CS 110
Computer Architecture

Sync & *OpenMP*

Instructors:
Chundong Wang & Siting Liu

https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html
School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkeley's CS61C

Review: TLP, OpenMP, and Sync

- Multicore
  - Hyperthreading
- OpenMP
  - Shared memory
  - Language extension
- Lock for synchronization
  - Data race
    - At least one write operation

```
wangc@HP:~/TT$ gcc omp.c -o p -O3 -fopenmp
wangc@HP:~/TT$ ./p
Hello World from thread = 1
Hello World from thread = 7
Hello World from thread = 6
Hello World from thread = 5
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 2
Hello World from thread = 4
Hello World from thread = 3
wangc@HP:~/TT$ ./p
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 5
Hello World from thread = 6
Hello World from thread = 4
wangc@HP:~/TT$ ./p
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 5
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 6
Hello World from thread = 4
Hello World from thread = 1
wangc@HP:~/TT$
```

Possible Lock Implementation

- Lock (a.k.a. busy wait)

```
Get_lock:                          # s0 -> addr of lock
        addiu t1,zero,1            # t1 = Locked value
Loop:   lw    t0,0(s0)             # load lock
        bne   t0,zero,Loop         # loop if locked
Lock:   sw    t1,0(s0)             # Unlocked, so lock
```

- Unlock

```
Unlock:
     sw zero,0(s0)
```

- Any problems with this?

Possible Lock Problem

- Thread 1

```
      addiu t1,zero,1
Loop: lw  t0,0(s0)




      bne t0,zero,Loop

Lock: sw t1,0(s0)
```

- Thread 2

```


      addiu t1,zero,1
Loop: lw t0,0(s0)




      bne t0,zero,Loop

Lock: sw t1,0(s0)
```

Time

Both threads think they have set the lock!
Exclusive *access not guaranteed!*

RISC-V: Two solutions!

- Option 1: Read/Write Pairs
  - Pair of instructions for “linked” read and write
  - Load reserved and Store conditional
  - No other access permitted between read and write
    - Must use *shared memory* (<u>multi</u>processing)

- Option 2: Atomic Memory Operations
  - Atomic swap of register $\leftrightarrow$ memory

Read/Write Pairs

- Load reserved: **lr rd, rs**
  - Load the word pointed to by **rs** into **rd**, and add a reservation
- Store conditional: **sc rd, rs1, rs2**
  - Store the value in **rs2** into the memory location pointed to by **rs1**, only if the reservation is still valid and set the status in **rd**
    - Returns 0 (success) if location has not changed since the **lr**
    - Returns nonzero (failure) if location has changed: ***Actual store will not take place***

Synchronization in RISC-V Example

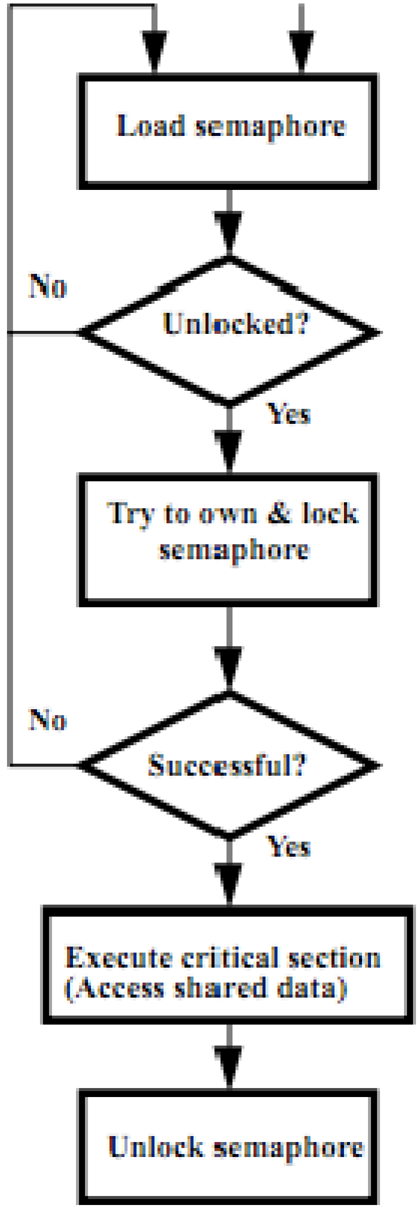
- Atomic swap (to test/set lock variable)
- Exchange contents of register and memory:
  s4 $\leftrightarrow$ Mem(s1)

```
try:
  lr   t1, s1         #load reserved
  sc   t0, s1, s4     #store conditional
  bne  t0, x0, try    #loop if sc fails
  add  s4, x0, t1     #load value in s4
```

