Shared Memory Parallel Programming with OpenMP

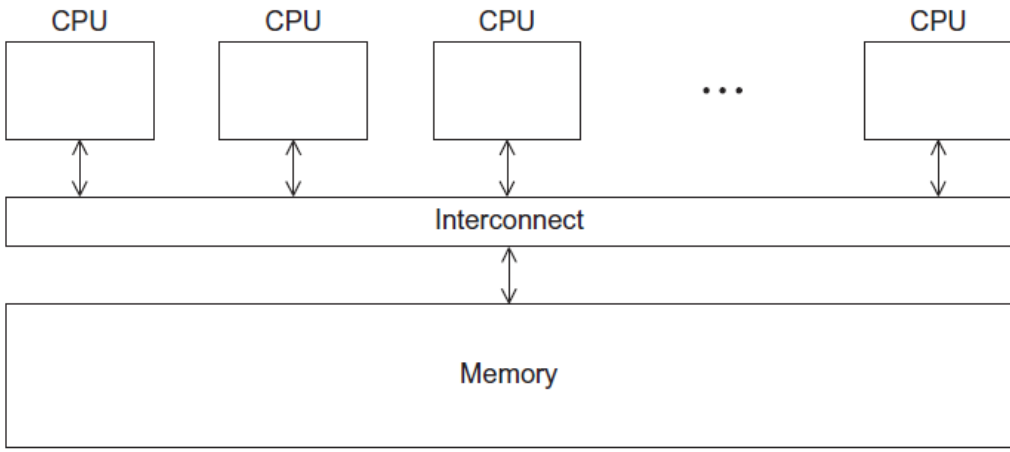
CS140, T. Yang

Shared Memory Programming with Threads

Several Thread Libraries/systems

- **OpenMP standard for application level programming**
  - Support for scientific programming on shared memory
- **Pthreads is the POSIX Standard:** Relatively low level
- **Java threads:** Built on top of POSIX threads
- **Python** packages often use multiple CPU cores with OpenMP and GPU threads when GPU is attached

Targeted platform: shared memory architecture

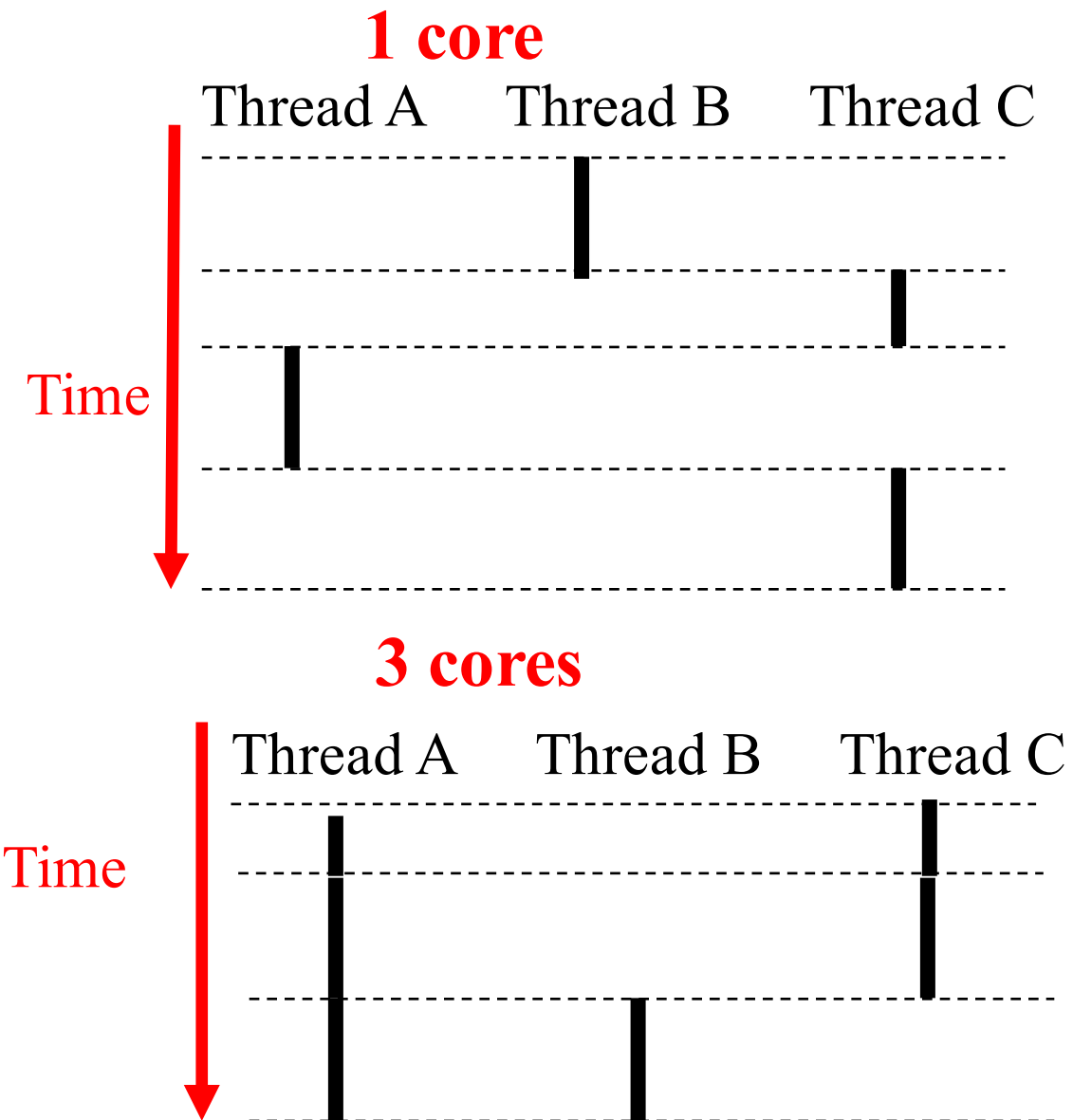


Thread Execution on Single/Multiple Cores

- **Two threads run concurrently if their logical flows overlap in time**
- **Examples:**
  - Concurrent:

