

Lecture 2

Announcements

- A1 posted by 9AM on Monday morning, probably sooner, will announce via Piazza
- Lab hours starting next week: will be posted by Sunday afternoon

CLICKERS OUT

Have you found a programming partner?

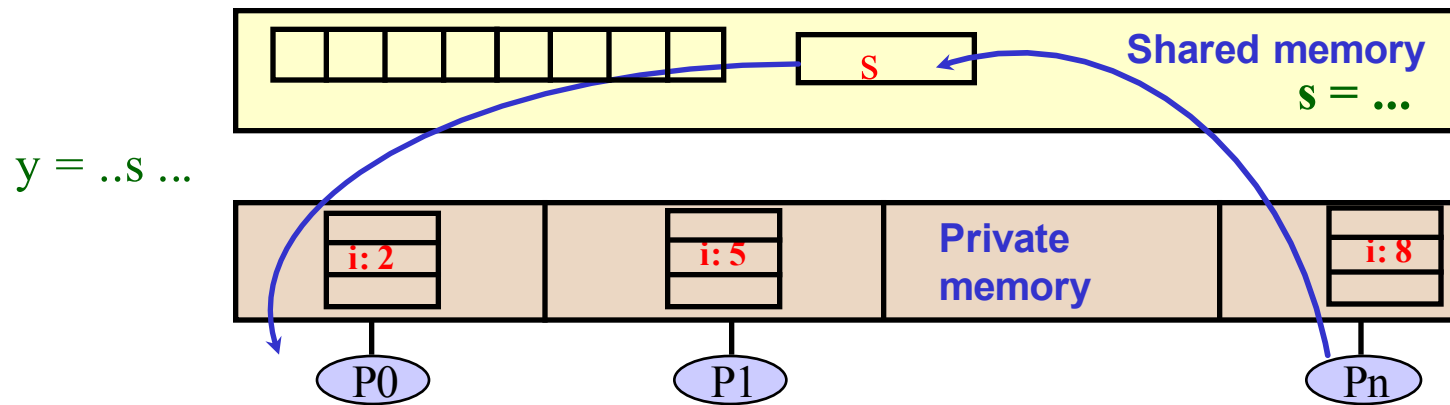
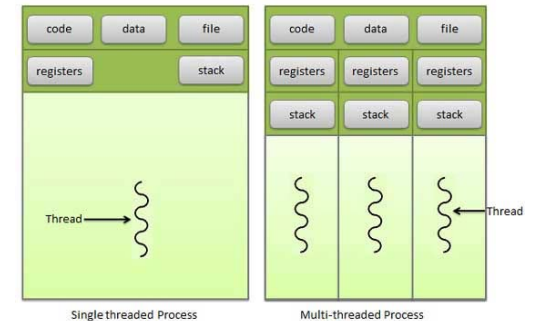
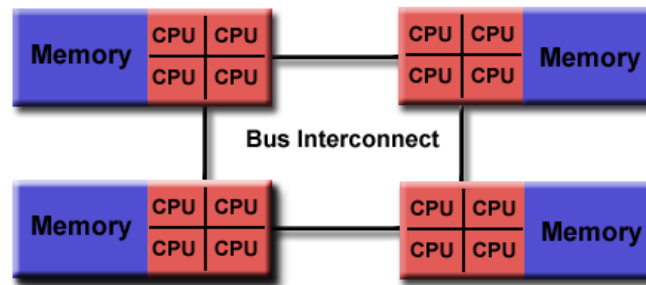
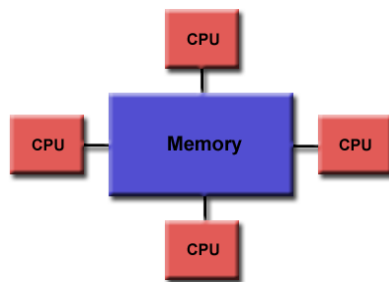
A. Yes