sc would fail if another thread executes **sc** here

Test-and-Set

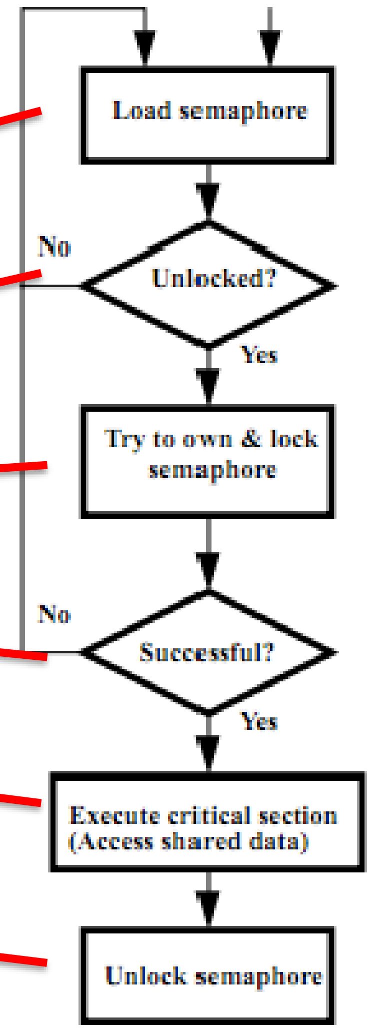
- In a single atomic operation:
  - ***Test*** to see if a memory location is set (contains a 1)
  - ***Set*** it (to 1) if it isn't (it contained a zero when tested)
    - Otherwise indicate that the Set failed, so the program can try again
  - While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions
- Useful for implementing lock operations



Test-and-Set in RSIC-V using lr/sc

- Example: RISC-V sequence for implementing a T&S at (s1)

```
        li t2, 1
Try:
        lr t1, s1
        bne t1, x0, Try
        sc t0, s1, t2
        bne t0, x0, Try
Locked:
        # critical section
Unlock:
        sw x0,0(s1)
```



Option 2: RISC-V Atomic Memory Operations (AMOs)

- Encoded with an R-type instruction format
  - swap, add, and, or, xor, max, min
  - **`AMOSWAP`** `rd, rs2, (rs1)`   rd = *rs1, *rs1 = rs2
  - **`AMOADD`** `rd, rs2, (rs1)`
- Take the value pointed to by `rs1`
  - Load it into `rd`

    aq(acquire) and rl(release) to insure *in order* execution
  - Apply the operation to that value with the contents in `rs2`
    - If `rs2==rd`, use the old value in `rd`
  - Store the result back to where `rs1` is pointed to
- This allows atomic swap as a primitive
  - It also allows "reduction operations" that are common to be efficiently implemented

RISC-V Critical Section

- Assume that the lock is in memory location stored in register a0
- The lock is "set" if it is 1; it is "free" if it is 0 (it's initial value)

```
      li            t0, 1          # Get 1 to set lock
Try:  amoswap.w.aq  t1, t0, (a0)   # t1 gets old lock value
                                   # while we set it to 1
      bnez          t1, Try        # if it was already 1, another
                                   # thread has the lock,
                                   # so we need to try again
          … critical section goes here …
      amoswap.w.rl  x0, x0, (a0)   # store 0 in lock to release
```

Lock Synchronization

Broken Synchronization

```
while (lock != 0) ;

lock = 1;

// critical section

lock = 0;
```

Fix (lock is at location (a0))

```
      li             t0, 1
Try:  amoswap.w.aq   t1, t0, (a0)
      bnez           t1, Try
Locked:

      # critical section

Unlock:
      amoswap.w.rl   x0, x0, (a0)
```

