CS410: Parallel Computing

Spring 2024

Nikhil Hegde
Milind Chabbi

Shared-Memory Programming OpenMP

What is OpenMP

- An open standard for shared-memory programming in C, C++, and Fortran
- Supported by IBM, Intel, GNU and others
- *Directive-based* programming approach removes the need for explicitly setting up initialization, mutexes, and condition variables.

What is OpenMP?

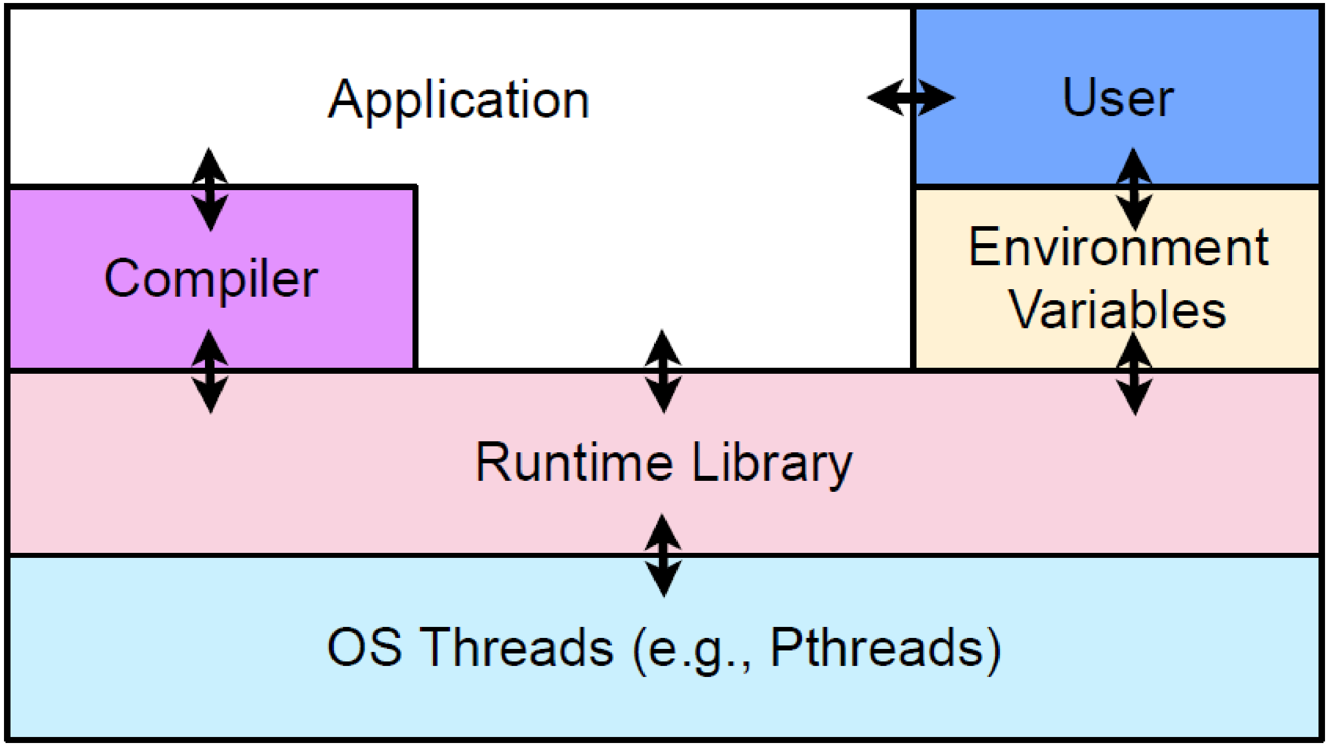
Open specifications for Multi Processing

- An API for explicit multi-threaded, shared memory parallelism
- Three components
  - —compiler directives
  - —runtime library routines
  - —environment variables
- Higher-level programming model than Pthreads
  - —implicit mapping and load balancing of work
- Portable
  - —API is specified for C/C++ and Fortran
  - —implementations on almost all platforms
- Standardized

OpenMP Is Not

- An automatic parallel programming model
  - —parallelism is explicit
  - —programmer full control (and responsibility) over parallelization
- Meant for distributed-memory parallel systems (by itself)
  - —designed for shared address spaced machines
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory
  - —no data locality control

