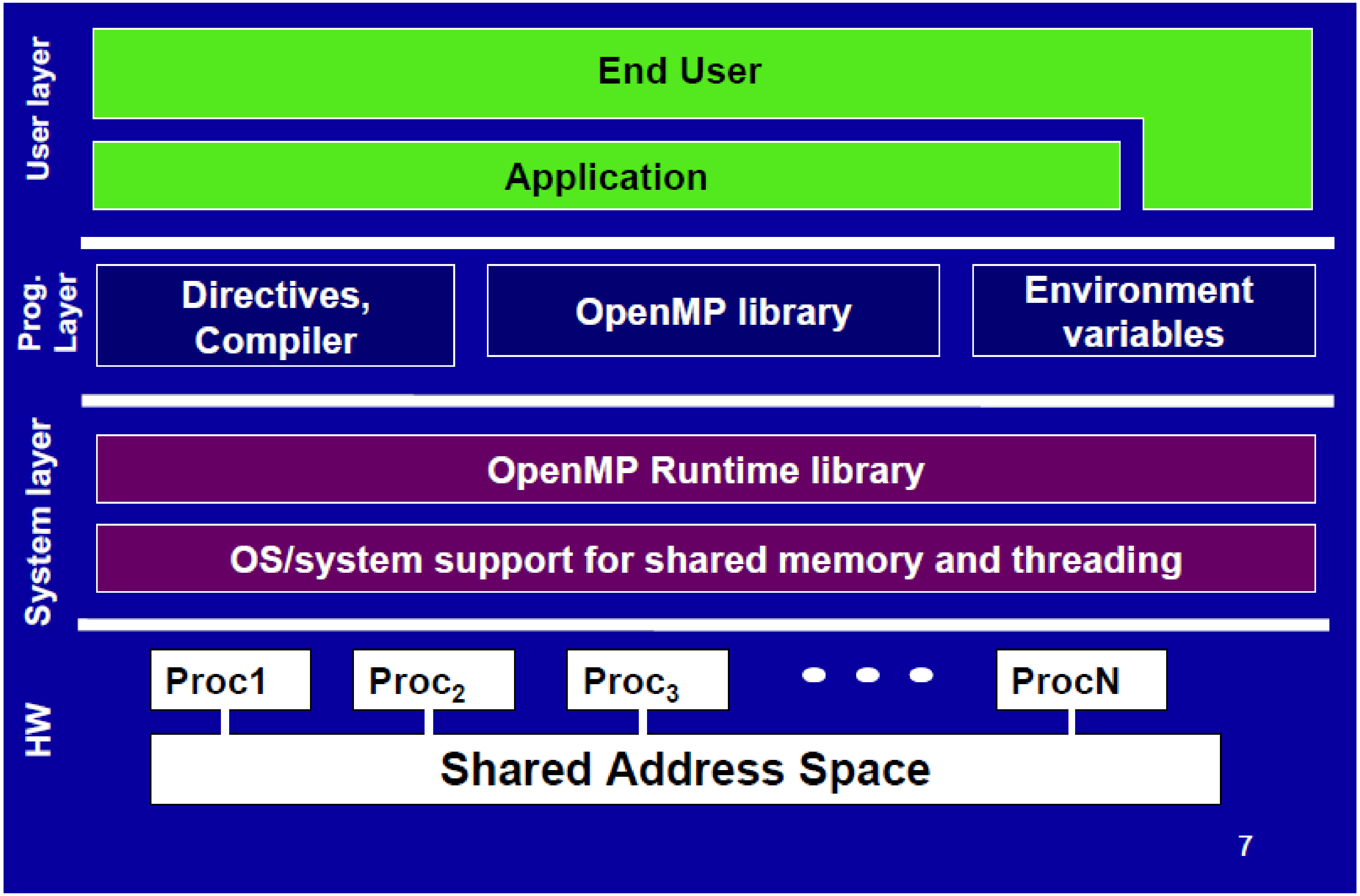
Open specifications for Multi Processing

Long version: Open specifications for MultiProcessing via collaborative work between interested parties from the hardware and software industry, government and academia.

