OpenMP - Beginners Guide

TABLE OF CONTENTS

NOTE

Write a general serial Code ⇒ Make it parallel by adding appropriate OpenMP clauses

compile : gcc -o filename-fopenmp filename.c

run : ./filename

OpenMP : shared memory space

- OpnMP : API for parallel programming of microprocessors
- OpenMP is 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
- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors

Shared Memory with **thread** based parallelism

OpenMP ⇒ Shared Memory

MPI ⇒ Distributed Memory

Create multiple threads for "for" block

#pragma omp parallel for

Pragma Stands for "pragmatic information"
A way for the programmer to communicate with the compiler

int n = omp_get_num_procs ( )

int omp_get_num_threads ( ) # No. of Threads in Use

int omp_get_thread_num (void) # Get current Thread ID

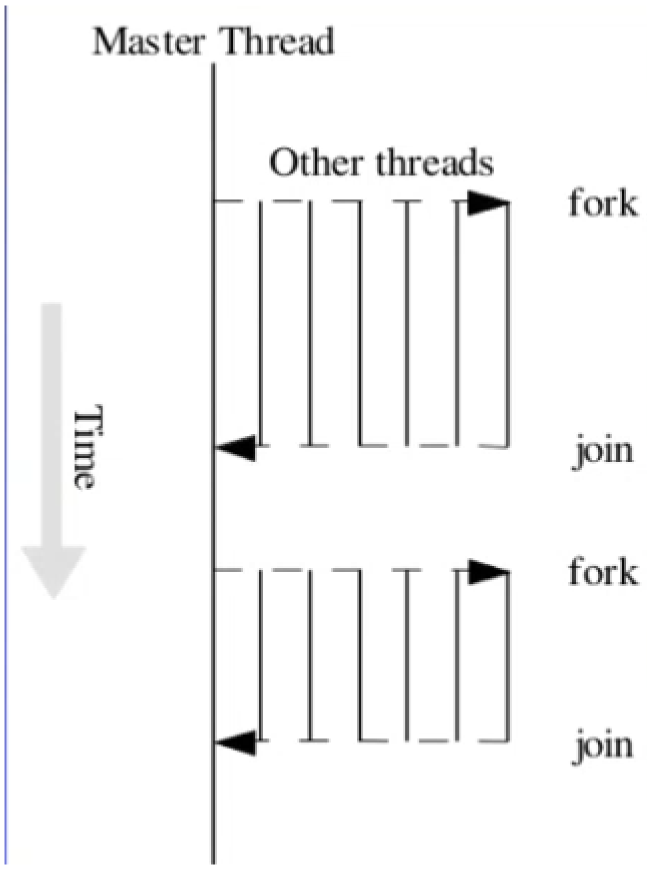
Master TID = 0

Thread

- Independent Instruction stream , **allows concurrent operations**
- Threads tend to **share state and memory** information and may have some (usually small) private data
- Similar (but distinct) from processes.
- Threads are usually lighter weight allowing faster **context switching** **[Multiple Processors can use a resource]**
- In OpenMP one usually wants **"no more than one thread per core"**
- Loop 1 to 5 ; 4 Threads ; Each threads executed the loop from1 to 5

OpenMP uses the fork-join model of parallel execution

- OpenMP programs begin with a single master thread.
- The **master thread executes sequentially** until a parallel region is encountered, when it **creates** a team of **parallel threads (FORK)**.
- When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN).



Header ⇒ #include<omp.h>

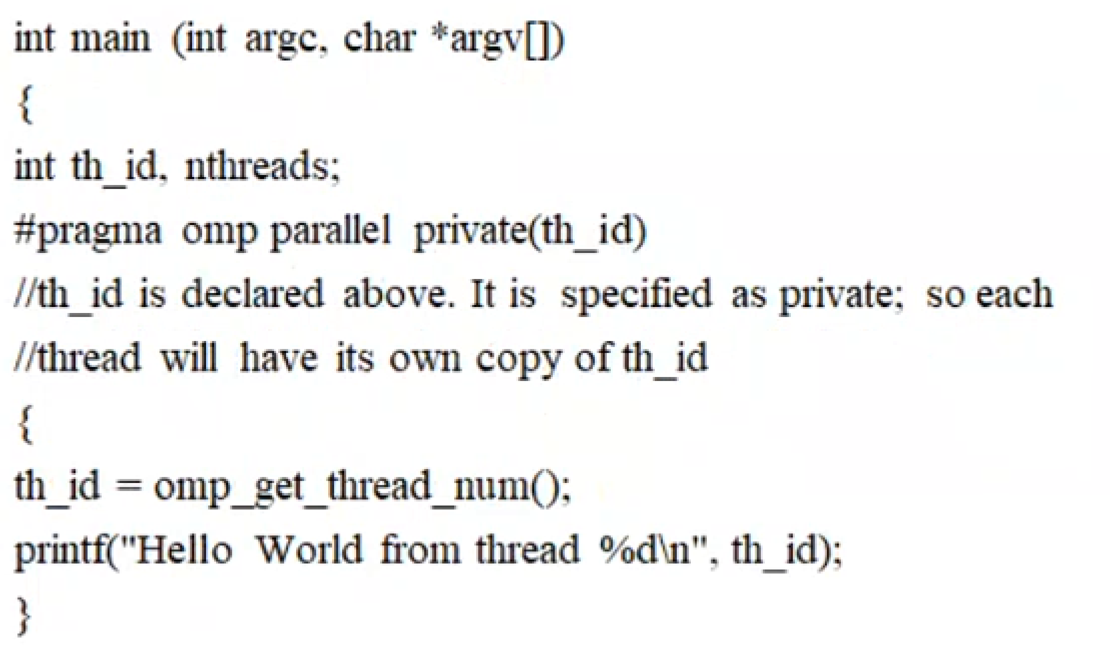
Incremental Parallelization

- Start with a serial program that is organized to perform its analysis using a looping construct
- Execute and profile the sequential program
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit at a time
- Parallel shared-memory programs may only have a **single parallel loop**
- Stop when further effort not warranted – no increase in speed or efficiency is observed

---

First OpenMP Code “Hello World”

> ### Each Thread prints “Hello” and it's own copy of “Thread_id”



```
int main (int argc, char *argv[])
{
int th_id, nthreads;
#pragma omp parallel private(th_id)
//th_id is declared above. It is specified as private; so each
//thread will have its own copy of th_id
{
th_id = omp_get_thread_num();
printf("Hello World from thread %d\n", th_id);
}
```

