OpenMP

Riccardo Caprile

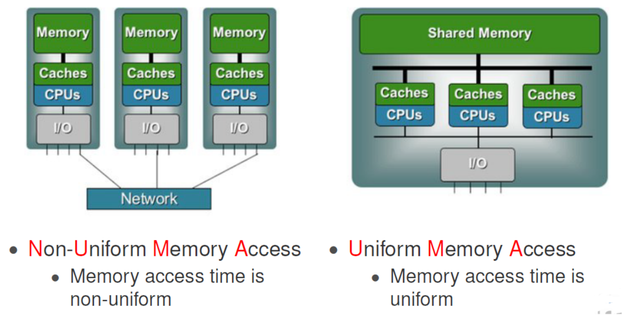
November 2022

1 OpenMP_Core

OpenMP = Opem Multi-Parallelism
It is an API to explicitly direct multi-threaded shared memory parallelism

1.1 Shared Memory Architectures

All processors may access the whole main memory.


Memory
Caches
CPUs
I/O
Network
Shared Memory
Non-Uniform Memory Access
Memory access time is non-uniform
Uniform Memory Access
Memory access time is uniform

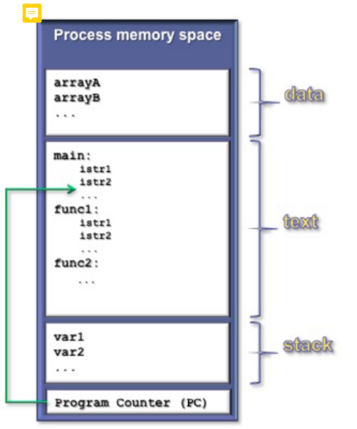
1.2 Process and Thread

A process is an instance of a computer program.
Some information included in a process are :

- Text : Machine code
- Data: Global variables
- Stack: Local Variables
- Program Counter (PC) : A pointer to the instruction to be executed.
- Heap : Pointers

1.3 Operating System Memory Model

A process , owns a lot of state information including the memory, file handles.
OS provide a separate address space for each process.
One process cannot see the memory of another process.
Memory within a process is managed separately : text and data segment (static read only areas for code and data known at compile time), the heap (an area of memory managed by the operating system for dynamic data allocation), the stack (a piece of memory) managed by the process itself.
Multiple threads launched from the same process share the same address space and memory.



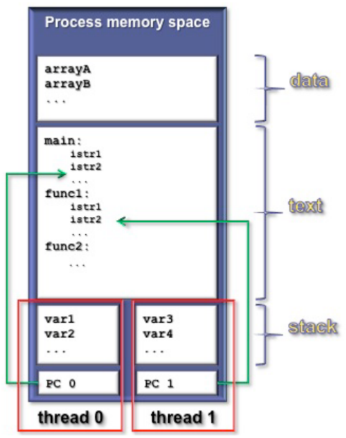
1.4 Multi-Threading

The process contains several concurrent execution flows(threads).

Each thread has its own program counter.

Each thread has its own private stack.

The instruction executed by a thread can access : the process gloval memory(data) and the thread local stack.



Threading exploits more parallelism to increase scaling and performance.

1.4.1 Alternatives 1 - Pthreads

Pthreads provides the ability to dynamically create threads which are launched to run a specific task.

Provides mutex and condition variables for synchronisation.

1.4.2 Alternatives 2 - TBB

Intel Thread Building Block is a C++ template library that adds parallel programming for C++ programmers.

1.5 So why OpenMP?

OpenMP is ubiquitous : available with almost all compilers.

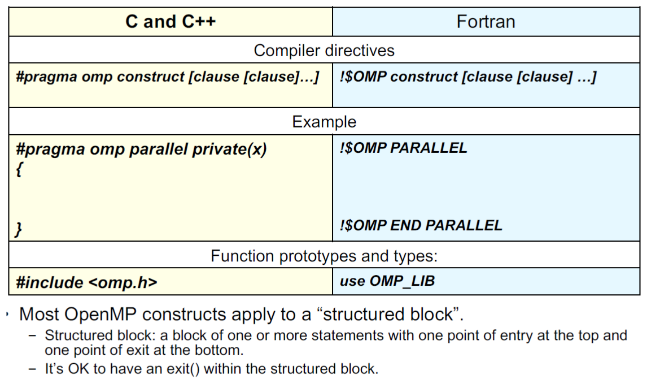
Easy interface available for incremental parallelism, best fit for data-parallel codes.

1.6 How does OpenMP work?

Teams of OpenMP threads are created to perform the computation in a code :

- Work is divided among the threads, which run on the different cores
- The threads collaborate by sharing variables
- Threads synchronize to order acesses and prevent data corruption
- Structured programming is encouraged to reduce likelihood of bugs

1.7 OpenMP Basic Syntax



C and C++	Fortran
Compiler directives	
#pragma omp construct [clause [clause]…]	***!$OMP construct [clause [clause] …]***
Example	

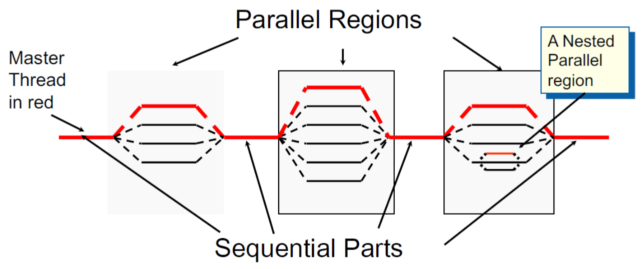
***#pragma omp parallel private(x)
{
 	
}***
	***!$OMP PARALLEL
	
	!$OMP END PARALLEL***

Function prototypes and types:	
#include <omp.h>	***use OMP_LIB***

- Most OpenMP constructs apply to a “structured block”.
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It’s OK to have an exit() within the structured block.

1.8 OpenMP Programming Model

Fork-Join Parallelism : Master thread spawns a team of threads as needed. Parallelism added incrementally until performance goals are met (the sequential program evolves into a parallel program).



1.9 OpenMP Memory Model

All threads have access to the same , globally shared, memory.

Data can be shared or private.

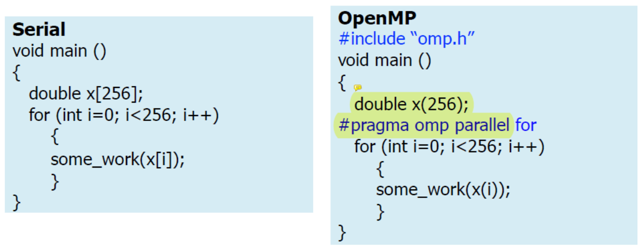
Shared data is accessible by all threads.