OpenMP at a Glance

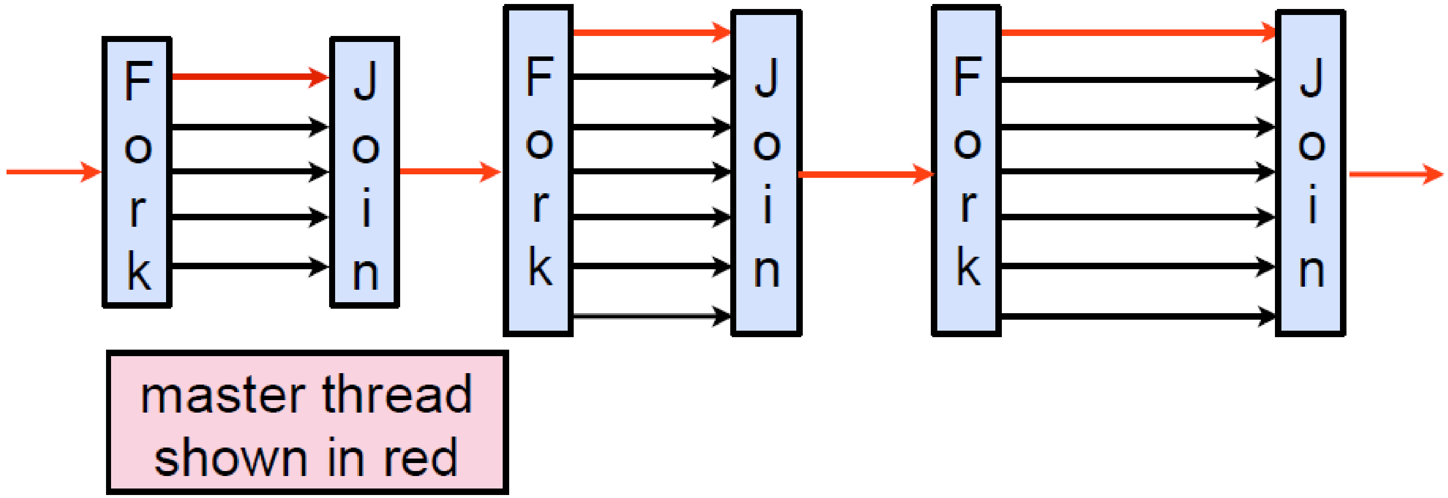


OpenMP Targets Ease of Use

- OpenMP does not require that single-threaded code be changed for threading
  - —enables incremental parallelization of a serial program
- OpenMP only adds compiler directives
  - —pragmas (C/C++); significant comments in Fortran
    - – if a compiler does not recognize a directive, it simply ignores it
  - —simple & limited set of directives for shared memory programs
  - —significant parallelism possible using just 3 or 4 directives
    - – both coarse-grain and fine-grain parallelism
- If OpenMP is disabled when compiling a program, the program will execute sequentially

OpenMP: Fork-Join Parallelism

- OpenMP program begins execution as a single master thread
- Master thread executes sequentially until 1st parallel region
- When a parallel region is encountered, master thread
  - —creates a group of threads
  - —becomes the master of this group of threads
  - —is assigned the thread id 0 within the group



OpenMP: Fork-Join Parallelism

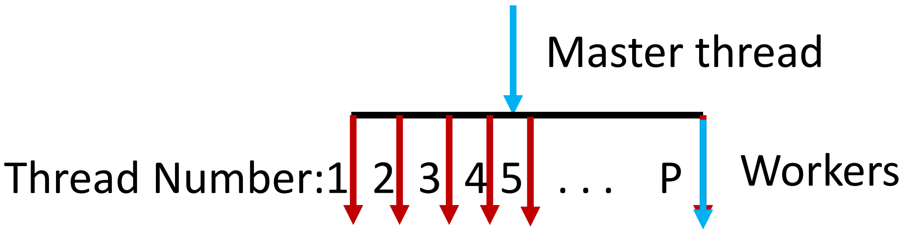
```
//code region 0
```

Master thread

- Execution begins with a master thread

OpenMP: Fork-Join Parallelism

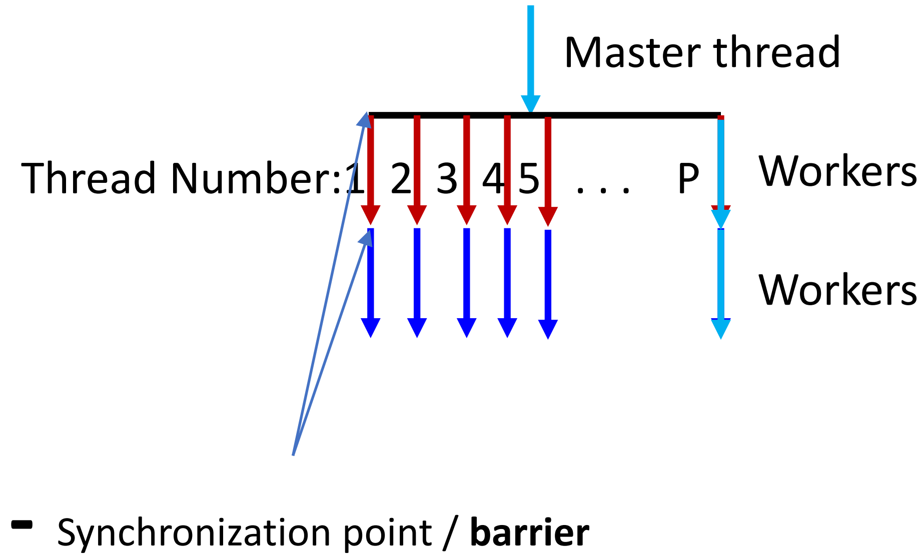
```
//code region 0
#pragma omp parallel
{
    //code region 1
}
```



- Master thread creates / forks worker threads (threads execute **code region 1**)