B. Not yet, but I may have a lead

C. No

Recapping from last time

- We will program multicore processors with multithreading
 - Multiple program counters
 - A new storage class: shared data
 - Synchronization may be needed when updating shared state (*thread safety*)



Hello world with <Threads>

```
#include <thread>
void Hello(int TID) {
    cout << "Hello from thread " << TID << endl;
}

int main(int argc, char *argv[ ]){
    thread *thrds = new thread[NT];

    // Spawn threads
    for(int t=0;t<NT;t++){
        thrds[t] = thread(Hello, t );
    }

    // Join threads
    for(int t=0;t<NT;t++)
        thrds[t].join();
}
```

```
$ ./hello_th 3
Hello from thread 0
Hello from thread 1
Hello from thread 2
$ ./hello_th 3
Hello from thread 1
Hello from thread 0
Hello from thread 2
$ ./hello_th 4
Running with 4 threads
Hello from thread 0
Hello from thread 3
Hello from thread Hello from
thread 21
```

\$PUB/Examples//Threads/Hello-Th

PUB = /share/class/public/cse160-wi16

What things can threads do?

- A. Create even more threads
- B. Join with others created by the parent
- C. Run different code fragments
- D. Run in lock step
- E. A, B & C

Steps in writing multithreaded code

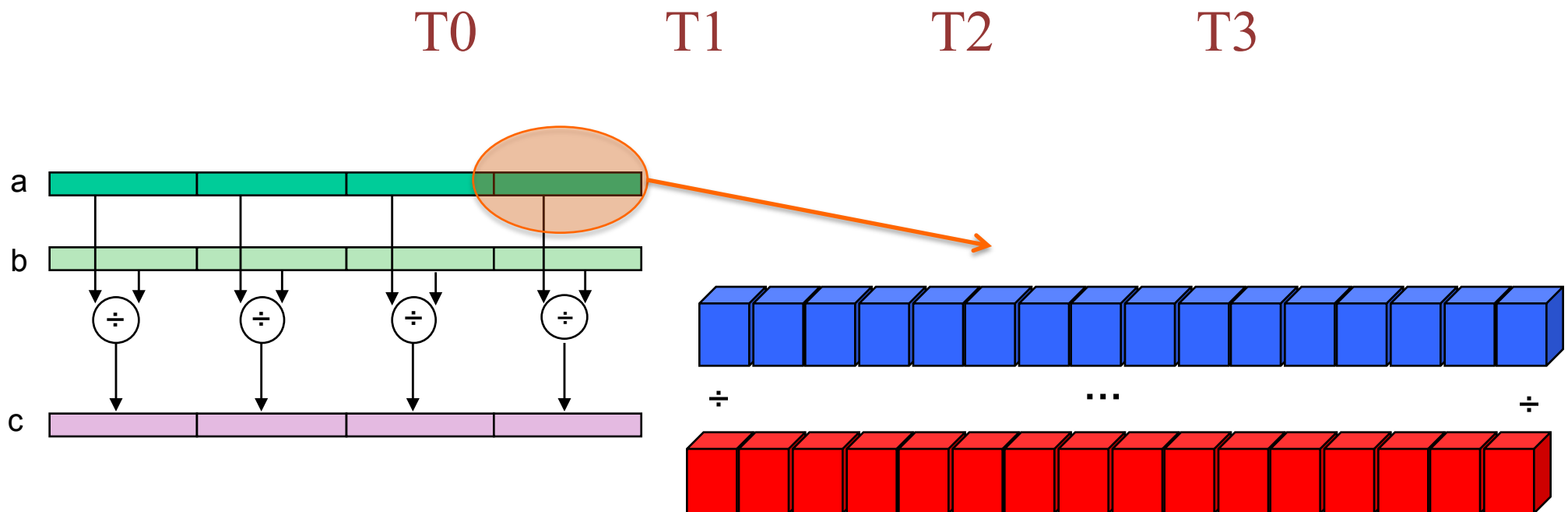
- We write a *thread function* that gets called each time we spawn a new thread
- *Spawn* threads by constructing objects of class Thread (in the C++ library)
- Each thread runs on a separate processing core (If more threads than cores, the threads share cores)
- Threads share memory, declare shared variables outside the scope of any functions
- Divide up the computation fairly among the threads
- *Join* threads so we know when they are done