Private data can only be accessed by the thread that owns it.

Data transfer is transparent to the programmer.

Synchronization takes place , but it is mostly implicit.

1.10 Serial vs. OpenMP



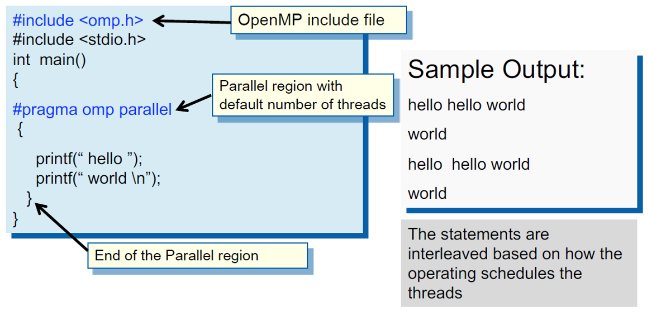
Serial
```
void main ()
{
   double x[256];
   for (int i=0; i<256; i++)
      {
      some_work(x[i]);
      }
}
```

OpenMP
```
#include "omp.h"
void main ()
{
   double x(256);
#pragma omp parallel for
   for (int i=0; i<256; i++)
      {
      some_work(x(i));
      }
}
```

OpenMP is a simple programming model. Single source code for serial and parallel codes, portable implementation.

1.11 A Multi-Threaded "Hello World" Program

Write a multithreaded program where each threads prints "hello world".



```
#include <omp.h>
#include <stdio.h>
int  main()
{
#pragma omp parallel
 {
    printf(" hello ");
    printf(" world \n");
  }
}
```

OpenMP include file

Parallel region with default number of threads

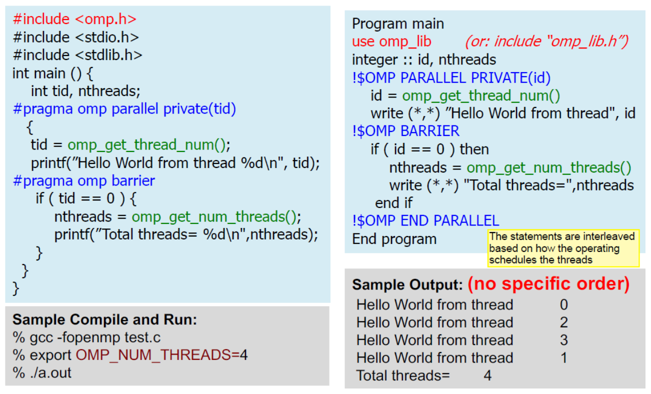
End of the Parallel region

Sample Output:

```
hello hello world
world
hello  hello world
world
```

The statements are interleaved based on how the operating schedules the threads

1.12 A simple OpenMP program



```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main () {
    int tid, nthreads;
#pragma omp parallel private(tid)
   {
    tid = omp_get_thread_num();
    printf("Hello World from thread %d\n", tid);
#pragma omp barrier
     if ( tid == 0 ) {
         nthreads = omp_get_num_threads();
         printf("Total threads= %d\n",nthreads);
     }
   }
}
```

Sample Compile and Run:

```
% gcc -fopenmp test.c
% export OMP_NUM_THREADS=4
% ./a.out
```

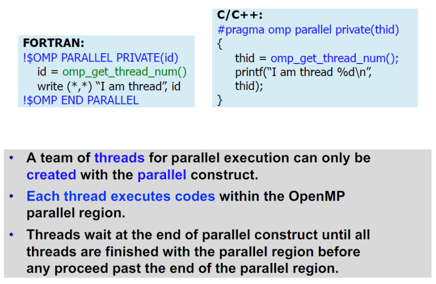
```
Program main
use omp_lib       (or: include "omp_lib.h")
integer :: id, nthreads
!$OMP PARALLEL PRIVATE(id)
    id = omp_get_thread_num()
    write (*,*) "Hello World from thread", id
!$OMP BARRIER
    if ( id == 0 ) then
        nthreads = omp_get_num_threads()
        write (*,*) "Total threads=",nthreads
     end if
!$OMP END PARALLEL
End program
```

The statements are interleaved based on how the operating schedules the threads

Sample Output: **(no specific order)**

```
Hello World from thread        0
Hello World from thread        2
Hello World from thread        3
Hello World from thread        1
Total threads=        4
```

1.13 Thread Creation : The parallel construct



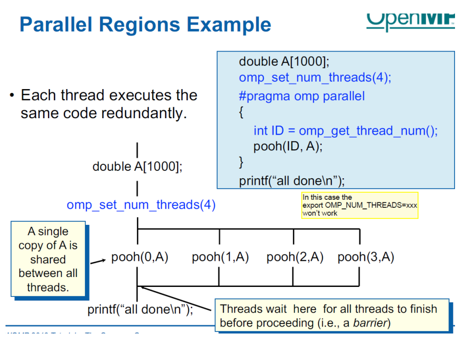
FORTRAN:

```
!$OMP PARALLEL PRIVATE(id)
   id = omp_get_thread_num()
   write (*,*) "I am thread", id
!$OMP END PARALLEL
```

C/C++:

```
#pragma omp parallel private(thid)
{
    thid = omp_get_thread_num();
    printf("I am thread %d\n",
    thid);
}
```

- **A team of threads for parallel execution can only be created with the parallel construct.**
- **Each thread executes codes within the OpenMP parallel region.**
- **Threads wait at the end of parallel construct until all threads are finished with the parallel region before any proceed past the end of the parallel region.**



Parallel Regions Example

- Each thread executes the same code redundantly.