OpenMP: Fork-Join Parallelism

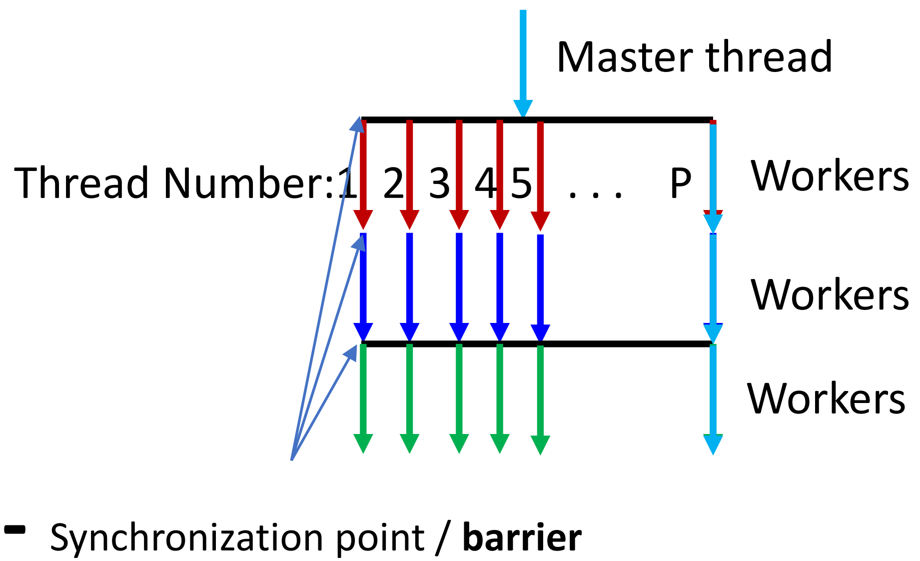
```
//code region 0
#pragma omp parallel
{
    //code region 1
    #pragma omp for
    for(i=0;i<N;i++) {
        //code region 2
    }
}
```



- Implicit barriers hinder worker threads' free run
    - Worker threads all begin to execute **code region 2** at the same time

OpenMP: Fork-Join Parallelism

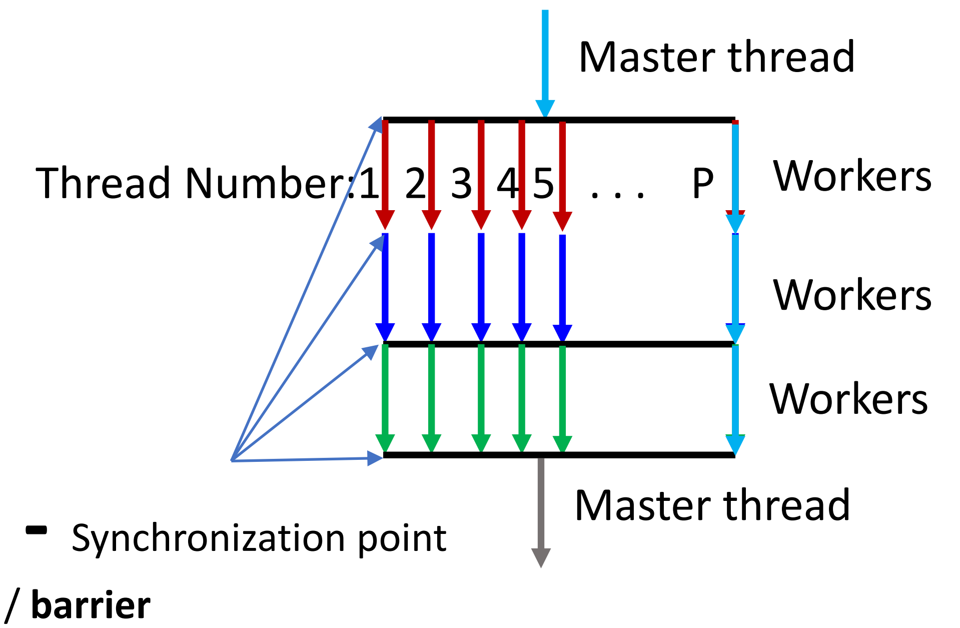
```
//code region 0
#pragma omp parallel
{
    //code region 1
    #pragma omp for
    for(i=0;i<N;i++) {
        //code region 2
    }
    //code region 3
}
```



- Worker threads cross another implicit barrier
  - Start executing **code region 3** all at the same time