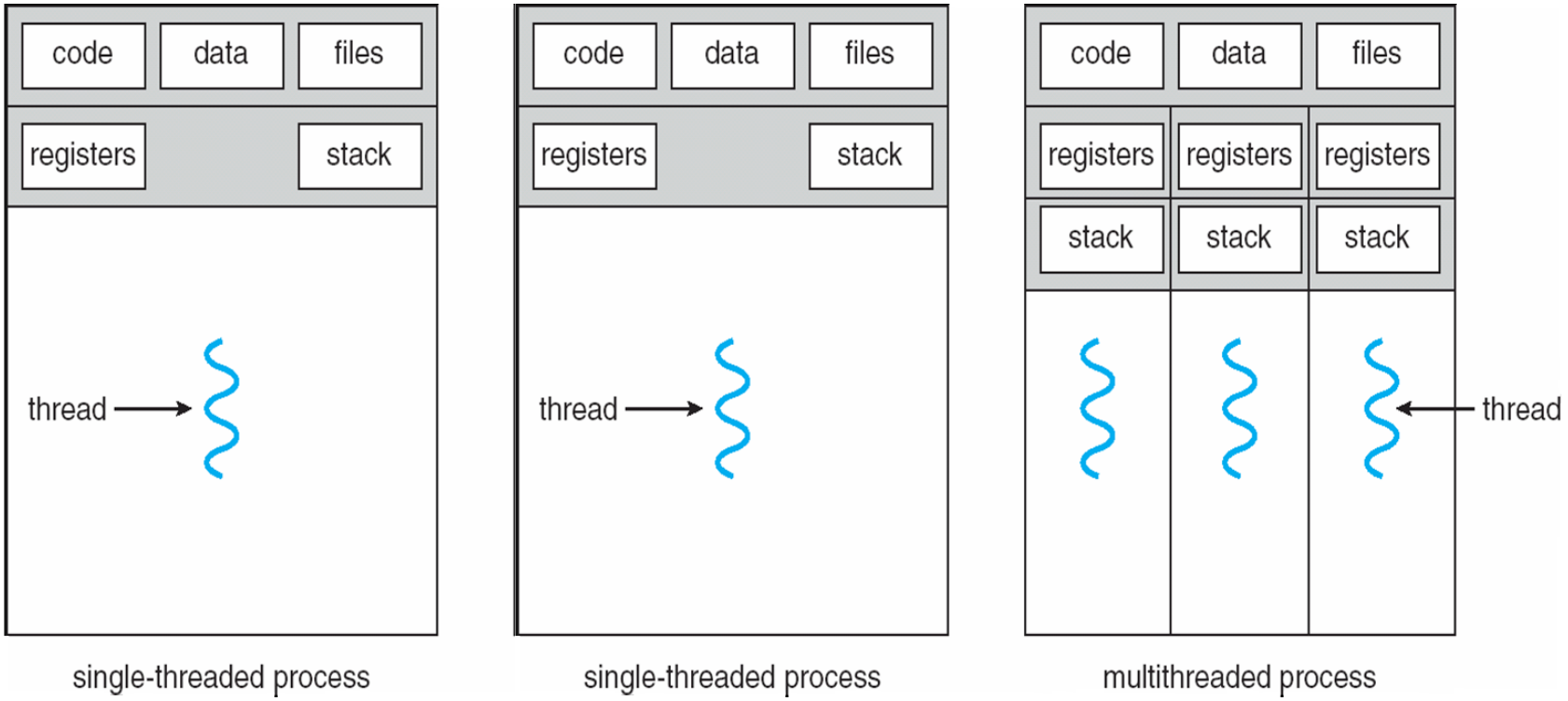
    A & B, A&C
  - Sequential:

    B & C



Difference between Threads and Processes

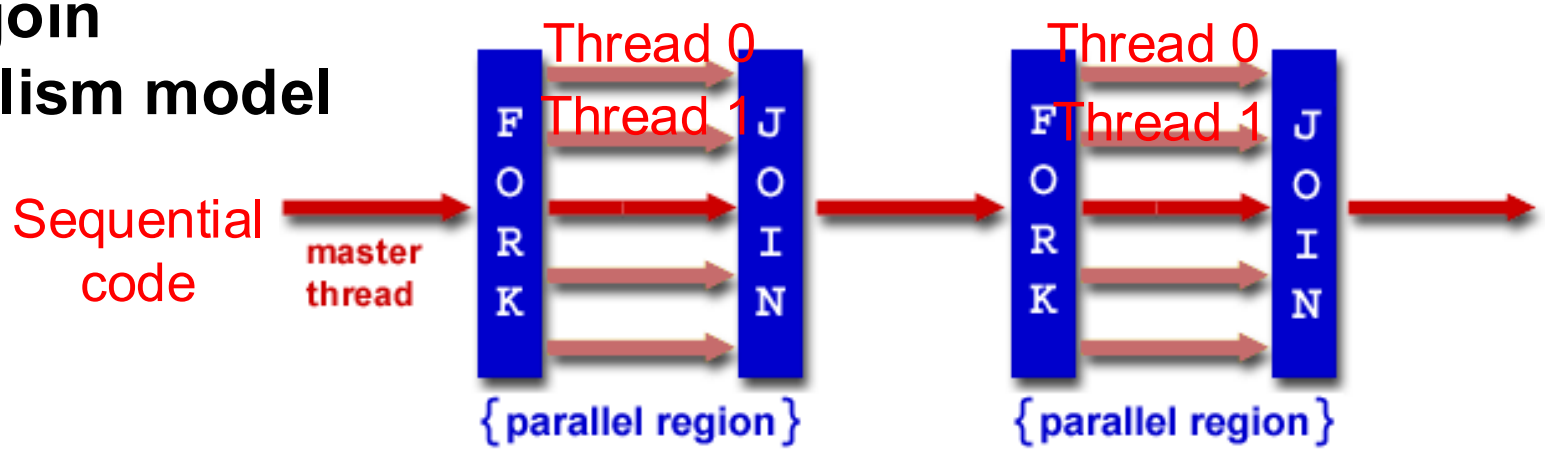
Processes: Nothing is shared between two processes
Separate address spaces
Inter-process communication: message sending/receiving



Thread: Shared memory access for code/global variables/heap space
Inter-thread communication: through shared global variables
Separate thread control flow by using separate stack/registers