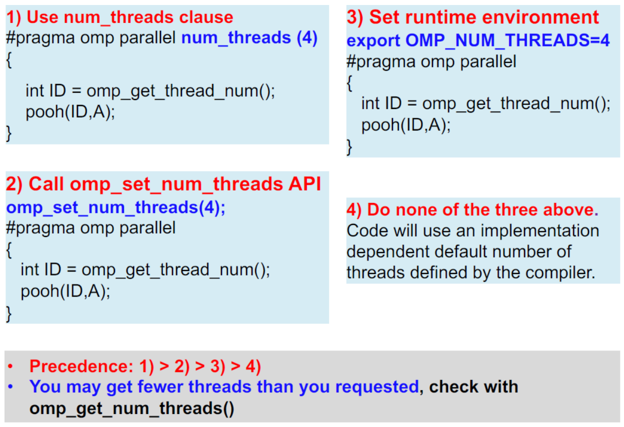
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID, A);
}
printf("all done\n");
```

double A[1000];
omp_set_num_threads(4)
In this case the export OMP_NUM_THREADS=xxx won't work
A single copy of A is shared between all threads.
pooh(0,A)
pooh(1,A)
pooh(2,A)
pooh(3,A)
printf("all done\n");
Threads wait here for all threads to finish before proceeding (i.e., a *barrier*)

1.14 Single Worksharing Construct

The single construct denotes a block of code that is executed by only one thread. A barrier is implied at the end of the single block. (#pragma omp parallel).

1.15 Various Methods to set #threads



1) Use num_threads clause

```
#pragma omp parallel num_threads (4)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

2) Call omp_set_num_threads API

```
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
```

3) Set runtime environment

```
export OMP_NUM_THREADS=4
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
```

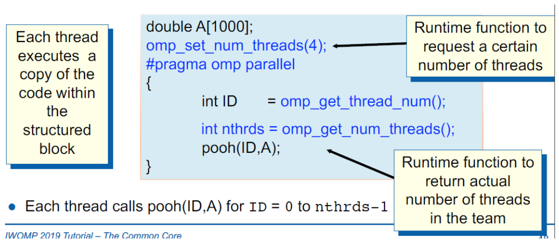
4) Do none of the three above.
Code will use an implementation dependent default number of threads defined by the compiler.

- **Precedence: 1) > 2) > 3) > 4)**
- **You may get fewer threads than you requested, check with omp_get_num_threads()**

1.16 Thread creation : How Many Threads Did you Actually Get?

You create a team threads in OpenMP with the parallel construct.

You can request a number of threads with **omp_set_num_threads()**. But is the number of threads requested the number you actually get? NO!, an implementation can silently decide to give you a team with fewer threads.



```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID     = omp_get_thread_num();
        int nthrds = omp_get_num_threads();
        pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block
Runtime function to request a certain number of threads
Runtime function to return actual number of threads in the team
IWOMP 2019 Tutorial – The Common Core

- Each thread calls pooh(ID,A) for `ID = 0` to `nthrds-1`

1.17 Performance Tips

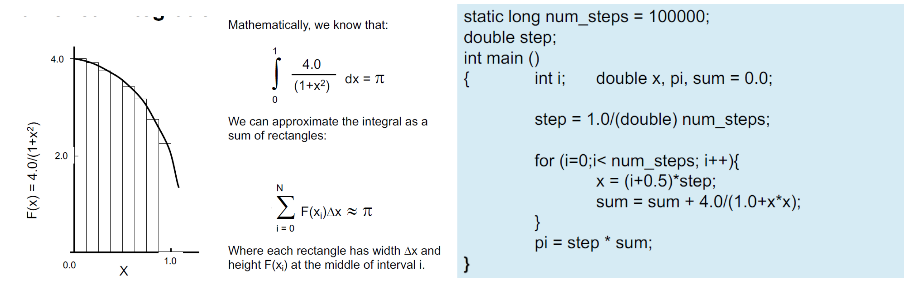
Experiment to find the best number of threads on your system. Put as much code as possible inside parallel region , have large parallel regions , run time routines are your friend , barriers are expensive.

1.18 Orphaned Directives

Orphaning is a situation when directives related to a parallel region are not required to occur lexically within a single program unit.

Orphaned directives enable parallelism to be inserted into existing code with a minimum of code restructuring.

1.19 An interesting Pi Program - Numerical Integration



Mathematically, we know that:

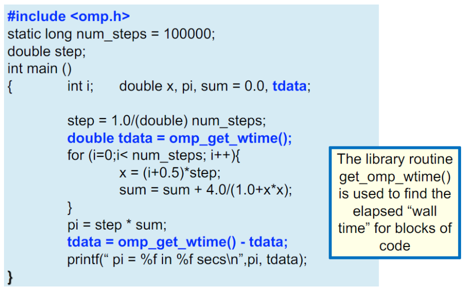
$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

```
static long num_steps = 100000;
double step;
int main ()
{           int i;      double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```



```
#include <omp.h>
static long num_steps = 100000;
double step;
int main ()
{           int i;      double x, pi, sum = 0.0, tdata;

            step = 1.0/(double) num_steps;
            double tdata = omp_get_wtime();
            for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
            tdata = omp_get_wtime() - tdata;
            printf(" pi = %f in %f secs\n",pi, tdata);
}
```

The library routine get_omp_wtime() is used to find the elapsed "wall time" for blocks of code

1.19.1 Exercise : the Parallel Pi Program

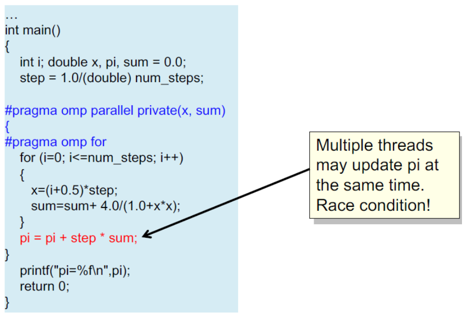
Create a parallel version of the pi program using a parallel construct : #pragma omp parallel

Pay close attention to shared versus private variables. In addition to a parallel construct , you will need the runtime library routines : int omp_get_num_threads() : number of threads in the team , int omp_get_thread_nume() : thread id or rank , double omp_get_wtime() : time in seconds since a fixed point in the past , omp_set_num_threads() : request a number of threads in the team

Hints : use a parallel construct , pragma omp parallel.

The challenge is to divide loop operations between threads and create an accumulator for each thread to hold partial sums that you can later combine to generate the global sum.

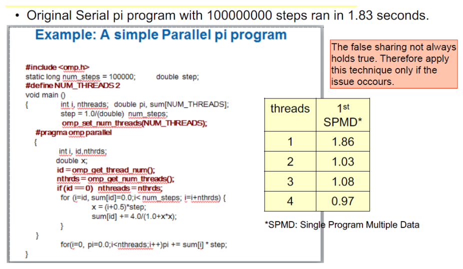
1.19.2 Wrong Code : Has data race



```
...
int main()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;

#pragma omp parallel private(x, sum)
{
#pragma omp for
   for (i=0; i<=num_steps; i++)
   {
      x=(i+0.5)*step;
      sum=sum+ 4.0/(1.0+x*x);
   }
   pi = pi + step * sum;
}
   printf("pi=%f\n",pi);
   return 0;
}
```

Multiple threads may update pi at the same time. Race condition!