OpenMP: Fork-Join Parallelism

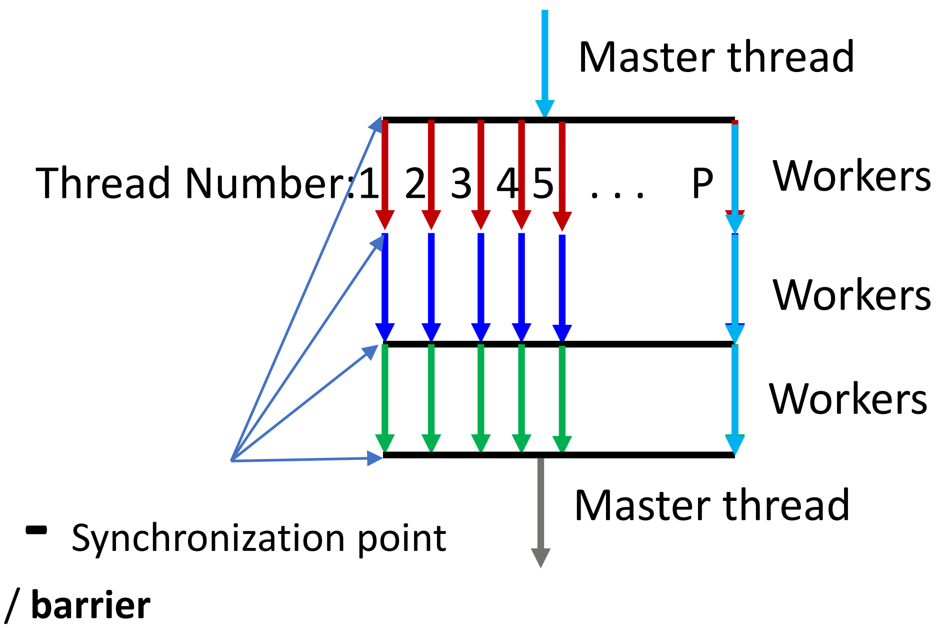
```
//code region 0
#pragma omp parallel
{
    //code region 1
    #pragma omp for
    for(i=0;i<N;i++) {
        //code region 2
    }
    //code region 3
}
//code region 4
```



- Worker threads join master thread
  - Master thread continues executing **code region 4**

OpenMP: Fork-Join Parallelism

```
//code region 0
#pragma omp parallel
{
    //code region 1
    #pragma omp for
    for(i=0;i<N;i++) {
        //code region 2
    }
    //code region 3
}
//code region 4
```



1. Execution begins with a master thread
2. Master thread creates / forks worker threads
3. Worker threads join master thread

fork / join parallelism

Programming in OpenMP - Directives

Compiler directives

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

- `#pragma parallel`

default behavior: executes as many threads as there are processors

- `#pragma omp for`

divides the whole work among available threads

- Allows loop-by-loop & region-by-region parallelization of *sequential programs*

Programming in OpenMP - Directives

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

- Example of loop parallelism
  - Common in scientific codes
- Programmer is still responsible for handling data races.

Programming in OpenMP - Directives

```
#pragma omp parallel for
   for(i=0;i<N;i++)
         sum = sum + arr[i];
```

- Combining `parallel` and `for`
  - `Meaning:` the iterations of the `for` loop is to be executed in parallel

OpenMP `parallel` Region Directive

`#pragma omp parallel` [clause list]

A few more clauses on slide 37

Typical clauses in [clause list]

- Conditional parallelization
  - `if (`scalar expression`)`
    - determines whether the `parallel` construct creates threads
- Degree of concurrency
  - `num_threads(`integer expression`): #` of threads to create
- Data 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 that variables are shared across all the threads
  - `default` (data scoping specifier)
    - default data scoping specifier may be `shared` or `none`

Interpreting an OpenMP Parallel Directive

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

Meaning

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

Meaning of OpenMP Parallel Directive

```
int a, b;
main() {
  // serial segment
  #pragma omp parallel num_threads (8) private (a) shared (b)
  {
    // parallel segment
  }
  // rest of serial segment
}
```

Sample OpenMP program

OpenMP

```
int a, b;
main() {
  // serial segment
    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);
  // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;
  // parallel segment
}
```

Code inserted by the OpenMP compiler

Corresponding Pthreads translation

Pthreads equivalent

Specifying Worksharing

Within the scope of a parallel directive, worksharing directives allow concurrency between iterations or tasks

- **OpenMP provides two directives**
  - `DO/for`: concurrent loop iterations
  - `sections`: concurrent tasks

Worksharing DO/for Directive

for directive partitions parallel iterations across threads

DO is the analogous directive for Fortran

- Usage:

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

- Possible clauses in [clause list]
  - private, firstprivate, lastprivate
  - reduction
  - schedule, nowait, and ordered
- Implicit barrier at end of for loop

A Simple Example Using `parallel` and `for`

Program

```
void main()  {
#pragma omp parallel num_threads(3)
{
  int i;
  printf("Hello world\n");
  #pragma omp for
  for (i = 1; i <= 4; i++) {
     printf("Iteration %d\n",i);
  }
  printf("Goodbye world\n");
}
}
```

Output

```
Hello world
Hello world
Hello world
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Goodbye world
Goodbye world
Goodbye world
```

Reduction Clause with Parallel Directive

- reductions - take something complex and reduce it to something simpler. E.g. take a vector and produce a scalar
- The `reduction` clause in OpenMP specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
- The usage of the `reduction` clause is `reduction (operator: variable list)`.