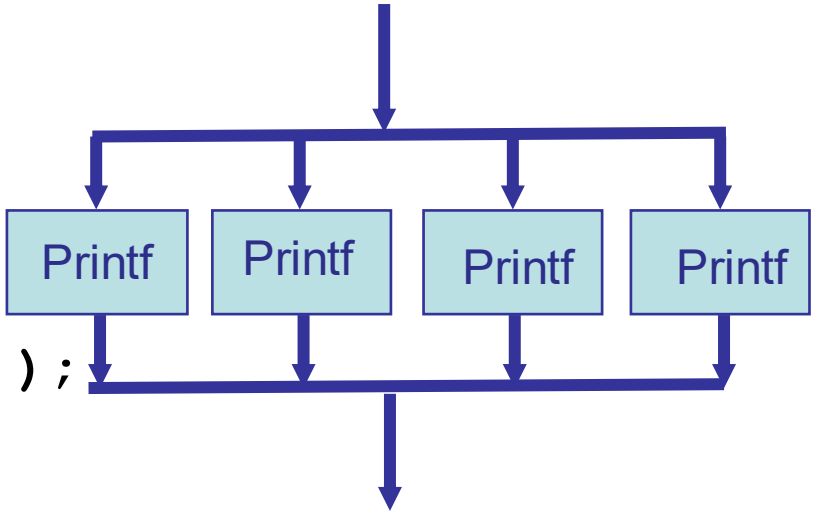
A Programmer's View of OpenMP

- **What is OpenMP?**
  - "Standard" API for multi-threaded shared-memory programs
  - openmp.org – Talks, examples, forums, etc.
- **OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax**
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than explicitly define concurrently-executing threads.
- **Fork - join parallelism model**



Hello World Example with OpenMP

```
int main() {
  omp_set_num_threads(4);
  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }
  return 0;
}
```



- **Add special preprocessor instructions to C**
  - All OpenMP directives begin with #pragma
  - Compilers that don't support the pragmas ignore them

- **How to compile and run in CSIL:**

  gcc –O –fopenmp hello.c –o hello

  ./hello

OpenMP parallel region construct

- **C/C++ syntax in SPMD style**

  #pragma omp parallel [clause [clause] ...] new-line

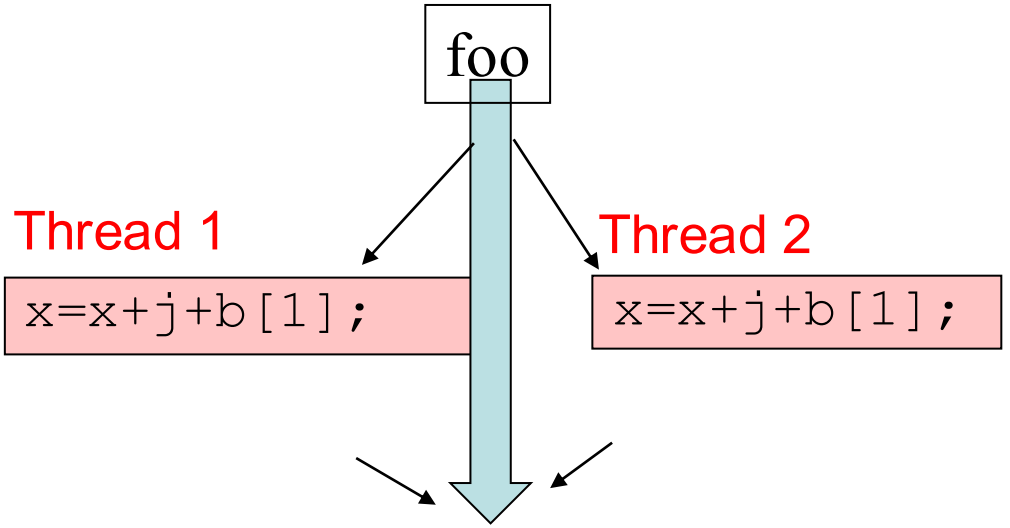
  structured-block

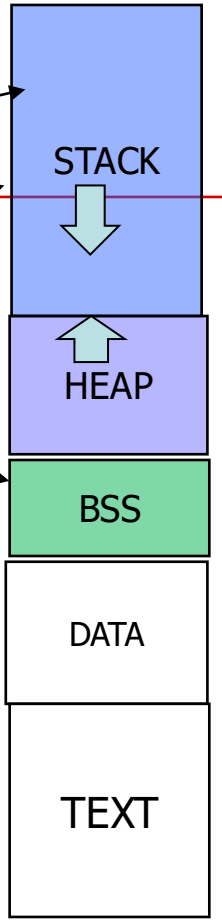
- **Clause** is the text that modifies a directive, and can include **private (list)** or **shared (list)**
  - Private data, visible to a single thread (often stack-allocated)
  - Shared data within a scope, visible to all threads. Default setting.

    #pragma omp parallel private (x,y)

Data Sharing: OpenMP

```
int b[1024]; //shared, global

void* foo(int bar) {

  int x;  //shared among threads within foo

  #pragma omp parallel

   {  int j=1;

      x=x+j+b[1];

    }

}
```