Private and shared variables

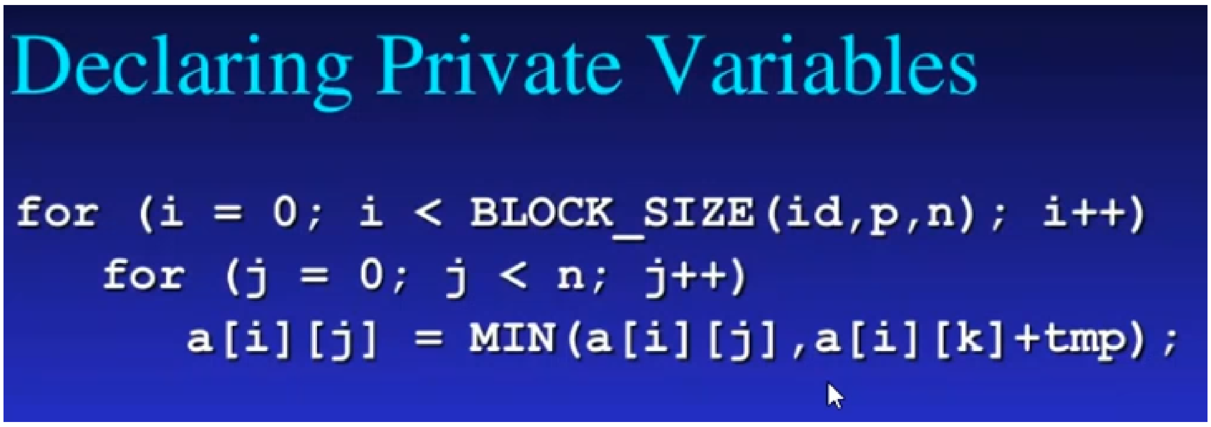
- Variables in the global data space are accessed by all parallel threads (shared variables)
- Variables in a thread's private space can only be accessed by the thread (private variables)

In a parallel section variables can be private or shared:

private

- The variable is private to each thread, which means each thread will have its own local copy.
- A private variable is not initialized and the value is not maintained for use outside the parallel region.
- By default, the **loop iteration counters** in the OpenMP loop constructs are **private**.

  i is different (private) by default



Declaring Private Variables

```
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
   for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j],a[i][k]+tmp);
```

j is common ⇒ Make it private to avoid race condition

Used to **create private variables having initial values identical** to the variable controlled by the master thread as the loop is entered

> Variables are initialized **once per thread**, not once per loop iteration

If a thread modifies a variable's value in an iteration, subsequent iterations will get the modified value

shared

The variable Is shared, which means it is visible to and accessible by all threads simultaneously.

By default, all variables in the work sharing region are shared except the loop iteration counter.

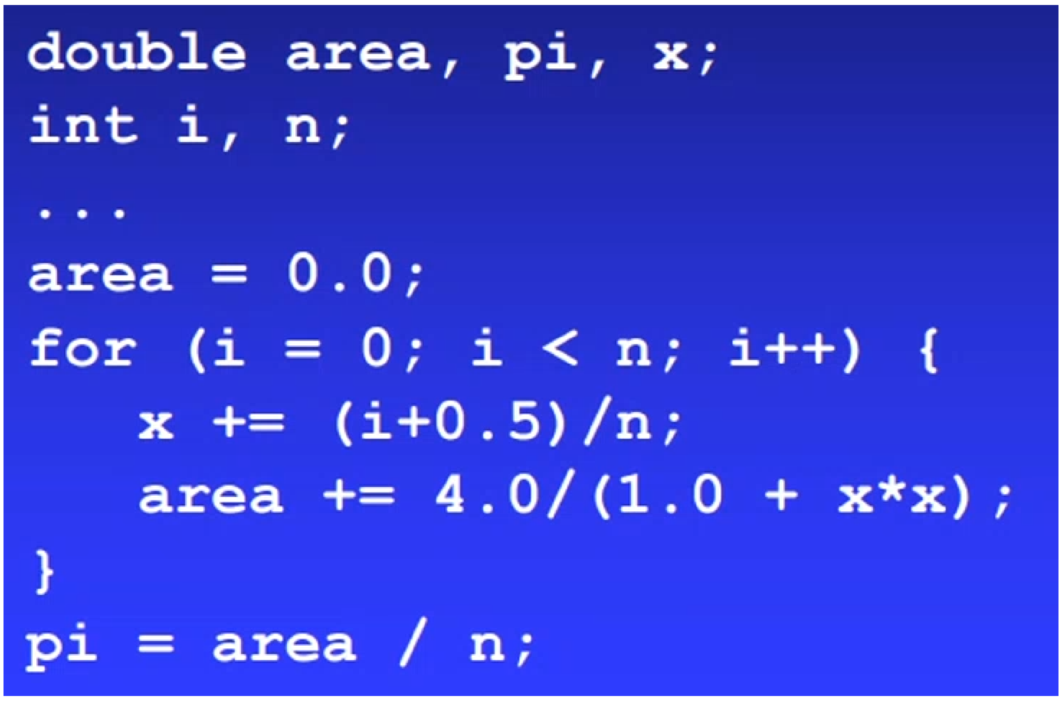
> Shared variables must be used with care because they cause race conditions.

Critical Section and Race Condition

Only one thread executes in a critical section (shared variable) during concurrent processing