1.20 Results : Use an array for local sum



- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: A simple Parallel pi program

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS];
            step = 1.0/(double)  num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
            int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)  nthreads = nthrds;
            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                     x = (i+0.5)*step;
                     sum[id] += 4.0/(1.0+x*x);
            }
   }
            for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

The false sharing not always holds true. Therefore apply this technique only if the issue occours.

threads	1st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

*SPMD: Single Program Multiple Data

1.21 Why such poor scaling?

If independent data elements happen to sit on the same cache line , each update will cause the cache lines to slosh back and forth between thread, this is called **False sharing**.

If you promote scalars to an array to support creation of an SPMD program , the array elements are contiguous in memory and hence share cache lines, result in poor scalability.

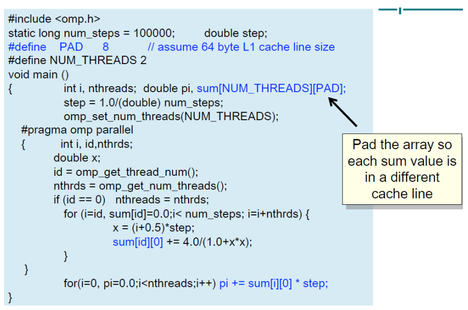
Solution : Pad arrays so elements you use are on distinct cache lines

1.22 Eliminate false sharing via padding and results

1.23 Synchronization

High level synchronization included in the common core : Critical , Barrier.

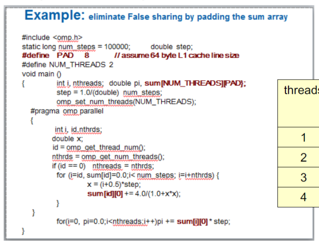
Synchronization is used to impose order constraints and to protect access to shared data.



```
#include <omp.h>
static long num_steps = 100000;        double step;
#define   PAD    8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{            int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {         int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)   nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                         x = (i+0.5)*step;
                         sum[id][0] += 4.0/(1.0+x*x);
             }
    }
             for(i=0, pi=0.0;i<nthreads;i++) pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.



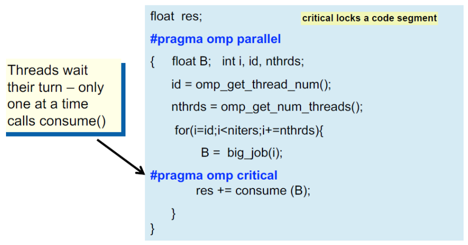
Example: eliminate False sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define   PAD    8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{            int i, nthreads;  double pi, sum[NUM_THREADS][PAD};
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
             int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                         x = (i+0.5)*step;
                         sum[id][0] += 4.0/(1.0+x*x);
             }
    }
             for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

1.24 Synchronization : critical

Mutual exclusion : only one thread at a time can enter a critical region



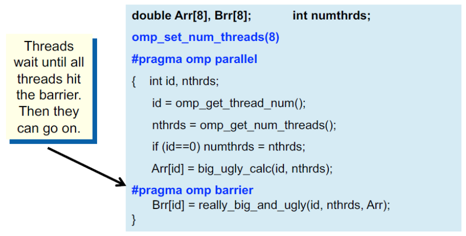
critical locks a code segment
Threads wait their turn – only one at a time calls consume()

```
float  res;
#pragma omp parallel
{     float B;   int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
       for(i=id;i<niters;i+=nthrds){
            B =  big_job(i);
#pragma omp critical
            res += consume (B);
      }
}
```

1.24.1 Synchronization : Barriers

Barrier : a point in a program all threads must reach before any threads are allowed to proceed.

It is a stand alone pragma meaning it is not associated with user code.
Barrier makes sure all the shared variables are synchronized.

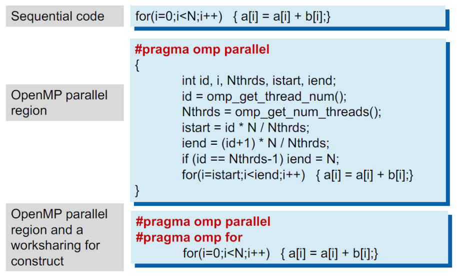


Threads wait until all threads hit the barrier. Then they can go on.

```
double Arr[8], Brr[8];          int numthrds;
omp_set_num_threads(8)
#pragma omp parallel
{    int id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
     if (id==0) numthrds = nthrds;
     Arr[id] = big_ugly_calc(id, nthrds);
#pragma omp barrier
     Brr[id] = really_big_and_ugly(id, nthrds, Arr);
}
```

1.25 Parallel loops - The worksharing-loop constructs

The worksharing-loop construct splits up loop iterations among the threads in a team.



Sequential code

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

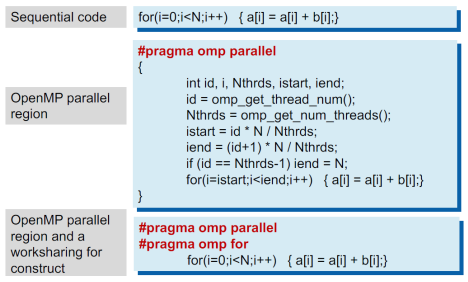
OpenMP parallel region

```
#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++)  { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

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

1.25.1 A motivating Example



Sequential code

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

OpenMP parallel region

```
#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++)  { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

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

1.25.2 The schedule clause

The schedule clause affects how loop iterations are mapped onto threads : schedule(static [,chunk]) (deal out blocks of iterations of size chunk to each thread

Schedule(dynamic[,chunk]) (each thread grabs chunk iterations off a queue until all iterations have been handled

1.25.3 Loop schedules

Default : static scheduling of iterations. Very efficient. Good if all iterations take the same amount of time. (schedule (static))

Other possibility : dynamic. Better if iterations do not take the same amount of time

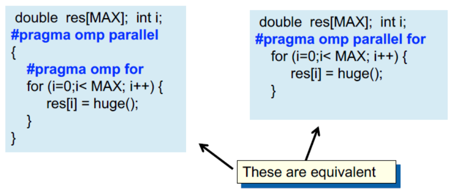
1.25.4 Chunk size

With N iterations and t threads :

- Static : each thread gets N/t iterations. explicit chunk size : schedule(static , 123)
- Dynamic : each thread gets 1 iteration at a time.
- Help from OpenMP : guided schedule uses decreasing chunk size

1.26 Combined Parallel/Worksharing Construct

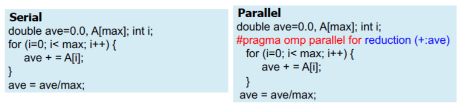
OpenMP shortcut : put the "parallel" and the worksharing directive on the same line.