Reduction Clause with Parallel Directive

- Operations supported in reductions:
  - +: addition
  - *: multiplication
  - |: bitwise OR
  - &: bitwise AND
  - ^: bitwise exclusive OR
  - ||: logical OR
  - &&: logical AND

  Note the *commutative nature* of these operations
- Usage:

```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local sums here */
}
```

OpenMP Reduction Clause Example

OpenMP threaded program to estimate 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++) {
        coord_x =(double)(rand_r(&seed))/(double)((2<<14)-1) - 0.5;
        coord_y =(double)(rand_r(&seed))/(double)((2<<14)-1) - 0.5;
        if ((coord_x * coord_x + coord_y * coord_y) < 0.25)
            sum ++;
    }
}
```

here, user manually divides work

- a local copy of sum for each thread
- all local copies of sum added together and stored in master

Using Worksharing for Directive

```
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
    sum = 0;
    #pragma omp for
    for (i = 0; i < npoints; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum ++;
    }
}
```

worksharing for divides work

Implicit barrier at end of loop

Mapping Iterations to Threads

`schedule` clause of the `for` directive

- Recipe for mapping iterations to threads
- Usage: `schedule(`scheduling_class`[,` $\overbrace{\text{chunk}}^{\text{optional}}$ `]).`
- Four scheduling classes
  - `static`: work partitioned at compile time
    - iterations statically divided into pieces of size *chunk*
    - statically assigned to threads
  - `dynamic`: work evenly partitioned at run time
    - iterations are divided into pieces of size *chunk*
    - chunks dynamically scheduled among the threads
    - when a thread finishes one chunk, it is dynamically assigned another
    - default chunk size is 1
  - `guided`: guided self-scheduling
    - chunk size is exponentially reduced with each dispatched piece of work
    - the default minimum chunk size is 1
  - `runtime`:
    - scheduling decision from environment variable OMP_SCHEDULE
    - illegal to specify a chunk size for this clause.

Mapping Iterations to Threads

- The schedule clause can guide how iterations of a loop are assigned to threads
- Two kinds of schedules:
    - static: iterations are assigned to threads at the start of the loop. Low overhead but possible load balance issues.
    - dynamic: some iterations are assigned at the start of the loop, others as the loop progresses. Higher overheads but better load balance.
- A *chunk* is a contiguous set of iterations

The schedule clause - static

Example:

- *(type [,chunk])* is
  - (static): chunks of ~ *n/t* iterations per thread, no chunk specified. The default.
  - (static, chunk): chunks of size *chunk* distributed round-robin statically.

The schedule clause - dynamic

Example:

- *(type [,chunk])* is
  - (dynamic): chunks of size 1 iteration distributed dynamically
  - (dynamic, chunk): chunks of size *chunk* distributed dynamically

Static

	thread 0	thread 1	thread 2
Chunk = 1	0, 3, 6, 9, 12	1, 4, 7, 10, 13	2, 5, 8, 11, 14

	thread 0	thread 1	thread 2
Chunk = 2	0, 1, 6, 7, 12, 13	2, 3, 8, 9, 14, 15	4, 5, 10, 11, 16, 17

With *dynamic* chunks go to processors as work needed.

The schedule clause

- schedule(*type[, chunk])*) (type [,chunk]) is
    - (guided,chunk): uses *guided self scheduling* heuristic. Starts with big chunks and decreases to a minimum chunk size of *chunk*
    - runtime - type depends on value of OMP_SCHEDULE environment variable, e.g. setenv OMP_SCHEDULE=”static,1”

Avoiding Unwanted Synchronization

- Default: worksharing `for` loops end with an implicit barrier
- Often, less synchronization is appropriate
  - —series of independent `for`-directives within a parallel construct
- `nowait` clause
  - —modifies a `for` directive
  - —avoids implicit barrier at end of for

Avoiding Synchronization with nowait

```
#pragma omp parallel
{
   #pragma omp for nowait
      for (i = 0; i < nmax; i++)
         if (isEqual(name, current_list[i])
            processCurrentName(name);
   #pragma omp for
      for (i = 0; i < mmax; i++)
         if (isEqual(name, past_list[i])
            processPastName(name);
}
```

any thread can begin second loop immediately without waiting for other threads to finish first loop

OpenMP Synchronization Constructs

```
#pragma omp barrier
#pragma omp single [clause list]
   structured block
#pragma omp master
   structured block
#pragma omp critical [(name)]
   structured block
#pragma omp ordered
   structured block
```

Querying the number of physical processors

- Can query the number of physical processors
  - returns the number of *cores* on a multicore machine
  - returns the number of possible *hyperthreads* on a hyperthreaded machine

int omp_get_num_procs(void);

Setting the number of threads

- Number of threads can be more or less than the number of processors
  - if less, some processors or cores will be idle
  - if more, more than one thread will execute on a core/processor
    - Operating system and runtime will assign threads to cores
    - No guarantee same threads will always run on the same cores
- Default is number of threads equals number of cores/processors

int omp_set_num_threads(int t);

OpenMP: `#pragma omp critical`

- use a *critical* section in the code
- executes the following (possible compound) statement atomically

```
t = 0
#pragma omp parallel for
for (i=0; i < n; i++) {
#pragma omp critical
  t += a[i];
}
t = t/n
```

Slower, sequential way to compute average of array elements. Why?

The single directive

```
#pragma omp parallel for
for (i=0; i < n; i++) {
   if (a[i] > 0) {
      a[i] += b[i];
#pragma omp single
      printf("exiting");
   }
}
```

master clause forces execution on the master thread.

Requires statement following the pragma to be executed by a single available thread.

Differs from critical in that critical lets the statement execute on many threads, just one at a time.

OpenMP Synchronization Constructs - ordered

```
#pragma omp parallel for
for(i=1;i<N;i++) {
  A //statement A.