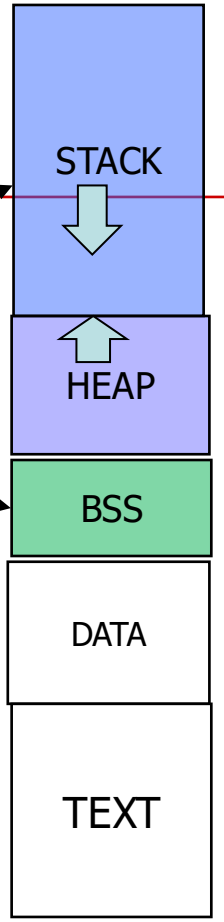




1. Treat each thread as a dynamically-created function call
2. Figure out data sharing among functions

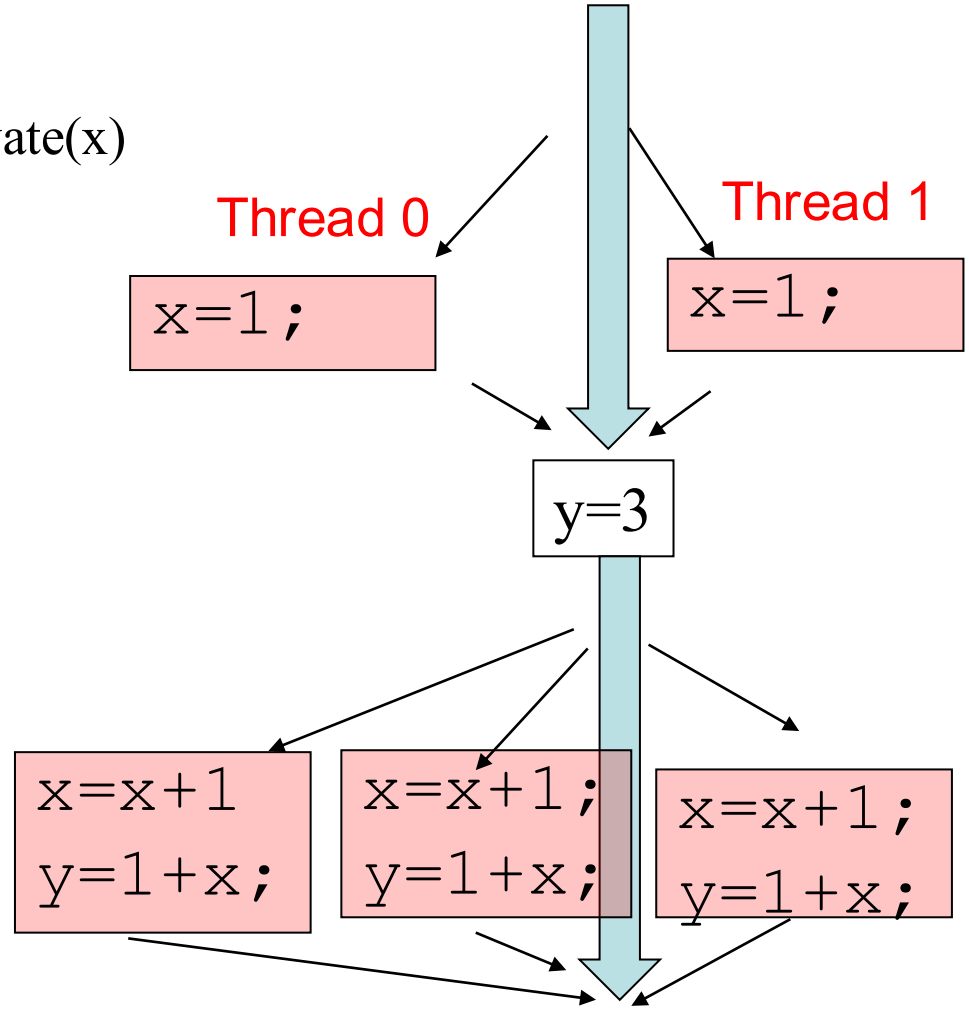
Parallel Pragma

```
int y; //globally shared
void foo( int x) {
#pragma omp parallel num_threads(2) private(x)
{    //same as declaring "int x;" local x here
      x=1;
}
y=3;
 #pragma omp parallel num_threads(3)
{    //x  is shared among all threads
    x=x+1;
    y=1+x;
}
```



Parallel Pragma

```
int y; //globally shared
void foo( int x) {
#pragma omp parallel num_threads(2) private(x)
{    //same as declaring "int x;" local x here
     x=1;
}
y=3;
 #pragma omp parallel num_threads(3)
{  //x  is shared among all threads
   x=x+1;
   y=1+x;
}
```



Loop Parallelization with Explicit Computation-to-Thread Mapping

```
for (int i=0; i<8; i++)
   x[i]=0; //Run on multiple threads with cyclic mapping
```

```
#pragma omp parallel
        {                                          // Assume number of threads=4
       int numt=omp_get_num_thread();
       int id = omp_get_thread_num(); //id=0, 1, 2, or 3
            for (int i=id; i<8; i +=numt)
                    x[i]=0;
            }
```

Thread 0	Thread 1	Thread 2	Thread 3

`Id=0;`
`x[0]=0;`
`X[4]=0;`
	`Id=1;`
	`x[1]=0;`
	`X[5]=0;`
		`Id=2;`
		`x[2]=0;`
		`X[6]=0;`
			`Id=3;`
			`x[3]=0;`
			`X[7]=0;`

Use “pragma parallel for “

```
for (int i=0; i<8; i++)
    x[i]=0;
```

```
#pragma omp parallel for schedule(static,1)
{
        for (i=0; i<8; i++)
                x[i]=0;
}
```

OpenMP parallelizes iterations of **the first loop below “parallel for”**
OpenMP make iteration variable *i* private automatically for each thread.

```
Id=0;
x[0]=0;
X[4]=0;
```

```
Id=1;
x[1]=0;
X[5]=0;
```

```
Id=2;
x[2]=0;
X[6]=0;
```

```
Id=3;
x[3]=0;
X[7]=0;
```

Parallel region vs. parallel for

```
#pragma omp parallel
{
   #pragma omp for
   for(i=0;i<max;i++) { … }
}
```

This is the only directive in the parallel section

can be shortened to:

```
#pragma omp parallel for
  for(i=0;i<max;i++) { … }
```

This index variable is private

Parallel region vs parallel for: More flexibility

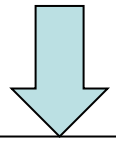
```
#pragma omp parallel
{
  Preprocessing…
    #pragma omp for
    for(i=0;i<max;i++) { … }
  Postprocessing…
}
```

CANNOT be shortened to:

```
#pragma omp parallel for
  for( i=0;i<max;i++) { … }
```

Parallel for with collapse directive

```
#pragma omp parallel collapse(2)
    for(i=0;i< m;i++)
        for(j=0;j< n;j++)
            a[i][j]=i+j;
```



