Lecture 3

Synchronization
Performance measurement

Announcements

- Partners?
- Office hours
  - Mondays 2.15p to 3.15p
  - Thursdays 4p to 5p

Today's lecture

- Synchronization
  - Mutual Exclusion
  - Barriers
- Performance characterization and measurement

Race conditions

- Consider the following thread procedure, where x is a shared variable initialized to 0

```
void *threadFn(void *arg) {
    x++;
    pthread_exit(NULL);  return 0;
}
```

- What is the value of x after all threads have joined?
- For 2 threads, there are 2 possibilities
- A *race condition* arises because the timing of accesses to shared memory can affect the outcome
- We say we have a *non-deterministic* computation
- Normally, if we repeat a computation using the same inputs we expect to obtain the same results
- This is true because we have a *side effect* (global variables, I/O and random number generators)

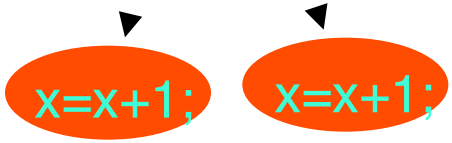
Under the hood of a race condition

- Consider this statement, assume x == 0

  ```
  x=x+1;
  ```

- Generated code
  - r1 ← (x)
  - r1 ← r1 + #1
  - r1 → (x)





- Possible interleaving with two threads

P1	P2	
r1 ← x		*r1(P1) gets 0*
	r1 ← x	*r2(P2) also gets 0*
r1 ← r1+ #1		*r1(P1) set to 1*
	r1 ← r1+#1	*r1(P1) set to 1*
x ← r1		*P1 writes its R1*
	x ← r1	*P2 writes its R1*



Avoiding race conditions

- Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness
- We need to take steps to avoid race conditions through appropriate program synchronization
  - Critical sections
  - Barriers
  - Atomic functions

Critical Sections

- Each process samples and increments the shared variable *x*
- The code performing the operation is called a *critical section*
- We use *mutual exclusion* to implement a critical section
- A critical section is non-parallelizing computation.. sensible guidelines?

Begin Critical Section

x++;

End Critical Section

Mutual exclusion

- Pthreads provides mutex variables (locks)
- May be CLEAR or SET
- Lock() waits if the lock is set, else sets the lock
- Unlock() clears the lock if set

```
pthread_mutex_t mutex_sum;
pthread_mutex_init(&mutex_sum, NULL));
pthread_mutex_lock (&mutex_sum);
        x++ ;                    // Critical Section
pthread_mutex_unlock (&mutex_sum);
```

Computing a sum in parallel

- See $PUB/Examples/ExThreads/summ.cpp

Globals

```
pthread_mutex_t mutex_sum;
int64_t  sum, *x[ ] = …;
int N, NT;
```

Main

```
for (i=0; i < N; i++) x[i] = i+1;
global_sum = 0;
pthread_t *thrd = new pthread_t[NT];
for(int t=0;t<NT;t++)
    pthread_create(&thrd[t], NULL, summ, reinterpret_cast<void *>(t)));
    //Join threads…
 cout << "The sum of 1 to " << N << " is: " << sum << endl;
```

The computation

```
void *summ(void *arg){
    int64_t _tid = reinterpret_cast<int64_t>(arg);
    int TID = _tid;
    int  i0 = TID*(N/NT),  i1   = i0 + (N/NT);
    for ( i=i0;  i<i1;  i++)
        sum +=  x[i] ;
    pthread_exit(NULL);   return NULL;
}
```

```
% summ 2 134217728  // 128M on 2 threads
The sum of 1 to 134217728 is: 454881302491991 1
Result verified to be INCORRECT, should be 9007199321849856
Run took 1798.06 milliseconds
```

With synchronization

Globals

```
pthread_mutex_t mutex_sum;
```

make clean; make critical=1

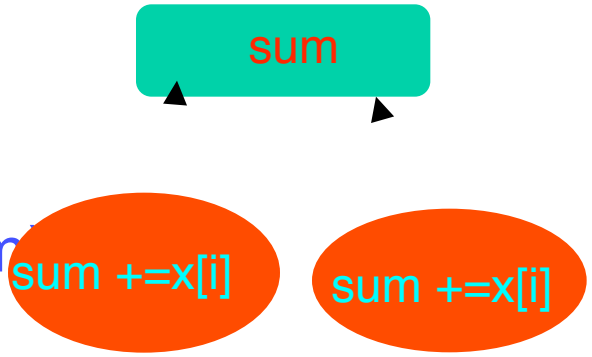
Main

```
…..
assert(!pthread_mutex_init(&mutex_sum, NULL));
…..
for(int t=0;t<NT;t++)
   pthread_create(&thrd[t], NULL, summ, NULL);

    //Join threads…
cout << "The sum of 1 to " << N << " is: " << sum << endl;
```