- An Application Program Interface (API) that is used to explicitly direct multi-threaded, shared memory parallelism.
- API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portability
  - API is specified for C/C++ and Fortran
  - Implementations on almost all platforms including Unix/Linux and Windows
- Standardization
  - Jointly defined and endorsed by major computer hardware and software vendors
  - Possibility to become ANSI standard

Brief History of OpenMP

- In 1991, Parallel Computing Forum (PCF) group invented a set of directives for specifying loop parallelism in Fortran programs.
- X3H5, an ANSI subcommittee developed an ANSI standard based on PCF.
- In 1997, the first version of OpenMP for Fortran was defined by OpenMP Architecture Review Board.
- Binding for C/C++ was introduced later.
- Version 3.1 is available since 2011.


User layer
End User
Application
Prog. Layer
Directives, Compiler
OpenMP library
Environment variables
System layer
OpenMP Runtime library
OS/system support for shared memory and threading
HW
Proc1
$Proc_2$
$Proc_3$
• • •
ProcN
Shared Address Space
7

Thread

- A **process** is an instance of a computer program that is being executed. It contains the program code and its current activity.
- A **thread of execution** is the smallest unit of processing that can be scheduled by an operating system.
- Differences between threads and processes:
  - A thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory. The threads of a process share the latter's instructions (code) and its context (values that its variables reference at any given moment).
  - Different processes do not share these resources.

http://en.wikipedia.org/wiki/Process_(computing)

Process

- A process contains all the information needed to execute the program
  - Process ID
  - Program code
  - Data on run time stack
  - Global data
  - Data on heap

  Each process has its own address space.
- In multitasking, processes are given time slices in a round robin fashion.
  - If computer resources are assigned to another process, the status of the present process has to be saved, in order that the execution of the suspended process can be resumed at a later time.

Threads

- Thread model is an extension of the process model.
- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure.
- Threads of a process share the address space of this process.
  - Global variables and all dynamically allocated data objects are accessible by all threads of a process
- Each thread has its own run time stack, register, program counter.
- Threads can communicate by reading/writing variables in the common address space.

OpenMP Programming Model

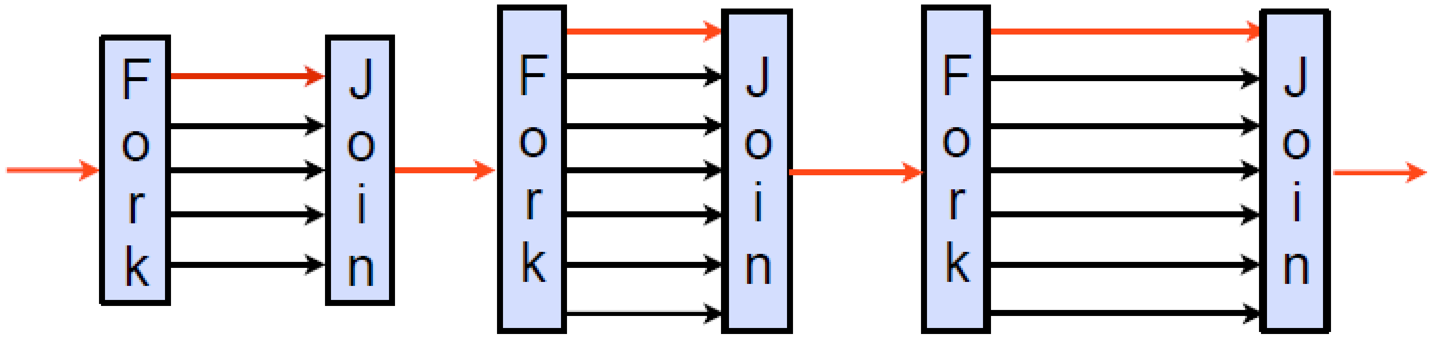
- Shared memory, thread-based parallelism
  - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm.
  - A shared memory process consists of multiple threads.
- Explicit Parallelism
  - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model.
- Compiler directive based
  - Most OpenMP parallelism is specified through the use of compiler directives which are embedded in the source code.

OpenMP is not

- Necessarily implemented identically by all vendors
- Meant for distributed-memory parallel systems (it is designed for shared address spaced machines)
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel.

Fork-Join Parallelism

- OpenMP program begin as a single process: the *master thread*. The master thread executes sequentially until the first *parallel region* construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by **FORK**.
  - Becomes the master of this group of threads, and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the *parallel region* construct are then executed in parallel among these threads.
- **JOIN**: When the threads complete executing the statement in the *parallel region* construct, they synchronize and terminate, leaving only the master thread.