Same as

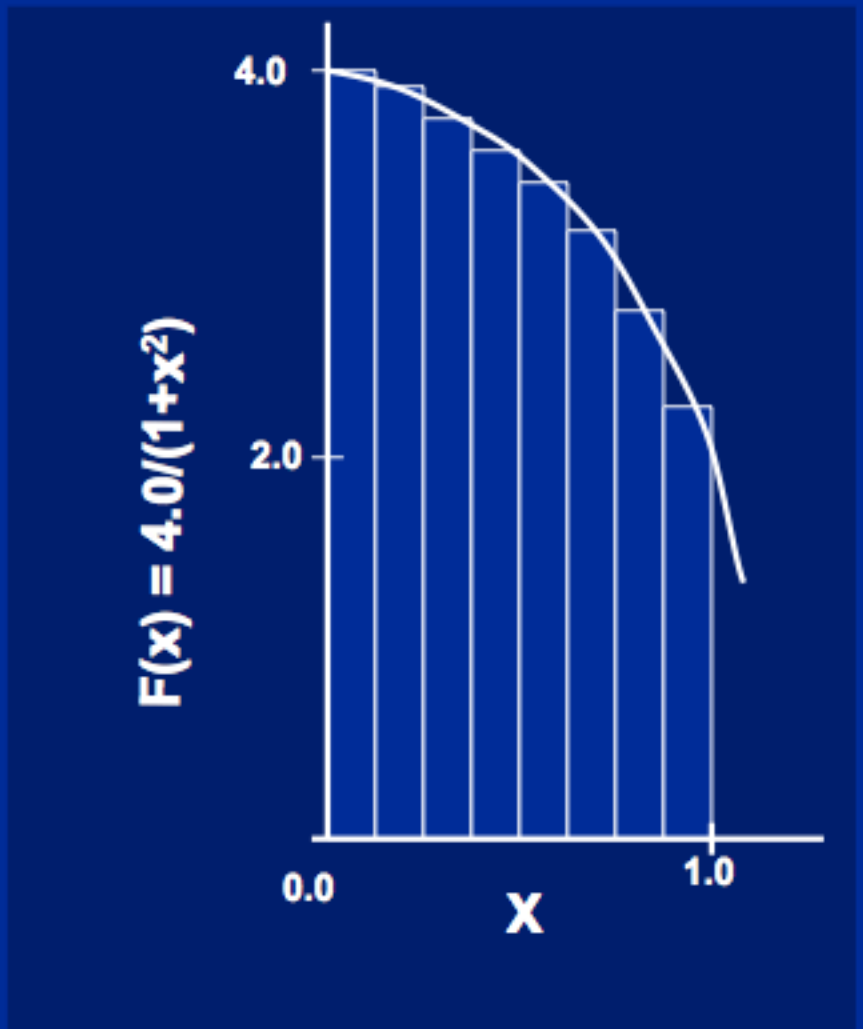
```
#pragma omp parallel for
  for( k=0;k< m*n; k++) {
        int i = k /n;
        int j = k%n;
        a[i][j]=i+j;
}
```

Apply #pragma omp parallel for collapse(2),

- the compiler mathematically "flattens" the two loops to exploit more parallelism
- It calculates a single total iteration count (N×M) and then divides that total across the available threads

Example: Calculating π

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.

Sequential Calculation of π in C

```
#include <stdio.h>          /* Serial Code */
static long num_steps = 100000;
double step;
void main () {
     int i;
     double x, pi, sum = 0.0;
     step = 1.0/(double)num_steps;
     for (i = 1; i <= num_steps; i++) {
       x = (i - 0.5) * step;
       sum = sum + 4.0 / (1.0 + x*x);
     }
     pi = sum / num_steps;
     printf ("pi = %6.12f\n", pi);
}
```

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

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

Parallel OpenMP Version (1)

```
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

void main () {
  int i;       double  x, pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  #pragma omp parallel private ( i, x )
  {
    int id = omp_get_thread_num();
    for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS)
    {
      x = (i+0.5)*step;
      sum[id] += 4.0/(1.0+x*x);
    }
  }
  for(i=1; i<NUM_THREADS; i++)
    sum[0] += sum[i];  pi = sum[0] / num_steps
  printf ("pi = %6.12f\n", pi);
}
```

Consider "Parallel for"
Can we also do global sum in parallel?

Reduction clause

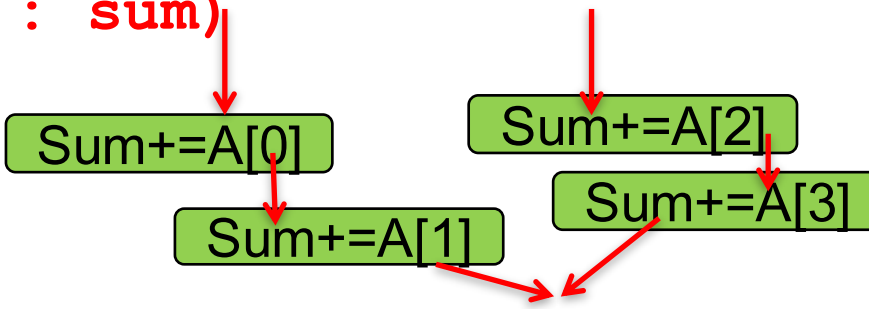
A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```

→ +, *, -, &, |, ˆ, &&, ||

- ***Variable list***: specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region:

```
#pragma omp for reduction(+ : sum)
for (i = 0; i < MAX ; i++)
      sum += A[i];
```