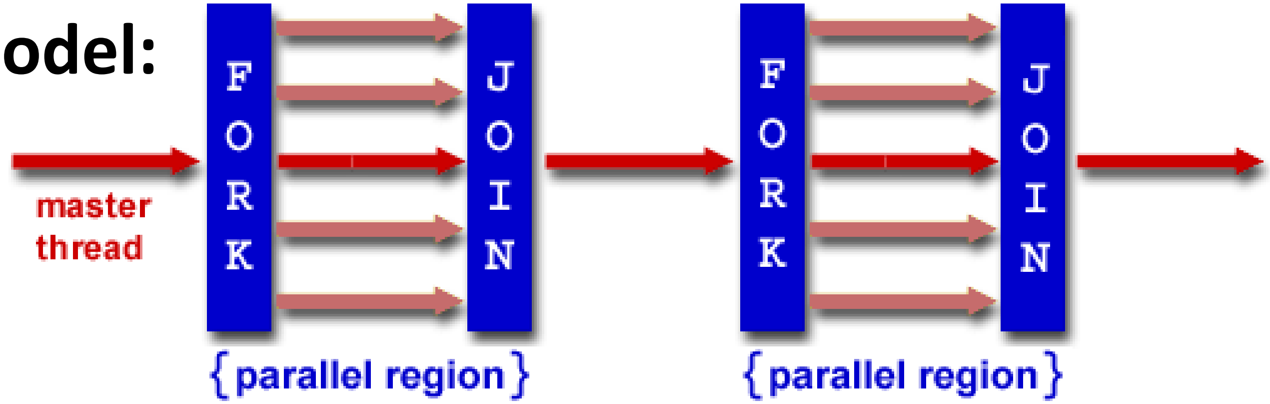
How to use

- Don't implement yourself!

- Use according library – e.g.:
  - pthread
  - C++:
    - std::thread C++11 https://en.cppreference.com/w/cpp/thread
    - std::jthread C++20
    - std::mutex; std::lock_guard; std::scoped_lock; std::shared_lock
    - std::condition_variable; std::counting_semaphore; std::latch; std::barrier
    - std::promise; std::future
  - Qt QThread
  - OpenMP

OpenMP Programming Model - Review

- **Fork - Join Model:**



- OpenMP programs begin as single process (*master thread*) and executes sequentially until the first parallel region construct is encountered
  - *FORK:* Master thread then creates a team of parallel threads
  - Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads
  - *JOIN:* When the team of threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

`parallel` Pragma and Scope - Review

- Basic OpenMP construct for parallelization:

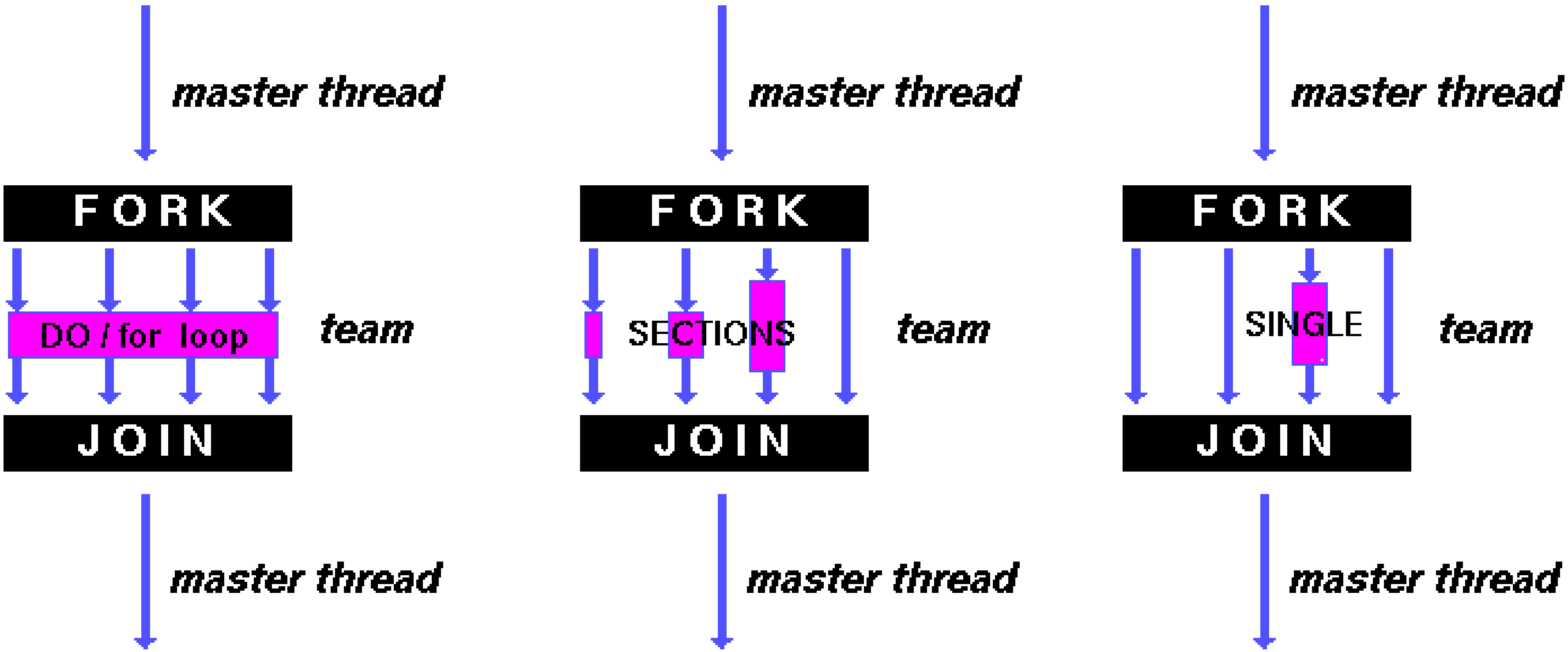
```
#pragma omp parallel
{
   /* code goes here */
}
```