Master thread is shown in red.

I/O

- OpenMP does not specify parallel I/O.
- It is up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Memory Model

- Threads can “cache” their data and are not required to maintain exact consistency with real memory all of the time.
- When it is critical that all threads view a shared variable identically, <u>the programmer is responsible</u> for insuring that the variable is updated by all threads as needed.

OpenMP Code Structure

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

int main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("Hello (%d)\n", ID);
        printf(" world (%d)\n", ID);
    }
}
```

Set # of threads for OpenMP
In csh
```
setenv OMP_NUM_THREAD 8
```

Compile: `g++  -fopenmp  hello.c`

Run: `./a.out`

See: http://wiki.crc.nd.edu/wiki/index.php/OpenMP

OpenMP Core Syntax

```
#include “omp.h”
int 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 . . .
}
```

OpenMP C/C++ Directive Format

OpenMP directive forms

- C/C++ use compiler directives
  - Prefix: #pragma omp …
- A directive consists of a directive name followed by *clauses*

  Example: #pragma omp parallel default (shared) private (var1, var2)

OpenMP Directive Format (2)

General Rules:

- Case sensitive
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be “continued” on succeeding lines by escaping the newline character with a backslash “\” at the end of a directive line.

OpenMP parallel Region Directive

#pragma omp parallel [*clause list*]

Typical clauses in [*clause list*]

- Conditional parallelization
  - if (scalar expression)
    - Determine whether the parallel construct creates threads
- Degree of concurrency
  - num_threads (integer expresson)
    - number of threads to create
- Date Scoping
  - private (variable list)
    - Specifies variables local to each thread
  - firstprivate (variable list)
    - Similar to the private
    - Private variables are initialized to variable value before the parallel directive
  - shared (variable list)
    - Specifies variables that are shared among all the threads
  - default (data scoping specifier)
    - Default data scoping specifier may be shared or none

Example:

```
#pragma omp parallel if (is_parallel == 1)  num_threads(8) shared (var_b)
private (var_a)   firstprivate (var_c) default (none)
{
/* structured block */
}
```

- if (is_parallel == 1) num_threads(8)
  - If the value of the variable is_parallel is one, create 8 threads
- shared (var_b)
  - Each thread shares a single copy of variable b
- private (var_a)   firstprivate (var_c)
  - Each thread gets private copies of variable var_a and var_c
  - Each private copy of var_c is initialized with the value of var_c in main thread when the parallel directive is encountered
- default (none)
  - Default state of a variable is specified as none (rather than shared)
  - Singals error if not all variables are specified as shared or private

Number of Threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
    1. Evaluation of the if clause
    2. Setting of the num_threads() clause
    3. Use of the omp_set_num_threads() library function
    4. Setting of the OMP_NUM_THREAD environment variable
    5. Implementation default – usually the number of cores on a node
- Threads are numbered from 0 (master thread) to N-1

Thread Creation: Parallel Region Example

- **Create threads with the parallel construct**

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

int main()
{
   int nthreads, tid;
   #pragma omp parallel num_threads(4) private(tid)
   {
      tid = omp_get_thread_num();
      printf("Hello world from (%d)\n", tid);
      if(tid == 0)
      {
          nthreads = omp_get_num_threads();
          printf("number of threads = %d\n", nthreads);
      }
   } // all threads join master thread and terminates
}
```

Clause to request threads

Each thread executes a copy of the code within the structured block

Thread Creation: Parallel Region Example

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

int main(){
   int nthreads,  A[100] , tid;
   // fork a group of threads with each thread having a private tid variable
   omp_set_num_threads(4);
   #pragma omp parallel private (tid)
   {
      tid = omp_get_thread_num();
      foo(tid, A);
    } // all threads join master thread and terminates
}
```

A single copy of A[] is shared between all threads

SPMD vs. Work-Sharing

- A parallel construct by itself creates a “single program multiple data” program, i.e., each thread executes the same code.
- Work-sharing is to split up pathways through the code between threads within a team.
  - Loop construct
  - Sections/section constructs
  - Single construct
  - …

Work-Sharing Construct

- Within the scope of a parallel directive, work-sharing directives allow concurrency between iterations or tasks
- Work-sharing constructs do not create new threads
- 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.
- Two directives to be studied
  - Do/for: concurrent loop iterations
  - sections: concurrent tasks

Work-Sharing Do/for Directive

Do/for

- Shares iterations of a loop across the group
- Represents a “data parallelism”.

for directive partitions parallel iterations across threads

Do is the analogous directive in Fortran

Usage:

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

- Implicit barrier at end of for loop

Example Using *for*

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

int main()
{
   int nthreads, tid;

   omp_set_num_threads(3);
   #pragma omp parallel private(tid)
   {
      int i;
      tid = omp_get_thread_num();
      printf("Hello world from (%d)\n", tid);
      #pragma omp for
      for(i = 0; i <=4; i++)
      {
         printf(“Iteration %d by  %d\n”, i, tid);
      }
   } // all threads join master thread and terminates
}
```

