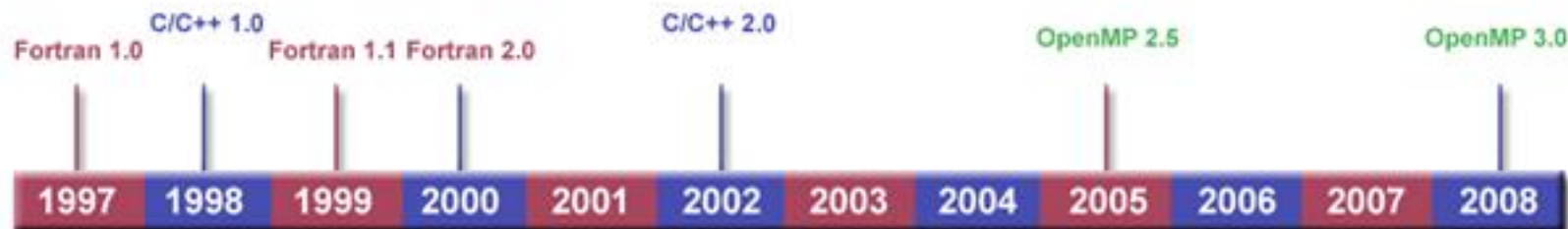


# Shared-memory parallel programming using OpenMP

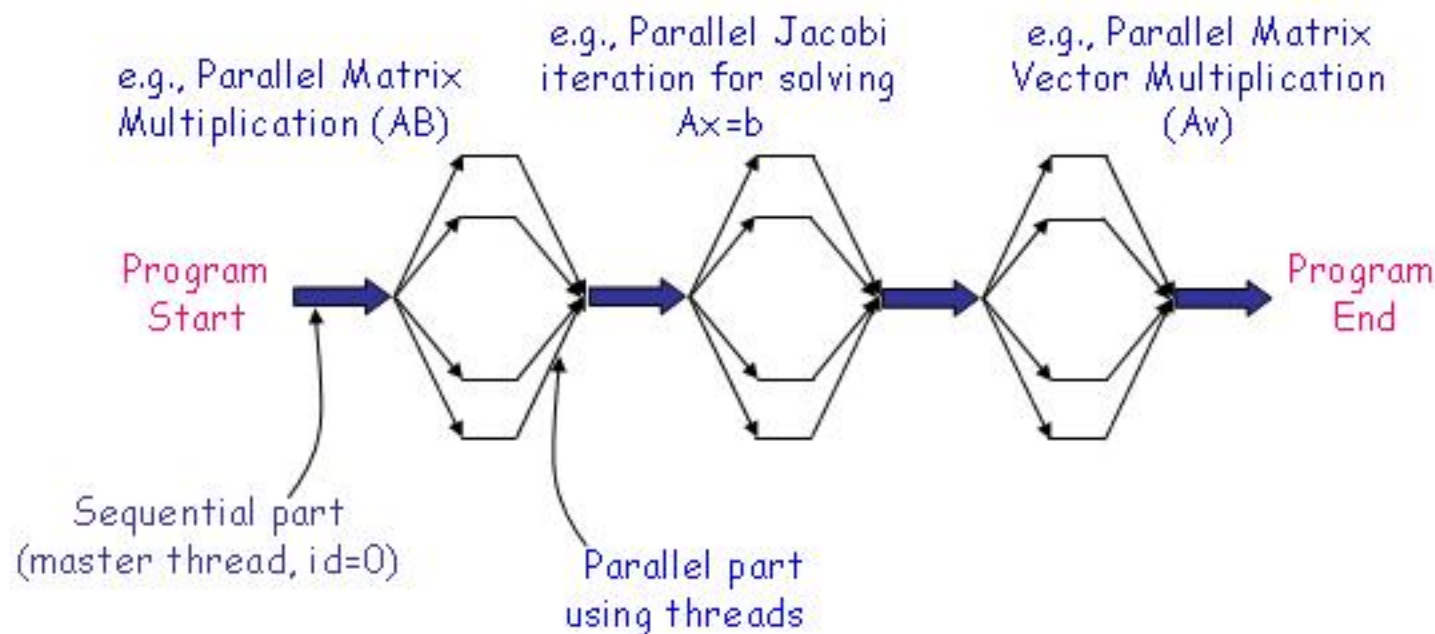
- **OpenMP is an API for writing multithreaded applications in a shared memory environment**
  - **Consists of a set of compiler directives and library routines**
  