Today's lecture

- A first application
- Performance characterization
- Data races

A first application

- Divide one array of numbers into another, *pointwise*
for $i = 0:N-1$
 $c[i] = a[i] / b[i];$
- Partition arrays into intervals, assign each to a unique thread
- Each thread sweeps over a reduced problem



Pointwise division of 2 arrays with threads

```
#include <thread>
int *a, *b, *c;
void Div(int TID, int N, int NT) {
    int64_t i0 = TID*(N/NT), i1 = i0 + (N/NT);
    for (int r = 0; r < REPS; r++)
        for (int i = i0; i < i1; i++)
            c[i] = a[i] / b[i];
}
int main(int argc, char *argv[]) {
    thread *thrs = new thread[NT];
    // allocate a, b and c
    // Spawn threads
    for (int t = 0; t < NT; t++) {
        thrs[t] = thread(Div, t, N, NT);
    }
    // Join threads
    for (int t = 0; t < NT; t++)
        thrs[t].join();
}
```

qlogin

\$./div 1 50000000 (50m)

0.3099 seconds

\$./div 2 50000000

0.1980 seconds

\$./div 4 50000000

0.1258 seconds

\$./div 8 50000000

0.1185 seconds

\$PUB/Examples/Threads/Div

PUB = /share/class/public/cse160-wi16

Why did the program run only a little faster on 8 cores than on 4?

- A. There wasn't enough work to give out so some were starved
- B. Memory traffic is saturating the bus**
- C. The workload is shared unevenly and not all cores are doing their fair share

Today's lecture

- A first application
- Performance characterization
- Data races

Measures of Performance

- Why do we measure performance?
- How do we report it?
 - ▶ Completion time
 - ▶ Processor time product
Completion time $\times$ # processors
 - ▶ Throughput: amount of work that can be accomplished in a given amount of time
 - ▶ Relative performance: given a reference architecture or implementation
AKA *Speedup*

Parallel Speedup and Efficiency

- How much of an improvement did our parallel algorithm obtain over the serial algorithm?
- Define the *parallel speedup*, $S_P = T_1 / T_P$

$$S_P = \frac{\text{Running time of the best serial program on 1 processor}}{\text{Running time of the parallel program on P processors}}$$

- T_1 is defined as the running time of the “best serial algorithm”
- In general: *not* the running time of the parallel algorithm on 1 processor
- **Definition:** *Parallel efficiency* $E_P = S_P / P$

What can go wrong with speedup?

- Not always an accurate way to compare different algorithms....
- .. or the same algorithm running on different machines
- We might be able to obtain a better running time even if we lower the speedup
- If our goal is performance, the bottom line is running time T_P

Program P gets a higher speedup on machine A than on machine B.

Does the program run faster on machine A or B?

A. A

B. B

C. Can't say

Superlinear speedup

- We have a *super-linear* speedup when

$$E_P > 1 \Rightarrow S_P > P$$

- Is it believable?
 - ▶ Super-linear speedups are often an artifact of inappropriate measurement technique
 - ▶ Where there is a super-linear speedup, a better serial algorithm may be lurking

What is the maximum possible speedup of any program running on 2 cores ?