  #pragma omp ordered
  B //statement B

  C //statement C

  #pragma omp ordered
  D //statement D

  E //statement E
}
```

A, C, E, can also be a sequence of statements. B and D can be a structured block of statements enclosed in { and }

OpenMP Synchronization Constructs - `ordered`

```
#pragma omp parallel for
for(i=1;i<N;i++) {
  A //statement A.

  #pragma omp ordered
  B //statement B

  C //statement C

  #pragma omp ordered
  D //statement D

  E //statement E
}
```

“preceeds”

Program order:

A

B

C

D

E

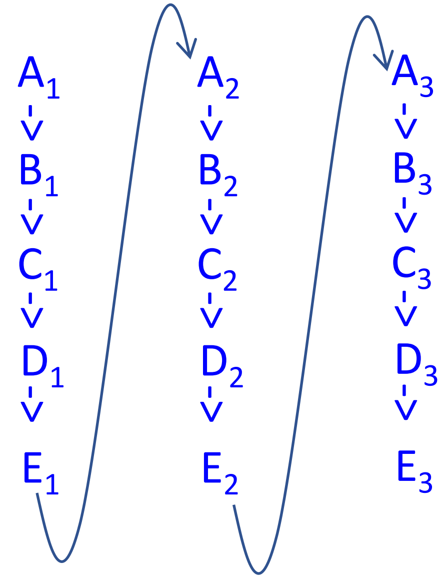
A, C, E, can also be a sequence of statements. B and D can be a structured block of statements enclosed in { and }

OpenMP Synchronization Constructs - ordered

```
#pragma omp parallel for
for(i=1;i<N;i++) {
  A //statement A.

  #pragma omp ordered
  B //statement B

  C //statement C

  #pragma omp ordered
  D //statement D

  E //statement E
}
```

Suppose N=4 (i.e. i varies from 1 to 3) and A_i denotes statement A in i^{th} iteration

Program order:



OpenMP Synchronization Constructs - `ordered`

```
#pragma omp parallel for
for(i=1;i<N;i++) {
  A //statement A.

  #pragma omp ordered
  B //statement B

  C //statement C

  #pragma omp ordered
  D //statement D

  E //statement E
}
```

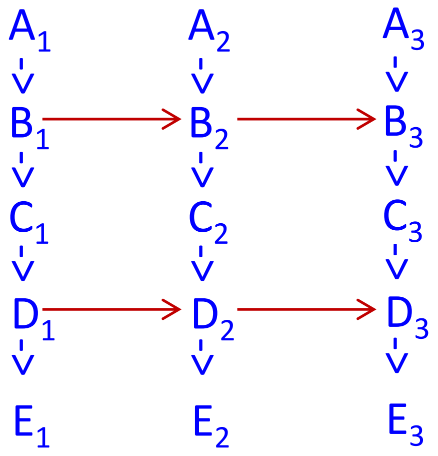
`#pragma omp parallel for`
takes away the constraints that exist across iterations

$$
\begin{array}{ccc}
A_1 & A_2 & A_3 \\
\downarrow & \downarrow & \downarrow \\
B_1 & B_2 & B_3 \\
\downarrow & \downarrow & \downarrow \\
C_1 & C_2 & C_3 \\
\downarrow & \downarrow & \downarrow \\
D_1 & D_2 & D_3 \\
\downarrow & \downarrow & \downarrow \\
E_1 & E_2 & E_3
\end{array}
$$

OpenMP Synchronization Constructs - ordered

```
#pragma omp parallel for
for(i=1;i<N;i++) {
  A //statement A.

  #pragma omp ordered
  B //statement B

  C //statement C

  #pragma omp ordered
  D //statement D

  E //statement E
}
```

ordered clause adds the constraints



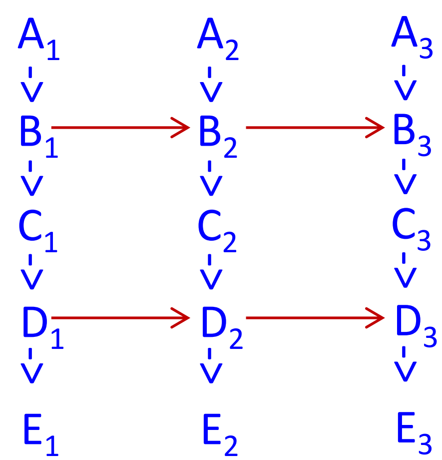
OpenMP Synchronization Constructs - `ordered`

```
#pragma omp parallel for
for(i=1;i<N;i++) {
  A //statement A.