- **Advantages:**
  - **Parallelize small parts of application, one at a time**
  - **Easy to create multi-threaded applications in Fortran, C, and C++**
  - **Code size grows only modestly; code is easy to read**
  - **Single source code for OpenMP and non-OpenMP**
    - **Non-OpenMP compilers simply ignore OMP directives**

- **Industry-standard shared memory programming model**
  - **Involved Hardware, Software, and Applications vendors**
    - **Intel, HP, SGI, Sun, IBM, KAI, PGI, PSR, APR, Absoft, ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, etc.**
- **Developed in 1997**
- **OpenMP Architecture Review Board (ARB) determines additions and updates to standard**



- **Are combination of**
  - **Directives**
  - **Runtime library routines**
  - **Environment variables**
- **API falls into three categories**
  - **Expression of parallelism (flow control)**
  - **Data sharing among threads (communication)**
  - **Synchronization (coordination or interaction)**

- A master thread spawns teams of threads as needed
- Parallelism is added incrementally; the serial program evolves into a parallel program



- Are based on "pragmas" in C and C++  
`#pragma omp directive [clause list]`
- All directives followed by newline
- Uses pragma construct (pragma = Greek for "thing")
- Case sensitive
- Directives follow standard rules for C/C++ compiler directives
- Long directive lines can be continued by escaping newline character with \

- **Basic format**

*sentinal directive [clause list]*

- **Three accepted sentinels: !\$omp \*\$omp c\$omp**

- **Fixed-form code:**

- **Any of three sentinels beginning at column 1**
- **Initial directive line has space/zero in column 6**
- **Continuation directive line has non-space/zero in column 6**

- **Free-form code:**

- **!\$omp only accepted sentinel**
- **Sentinel can be in any column, preceded/followed by white space**
- **Continuation dir. line ends in &, following line begins with sentinel**

- **OpenMP programs execute serially until they encounter the parallel directive**

```
!$OMP parallel [clause list]
```

```
...Program statements ...
```

```
!$OMP end parallel
```

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

```
{ /* structured block */ }
```

- This directive is responsible for creating a group of threads
  - **Exact number specified in directive, environment variable, or at runtime using OpenMP functions**
- Each thread executes the structured block concurrently
- Main thread that encounters this directive becomes the "master" of this group having a thread id 0

- **Conditional parallelism**

- **The `if (scalar expression)` clause determines whether the parallel construct results in creation of threads**

```
#pragma omp parallel if(is_parallel == 1) num_threads(8) \
private (a) shared (b, c) firstprivate(d)
{ /* Structured block */ }
```

- **Degree of concurrency**

- **The `num_threads(integer expression)` clause specifies number of threads created by the parallel directive**

- `private(variable list)`
  - **clause specifies local thread variables; each thread has its own copy of the variables for duration of the parallel code**
  
- `firstprivate(variable list)`
  - **Private variables that are initialized when parallel code is entered**
  
- `shared(variable list)`
  - **specifies shared thread variables (threads access same memory locations)**
  
- `default(private|shared|none)`
  - **specifies default scoping for variables in parallel code**

- `reduction(operator: variable list)`
  - specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exist
  - Variables in the list are implicitly specified as being private to threads

```
#pragma omp parallel reduction(+: sum) num_threads(8)
{
    /* Compute local sums here */
}
/* Sum here contains sum of all local instances of sums */
```

	Output
<pre>#include "omp.h" void main() {     printf("Region A: %d ", omp_get_thread_num());</pre>	Region A: 0
<pre>#pragma omp parallel num_threads(4) { printf("Region B: %d ", omp_get_thread_num()); }</pre>	Region B: 3 Region B: 0 Region B: 2 Region B: 1
<pre>printf("Region C: %d ", omp_get_thread_num());</pre>	Region C: 0
<pre>#pragma omp parallel num_threads(2) { printf("Region D: %d ", omp_get_thread_num()); }</pre>	Region D: 1 Region D: 0
<pre>printf("Region E: %d ", omp_get_thread_num());</pre>	Region E: 0
<pre>#pragma omp parallel num_threads(3) { printf("Region F: %d ", omp_get_thread_num()); }</pre>	Region F: 0 Region F: 1 Region F: 2
<pre>printf("Region G: %d ", omp_get_thread_num()); }</pre>	Region G: 0

```
#include "omp.h"
void main()
{
    #pragma omp parallel num_threads(4)
    {
        int thread_ID = omp_get_thread_num();
        printf("Hello World! I am thread no.: %d ", thread_ID);
    }
}
```