A. 1

B. 2

C. 4

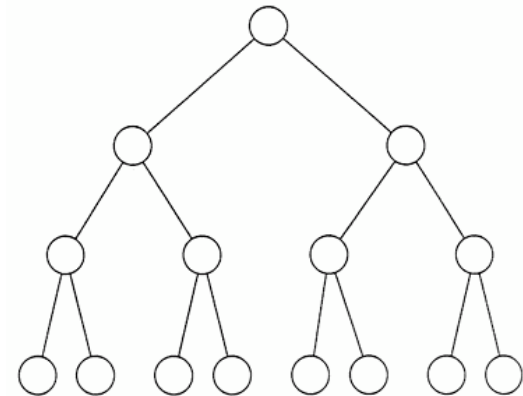
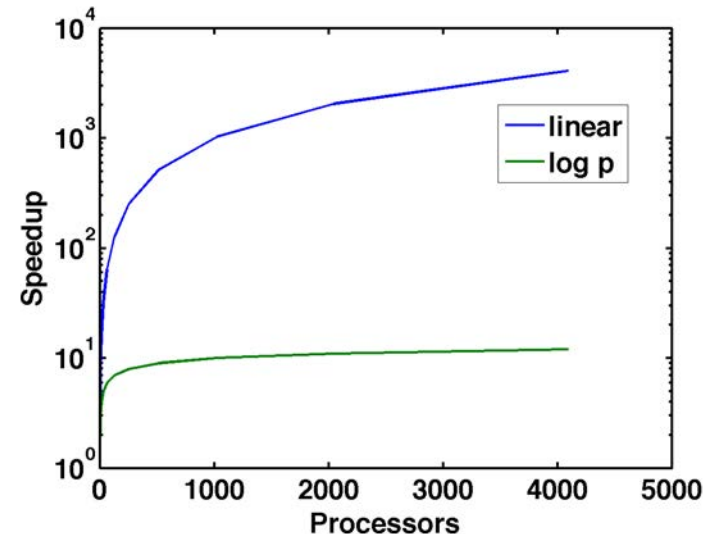
D. 10

E. None of these

Scalability

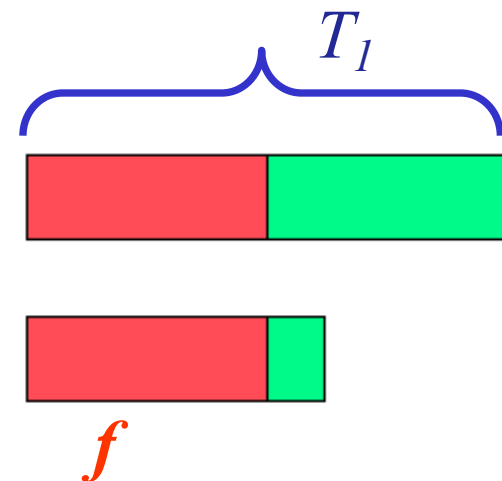
- A computation is **scalable** if performance increases as a “nice function” of the number of processors, e.g. linearly
- In practice scalability can be hard to achieve
 - ▶ Serial sections: code that runs on only one processor
 - ▶ “Non-productive” work associated with parallel execution, e.g. synchronization
 - ▶ Load imbalance: uneven work assignments over the processors
- Some algorithms present intrinsic barriers to scalability leading to alternatives