- **Initial value for reduction**

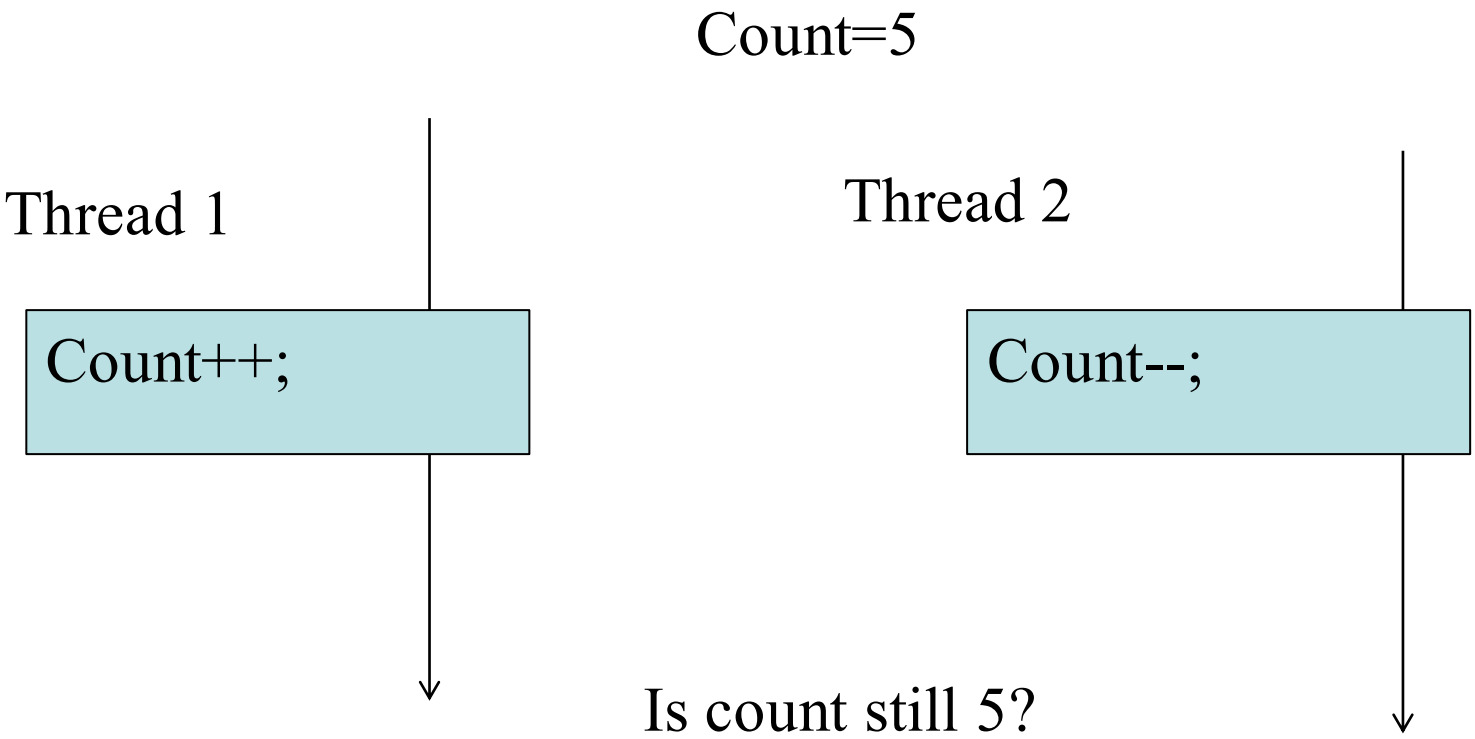
-	0	bitwise |	0	logical |	0
*	1	bitwise ^	0		

Calculating π Version 2 with parallel for, reduction

```
#include <omp.h>
#include <stdio.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 for private(x) reduction(+:sum)
     for (i=1; i<= num_steps; i++){
          x = (i-0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
     }
     pi = sum / num_steps;
  printf ("pi = %6.8f\n", pi);
}
```

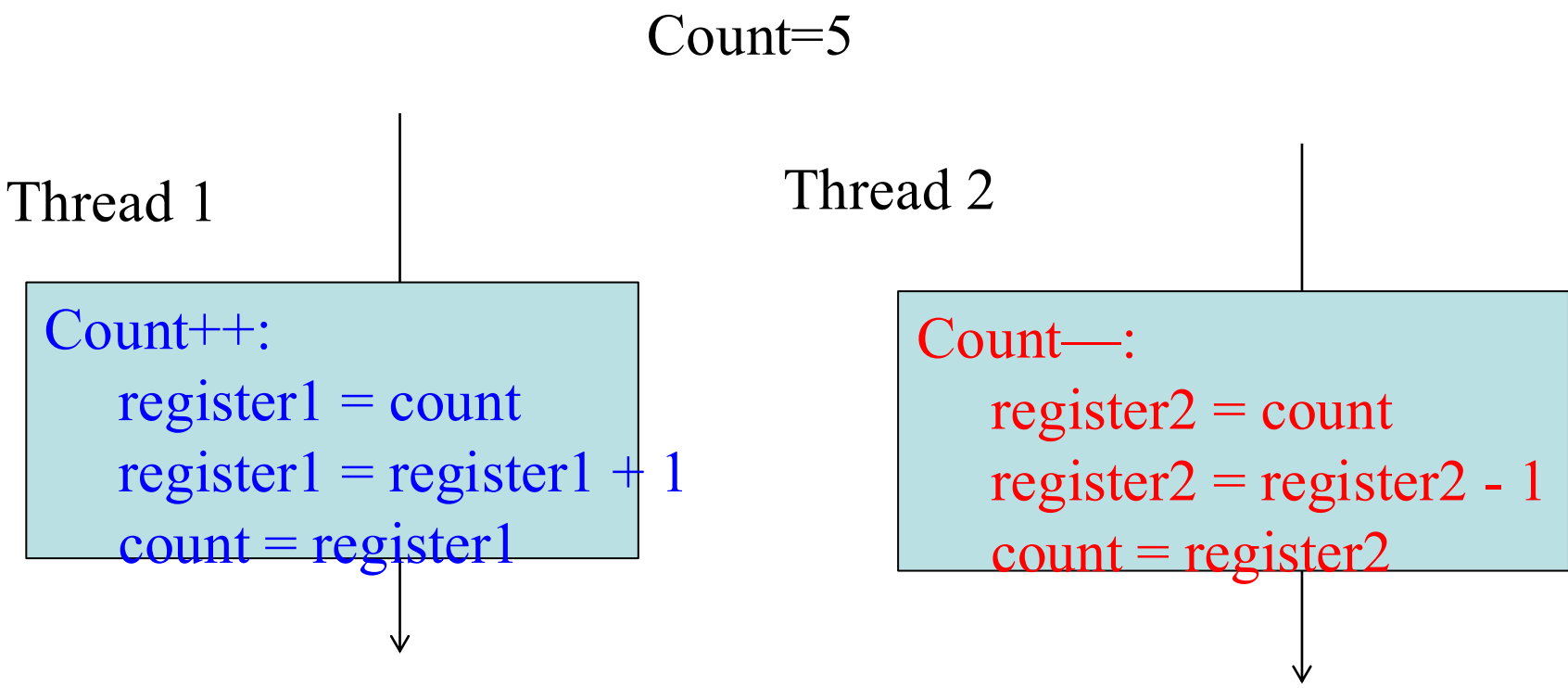
Race Conditions and Synchronization

Race Conditions: Example



Race Conditions: two or more threads are reading and writing on shared data and the final result depends on who runs precisely & when