Another Example Using *for*

- Sequential code to add two vectors

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

- **OpenMP implementation 1 (not desired)**

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

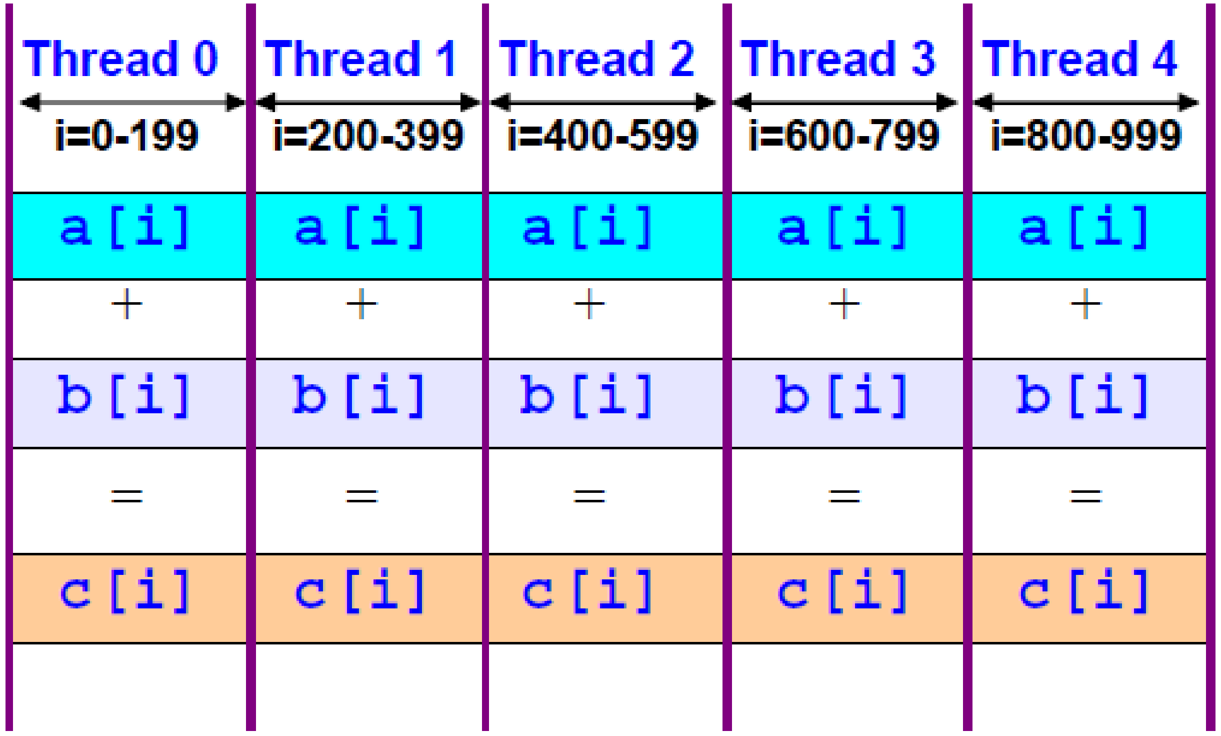
- **A worksharing for construct to add vectors**

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

or

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

Execution for loop in parallel



C/C++ for Directive Syntax

```
#pragma omp for [clause list]
                schedule (type [,chunk])
                ordered
                private (variable list)
                 firstprivate (variable list)
                 shared (variable list)
                 reduction (operator: variable list)
                 collapse (n)
                 nowait
/* for_loop */
```

Reduction

- **Serial code**

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

- How to combine values into a single accumulation variable (avg)?