for $i=0:n-1$ $sum = sum + x[i]$



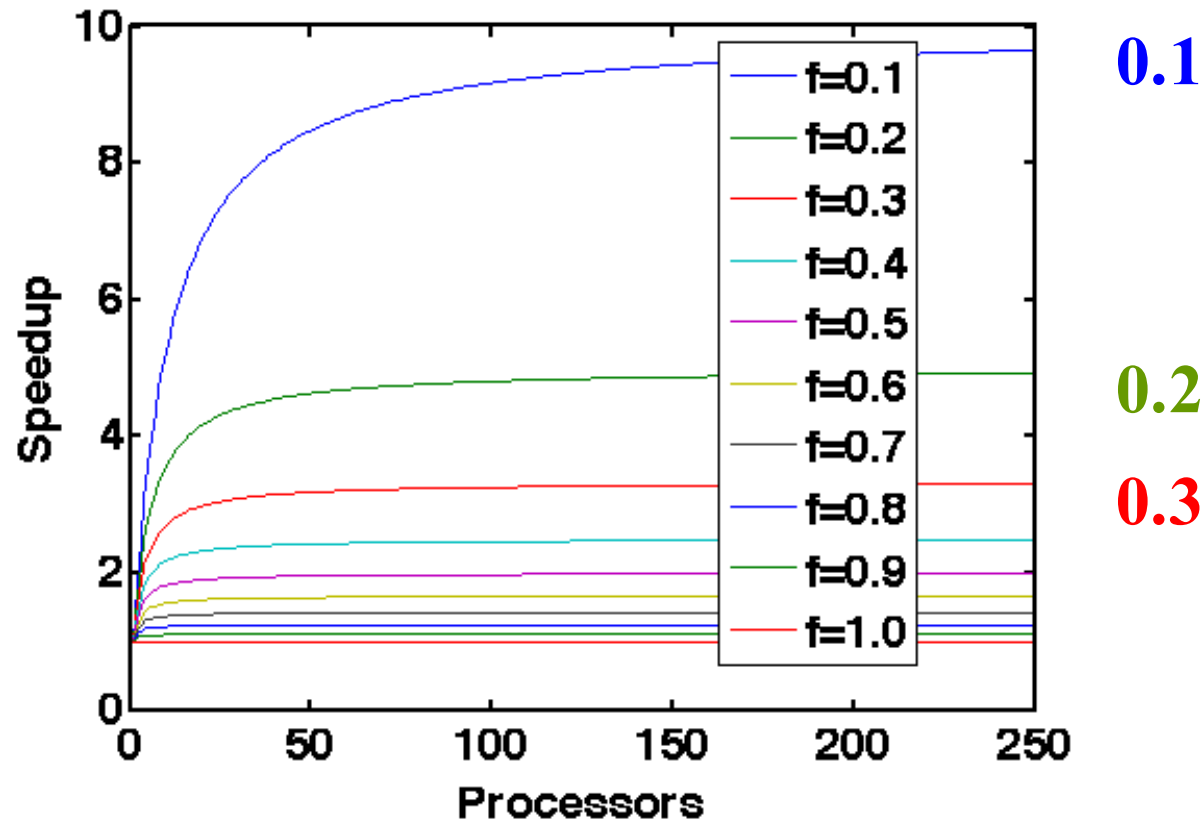
Serial Sections

- Limit scalability
- Let f = the fraction of T_1 that runs serially
- $T_1 = f \times T_1 + (1-f) \times T_1$
- $T_P = f \times T_1 + (1-f) \times T_1 / P$
Thus $S_P = 1/[f + (1 - f)/p]$
- As $P \rightarrow \infty$, $S_P \rightarrow 1/f$
- This is known as *Amdahl's Law* (1967)



Amdahl's law (1967)

- A serial section limits scalability
- Let f = fraction of T_1 that runs serially
- *Amdahl's Law* (1967) : As $P \rightarrow \infty$, $S_P \rightarrow 1/f$

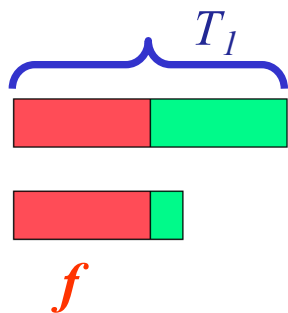


Performance questions

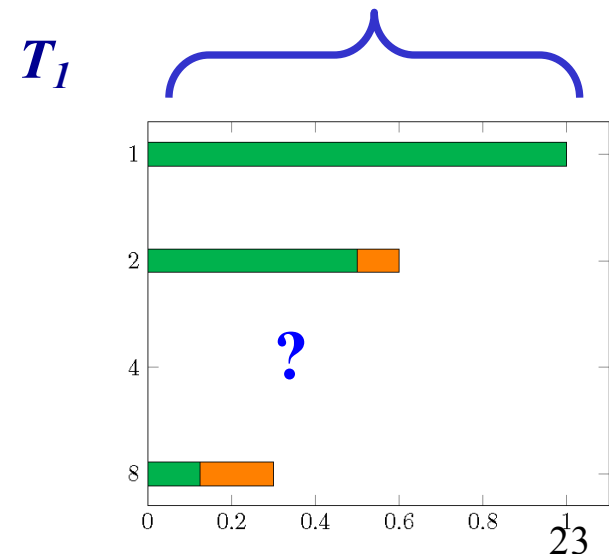


- You observe the following running times for a parallel program running a fixed workload N
- Assume that the only losses are due to serial sections
- What are the speedup and efficiency on 2 processors?
- What is the maximum possible speedup on an infinite number of processors?

$$S_p = 1/[f + (1 - f)/p]$$
- What is the running time on 4 processors?