The computation with a critical section

```
void *summ(void *arg){
   …
   for ( i=i0;  i<i1;  i++){
        pthread_mutex_lock (&mutex_sum);
        sum +=  x[i] ;
        pthread_mutex_unlock (&mutex_sum);
}
   pthread_exit((void*) 0); return 0;
}
```




```
% summ 2 134217728  // 128M on 2 threads
The sum of 1 to 134217728 is: 9007199321849856
Run took 43215.4 milliseconds
Result verified to be CORRECT.
```

Fixing the performance bug

```
void *summ(void *arg){
   int mysum =  0;
   for ( i=i0;  i<i1;  i++){
         mysum +=  x[i] ;
   }
   pthread_mutex_lock (&mutex_sum);
   global_sum += mysum;
   pthread_mutex_unlock (&mutex_sum);
   pthread_exit((void*) 0); return 0;
}
```

```
% summ 2 134217728  // 128M on 2 threads
The sum of 1 to 134217728 is: 9007199321849856
Run took 1830.75 milliseconds
Result verified to be CORRECT.
```

Implementation issues

- The program won't always fail!

```
% summ 2 16384    // 16K on 2 threads
The sum of 1 to 16384 is: 134225920
Run took 0.194073 milliseconds
Result verified to be CORRECT.
```

- Scheduling issues
  - Busy waiting or spinning
  - Pre-emption by scheduler forces thread to *yield*
- Hardware support
  - Test and set: atomically test a memory location and then set it
  - Cache coherence protocol provides synchronization

More on Correctness

```
Int64_t   sum = 0;       // Global
void *sumIt(void *arg){
   int TID =  unique thread ID (arg);
   pthread_mutex_lock (&mutex_sum);
   sum += (TID+1);
   pthread_mutex_unlock (&mutex_sum);
   if (TID == 0)
       cout << "Sum of 1 : " << NT << " = " << sum << endl;
   pthread_exit(NULL); return NULL; }
```

```
% ./summ2 5
# threads: 5
The sum of 1 to 5 is 10
After join returns, the sum of 1 to 5 is: 15
```

Barrier synchronization

- Why was the sum reported incorrectly ?
- Don't read a location updated by other threads that had not had the chance to produce its contribution (true dependence)
- Don't overwrite the values used by other processes in the current iteration until they have been consumed (anti-dependence)
- No thread can move past a barrier until all have arrived

```
pthread_mutex_lock (&mutex_sum);
sum += 2*(TID+1);
pthread_mutex_unlock (&mutex_sum);
Barrier();
if (TID == 0)
    cout << "Total sum is " << sum << endl;
```

Building a linear time barrier with locks

```
Mutex arrival=UNLOCKED, departure=LOCKED;
 int count=0;

 void Barrier( )
    arrival.lock( );                    // atomically count the
    count++;                            // waiting threads
    if (count < $NT)  arrival.unlock( );
    else departure.unlock( );           // last  processor
                                        // enables all to go
     departure.lock( );
    count--;                            // atomically decrement
    if (count > 0) departure.unlock( );
    else arrival.unlock( );             //  last processor resets state
```

Performance metrics

Measures of Performance

- Why do we measure performance?
- How do we report it?
  - Completion time
  - Processor time product

    Completion time × # 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* $E_P = S_P/P$

Performance Anomalies

- *Super-linear* speedup: $S_P > P$
- Is it real?
- A better serial algorithm may be lurking

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

Superlinear speedup

- We have a *super-linear* speedup when

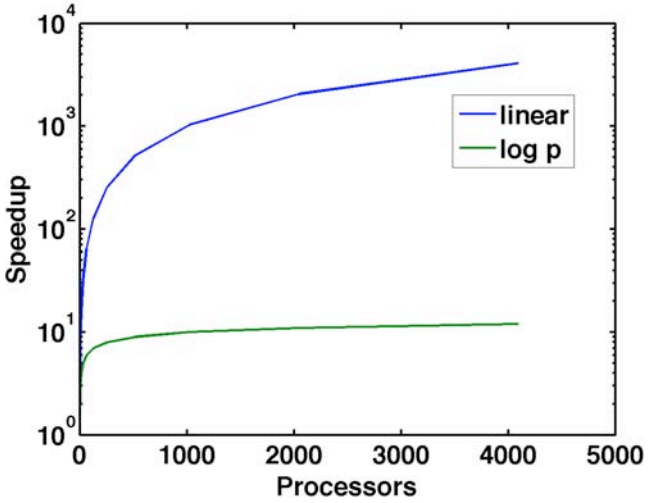
$$E_P > 1 \Rightarrow S_P > P$$

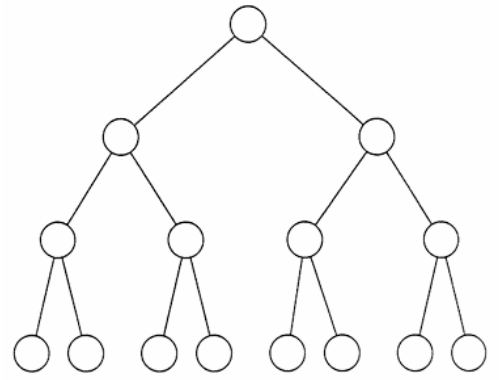
  - Super-linear speedups are often an artifact of inappropriate measurement technique
  - Where there is a super-linear speedup, a better serial algorithm may be lurking