## Output of the program:

```
Hello World! I am thread no.: 2
Hello World! I am thread no.: 0
Hello World! I am thread no.: 1
Hello World! I am thread no.: 3
```

```

program hello
integer tid, omp_get_thread_num
print 'Hello world from threads:'
!$OMP parallel private(tid)
tid = omp_get_thread_num()
print '<', tid, '>'
!$omp end parallel
print 'I am sequential now'
end

```

## Output

Hello world from threads:

<1>

<2>

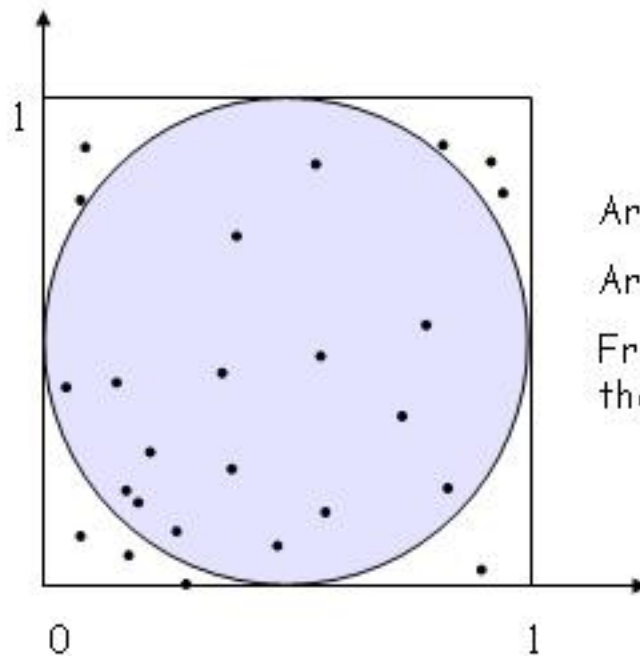
<0>

<4>

<3>

I am sequential now

- Based on generating random numbers in a unit length square and counting number of points that fall within largest circle inscribed in the square