NT	Time
1	1.0
2	0.6
8	0.3



Performance questions



- You observe the following running times for a parallel program running a fixed workload and the only losses are due to serial sections

- What are the speedup and efficiency on 2 processors?

$$S_2 = T_1 / T_2 = 1.0 / 0.6 = 5/3 = 1.67; E_2 = S_2 / 2 = 0.83$$

- What is the maximum possible speedup on an infinite number of processors?

$$S_P = 1/[f + (1 - f)/p]$$

Do compute the max speedup, we need to determine f

Do determine f , we plug in known values (S_2 and p):

$$5/3 = 1/[f + (1-f)/2] \Rightarrow 3/5 = f + (1-f)/2 \Rightarrow f = 1/5$$

So what is S_∞ ?

- What is the running time on 4 processors?

Plugging values into the S_P expression:

$$S_4 = 1/[1/5 + (4/5)/4] \Rightarrow S_4 = 5/2$$

$$\text{But } S_4 = T_1 / T_4, \text{ So } T_4 = T_1 / S_4$$

NT	Time
1	1.0
2	0.6
8	0.3

Weak scaling

- Is Amdahl's law pessimistic?
- Observation: Amdahl's law assumes that the workload (W) remains fixed
- But parallel computers are used to tackle more ambitious workloads
- If we increase W with P we have **weak scaling**

f often decreases with W

- We can continue to enjoy speedups
 - Gustafson's law [1992]
http://en.wikipedia.org/wiki/Gustafson's_law
www.scl.ameslab.gov/Publications/Gus/FixedTime/FixedTime.pdf

Isoefficiency

- The consequence of Gustafson's observation is that we increase N with P
- We can maintain constant efficiency so long as we increase N appropriately
- The *isoefficiency* function specifies the growth of N in terms of P
- If N is linear in P , we have a scalable computation
- If not, memory per core grows with P !
- Problem: the amount of memory per core is shrinking over time