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. sycnhronization
  - 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]
```

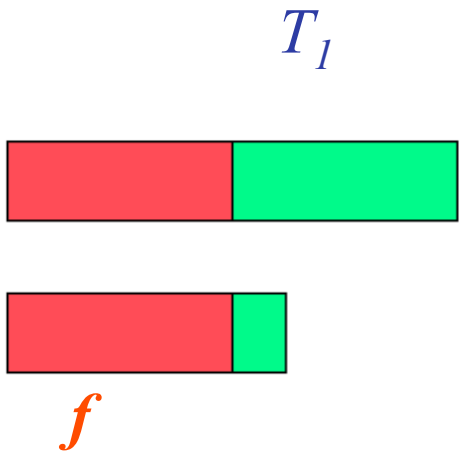




Serial Section

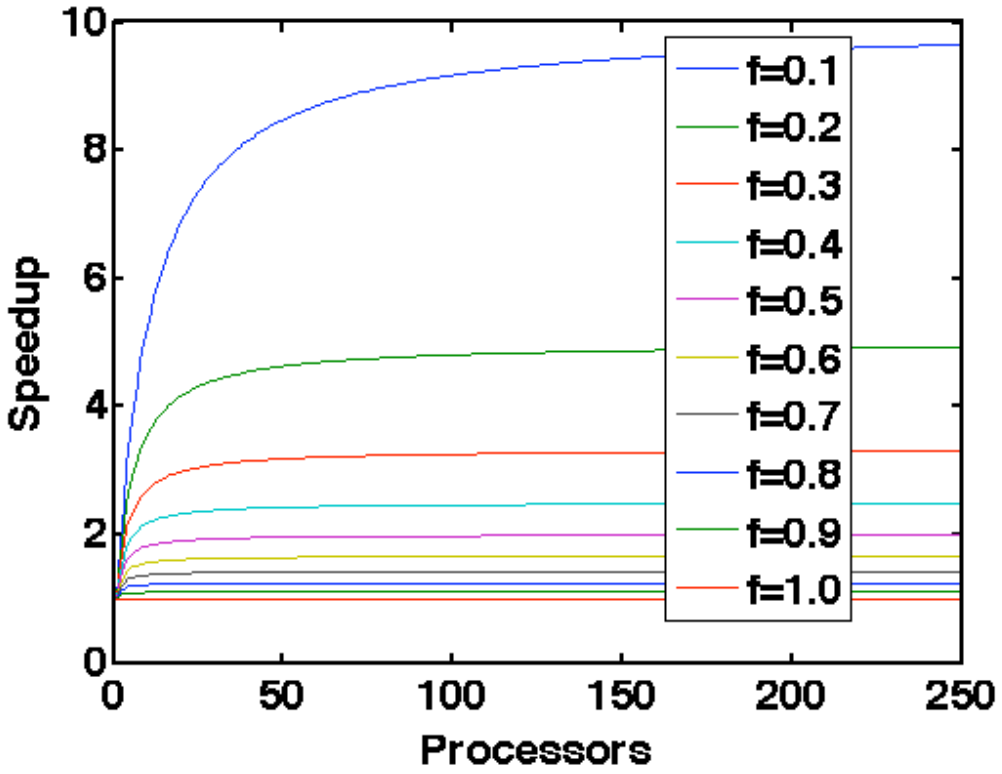
- Limits scalability
- Let f = the fraction of T_1 that runs serially
- $T_1 = f \times T_1 + (1\text{-}f) \times T_1$
- $T_P = f \times T_1 + (1\text{-}f) \times T_1 / P$

  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 $P \rightarrow \infty$, $S_P \rightarrow 1/f$



0.1

0.2

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]
    www.scl.ameslab.gov/Publications/Gus/FixedTime/FixedTime.pdf

Isoefficiency

- Consequence of Gustafson's observation is that we increase N with P
- Kumar: 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
- Problem: the amount of memory per core is shrinking

Measuring performance

Challenges to measuring performance

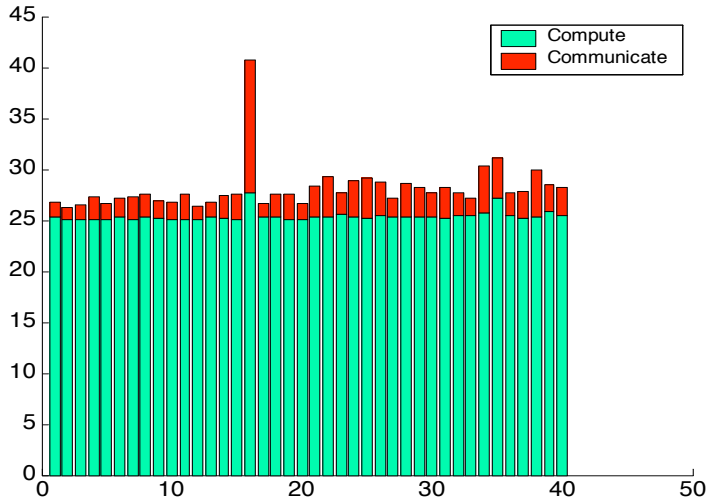