```
double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
double  res[MAX];  int i;
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
```

1.27 The Reduction clause



Serial

```
double ave=0.0, A[max]; int i;
for (i=0; i< max; i++) {
    ave + = A[i];
}
ave = ave/max;
```

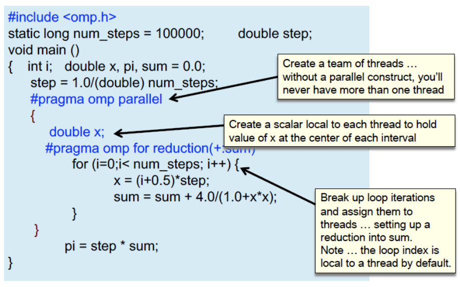
Parallel

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

Common accumulation pattern , with loop dependencies in serial code.
Syntax : Reduction(operator:list)
Reduces list of variables into one, using operator.
Reduced variables must be shared variables.
Operator : +(initial value 0) - (0) * (1) ,max (largest pos number) , min (most neg number
Each threads does its local accumulation first , then a global reduction is done at the end.

1.27.1 Exercise : Pi with loops and Reduction

The goal is to minimize the number of changes made to the serial program.

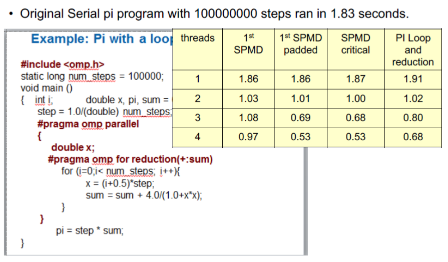


```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{   int i;   double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
        double x;
       #pragma omp for reduction(+:sum)
            for (i=0;i< num_steps; i++) {
                  x = (i+0.5)*step;
                  sum = sum + 4.0/(1.0+x*x);
            }
      }
          pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.



• Original Serial pi program with 100000000 steps ran in 1.83 seconds.

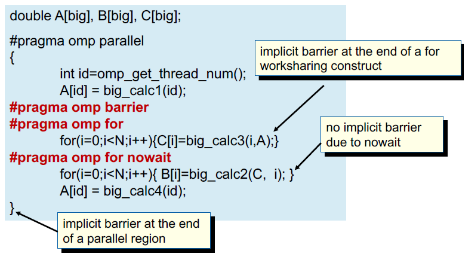
Example: Pi with a loop

```
#include <omp.h>
static long num_steps = 100000;
void main ()
{   int i;          double x, pi, sum =
     step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
         double x;
        #pragma omp for reduction(+:sum)
            for (i=0;i< num_steps; i++){
                  x = (i+0.5)*step;
                  sum = sum + 4.0/(1.0+x*x);
            }
      }
          pi = step * sum;
}
```

threads	1st SPMD	1st SPMD padded	SPMD critical	PI Loop and reduction
1	1.86	1.86	1.87	1.91
2	1.03	1.01	1.00	1.02
3	1.08	0.69	0.68	0.80
4	0.97	0.53	0.53	0.68

1.28 The nowait Clause

Barriers are really expensive. You need to understand when they are implied and how to ski[them when it's safe to do so.



```
double A[big], B[big], C[big];

#pragma omp parallel
{
        int 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);
}
```

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

1.29 Performance Tips

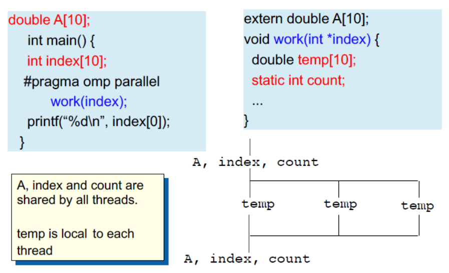
Minimize synchronization , use nowait where possible. If performance is bad , look for false sharing.

1.30 OpenMP Data Environment

Most variables are shared by default. (file scope variables , static variables, dynmically allocated variabels)

Some variable are private by default : certain loop indices , local variables within a statement block.

1.30.1 A data sharing example



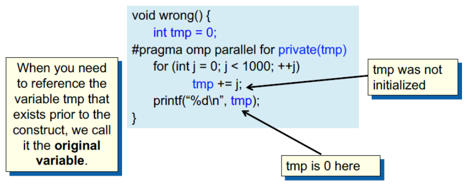
```
double A[10];
  int main() {
  int index[10];
  #pragma omp parallel
       work(index);
  printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

A, index, count
temp
temp
temp
A, index, count
A, index and count are shared by all threads.
temp is local to each thread

1.30.2 Data sharing : private Clause

private(var) creates a new local copy of var for each thread. The value of the private copies is uninitialized, the storage of the private copy will be on the each thread's stack memory and is unassociated with the original variable. The value of the original variable is unchanged after the region.

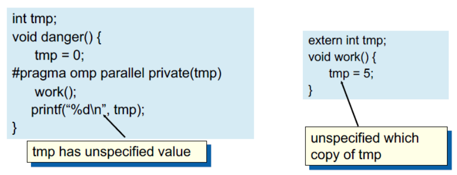


```
void wrong() {
    int tmp = 0;
#pragma omp parallel for private(tmp)
    for (int j = 0; j < 1000; ++j)
          tmp += j;
    printf("%d\n", tmp);
}
```

When you need to reference the variable tmp that exists prior to the construct, we call it the **original variable**.
tmp was not initialized
tmp is 0 here

1.30.3 Data sharing : private clause , when is the original variable valid?

The original variable's value is unspecified if it is referenced outside of the construct. Implementations may reference the original variable or a copy .

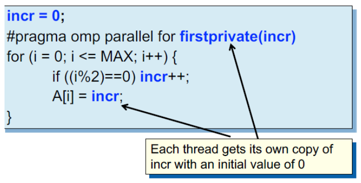


```
int tmp;
void danger() {
      tmp = 0;
#pragma omp parallel private(tmp)
      work();
     printf("%d\n", tmp);
}
```

```
extern int tmp;
void work() {
     tmp = 5;
}
```

1.30.4 The Firstprivate Clause

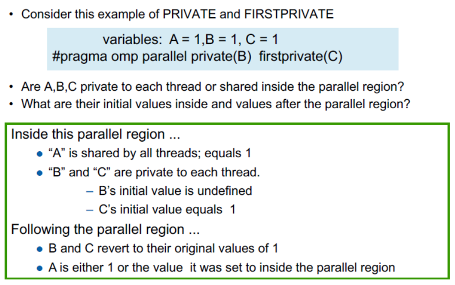
Initializes the variables in the list with the value of the shared variable when they first enter the construct.