Today's lecture

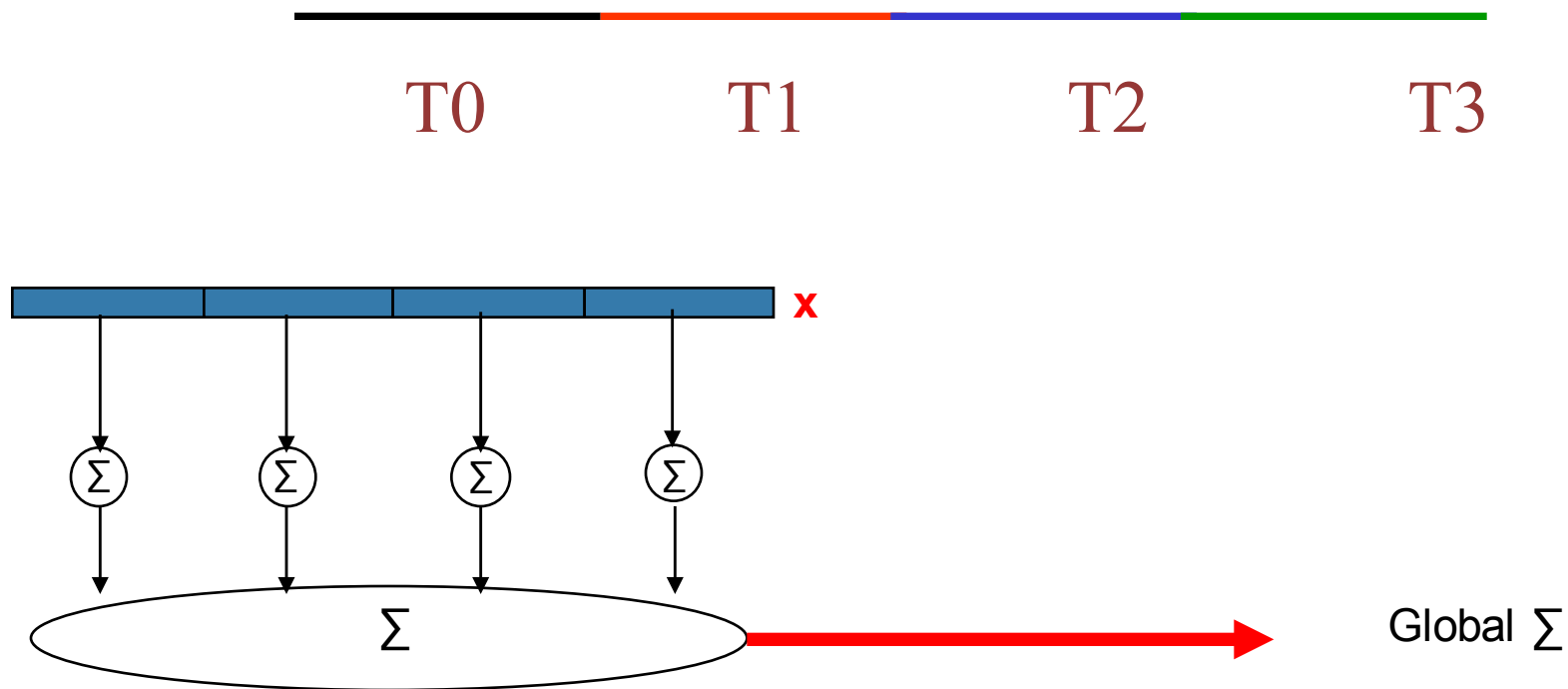
- A first application
- Performance characterization
- Data races

Summing a list of integers

for $i = 0:N-1$

$\text{sum} = \text{sum} + x[i];$

- Partition $x[]$ into intervals, assign each to a unique thread
- Each thread sweeps over a reduced problem



First version of summing code

```
void sum(int TID, int N, int NT){  
    int64_t i0 = TID*(N/NT), i1 = i0 + (N/NT);  
    int64_t local_sum=0;  
    for (int64_t i=i0; i<i1; i++)  
        local_sum += x[i];  
    global_sum += local_sum  
}
```

```
int* x;  
Main():  
    int64_t global_sum;  
    for(int64_t t=0; t<NT; t++){  
        thrds[t] = thread(sum,t,N,NT);
```

Results

- The program usually runs correctly
- But sometimes it produces incorrect results:

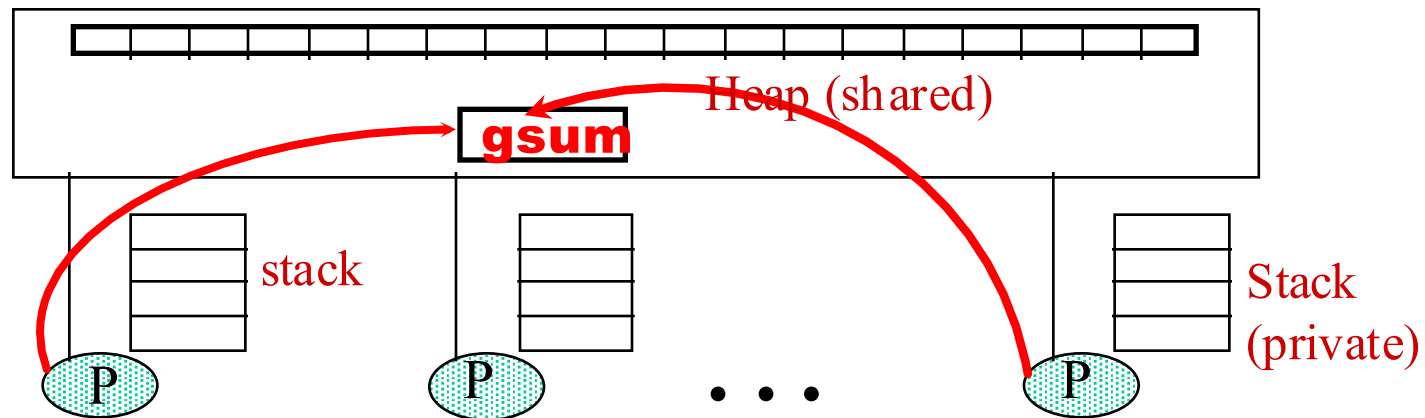
Result verified to be INCORRECT, should be 549756338176

```
void sum(int TID, int N, int NT){
```

```
...  
    gsum += local_sum
```

```
}
```