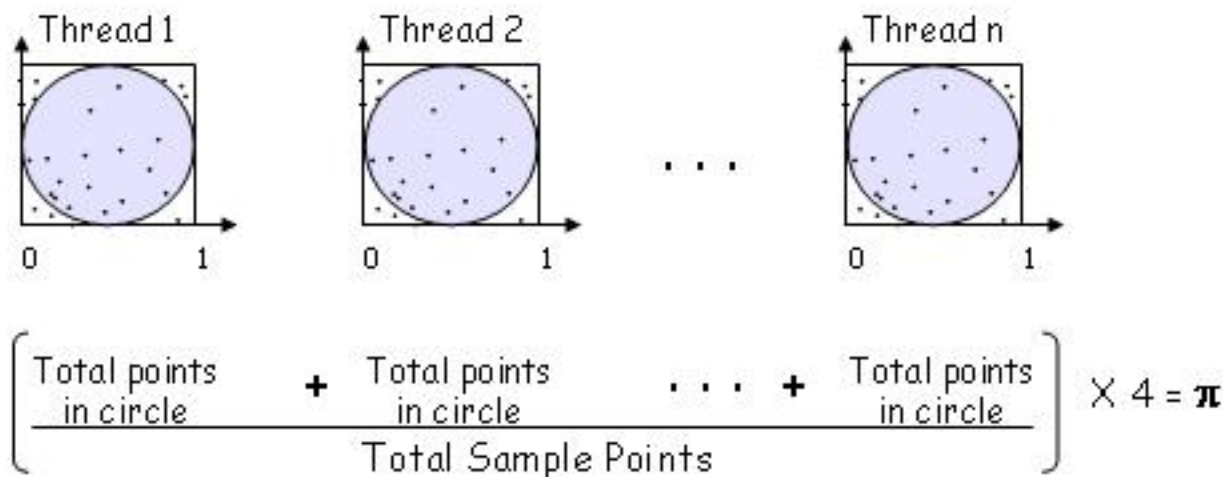


Area of the circle =  $\pi r^2 = \pi/4$

Area of the square =  $1 \times 1$

Fraction of random points that fall in the circle  $\sim \pi/4$

- Assign fixed number of points to each thread
- Each thread generates random points and keeps track of no. of points that land in circle locally
- After all threads finish execution, their counts are combined to compute the value of  $\pi$



```

#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
main()
{
    int num_threads, sample_points_per_thread, npoints=800000, i, sum=0;
    double x, y, d, pi;
    unsigned int seed=35791246;
    #pragma omp parallel private(i) shared(npoints) reduction(+: sum) num_threads(3)
    {
        num_threads = omp_get_num_threads(); // OpenMP function
        sample_points_per_thread = npoints/num_threads;
        int thseed=seed/(omp_get_thread_num()+1);
        srand(thseed);
        for(i=0; i<sample_points_per_thread; i++)
        {
            x = (double)rand()/RAND_MAX;
            y = (double)rand()/RAND_MAX;
            d = (x-0.5)*(x-0.5)+(y-0.5)*(y-0.5);
            if (d<=0.25)sum++;
            //if(i==(sample_points_per_thread-1))
                //printf("\nseed=%d, sum=%d, Tid=%d\n", thseed, sum, omp_get_thread_num());
        }
    }
    pi=(double)sum/npoints*4;
    printf("\nSum=%d pi=%g\n", sum, pi);
}

```

- **The parallel directive can be used in conjunction with other directives**
  - **Parallel for directive**
  - **Parallel sections directive**
  - **single directive**

- Used to split parallel iteration spaces across threads

```
!$OMP DO [clause list]
```

```
...DO Loop...
```

```
!$OMP end DO [nowait]
```

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

```
/* for loop */
```

- The clauses that can be used with for directive

- private, firstprivate, lastprivate, reduction, schedule, nowait, **and** ordered

- lastprivate(var1, var2, ...)

- Private variables that save their values at the last (serial) iteration

- schedule(type [,chunk])

- Controls how loop iterations are distributed among threads

Shared Memory Programming using OpenMP

```

#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
main()
{
    int num_threads,sample_points_per_thread,npoints=800000,i,sum=0;
    double x,y,d,pi;
    unsigned int seed=35791246;
    #pragma omp parallel private(i) shared(npoints) num_threads(3) reduction(+: sum)
    {
        int thseed=seed/(omp_get_thread_num()+1);
        srand(thseed);
        printf("\nseed=%d, Tid=%d\n",thseed,omp_get_thread_num());
        #pragma omp for
        for(i=0; i<npoints; i++)
        {
            x = (double)rand()/RAND_MAX;
            y = (double)rand()/RAND_MAX;
            d = (x-0.5)*(x-0.5)+(y-0.5)*(y-0.5);
            if (d<=0.25)sum++;
        }
    }
    pi=(double)sum/npoints*4;
    printf("\nSum=%d pi=%g\n", sum,pi);
}

```

- **By default, all variables shared except**
  - **Certain loop index values - private by default**
  - **Local variables and value parameters within subroutines called within parallel region - private**
  - **Variables declared within lexical extent of parallel region - private**
  
- **Good programming practice: explicitly declare scope of all variables**
  - **Helps in understanding how variables are used in program**
  - **Reduces chances of data race conditions or unexplained behavior**

```
void caller(int *a, int n)
{
    int i,j,m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++)
    {
        int k=m;
        for (j=1; j<=5; j++)
            callee(&a[i], &k, j);
    }
}
```

```
void callee(int *x, int *y,
            int z)
{
    int ii;
    static int cnt;
    cnt++;
    for (ii=1; ii<z; ii++)
        *x = *y + z;
}
```