Race Condition! : N threads accessing a shared variable at the same time

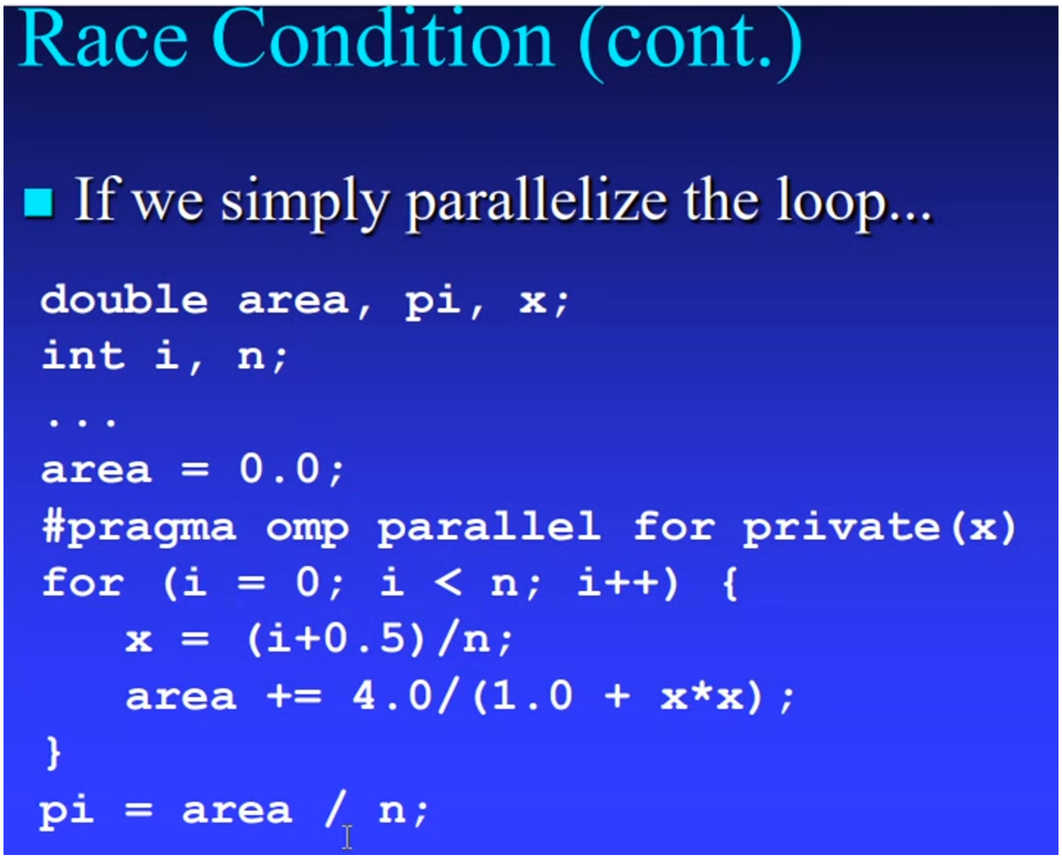
Consider this C program segment to compute pi using the rectangle rule:



```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
    x += (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

> **x** : private ; pi : shared

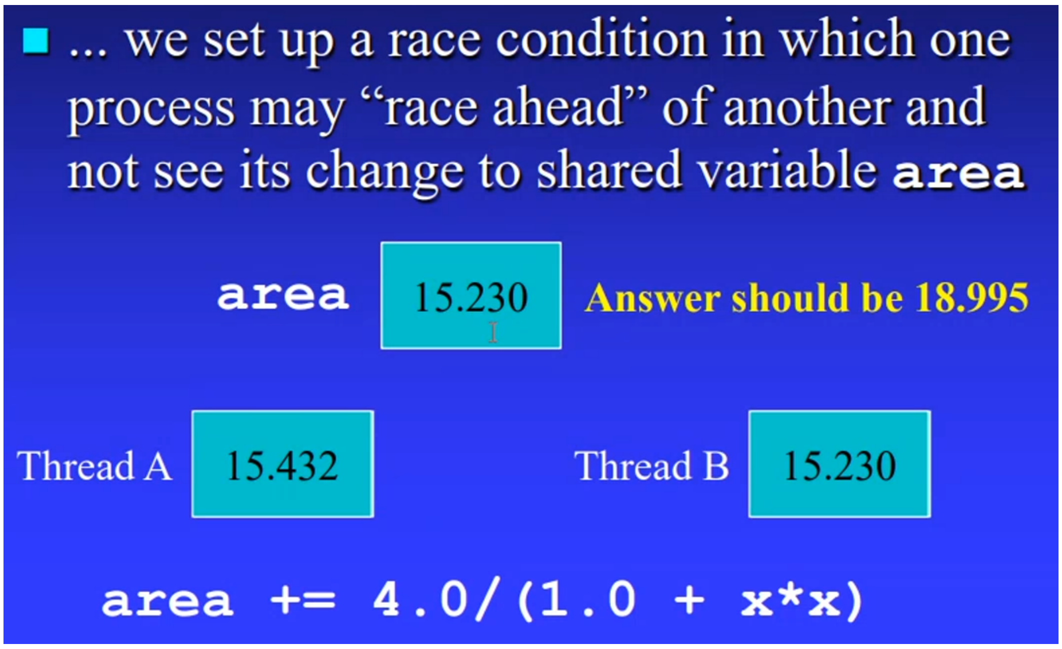
Simple Parallelization **without** critical section



Race Condition (cont.)

- If we simply parallelize the loop...

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```



- ... we set up a race condition in which one process may "race ahead" of another and not see its change to shared variable **area**

Critical pragma

- **Critical Section :** Sometimes it is necessary to limit a piece of code so that it can be executed by only one tread at a time. Such a piece of code is called a critical section
- We denote a critical section by putting the pragma

#pragma omp critical

- **Critical** : It is general and can contain an arbitrary sequence of Instructions;
- If your code has multiple critical sections, they are all **mutually exclusive**: if a thread is in one critical section, the other ones are all blocked.
- Critical sections are an easy way to turn an existing code into a correct parallel code. However, there are performance disadvantages to critical sections, and sometimes a more drastic rewrite is called for.

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  double tmp = some_function(mytid);
#pragma omp critical
  sum += tmp;
}
```

Make the area where we update as critical

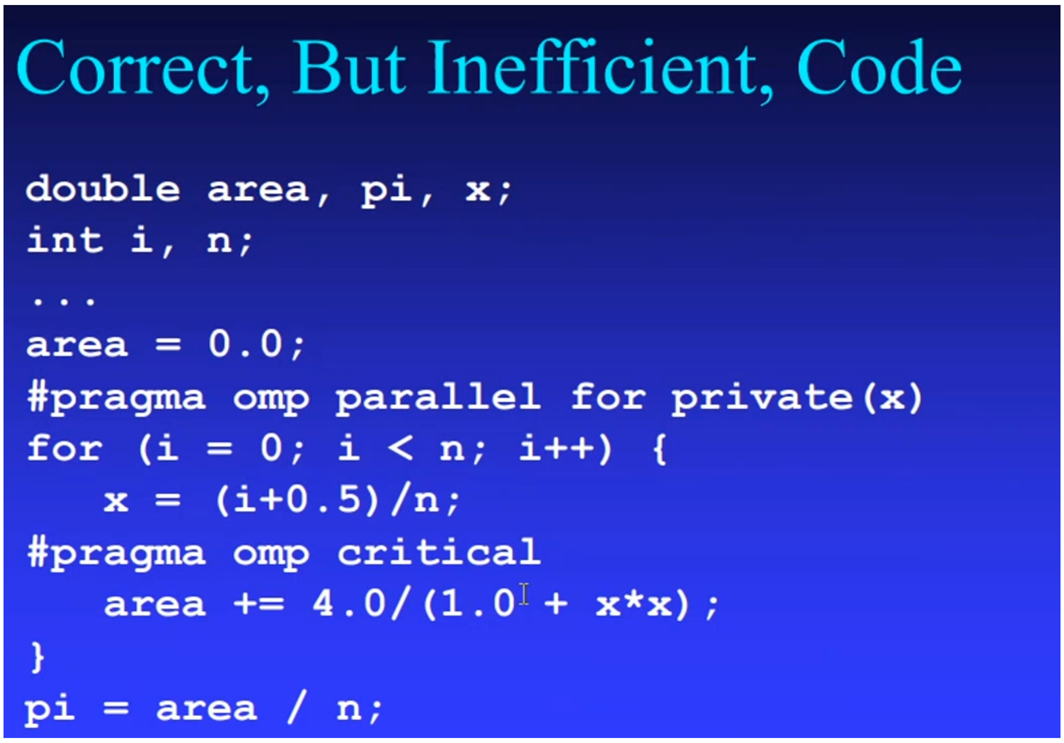
```
int count[100];
float x = some_function();
int ix = (int)x;
if (ix>=100)
  error();
else
  count[ix]++; {Updation => Make Critical)
```

It would be possible to guard the last line:

```
#pragma omp critical
  count[ix]++;
```

EREW ⇒ Worst / Waste

ERCW / CRCW ⇒ Better



Correct, But Inefficient, Code

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
#pragma omp critical
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Source of Inefficiency - Critical clause

• Update to area inside a critical section
• Only one thread at a time may execute the statement; i.e., it is sequential code
• Time to execute statement significant part of loop
• By Amdahl's Law we know speedup will be severely constrained