Reduction Clause

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

```
#pragma omp parallel reduction(+: sums) num_threads(4)
{
   /* compute local sums in each thread
}
 /* sums here contains sum of all local instances of sum */
```

Reduction in OpenMP for

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on *operator* (e.g. 0 for “+”)
  - Compiler finds standard reduction expressions containing *operator* and uses it to update the local copy.
  - Local copies are reduced into a single value and combined with the original global value when returns to the master thread.

```
{
    double avg = 0.0, A[MAX];
    int i;
     …
    #pragma omp parallel for reduction (+:avg)
    for(i =0; i<MAX; i++) {avg += a[i];}
     avg /= MAX;
}
```

Reduction Operators/Initial-Values

C/C++:

Operator	Initial Value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Monte Carlo to estimate PI

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

int main(int argc, char *argv[])
{
   long int    i, count;               // count points inside unit circle
   long int   samples;     // number of samples
   double pi;
   unsigned short xi[3] = {1, 5, 177};     // random number seed
   double x, y;
  samples = atoi(argv[1]);
   count = 0;
   for(i = 0; i < samples; i++)
   {
      x = erand48(xi);
      y = erand48(xi);
      if(x*x + y*y <= 1.0) count++;
   }

   pi = 4.0*count/samples;
   printf("Estimate of pi: %7.5f\n", pi);
}
```

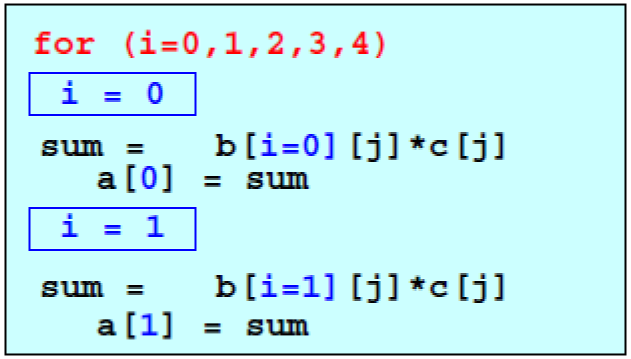
OpenMP version of Monte Carlo to Estimate PI

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

main(int argc, char *argv[])
{
    int          i, count;      /* points inside the unit quarter circle */
    unsigned short  xi[3];      /* random number seed */
    int           samples;        /* samples Number of points to generate */
    double       x,y;         /* Coordinates of points */
    double       pi;     /* Estimate of pi */

    samples = atoi(argv[1]);

    #pragma omp parallel
    {
        xi[0] = 1;      /* 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);
}
```

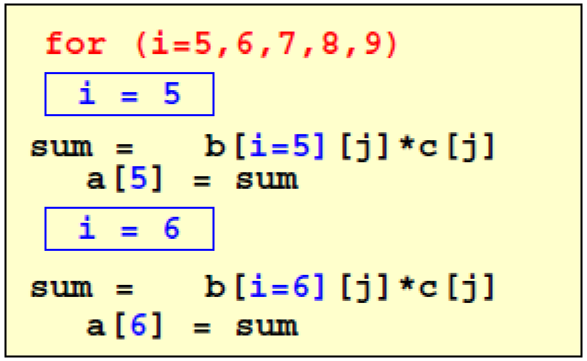
- A local copy of “count” for each thread
- All local copies of “count” added together and stored in master thread
- Each thread needs different random number seeds.

Matrix-Vector Multiplication

```
#pragma omp parallel default (none) \
shared (a, b, c, m,n) private (i,j,sum)
num_threads(4)
for(i=0; i < m; i++){
    sum = 0.0;
    for(j=0; j < n; j++)
          sum += b[i][j]*c[j];
     a[i] =sum;
}
```



```
for (i=0,1,2,3,4)
 i = 0
sum =    b[i=0][j]*c[j]
    a[0] = sum
 i = 1
sum =    b[i=1][j]*c[j]
    a[1] = sum
```



```
for (i=5,6,7,8,9)
 i = 5
sum =    b[i=5][j]*c[j]
    a[5] = sum
 i = 6
sum =    b[i=6][j]*c[j]
    a[6] = sum
```

Thread 0,   Thread 1,   ...etc...

schedule clause

- Describe how iterations of the loop are divided among the threads in the group. The default schedule is implementation dependent.
- Usage: schedule (scheduling_class[, parameter]).
  - *static*

    Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iteration are evenly (if possible) divided contiguously among the threads.
  - *dynamic*

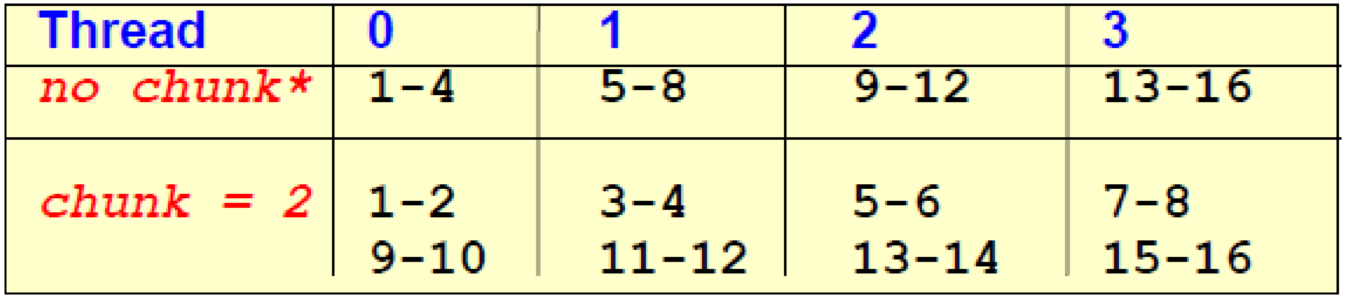
    Loop iterations are divided into pieces of size chunk and then dynamically assigned to threads. When a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
  - *guided*

    For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value $k(k > 1)$, the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.
  - *runtime*

    The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause
  - *auto*

    The scheduling decision is made by the compiler and/or runtime system.

- Static scheduling
- 16 iterations, 4 threads:



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

Static Scheduling

```
// static scheduling of matrix multiplication loops
#pragma omp parallel default (private) \
shared (a, b, c, dim) num_threads(4)
#pragma omp for schedule(static)
for(i=0; i < dim; i++)
{
   for(j=0; j < dim; j++)
   {
       c[i][j] =  0.0;
        for(k=0; j < dim; k++)
           c[i][j] += a[i][k]*b[k][j];
    }
}
```

Static schedule maps iterations to threads at compile time

Environment Variables

- OMP_SCHEDULE “schedule[, chunk_size]”
  - Control how “omp for schedule (RUNTIME)” loop iterations are scheduled.
- OMP_NUM_THREADS integer
  - Set the default number of threads to use
- OMP_DYNAMIC TRUE|FALSE
  - Can the program use a different number of threads in each parallel region?
- OMP_NESTED TRUE |FALSE
  - Will nested parallel regions create new teams of threads, or will they be serialized?

By default, worksharing for loops end with an implicit barrier

- *nowait*: If specified, threads do not synchronize at the end of the parallel loop
- *ordered*: specifies that the iteration of the loop must be executed as they would be in serial program.
- *collapse*: specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iteration in all associated loops determines the order of the iterations in the collapsed iteration space.

Avoiding Synchronization with nowait

```
#pragma omp parallel shared(A,B,C) private(id)
{
   id = omp_get_thread_num();
   A[id] = big_calc1(id);
   #pragma omp barrier
   #pragma omp for
   for(i = 0; i < N; i++) { C[i] = big_calc3(i,A); }
   #pragma omp for nowait
   for(i = 0; i < N; i++) {B[i] = big_calc2(C,i); }
   A[id] = big_calc4(id);
}
```

Barrier: each threads waits till all threads arrive.

No implicit barrier due to nowait. Any thread can begin big_calc4() immediately without waiting for other threads to finish the loop

Implicit barrier at the end of the parallel region

- By default: worksharing for loops end with an implicit barrier
- nowait clause:
  - Modifies a for directive
  - Avoids implicit barrier at end of for

Loop Collapse

- Allows parallelization of perfectly nested loops without using nested parallelism
- Compiler forms a single loop and then parallelizes this

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

For Directive Restrictions

For the "*for* loop" that follows the *for* directive:

- It must not have a break statement
- The loop control variable must be an integer
- The initialization expression of the "*for* loop" must be an integer assignment.
- The logical expression must be one of $<, \leq, >, \geq$
- The increment expression must have integer increments or decrements only.