  - *Each* thread runs a copy of code within the block
  - Thread scheduling is *non-deterministic*
- OpenMP default is *shared* variables
  - To make private, need to declare with pragma:

```
#pragma omp parallel private (x)
```

OpenMP Directives (Work-Sharing)

- These are defined *within* a `parallel` section



Shares iterations of a loop across the threads

Each section is executed by a separate thread

Serializes the execution of a thread

`Parallel` Statement Shorthand

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

This is the only directive in the parallel section

can be shortened to:

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

- Also works for `sections`

Building Block: `for` loop

```
for (i=0; i<max; i++) zero[i] = 0;
```

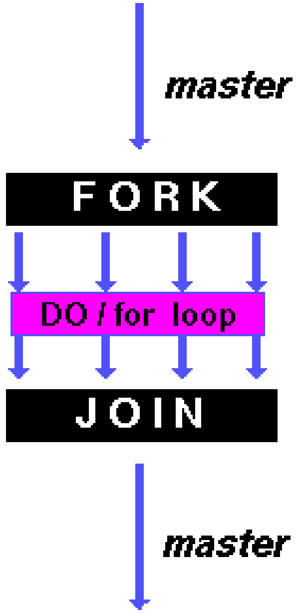
- Breaks *for loop* into chunks, and allocate each to a separate thread
  - e.g. if `max` = 100 with 2 threads:
    assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple "shape" for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed
  - i.e. No `break`, `return`, `exit`, `goto` statements

In general, don't jump outside of any pragma block

Parallel `for` *pragma*

```
#pragma omp parallel for
  for (i=0; i<max; i++) zero[i] = 0;
```

- Master thread creates additional threads, each with a separate execution context
- All variables declared outside for loop are shared by default, except for loop index which is *private* per thread (Why?)
- Implicit "barrier" synchronization at end of for loop
- Divide index regions sequentially per thread
  - Thread 0 gets 0, 1, …, (max/n)-1;
  - Thread 1 gets max/n, (max/n)+1, …, 2*(max/n)-1



OpenMP Example

```c
/* clang -Xpreprocessor -fopenmp -lomp -o for for.c */

#include <stdio.h>
#include <omp.h>
int main()
{
    omp_set_num_threads(4);
    int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int N = sizeof(a)/sizeof(int);

    #pragma omp parallel for
    for (int i=0; i<N; i++) {
        printf("thread %d, i = %2d\n",
            omp_get_thread_num(), i);
        a[i] = a[i] + 10 * omp_get_thread_num();
    }

    for (int i=0; i<N; i++) printf("%02d ", a[i]);
    printf("\n");
}
```

```
$ gcc-5 -fopenmp for.c;./a.out
% clang -Xpreprocessor -fopenmp -
lomp -o for for.c; ./for
thread 0, i =  0
thread 1, i =  3
thread 2, i =  6
thread 3, i =  8
thread 0, i =  1
thread 1, i =  4
thread 2, i =  7
thread 3, i =  9
thread 0, i =  2
thread 1, i =  5
00 01 02 13 14 15 26 27 38 39
```


The call to find the maximum number of threads that are available to do work is **`omp_get_max_threads()`** (from omp.h).