- What happened?
- There is a conflict when updating global_sum: a *data race*



Data Race

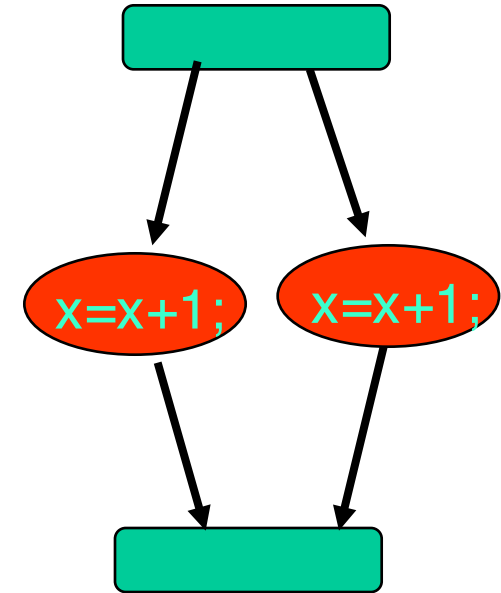
- Consider the following thread function, where x is shared and initially 0

```
void threadFn(int TID) {  
    x++;  
}
```

- Let's run on 2 threads
- What is the value of x after both threads have joined?
- A *data race* arises because the timing of accesses to shared data can affect the outcome
- We say we have a *non-deterministic* computation
- This is true because we have a *side effect* (changes to global variables, I/O and random number generators)
- Normally, if we repeat a computation using the same inputs we expect to obtain the same results

Under the hood of a race condition

- Assume x is initially 0
 $x = x + 1$;
- Generated assembly code
 - $r1 \leftarrow (x)$
 - $r1 \leftarrow r1 + \#1$
 - $r1 \rightarrow (x)$
- Possible interleaving with two threads



P1

$r1 \leftarrow x$

$r1 \leftarrow r1 + \#1$

$x \leftarrow r1$

P2

$r1 \leftarrow x$

$r1 \leftarrow r1 + \#1$

$x \leftarrow r1$

*$r1(P1)$ gets 0
 $r2(P2)$ also gets 0
 $r1(P1)$ set to 1
 $r1(P1)$ set to 1
 $P1$ writes its $R1$
 $P2$ writes its $R1$*

Avoiding the data race in summation

- Perform the global summation in `main()`
- After a thread joins, add its contribution to the global sum, one thread at a time
- We need to wrap `std::ref()` around reference arguments, `int64_t &`, compiler needs a hint*

```
void
sum(int TID, int N, int NT
    int64_t& localSum ){
....
for (int i=i0; i<i1; i++)
    localSum += x[i];
}

int64_t global_sum;
...
int64_t *locSums = new int64_t[NT];
for(int t=0; t<NT; t++)
    thrds[t] = thread(sum,t,N,NT,ref(locSums[t]));
for(int t=0; t<NT; t++){
    thrds[t].join();
    global_sum += locSums[t];
}
```

* Williams, pp. 23-4

Next time

- Avoiding data races
- Passing arguments by reference
- The Mandelbrot set computation