Review compiled code



Count can be 4,5,6

Thread Synchronization in OMP

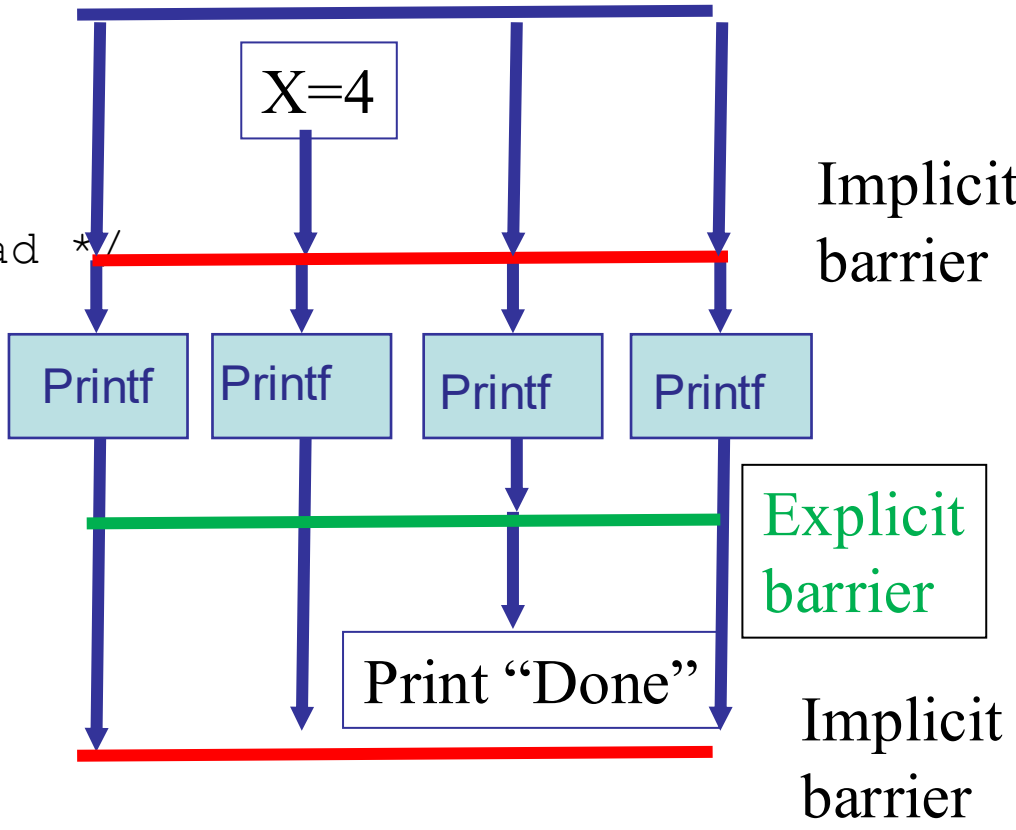
- OMP barrier: any thread must stop at this point and cannot proceed until all other threads reach this barrier.
- Implicit barrier
  - End of parallel region
  - End of parallel-for (removed with nowait clause)
  - End of OMP single
- OMP single specifies that a section of code should be executed **by single thread** (not necessarily the master thread)
- OMP atomic specifies that a statement below with a certain operator can be executed only in an atomic manner **by one thread at a time**
- OMP critical specifies that code is executed **by one thread at a time**
- Explicit lock functions

```
omp_set_lock( lock l );
   sum=sum+x;
omp_unset_lock( lock l );
```

```
#pragma omp critical
{
   /* Critical code here */
   sum=sum+x;
}
```

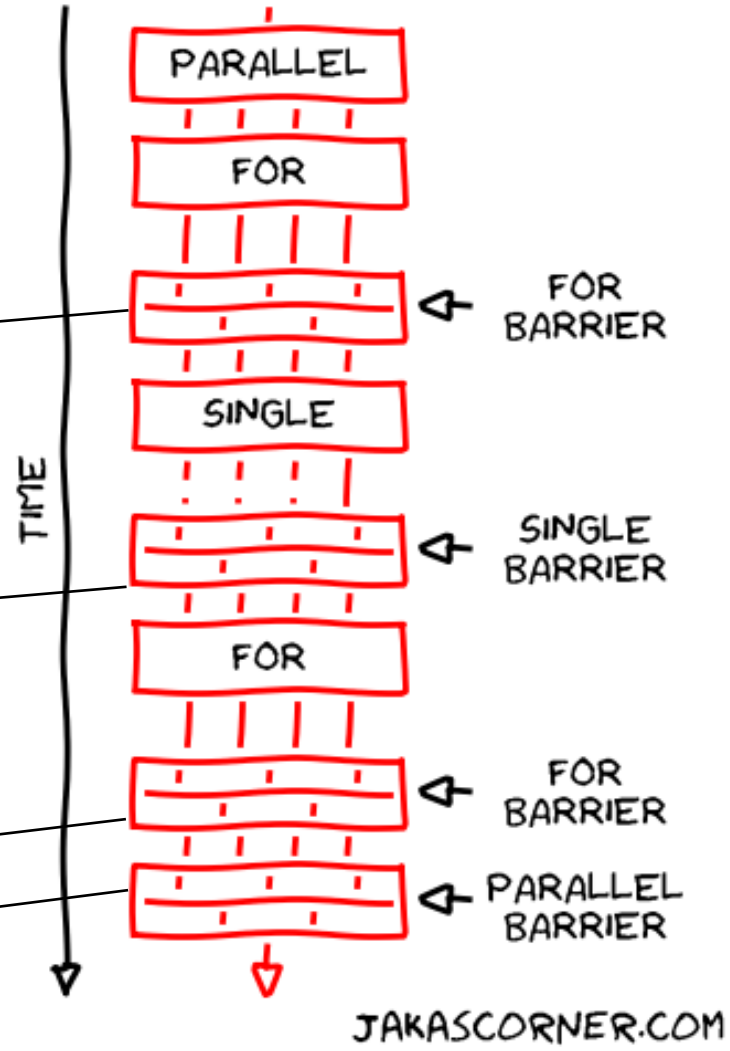
Example: Serial Tasks Inside Parallel Regions with Explicit Barrier

```
int x;
#pragma omp parallel
{
  /*Initialize with one thread */
 #pragma omp single
 {
       x=4;
 }
 printf("Hello %d\n",x);
 #pragma barrier
 #pragma omp single
 {
    printf("Done\n"); }

 }
}
```