Var	Scope	Comment
a	shared	Declared outside parallel construct
n	shared	same
i	private	Parallel loop index
j	shared	Sequential loop index
m	shared	Declared outside parallel construct
k	private	Automatic variable/parallel region
x	private	Passed by value
*x	shared	(actually a)
y	private	Passed by value
*y	private	(actually k)
z	private	(actually j)
ii	private	Local stack variable in called function
cnt	shared	Declared static (like global)

- The **schedule clause** controls how work is distributed among threads

```
schedule (type [, chunk-size])
```

- `chunk-size` is used to specify the size of each work packet (number of iterations)
  - Default values for `chunk-size` used when not specified
- `type` may be one of the following
  - `static`, `dynamic`, `guided`, and `runtime`

- **Iterations are divided into chunk-size work packets**
  - If there are N threads, each thread does every Nth chunk-size of work
- **When chunk-size is unspecified, iteration space is split into as many chunks as there are threads**

```
schedule(static[,chunk_size])
```

```
#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;
        for(k=0; k<dim; k++){
            c(i,j) += a(i,k) * b(k,j);
        }
    }
}
```

If  $dim=128$ , size of each partition is 32 columns (since there are 4 threads)

- Divides the workload into chunk-size work packets  
`schedule(dynamic[,chunk_size])`
- As a thread finishes one chunk-size packet, it grabs the next available chunk packet
- Default value for chunk-size is one (single iteration per chunk)
- More overhead, but potentially better load balancing

- Like `dynamic` scheduling, but the chunk size varies dynamically  
`schedule(guided[, chunk_size])`
- Chunk sizes depend on the number of unassigned iterations
- The chunk size decreases toward the specified value of `chunk-size`
- Achieves good load balancing with relatively low overhead
- Insures that no single thread will be stuck with a large number of leftovers while the others take a tea break

- **Scheduling method is determined at runtime**  
`schedule(runtime)`
- **Depends on value of environment variable `OMP_SCHEDULE` that determines the scheduling type and chunk size**
  - **Scheduling method is `static` by default**
- **Useful for experimenting with different scheduling methods without recompiling**

- **By default there is a barrier at end of the `for` loop**
  - **Threads wait until all are finished, then proceed**
  - **Use of `nowait` clause allows threads to continue without waiting**

- Each parallel section is run on a separate thread
- Allows functional decomposition
- Implicit barrier at the end of the sections construct
  - Use the `nowait` clause to suppress this

```

#pragma omp parallel
{
    #pragma omp sections [clause list]
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        . . .
    }
}
    
```

private,  
firstprivate,  
lastprivate,  
reduction, and  
nowait

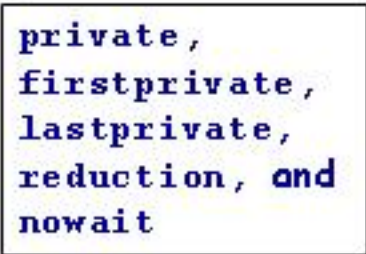
```

!$OMP parallel
!$OMP sections [clause list]

!$OMP section
  taskA

!$OMP section
  taskB
. . .

!$OMP end sections
!$OMP end parallel
  
```



private,  
firstprivate,  
lastprivate,  
reduction, and  
nowait

- **The parallel directive creates concurrent threads, for and sections directives farm out work to threads**
  - **If parallel not specified, for and sections will execute serially (all work assigned to single thread, the master thread)**
  
- **Consequently, for and sections directives are preceded by parallel directive**
  - **Can also merge to parallel for and parallel sections**
  - **Clause list for the merged directive can be from clause lists of either the parallel or for/sections directives**

```
#pragma omp parallel for default(private) shared(n)
{
    for(i=0; i<n; i++) {
        /* body of p'll for loop */
    }
}
```

```
!$OMP parallel DO [clause
list]
..DO Loop..
!$OMP end parallel DO
```

And ...

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        taskA();
    }
    #pragma omp section
    {
        taskB();
    }
}
```

```
!$OMP parallel sections
[clause list]

!$OMP section
    taskA

!$OMP section
    taskB
    . . .
!$OMP end parallel sections
```

- **A parallel directive inside another parallel directive establishes new set of threads**
- **Nesting is enabled by setting OMP\_NESTED environment variable to TRUE (by default it's FALSE)**
  - **New set of threads created when parallel for directives are nested**
    - **for, sections, and single that bind to same parallel directive cannot be nested**

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