OpenMP Timing

- Elapsed wall clock time:

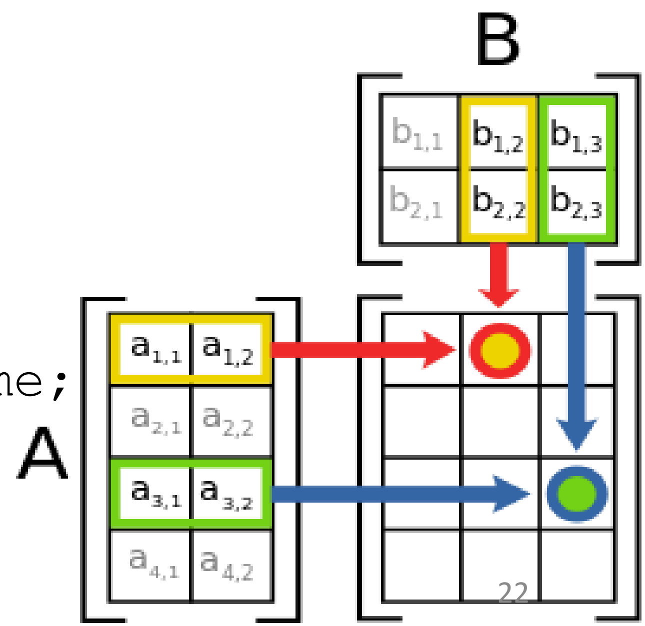
  `double omp_get_wtime(void);`

  - Returns elapsed wall clock time in seconds
  - Time is measured per thread, no guarantee can be made that two distinct threads measure the same time
  - Time is measured from “some time in the past,” so subtract results of two calls to `omp_get_wtime` to get elapsed time

Matrix Multiply in OpenMP

```
// C[M][N] = A[M][P] × B[P][N]
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, j, k)
  for (i=0; i<M; i++){
    for (j=0; j<N; j++){
      tmp = 0.0;
      for( k=0; k<P; k++){
        /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
        tmp += A[i][k] * B[k][j];
      }
      C[i][j] = tmp;
    }
  }
run_time = omp_get_wtime() - start_time;
```