  #pragma omp ordered
  B //statement B

  C //statement C

  #pragma omp ordered
  D //statement D

  E //statement E
}
```

Now a statement S_i can execute concurrently with a statement S_j if:

- No edge exists between S_i and S_j AND
- There is no path between S_i and S_j

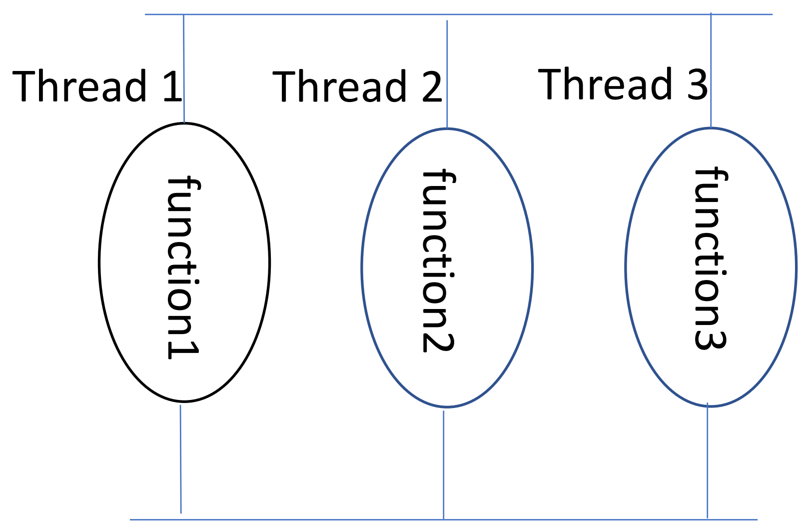
e.g. A_1 and B_2 cannot be executed concurrently.



OpenMP: Sections

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            function1();
        }
        #pragma omp section
        {
            function2();
        }
        #pragma omp section
        {
            function3();
        }
    }
}
```

- Open MP also supports task parallelism with `sections` clause
- As many work items exist as there are `section` clauses



OpenMP Orphaned Directive

```
#pragma omp parallel
{
    Blur(A, size)


}
```

```
void Blur(double* A, pair<int, int>
size) {

    #pragma omp for
    for(..) {

    }
}
```

- Useful for library developers

OpenMP Orphaned Directive

```
#pragma omp parallel
{
    Blur(A, size)
    ComputeGaussian(A, size)

}
```

```
void ComputeGaussian(double* A,
pair<int, int> size) {

   #pragma omp for
   for(..) {

   }
}
```

- Useful for library developers
- Can associate a parallel region with multiple (different) function calls.

OpenMP Orphaned Directive

```
Blur(A, size)

ComputeGaussian(A, size)
```

```
void ComputeGaussian(double* A,
pair<int, int> size) {

    #pragma omp for
    for(..) {

    }
}
```

- Useful for library developers
- Can associate a parallel region with multiple (different) function calls.
- The same function can be called from a non-parallel code region. When called from non-parallel code region, single thread executes the called function.

OpenMP Task Example:

```c
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
  if (p->left)
#pragma omp task   // p is firstprivate by default
      traverse(p->left);
  if (p->right)
#pragma omp task    // p is firstprivate by default
      traverse(p->right);
  process(p);
}
```

Source: https://www.openmp.org/wp-content/uploads/openmp-examples-4.0.2.pdf

- process(p) is called without waiting for the tasks to finish

OpenMP Task Example2:

```
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void postorder_traverse( struct node *p ) {
    if (p->left)
       #pragma omp task    // p is firstprivate by default
           postorder_traverse(p->left);
    if (p->right)
       #pragma omp task   // p is firstprivate by default
           postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

Source: https://www.openmp.org/wp-content/uploads/openmp-examples-4.0.2.pdf

- process(p) is called only after left and right subtrees are traversed and the tasks finish

OpenMP Task Example3:

- Traversing a linked list and processing a node.
    - One thread from the team creates several tasks (one per node in the list)
    - All threads do the processing on nodes in the tasks created.

```c
typedef struct node node;
struct node {
      int data;
      node * next;
};

void process(node * p)
{
    /* do work here */
}
void increment_list_items(node * head)
{
    #pragma omp parallel
    {
        #pragma omp single
            {
               node * p = head;
               while (p) {
                    #pragma omp task
                     // p is firstprivate by default
                           process(p);
                     p = p->next;
                  }
            }
     }
}
```

Source: https://www.openmp.org/wp-content/uploads/openmp-examples-4.0.2.pdf

OpenMP Library Functions