```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
```

Each thread gets its own copy of incr with an initial value of 0

1.30.5 A data environment test



- Consider this example of PRIVATE and FIRSTPRIVATE

```
        variables:  A = 1,B = 1, C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

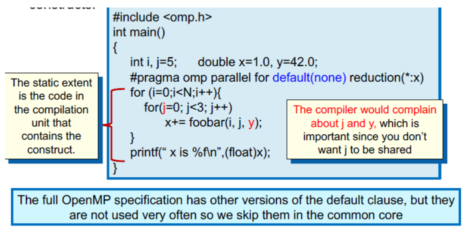
- Are A,B,C private to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...
- "A" is shared by all threads; equals 1
- "B" and "C" are private to each thread.
  - B's initial value is undefined
  - C's initial value equals 1

Following the parallel region ...
- B and C revert to their original values of 1
- A is either 1 or the value it was set to inside the parallel region

1.30.6 Data Sharing : default Clause

default(none) : forces you to define the storage attributes for variables that appear inside the static extent of the construct....if you fail the compiler will complain. You can put the default clause on parallel and parallel + workshare constructs.



```
#include <omp.h>
int main()
{
    int i, j=5;     double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0;i<N;i++){
        for(j=0; j<3; j++)
            x+= foobar(i, j, y);
    }
    printf(" x is %f\n",(float)x);
}
```

The static extent is the code in the compilation unit that contains the construct.

The compiler would complain about j and y, which is important since you don't want j to be shared

The full OpenMP specification has other versions of the default clause, but they are not used very often so we skip them in the common core

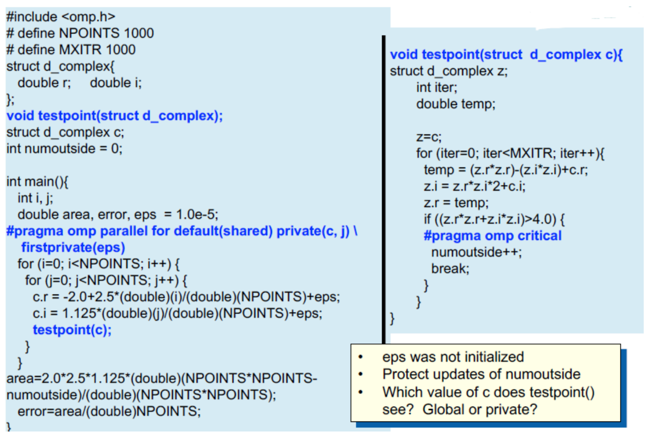
1.30.7 Performance and Correctness Tips

There is one version of shared data.

Private data is stored locally, so use of private variables can increase efficiency (avoids false sharing, may make it easier to parallelize loops). It is an error if multiple threads update the same variable at the same time. It is a good idea to use ”default none” while testing code.

1.30.8 Exercise : The Mandelbrot Area program

The supplied program computes the area of a Mandelbrot set. Find and fix the errors.



```
#include <omp.h>
# define NPOINTS 1000
# define MXITR 1000
struct d_complex{
   double r;    double i;
};
void testpoint(struct d_complex);
struct d_complex c;
int numoutside = 0;

int main(){
   int i, j;
   double area, error, eps  = 1.0e-5;
#pragma omp parallel for default(shared) private(c, j) \
    firstprivate(eps)
   for (i=0; i<NPOINTS; i++) {
     for (j=0; j<NPOINTS; j++) {
       c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
       c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
       testpoint(c);
     }
   }
area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
numoutside)/(double)(NPOINTS*NPOINTS);
   error=area/(double)NPOINTS;
}
```

```
void testpoint(struct  d_complex c){
struct d_complex z;
       int iter;
       double temp;

       z=c;
       for (iter=0; iter<MXITR; iter++){
         temp = (z.r*z.r)-(z.i*z.i)+c.r;
         z.i = z.r*z.i*2+c.i;
         z.r = temp;
         if ((z.r*z.r+z.i*z.i)>4.0) {
         #pragma omp critical
           numoutside++;
           break;
         }
       }
}
```

- eps was not initialized
- Protect updates of numoutside
- Which value of c does testpoint() see? Global or private?

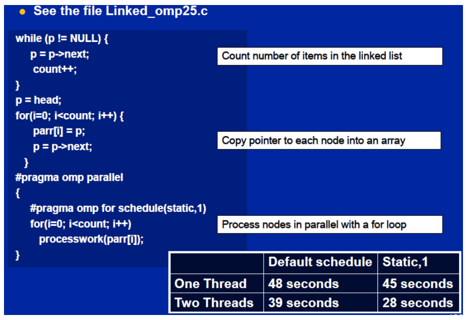
1.31 Irregular parallelism and tasks - What are OpenMP tasks?

Task construct : a structered block of code + a data environment.

Inside a parallel region , a thread encountering a task construct will package up the code block and its data for execution.

The task is executed immediately, or deferred for later execution.

1.31.1 Linked lists without tasks



- See the file Linked_omp25.c

```
while (p != NULL) {
    p = p->next;
     count++;
}
p = head;
for(i=0; i<count; i++) {
     parr[i] = p;
     p = p->next;
  }
#pragma omp parallel
{
    #pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
      processwork(parr[i]);
}
```

Count number of items in the linked list

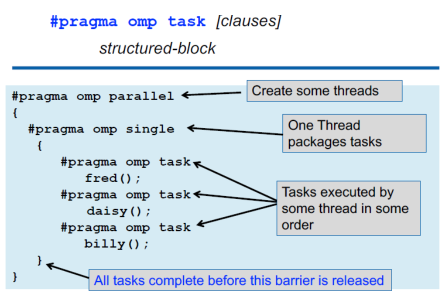
Copy pointer to each node into an array

Process nodes in parallel with a for loop

	Default schedule	Static,1
One Thread	48 seconds	45 seconds
Two Threads	39 seconds	28 seconds

We are able to parallelize the linked list traversal , but it was ugly and required multiple passes over the data. To move beyond its roots it the array based world of scientific computing we needed to support more general data structures and loops beyond basic for loops. We added task in OpenMP 3.0