Outer loop spread across n threads;
inner loops inside a single thread

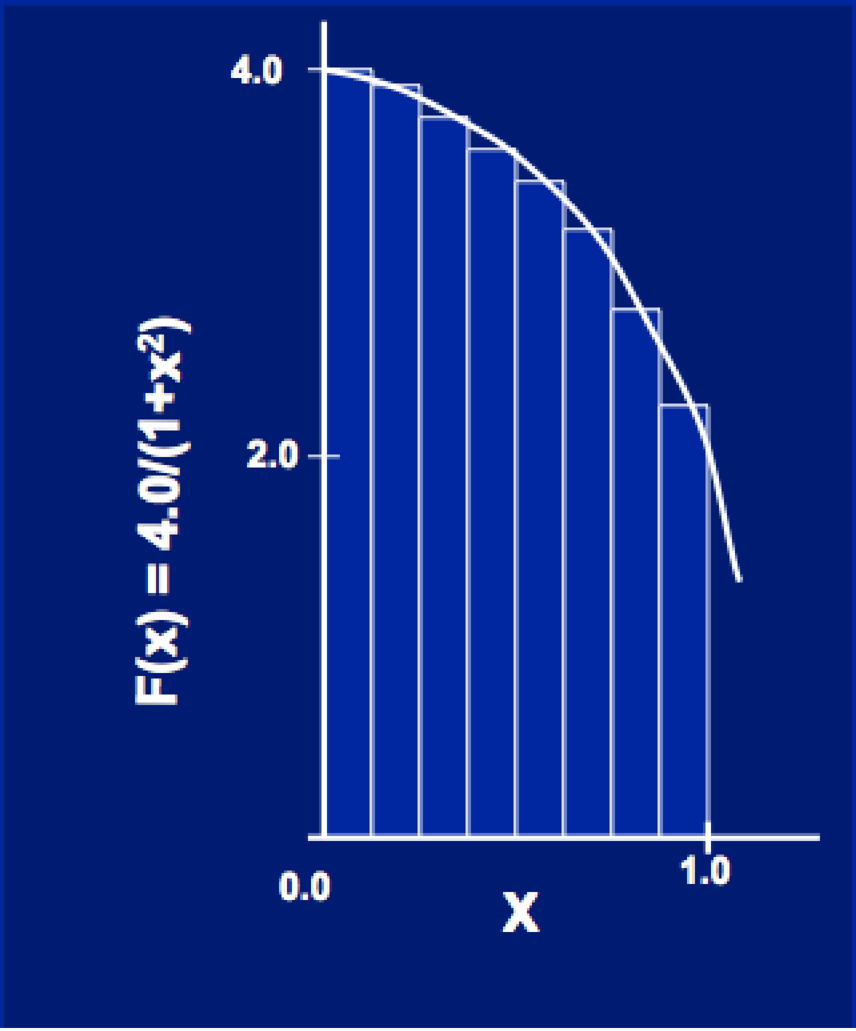


Notes on Matrix Multiply Example

- More performance optimizations available:
  - Higher *compiler optimization* (-O2, -O3) to reduce number of instructions executed
  - *Cache blocking* to improve memory performance
  - Using SIMD SSE instructions to raise floating point computation rate (*DLP*)

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 π

```
#include <stdio.h>

void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5) *step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

```
pi = 3.142425985001
```

- Resembles π, but not very accurate
- Let's increase **`num_steps`** and parallelize

Parallelize (1) …

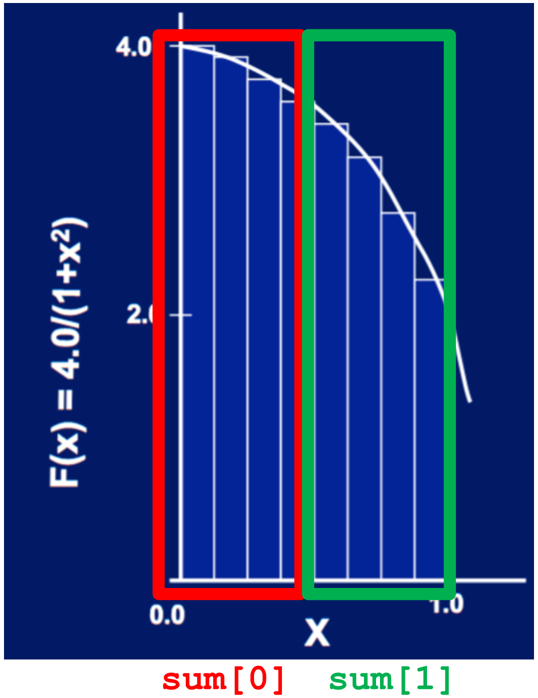
#include <omp.h>

```
#include <stdio.h>

void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
#pragma parallel for
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5) *step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

- Problem: each thread needs access to the shared variable **sum**
- Code runs sequentially …

Parallelize (2) …



1. Compute
   `sum[0]` and **`sum[1]`**
   in parallel

2. Compute
   `sum = sum[0] + sum[1]`
   sequentially

Parallel π—Trial Run

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

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5)*step;
            sum[id] += 4.0*step/(1.0+x*x);
            printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

```
i =  1,  id =  1
i =  0,  id =  0
i =  2,  id =  2
i =  3,  id =  3
i =  5,  id =  1
i =  4,  id =  0
i =  6,  id =  2
i =  7,  id =  3
i =  9,  id =  1
i =  8,  id =  0
pi = 3.142425985001
```

Scale up: `num_steps = 10`^{6}

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

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 1000000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            // printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

```
pi =
3.141592653590
```

You verify how many digits are correct …

Can We Parallelize Computing `sum`?

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

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

Always looking for ways to beat Amdahl's Law …

Summation inside parallel section

- Insignificant speedup in this example, but …
- **`pi = 3.138450662641`**
- Wrong! And value changes between runs?!
- What's going on?

What's Going On?

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

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

Can you resolve such a problem?

- Operation is really
  `pi = pi + sum[id]`
- What if >1 threads reads current (same) value of **`pi`**, computes the sum, stores the result back to **`pi`**?
- Each processor reads same intermediate value of **`pi`**!
- Result depends on who gets there when
  - A "race" → result is **not deterministic**

OpenMP Reduction

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp parallel for private ( sum )
for (i = 0; i <= MAX ; i++)
    sum += A[i];
avg = sum/MAX;  // bug
```

- *Problem is that we really want sum over all threads!*
- *Reduction*: specifies that, 1 or more variables that are private to each thread, are subject of reduction operation at end of parallel region:
  reduction(operation:var) where
  - *Operation*: operator to perform on the variables (var) at the end of the parallel region : +, *, -, &, ^, |, &&, or ||.
  - *Var*: One or more variables on which to perform scalar reduction.

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp for reduction(+ : sum)
for (i = 0; i <= MAX ; i++)
    sum += A[i];
avg = sum/MAX;
```