Inefficient ⇒ Multiple Forks and Joins

Reductions clause

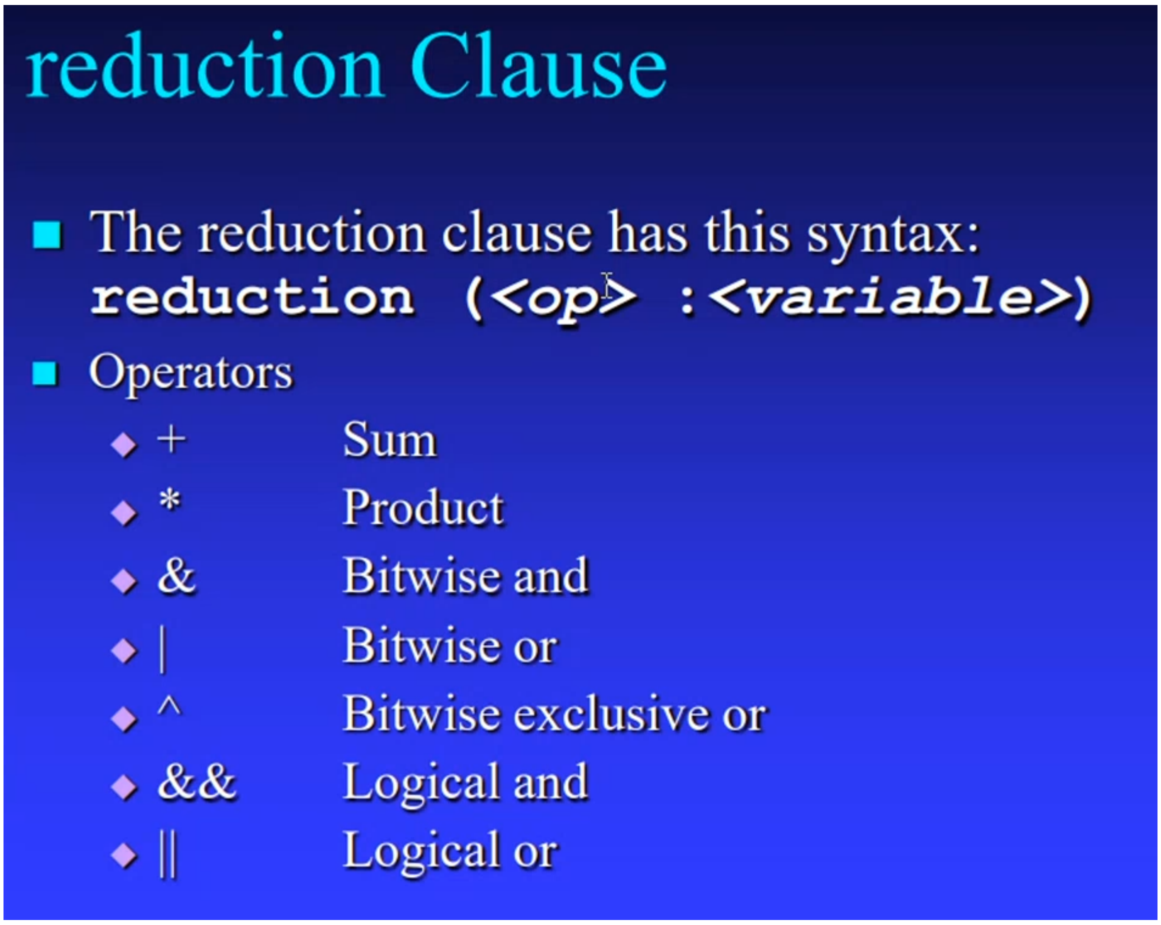
Too many fork/joins can lower performance

NOTE : You have to make an additional join in the critical section if you don't use Reduction

Instead of Critical Section use Reduction

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to parallel for pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of **storing partial results** in **private variables** and **combining partial results** after the loop

- During the Loop execution : It stores partial area results
- At the end of the Loop : Combine partial area results

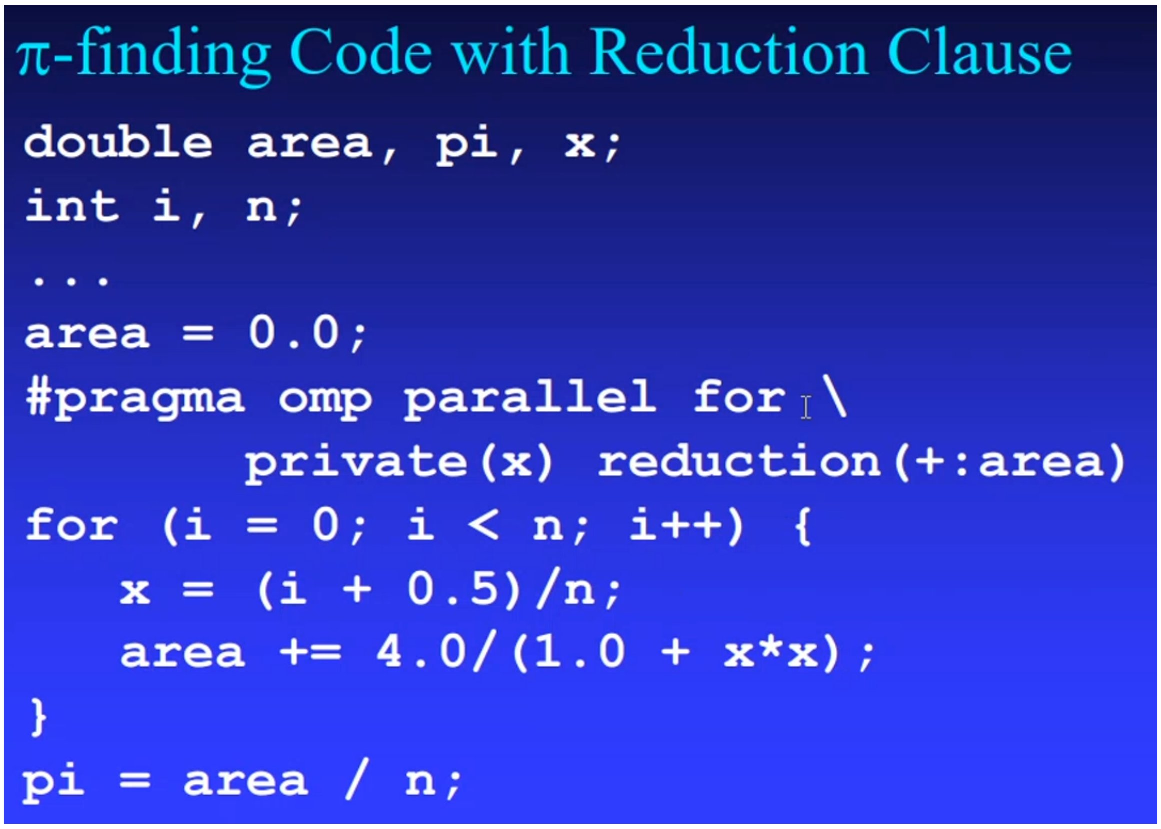


reduction Clause

- The reduction clause has this syntax:
  `reduction (<op> :<variable>)`
- Operators

+	Sum
*	Product
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
&&	Logical and
||	Logical or

op : operator $\Rightarrow$ Here, operator is +



π-finding Code with Reduction Clause

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \
       private(x) reduction(+:area)
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Performance Improvement #1

- Inverting loops may help performance if
  - Parallelism is in inner loop
  - After inversion, the **outer loop can be made parallel**
  - Inversion does not significantly lower cache hit rate
- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution

If : Performance Improvement #2

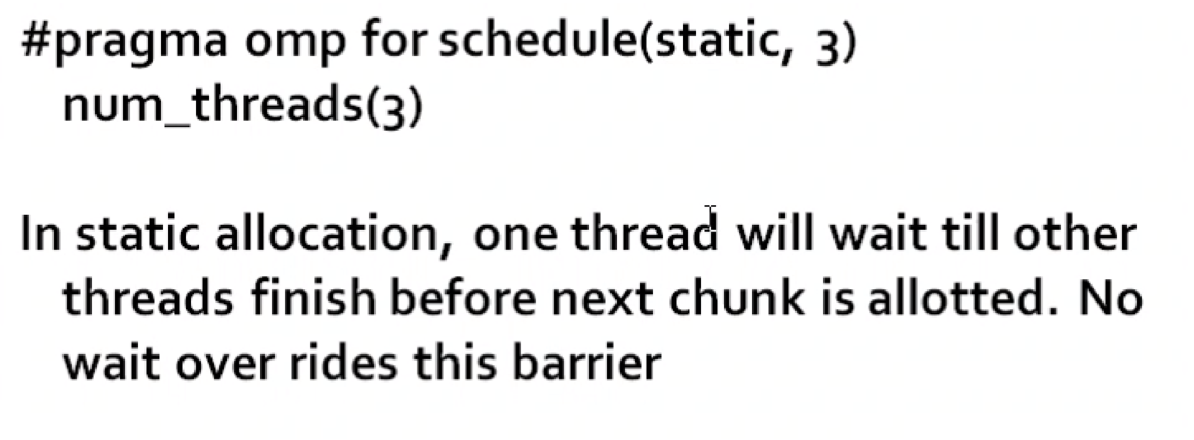
- The **if clause** instructs compiler to insert code that determines at run-time whether loop should be executed in parallel; e.g.,
  #pragma omp parallel for if (n > 5000)
- "n" is given at runtime
- Limit the loops executing in parallel

Schedule : Performance Improvement #3

- The **Schedule** clause specifies how iterations of a loop should be allocated to threads (loop work sharing construct)
- Static schedule: all iterations allocated to threads before any iterations executed
- Dynamic schedule: only some iterations allocated to threads at beginning of loop's execution.
  Remaining iterations allocated to threads that complete their assigned iterations.

1. Static

• Low overhead [Iterations of a loop are Pre-allocated to the threads]
• May exhibit high workload imbalance



> #pragma omp for schedule(static, 3) num_threads(3)
>
> In static allocation, one thread will wait till other threads finish before next chunk is allotted. No wait over rides this barrier

2. Dynamic

schedule(dynamic [,chunk])
Each thread grabs "chunk" iterations off a queue until all iterations have been handled.

In dynamic scheduling, using a "chunksize" of 10:

• thread 1 is assigned to do iterations 1 to 10.
• thread 2 is assigned to do iterations 1 to 20.
• thread 3 is assigned to do iterations 21 to 30.

The next chunk is iterations 31 to 40, and will be assigned to whichever thread finishes its current work first, and so on until all work is completed.

• Higher overhead
• Can reduce workload imbalance

Eg : Dynamic Scheduling : 0-5 Thread performs Iteration 1 , ...

no. of threads = no. of iterations ⇒ yields a good result

3. Guided

schedule(guided[,chunk])

Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.

Synchronization

Barrier Directive

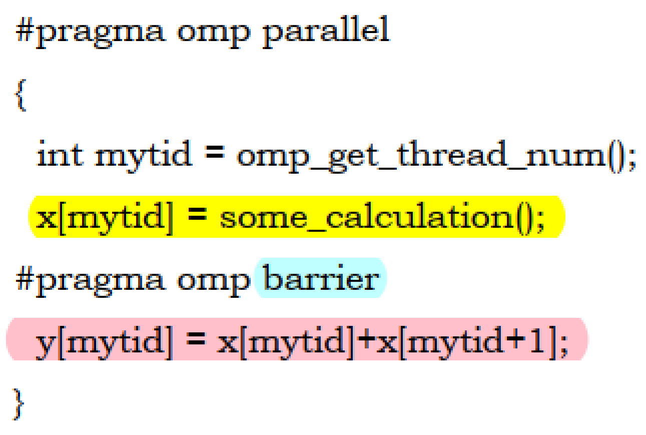
A point in the code where **all active threads will stop** until **all threads** have **arrived** at that point.

With this, we can **guarantee that certain calculations are finished**.

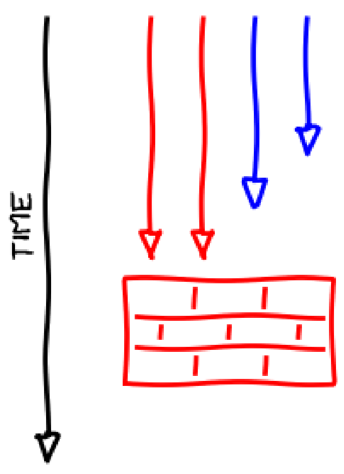
Eg:

Unless we have the value of X , we can't compute Y , thus barrier guarantees that value of X is being computed

Explicit Barrier



```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
#pragma omp barrier
  y[mytid] = x[mytid]+x[mytid+1];
}
```



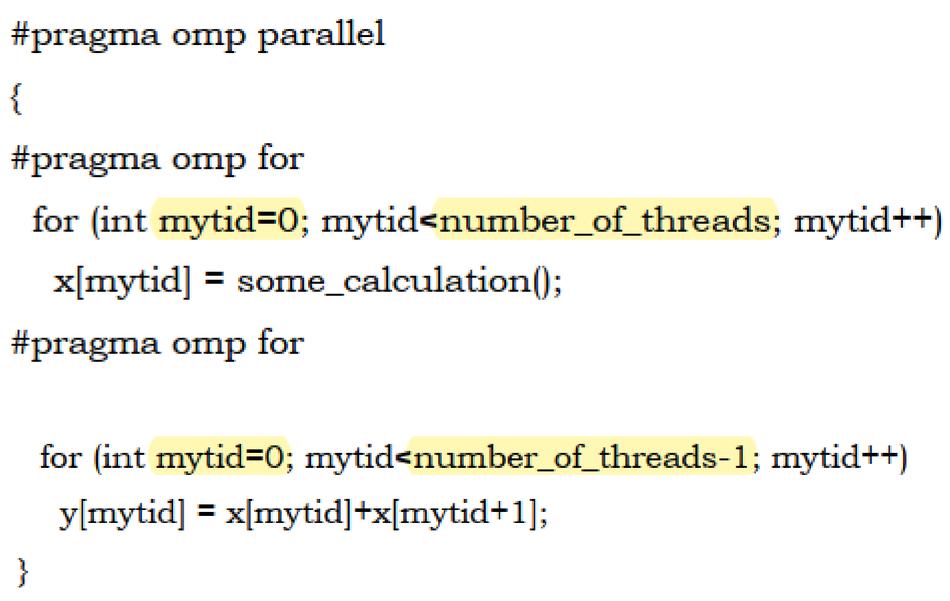
OpenMP: Barrier

What is a barrier? It is a point in the execution of a program where threads wait for each other. No thread is allowed to continue until all threads in a team reach the barrier. Basically, a barrier is a synchronization point in a program. We can visualize it with a wall.

http://jakascorner.com/blog/2016/07/omp-barrier.html

Implicit - looping construct

Using number_of_threads as n and thread_id as looping variable



```
#pragma omp parallel
{
#pragma omp for
  for (int mytid=0; mytid<number_of_threads; mytid++)
    x[mytid] = some_calculation();
#pragma omp for

   for (int mytid=0; mytid<number_of_threads-1; mytid++)
    y[mytid] = x[mytid]+x[mytid+1];
}
```