1.31.2 Task directive



```
#pragma omp task [clauses]
        structured-block
```

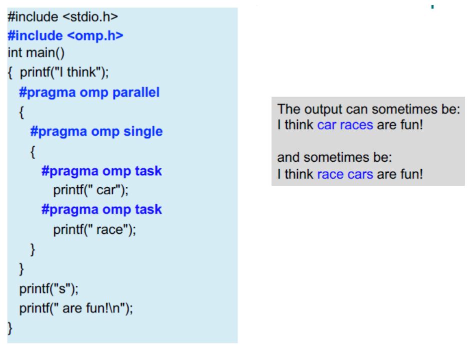
```
#pragma omp parallel
{
  #pragma omp single
  {
      #pragma omp task
         fred();
      #pragma omp task
         daisy();
      #pragma omp task
         billy();
  }
}
```

Create some threads
One Thread packages tasks
Tasks executed by some thread in some order
All tasks complete before this barrier is released

1.31.3 Exercise : Simple Tasks

Write a program using tasks that will randomly generate one of two strings : ”I think race cars are fun” , ”I think car races are fun”.

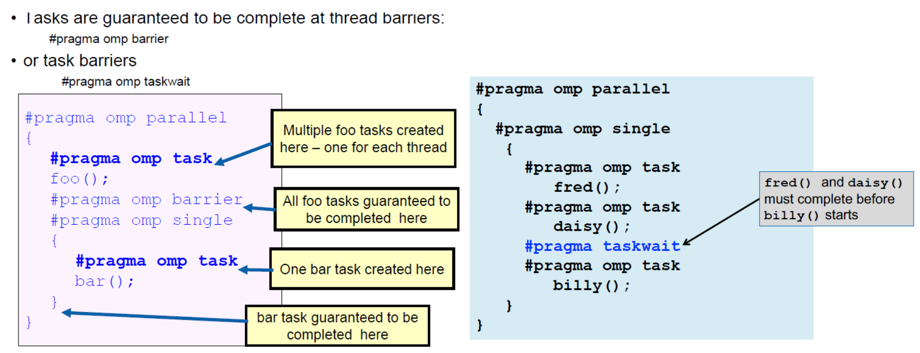
This is called a **race condition**. It occurs when the result of a program depends on how the OS schedules the threads.



```
#include <stdio.h>
#include <omp.h>
int main()
{  printf("I think");
   #pragma omp parallel
   {
      #pragma omp single
      {
         #pragma omp task
            printf(" car");
         #pragma omp task
            printf(" race");
      }
   }
   printf("s");
   printf(" are fun!\n");
}
```

```
The output can sometimes be:
I think car races are fun!

and sometimes be:
I think race cars are fun!
```

1.32 When are tasks guaranteed to complete + example



- Tasks are guaranteed to be complete at thread barriers:
  #pragma omp barrier
- or task barriers
  #pragma omp taskwait

```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

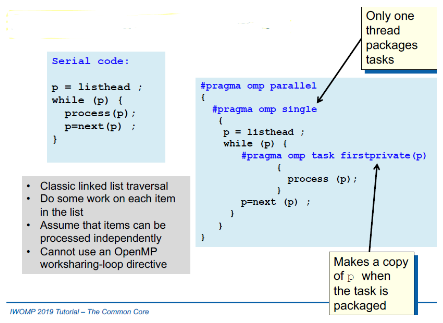
One bar task created here

bar task guaranteed to be completed here

```
#pragma omp parallel
{
  #pragma omp single
   {
      #pragma omp task
         fred();
      #pragma omp task
         daisy();
      #pragma taskwait
      #pragma omp task
         billy();
   }
}
```

fred() and daisy() must complete before billy() starts

1.32.1 Parallel linked list traversal



```
Serial code:

p = listhead ;
while (p) {
  process(p);
  p=next(p) ;
}
```

- Classic linked list traversal
- Do some work on each item in the list
- Assume that items can be processed independently
- Cannot use an OpenMP worksharing-loop directive

```
#pragma omp parallel
{
  #pragma omp single
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
             {
               process (p);
             }
       p=next (p) ;
     }
   }
}
```

Only one thread packages tasks

Makes a copy of p when the task is packaged

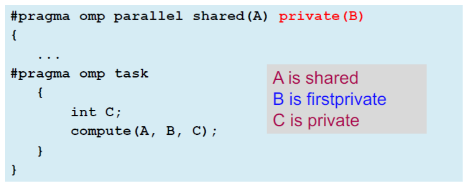
IWOMP 2019 Tutorial – The Common Core

1.32.2 OpenMP tasks : Data scoping defaults

The behavior you want for tasks is usually firstprivate , because the task may not be executed until later.

Variables that are private when the task construct is encountered are firstprivate by default.

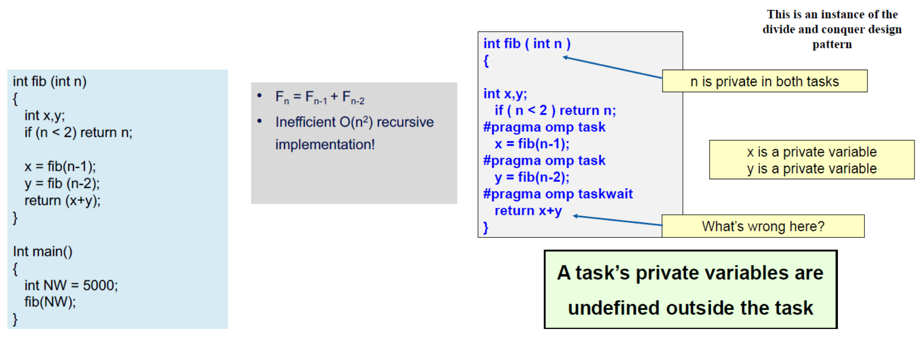
Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default.