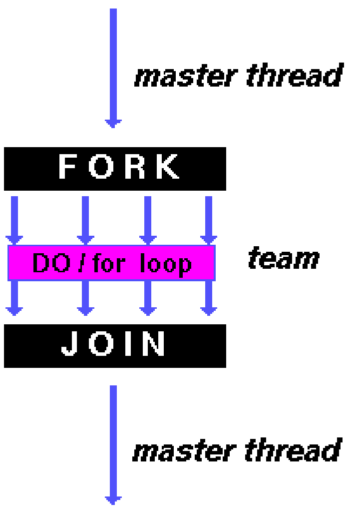
parallel for, reduction

```
#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 * step;
    printf ("pi = %6.12f\n", pi);
}
```

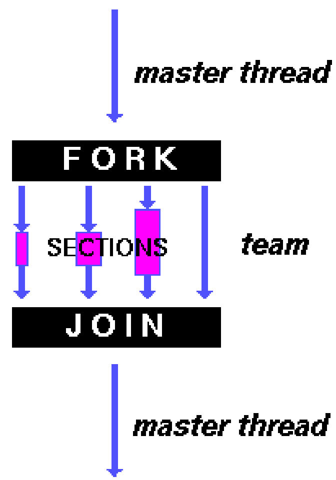
```
wangc@HP:~/TT$ gcc pi.c -o p -fopenmp
wangc@HP:~/TT$ ./p
pi = 3.141592653598
```

More on OpenMP

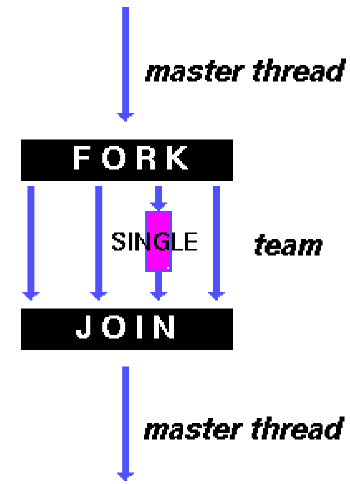
- These are defined *within* a `parallel` section



Shares iterations of a loop across the threads



Each section is executed by a separate thread



Serializes the execution of a thread

There are more, like `critical`, `barrier`, `atomic`, `master`, … Try them by yourself.

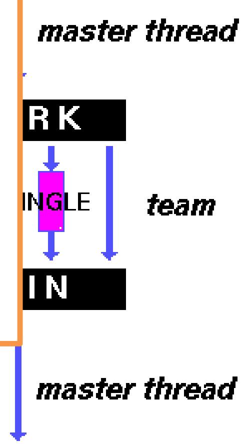
```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
        int i = 0;
        omp_set_num_threads(4); // Maximum 4 threads
        #pragma omp parallel private(i)
        {
                printf("thread %d start\n", omp_get_thread_num());

                #pragma omp single
                {
                        for (i = 0; i < 6; i++)
                        {
                                printf("Single, thread %d execute i = %d\n",
                                   omp_get_thread_num(), i);
                        }
                }

        }
}
```

ection



master thread

single: code block executed by one thread only;
Other threads will wait;
Useful for thread-unsafe code;
Useful for I/O operations.

Each section is executed by a separate thread

Serializes the execution of a thread

, barrier, atomic, master, ... Try them by yourself.

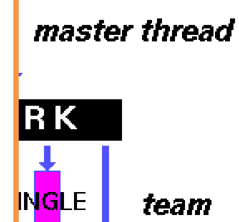
```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
        int i = 0;
        omp_set_num_threads(4); // Maximum 4 threads
        #pragma omp parallel private(i)
        {
                printf("thread %d start\n", omp_get_thread_num());

                #pragma omp single
                {
                        for (i = 0; i < 6; i++)
                        {
                                printf("Single, thread %d execute i = %d\n",
                                   omp_get_thread_num(), i);
                        }
                }

        }
}
```

ection



single: code block executed by one thread only;
Other threads will wait;
Useful for thread-unsafe code;
Useful for I/O operations.

```
wangc@HP:~/TT$ gcc single.c -o s -fopenmp
wangc@HP:~/TT$ ./s
thread 3 start
Single, thread 3 execute i = 0
Single, thread 3 execute i = 1
Single, thread 3 execute i = 2
Single, thread 3 execute i = 3
Single, thread 3 execute i = 4
Single, thread 3 execute i = 5
thread 1 start
thread 2 start
thread 0 start
wangc@HP:~/TT$
```