Where are OMP Implicit Barriers?

```
#pragma omp parallel shared(s1, s2)
{
  #pragma omp for reduction(+: s1)
  for (int e = 0; e < 250; e++) {
     s1 += fetch (e,1);
  }
  #pragma omp single
  {
    printf("%d\n" s1);
  }
   #pragma omp for reduction(+: s2)
  for (int e = 0; e < 250; e++) {
     s2 += fetch (e,2);
   }
}
```



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

OMP Parallel-for loop with schedule clause

#pragma parallel for schedule (type, chunksize)

- **Type can be:**
  - static: the iterations are assigned to the threads at compiler time before the code is executed.
    - Each thread receives `chunksize` iterations, rounding as necessary to account for all iterations
    - Default **`chunksize`** is
      - **`ceil( #iterations /# threads )`**
  - dynamic or guided: the iterations are assigned to the threads based on dynamic workload during run-time
  - runtime: the schedule for mapping iterations is determined at run-time

Examples of the Static Schedule Determined at Compile-Time

12 iterations, 0, 1, . . . , 11, and three threads

`schedule(static,1)`

Thread 0 : 0, 3, 6, 9

Thread 1 : 1, 4, 7, 10

Thread 2 : 2, 5, 8, 11

`schedule(static,2)`

Thread 0 : 0, 1, 6, 7

Thread 1 : 2, 3, 8, 9

Thread 2 : 4, 5, 10, 11

`schedule(static,4)`

Thread 0 : 0, 1, 2, 3

Thread 1 : 4, 5, 6, 7

Thread 2 : 8, 9, 10, 11

Which one is better?

Assigning loop scheduling type at runtime

void omp_set_schedule(omp_sched_t kind, int chunk_size)

```
Defined in omp.h
typedef enum omp_sched_t {
        omp_sched_static = 1,
        omp_sched_dynamic = 2,
        omp_sched_guided = 3,
        omp_sched_auto = 4 }
omp_sched_t;
```

schedule(runtime) means scheduling type is not known at compile-time

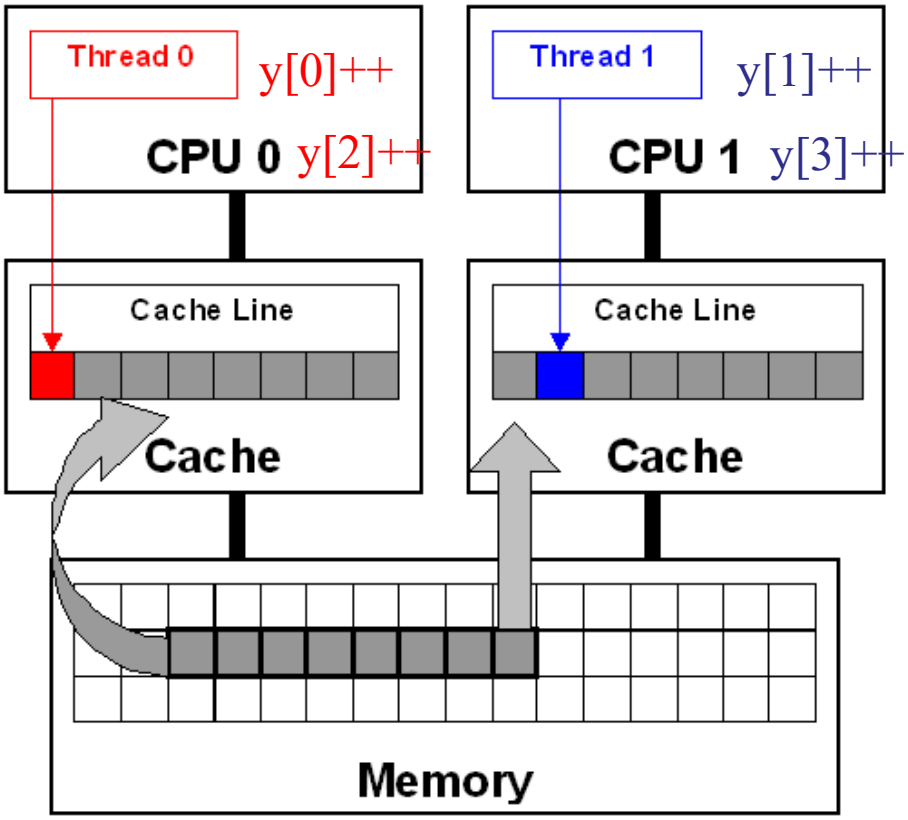
Example:

```
#include <omp.h>
omp_sched_t  schedtype=omp_sched_static;

omp_set_schedule(schedtype, 16);
#pragma parallel for schedule (runtime)
   for(i=0;i<n;i++)
        f(i);
```

False Sharing: Cache line/block movement between CPU cache and memory

- Example of false sharing with cyclic mapping

```
#pragma omp parallel for schedule(static,1)
for (int i=0; i<n; i++)
        y[i] = y[i] + 1
```



CPU0: Get cache line of y[0]

CPU1: Get cache line of y[1].
Invalidate CPU0's cache line

CPU0: Get cache line of y[2].
Invalidate CPU1's cache line

CPU1: Get cache line of y[3].
Invalidate CPU0's cache line

How to Avoid False Sharing

- **`Example of false sharing`**

```
#pragma omp parallel for schedule(static,1)
for (int i=0; i<n; i++)
        y[i] = y[i]+ 1
```

Block mapping or block cyclic mapping

```
#pragma omp parallel for schedule(static,32)
for (int i=0; i<n; i++)
        y[i] = y[i]+1;
```

Array padding

```
#pragma omp parallel for schedule(static,1)
for (int i=0; i<n; i++)
        y[i][0] ++;
```

Make shared variables private

OpenMP Summary

- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code
- OpenMP can enable (easy) parallelization of loop-based code with fork-join thread parallelism
  - #pragma omp parallel
  - #pragma omp parallel for
  - #pragma omp parallel private (i, x)
  - #pragma omp critical
  - #pragma omp for reduction(+ : sum)
- By default most systems use a block-partitioning of the iterations in a parallelized for loop.
  - OpenMP offers several loop scheduling options
- Make sure parallel region is thread-safe