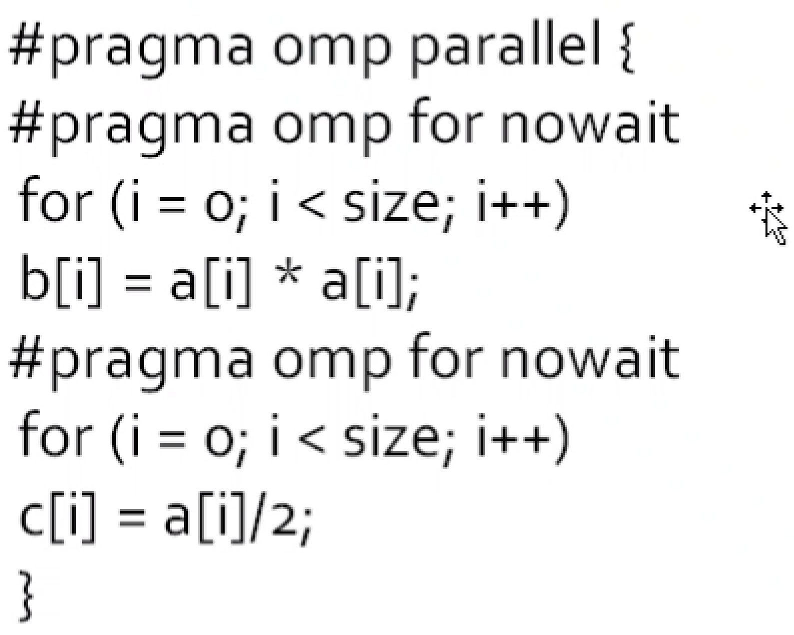
reduced barrier problem , but overhead is increased

No wait clause

- There is an implicit barrier at the end of the loop.
- Nowait overrides the implicit barrier in a directive.
- For example, if we have two parallel for loops
  #pragma omp parallel for
  #pragma omp for
- There will be a barrier at the end of the for loop, thus one thread will be waiting till all the other threads have finished working

Applies to following directives

- For
- Sections
- single

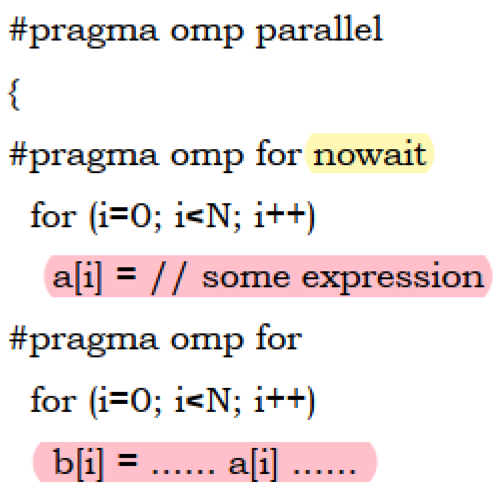


```
#pragma omp parallel {
#pragma omp for nowait
 for (i = 0; i < size; i++)
 b[i] = a[i] * a[i];
#pragma omp for nowait
 for (i = 0; i < size; i++)
 c[i] = a[i]/2;
 }
```

Where there is no dependency , we can use no wait

No Threads have to wait for other threads to complete their execution / iteration

Overhead (Waiting Time) is reduced



```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<N; i++)
    a[i] = // some expression
#pragma omp for
  for (i=0; i<N; i++)
   b[i] = ...... a[i] ......
```

Here the nowait clause implies that threads can start on the second loop
while other threads are still working on the first.

Mutual Exclusion

To avoid synchronization issues, race condition

Mutex Lock , Thread 1 won't be able to access shared variable until Thread 0 completes execution

Atomic

- It is more limited but has **performance advantages**.
- Atomic **provides mutual exclusion** but only applies to the **update of a memory location**

> #pragma omp atomic

- ATOMIC : Limits the update of a single memory location

Master

The MASTER directive specifies a **region that is to be executed only by the master thread** of
the team. All other threads on the team skip this section of code

There is no implied barrier associated with this directive

Ordered

Specifies that code under a parallelized for loop should be executed like a sequential loop.

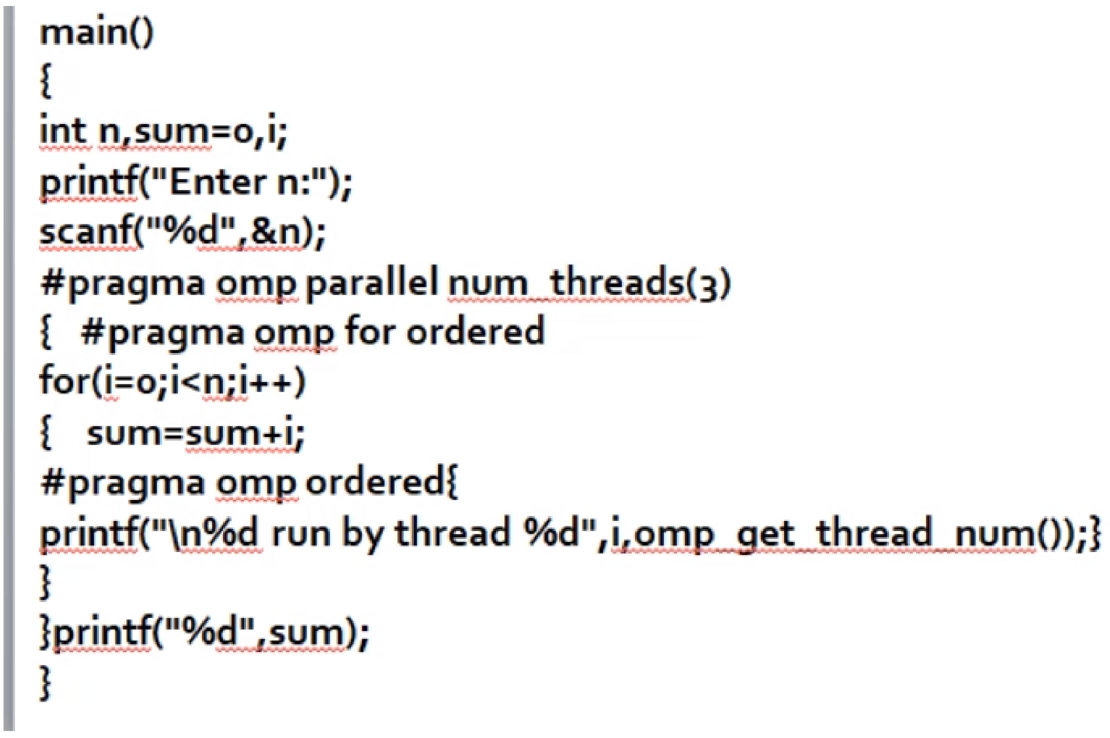
The **ordered** directive supports no OpenMP clauses other than **for** loop.
#pragma omp ordered
    structured-block

The omp ordered directive must be used as follows:

- It must appear within the extent of a omp forlor omp parallel for construct containing an ordered clause.
- It applies to the statement block immediately following it.
- Statements in that block are executed in the same order in which iterations are **executed in a sequential loop.**
- An iteration of a loop must not execute the same omp ordered directive more than once.
- An iteration of a loop must not execute more than one distinct omp ordered directive.