```
/* thread and processor count */
• void omp_set_num_threads (int num_threads);
• int omp_get_num_threads ();
• int omp_get_max_threads ();
• int omp_get_thread_num ();
• int omp_get_num_procs ();
• int omp_in_parallel();
```

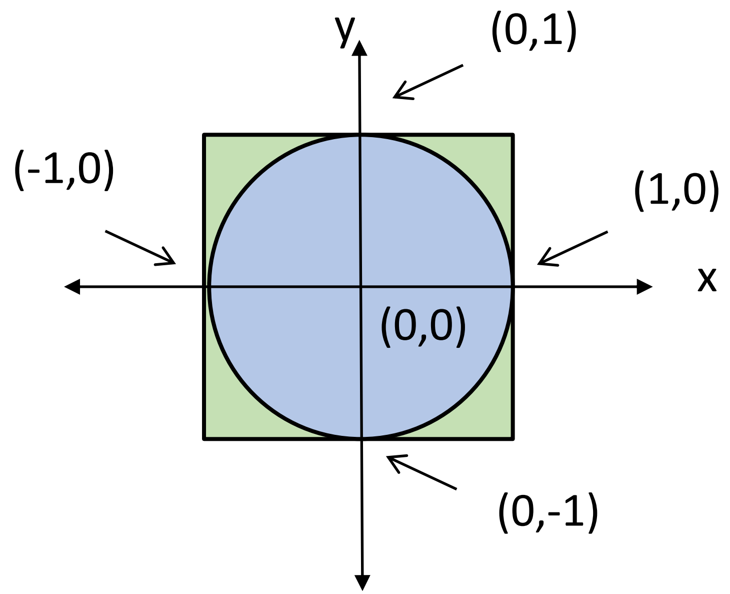
OpenMP Environment Variables

- OMP_NUM_THREADS: This environment variable specifies the default number of threads created upon entering a parallel region.
- OMP_SET_DYNAMIC: Determines if the number of threads can be dynamically changed.
- OMP_NESTED: Turns on nested parallelism.
- OMP_SCHEDULE: Scheduling of for-loops if the clause specifies runtime

OpenMP: Complete Reference

- https://www.openmp.org/specifications/

Estimate of π

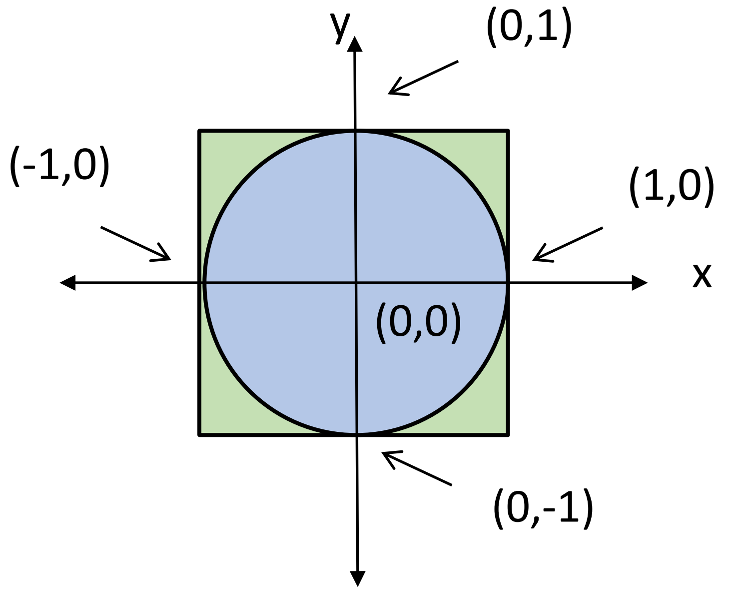


Circle

radius r = 1

Area of circle = $\pi r^2 = \pi$

Estimate of π



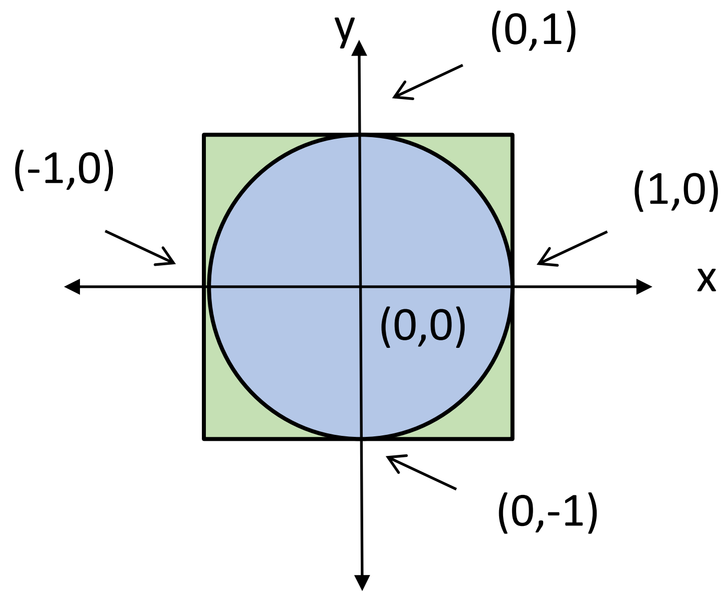
<u>Circle</u>

radius r = 1

Area of circle = $\pi r^2 = \pi$

Let f(x) = $\sqrt{1-x^2}$ describe the quarter circle for x = 0 ... 1

Estimate of π



Circle

radius r = 1

Area of circle = $\pi r^2 = \pi$

Let f(x) = $\sqrt{1-x^2}$ describe the quarter circle for x = 0 ... 1

pi / 4 = $\sum_{i=0}^{N-1} \Delta x \, f(x_i)$ where x_i = i Δx and $\Delta x = 1/N$

Numerical Integration Method: Estimate of π

- Reimann Sums Approach
  - Let f(x) = $\sqrt{1 - x^2}$ describe the quarter circle for x = 0 … 1
  - *pi* / 4 = $\sum_{i=0}^{N-1} \Delta x \; f(x_i)$ where x_i = i Δx and $\Delta x = 1/N$

- OpenMP Program (Demo)

Parallel Algorithms and Applications

Slides acknowledgement: Prof. Virendra Bhavsar

Embarrassingly Parallel Applications

- Application where:
  - A number of (almost) independent tasks
    - No or very little communication between tasks
  - Each task can be executed on a node
- Master-worker approach could be used
- Examples
  - Image Processing: e.g. blurring, scaling, rotation etc.
  - Computer Graphics: e.g. ray tracing
  - Monte Carlo method: e.g. estimation of pi
  - …

Interesting reads:
http://graphics.pixar.com/library/CurlyHairA/paper.pdf,
https://www.fxguide.com/fxfeatured/brave-new-hair/