```
#pragma omp parallel shared(A) private(B)
{
   ...
#pragma omp task
   {
       int C;
       compute(A, B, C);
   }
}
```

A is shared
B is firstprivate
C is private

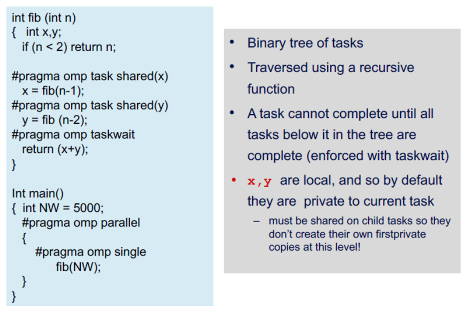
1.32.3 Example : Fibonacci Numbers


This is an instance of the divide and conquer design pattern
n is private in both tasks
x is a private variable
y is a private variable
What's wrong here?
A task's private variables are undefined outside the task

```
int fib (int n)
{
   int x,y;
   if (n < 2) return n;

   x = fib(n-1);
   y = fib (n-2);
   return (x+y);
}

Int main()
{
   int NW = 5000;
   fib(NW);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Inefficient $O(n^2)$ recursive implementation!

```
int fib ( int n )
{

int x,y;
  if ( n < 2 ) return n;
#pragma omp task
  x = fib(n-1);
#pragma omp task
  y = fib(n-2);
#pragma omp taskwait
  return x+y
}
```



```
int fib (int n)
{   int x,y;
   if (n < 2) return n;

#pragma omp task shared(x)
   x = fib(n-1);
#pragma omp task shared(y)
   y = fib (n-2);
#pragma omp taskwait
   return (x+y);
}

Int main()
{  int NW = 5000;
   #pragma omp parallel
   {
       #pragma omp single
           fib(NW);
   }
}
```

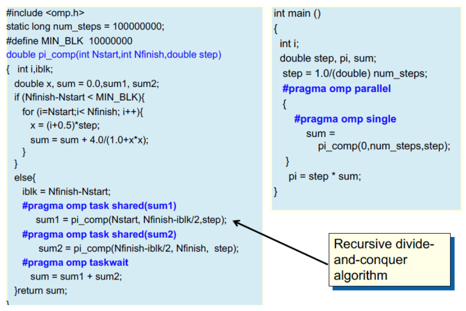
- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- `x,y` are local, and so by default they are private to current task
  - must be shared on child tasks so they don't create their own firstprivate copies at this level!

1.32.4 Divide and Conquer

Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly

1.32.5 Exercise: Pi with Tasks

Consider the program that uses a recursive algorithm in integrate the function in the pi program.



```
#include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
  double x, sum = 0.0,sum1, sum2;
  if (Nfinish-Nstart < MIN_BLK){
    for (i=Nstart;i< Nfinish; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
    }
  }
  else{
    iblk = Nfinish-Nstart;
    #pragma omp task shared(sum1)
        sum1 = pi_comp(Nstart, Nfinish-iblk/2,step);
    #pragma omp task shared(sum2)
         sum2 = pi_comp(Nfinish-iblk/2, Nfinish,  step);
    #pragma omp taskwait
      sum = sum1 + sum2;
  }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel
   {
      #pragma omp single
         sum =
            pi_comp(0,num_steps,step);
   }
    pi = step * sum;
}
```

1.32.6 Tips

Don't use tasks for thing already well supported by OpenMP (do/for). The overhead of using tasks is greater.

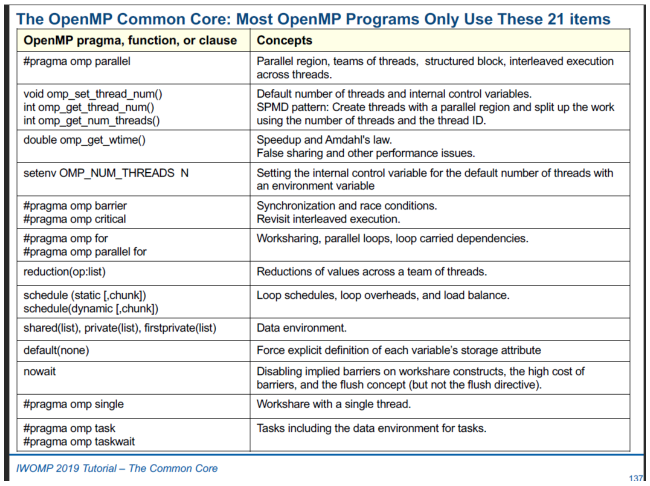
1.33 OpenMP and Performance

Do not parallelize what does not matter.

Do not share data unless you have to.

With NUMA, the data access time varies. The time depends on where the data is and there are techniques to avoid this.

1.33.1 Recap - The OpenMP Common Core



The OpenMP Common Core: Most OpenMP Programs Only Use These 21 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(none)	Force explicit definition of each variable's storage attribute
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).
#pragma omp single	Workshare with a single thread.
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

IWOMP 2019 Tutorial – The Common Core

137

Data declared outside a parallel region is shared.

Data declared in the parallel region is private.

The loop variable is automatically private.

Private data disappears after the parallel region, what if you want data to persist? Directive **threadprivate**.

Statically allocated arrays can be made private. Dynamically allocated ones can not : the pointer becomes private.

The loop and sections directives do not specify an ordering, sometimes you want to force an ordering.

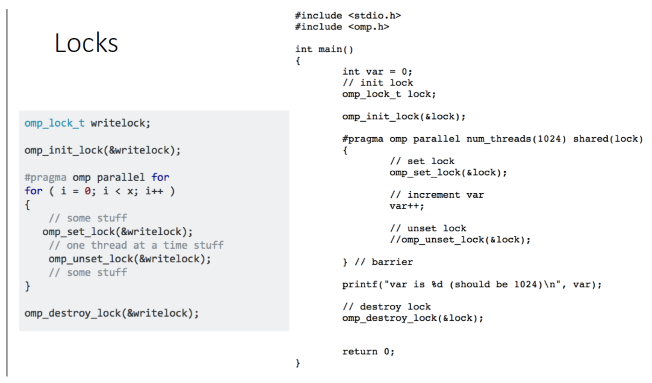
Barriers : global synchronization

Critical sections : only one process can execute a statement, this prevents race condition.

Locks : protect data items from being accessed.

Critical section are not cheap! Use only if minor amount of work. Do not use if a reduction suffices. Name your critical sections.

Locks protect a single data item



Locks

```c
omp_lock_t writelock;

omp_init_lock(&writelock);

#pragma omp parallel for
for ( i = 0; i < x; i++ )
{
    // some stuff
   omp_set_lock(&writelock);
    // one thread at a time stuff
    omp_unset_lock(&writelock);
    // some stuff
}

omp_destroy_lock(&writelock);
```

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

int main()
{
        int var = 0;
        // init lock
        omp_lock_t lock;

        omp_init_lock(&lock);

        #pragma omp parallel num_threads(1024) shared(lock)
        {
                // set lock
                omp_set_lock(&lock);

                // increment var
                var++;

                // unset lock
                //omp_unset_lock(&lock);

        } // barrier

        printf("var is %d (should be 1024)\n", var);

        // destroy lock
        omp_destroy_lock(&lock);


        return 0;
}
```