Iteration : 0 ; Thread_Id : 0

Iteration : 1 ; Thread_Id : 1



```
main()
{
int n,sum=0,i;
printf("Enter n:");
scanf("%d",&n);
#pragma omp parallel num_threads(3)
{   #pragma omp for ordered
for(i=0;i<n;i++)
{   sum=sum+i;
#pragma omp ordered{
printf("\n%d run by thread %d",i,omp_get_thread_num());}
}
}printf("%d",sum);
}
```

---

2 Methods

- Turning the update statement into a **critical section**.
- Letting the threads accumulate into a **private variable (temp)**and summing these after the loop.

LOCKS

> Protect resources with locks.

lock variable must have type **omp_lock_t**.

The **omp_init_lock** function initializes a simple lock.

The **omp_destroy_lock** function removes a simple lock.

The **omp_set_lock** function waits until a simple lock is available.

The **omp_unset_lock** function releases a simple lock.

Locks are much more efficient than critical section

Create/destroy:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

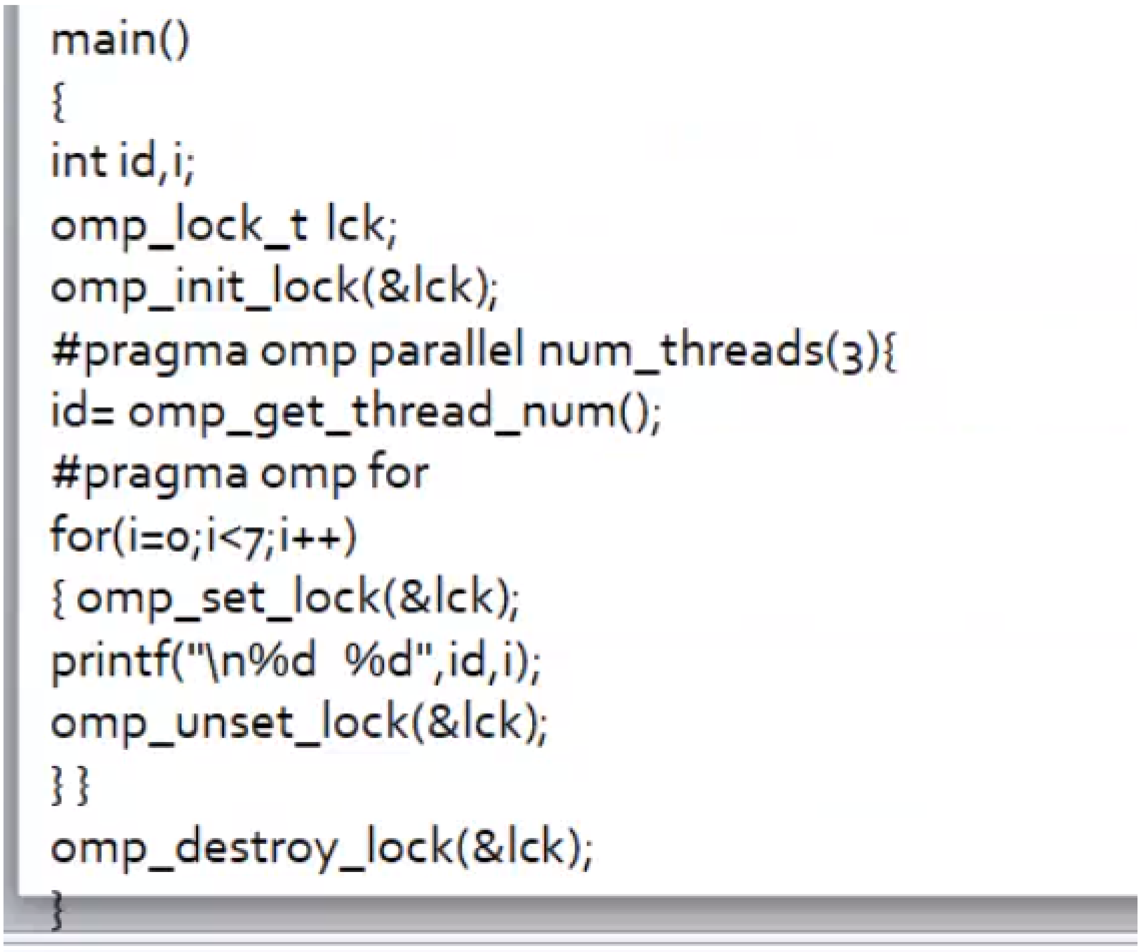
```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Since the set call is blocking, there is also

```
omp_test_lock();
```

Unsetting a lock needs to be done by the thread that set it.

Lock operations implicitly have a flush .



```
main()
{
int id,i;
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel num_threads(3){
id= omp_get_thread_num();
#pragma omp for
for(i=0;i<7;i++)
{ omp_set_lock(&lck);
printf("\n%d  %d",id,i);
omp_unset_lock(&lck);
} }
omp_destroy_lock(&lck);
}
```

NOTE : for creating or destroying locks you don't need OpenMP

Deadlock Problem - Solve using LOCKS

Process 1 ⇒ Set(A) ; Update(B)

Process 2 ⇒ Set(B) ; Update(A)

Shared (A , B , LockA , LockB)

▼ Deadlock Problem - LOCKS

```
#pragma omp parallel shared(a, b, nthreads, locka, lockb)

  #pragma omp sections nowait

    {
    // PROCESS 1

    #pragma omp section
      {

    // Setting Locks
      omp_set_lock(&locka);
      for (i=0; i<N; i++)
        // Set A
        a[i] = ..

      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
        // Update B
        b[i] = .. a[i] ..

    //Unsetting Locks
      omp_unset_lock(&lockb);
      omp_unset_lock(&locka);

      }

    //PROCESS 2

    #pragma omp section
      {
      // Setting Locks
      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
      // Set B
        b[i] = ...

      omp_set_lock(&locka);
      for (i=0; i<N; i++)

        // Update A
        a[i] = .. b[i] ..

      //Unsetting Locks
      omp_unset_lock(&locka);
      omp_unset_lock(&lockb);
      }
    }  /* end of sections */
  }  /* end of parallel region */
```

OpenMP compiler directives

Handling List of tasks

- Every thread takes a next task from the list and completes it.
- The threads continue to remove tasks from the list until there are no more tasks.
- Must ensure **no two threads take the same task** from the list, e.g., a critical section can be declared.

A sequential code version of the task list

```
int main (int argc, char *argv[])
{
  struct job_struct  *job_ptr;
  struct task_struct *task_ptr;

  ...
  task_ptr = get_next_task (&job_ptr);
  while (task_ptr != NULL) {
    complete_task (task_ptr);
    task_ptr = get_next_task (&job_ptr);
  }
  ...
}
```

get_next_task() function

```
char *get_next_task(structjob_struct**job_ptr)
{
   struct task_struct *answer;

   if (*job_ptr == NULL) answer = NULL;
   else {
      answer = (*job_ptr)->task;
      *job_ptr = (*job_ptr)->next;
   }
   return answer;
}
```

Job Assigns the Tasks , if not Null

The parallel pragma

The parallel pragma precedes a block of code that should be executed by *all* of the threads.