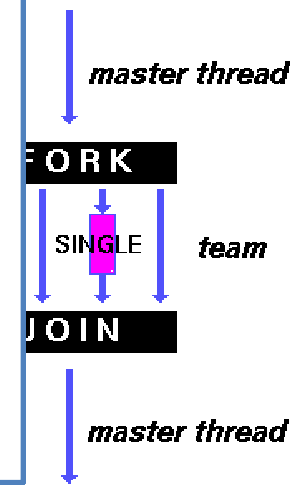
```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
        int i = 0;
        omp_set_num_threads(4); // Maximum 4 threads
        #pragma omp parallel private(i)
        {
                printf("thread %d start\n", omp_get_thread_num());

                #pragma omp master
                {
                        for (i = 0; i < 6; i++)
                        {
                                printf("Master, thread %d execute i = %d\n",
                                   omp_get_thread_num(), i);
                        }
                }
                printf("Outside master, thread %d execute i = %d\n",
                         omp_get_thread_num(), i);


        }
}
```

section



ch section is executed
a separate thread

Serializes the execution
of a thread

master Directive ensures that only the master threads executes instructions in the block. There is no implicit barrier, so other threads will not wait for master to finish

rrier, atomic, master, ... Try them by yourself.

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

int main(int argc, char **argv) {
        int i = 0;
        omp_set_num_threads(4); // Maximum 4 threads
        #pragma omp parallel private(i)
        {
                printf("thread %d start\n", omp_get_thread_num());

                #pragma omp master
                {
                        for (i = 0; i < 6; i++)
                        {
                                printf("Master, thread %d execute i = %d\n",
                                   omp_get_thread_num(), i);
                        }
                }
                printf("Outside master, thr
                         omp_get_thread_num


        }
}
```

section

master thread

```
wangc@HP:~/TT$ gcc master.c -o m -fopenmp
wangc@HP:~/TT$ ./m
thread 2 start
Outside master, thread 2 execute i = 0
thread 1 start
Outside master, thread 1 execute i = 0
thread 3 start
Outside master, thread 3 execute i = 0
thread 0 start
Master, thread 0 execute i = 0
Master, thread 0 execute i = 1
Master, thread 0 execute i = 2
Master, thread 0 execute i = 3
Master, thread 0 execute i = 4
Master, thread 0 execute i = 5
Outside master, thread 0 execute i = 6
wangc@HP:~/TT$
```

master Directive ensures that only the master threads executes instructions in the block. There is no implicit barrier, so other threads will not wait for master to finish

```
wangc@HP-Z2-G4:~/Works/TT$ ./p
thread 0 start
Master, thread 0 execute i = 0
Master, thread 0 execute i = 1
thread 2 start
Outside master, thread 2 execute i = 0
Master, thread 0 execute i = 2
Master, thread 0 execute i = 3
Master, thread 0 execute i = 4
Master, thread 0 execute i = 5
Outside master, thread 0 execute i = 6
thread 3 start
Outside master, thread 3 execute i = 0
thread 1 start
Outside master, thread 1 execute i = 0
```

```
4 threads
, omp_get_thread_num());
i++)
ster, thread %d execute i = %d\n",
_thread_num(), i);
omp_get_thread_num
        }
}
```

section

master thread

```
wangc@HP:~/TT$ gcc master.c -o m -fopenmp
wangc@HP:~/TT$ ./m
thread 2 start
Outside master, thread 2 execute i = 0
thread 1 start
Outside master, thread 1 execute i = 0
thread 3 start
Outside master, thread 3 execute i = 0
thread 0 start
Master, thread 0 execute i = 0
Master, thread 0 execute i = 1
Master, thread 0 execute i = 2
Master, thread 0 execute i = 3
Master, thread 0 execute i = 4
Master, thread 0 execute i = 5
Outside master, thread 0 execute i = 6
wangc@HP:~/TT$
```

master Directive ensures that only the master threads executes instructions in the block. There is no implicit barrier, so other threads will not wait for master to finish

And in Conclusion, …

- Multiprocessor/Multicore uses Shared Memory
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!
    - To be covered with "Advanced caches" :-)
- OpenMP as simple parallel extension to C
  - Threads, Parallel for, private, reductions …
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble
  - Much we didn't cover – including other synchronization mechanisms (locks, etc.)