```
#pragma omp parallel private(task_ptr)
{
   task_ptr = get_next_task (&job_ptr);
   while (task_ptr != NULL) {
      complete_task (task_ptr);
      task_ptr = get_next_task (&job_ptr);
   }
}
```

We are Updating Answer , Job_ptr ⇒ Need Critical Section

```
char *get_next_task(struct job_struct**job_ptr)
{
   struct task_struct *answer;
#pragma omp critical
   {
   if (*job_ptr == NULL) answer = NULL;
   else {
      answer = (*job_ptr)->task;
      *job_ptr = (*job_ptr)->next;
   }
   }
   return answer;
```

Single pragma

- The SINGLE directive specifies that the enclosed code is to be **executed by only one thread** in the team.

To **see the output of a calculation once** rather than printed by every thread.

May be Useful when dealing with sections of code that are not thread safe

Like : Exception Handling

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single   Only One Thread in execution
      printf ("low > high at (%d)\n", i);
   }
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}

Note, low and high are not private variables.
```

TO DO — Make Low , High Private & Add Nowait Clause

● The compiler puts a barrier synchronization at end of every parallel for statement, block or section.

● The nowait clause indicates that the barrier can be eliminated. That is, the threads can terminate or move ahead independently.

Use of the nowait clause

```
#pragma omp parallel private(i,j,low,high)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single
      printf ("low > high at (%d)\n", i);
   }
#pragma omp for nowait
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

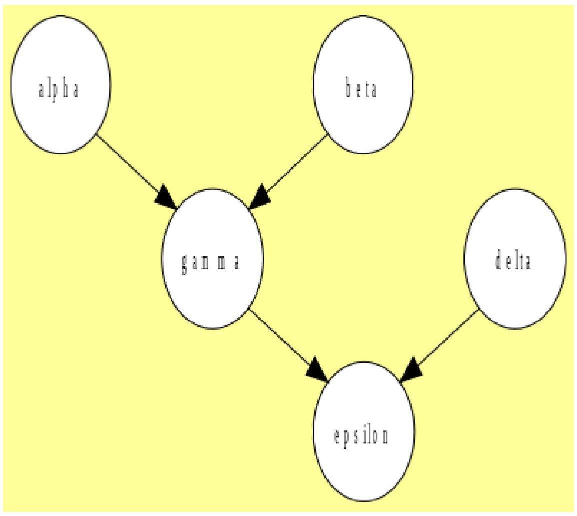
Functional Parallelism - “Sections”

Functional parallelism in OpenMP

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();

printf("%6.2f\n",epsilon(x,y));
```

Functions alpha(), beta(), and delta() may be executed in parallel.



2 Ways

1. Independent ⇒ Alpha , Beta , Delta ⇒ Make them execute in parallel

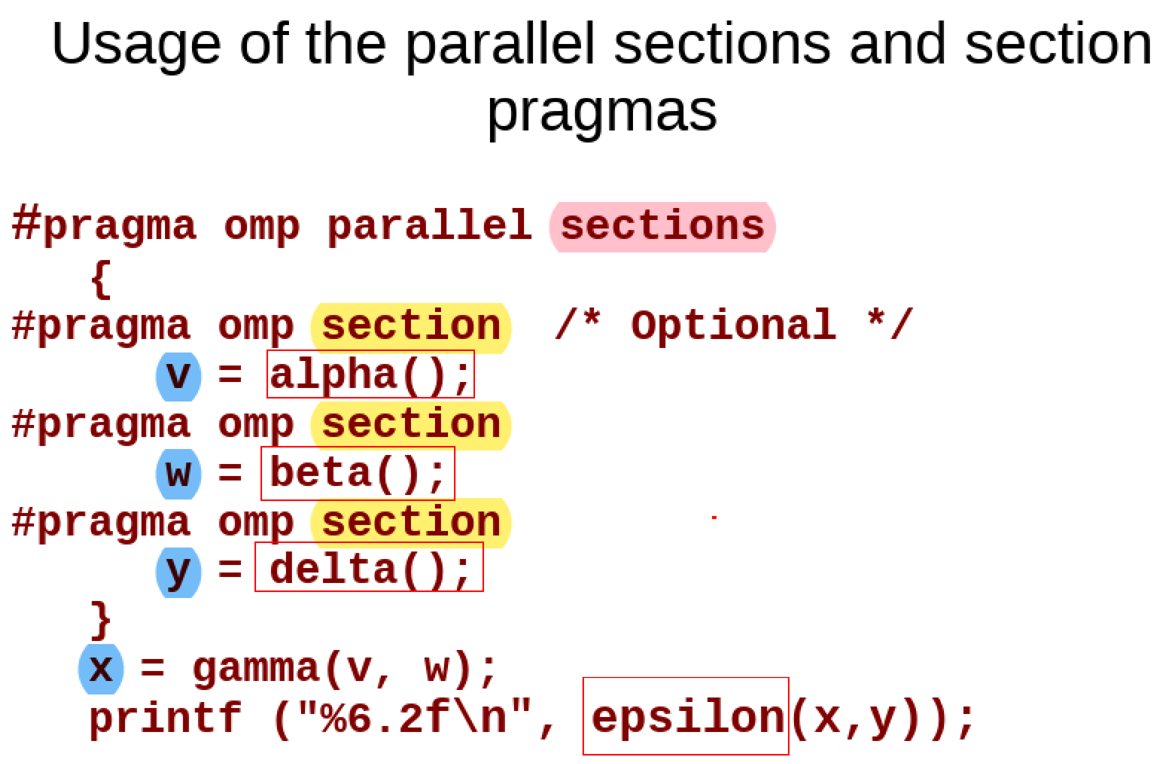
2. Alpha Beta execute in parallel

Once gamma is going to execute , also execute delta in parallel

---

Tasks can be shared among a pool of thread

At the end Master Thread will perform the Join



Usage of the parallel sections and section pragmas

```
#pragma omp parallel sections
    {
#pragma omp section  /* Optional */
        v = alpha();
#pragma omp section
        w = beta();
#pragma omp section
        y = delta();
    }
    x = gamma(v, w);
    printf ("%6.2f\n", epsilon(x,y));
```

The sections pragma

- The sections pragma is used separately from the parallel pragma.
- It appears inside a parallel block of code
- Used within a parallel pragma block, it has the same meaning as the parallel sections pragma

2nd Method (2 Independent Sections)

- Execute the functions alpha() and beta() in parallel.
- Execute gamma() and delta() in parallel.
- Only epsilon() has dependencies that require waiting for gamma() and delta().

Advantage

- If multiple **sections** pragmas are used inside one parallel block, the **fork/join costs may be reduced**.

```
#pragma omp parallel
   {
   #pragma omp sections
      {
         v = alpha();
      #pragma omp section
         w = beta();
      }
   #pragma omp sections
      {
         x = gamma(v, w);
      #pragma omp section
         y = delta();
      }
   }
   printf ("%6.2f\n", epsilon(x,y));
```

References

Computer Revolution (www.comrevo.com): OPENMP

OpenMP (llnl.gov)

Introduction to OpenMP: 01 Introduction - YouTube

Using OpenMP with C — Research Computing University of Colorado Boulder documentation (curc.readthedocs.io)

https://www.appentra.com/parallel-matrix-matrix-multiplication/

https://people.sc.fsu.edu/~jburkardt/c_src/mxm_openmp/mxm_openmp.c