- Reproducibility
  - Transient system operating conditions
  - Differing systems or program configuration
- Measurements are imprecise
  - "Heisenberg uncertainty principle:" measurement technique may affect performance
  - Overheads and inaccuracy
- Explain anomalous behavior, but ignore anomalies that are not significant

Complications

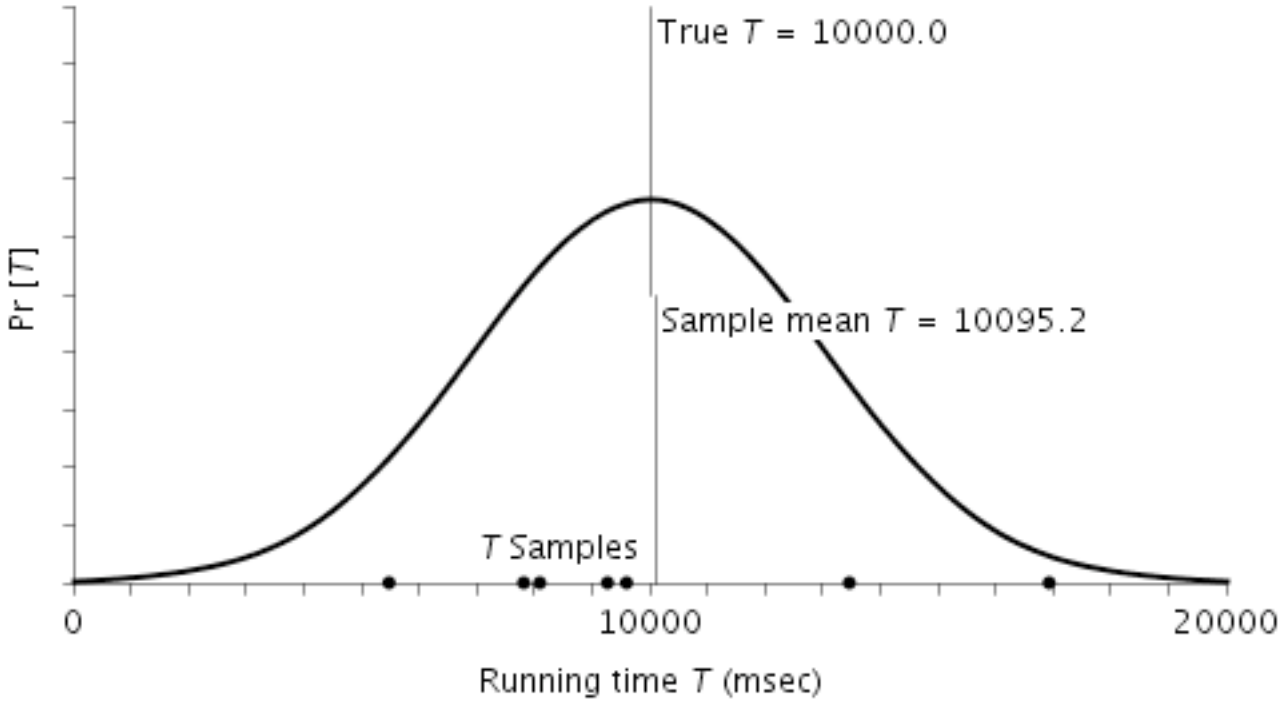
- Cost of measuring a full run is prohibitive
  - Ignore startup code if you plan to run for a much longer time in production
- Transient behavior
  - Repeat your measurements
  - "Warm up" the code before collecting measurements
  - Ignore outliers unless their behavior is important to you
  - Average time, maximum time, minimum time?

Measurement collection

- Report the *best* timings
  - Repeat results ×3 to 5 until at least 2 measures agree to within… 5%, 10%
  - Report the minimum time
- Also report outliers
- A scatter plot or error bar can be useful

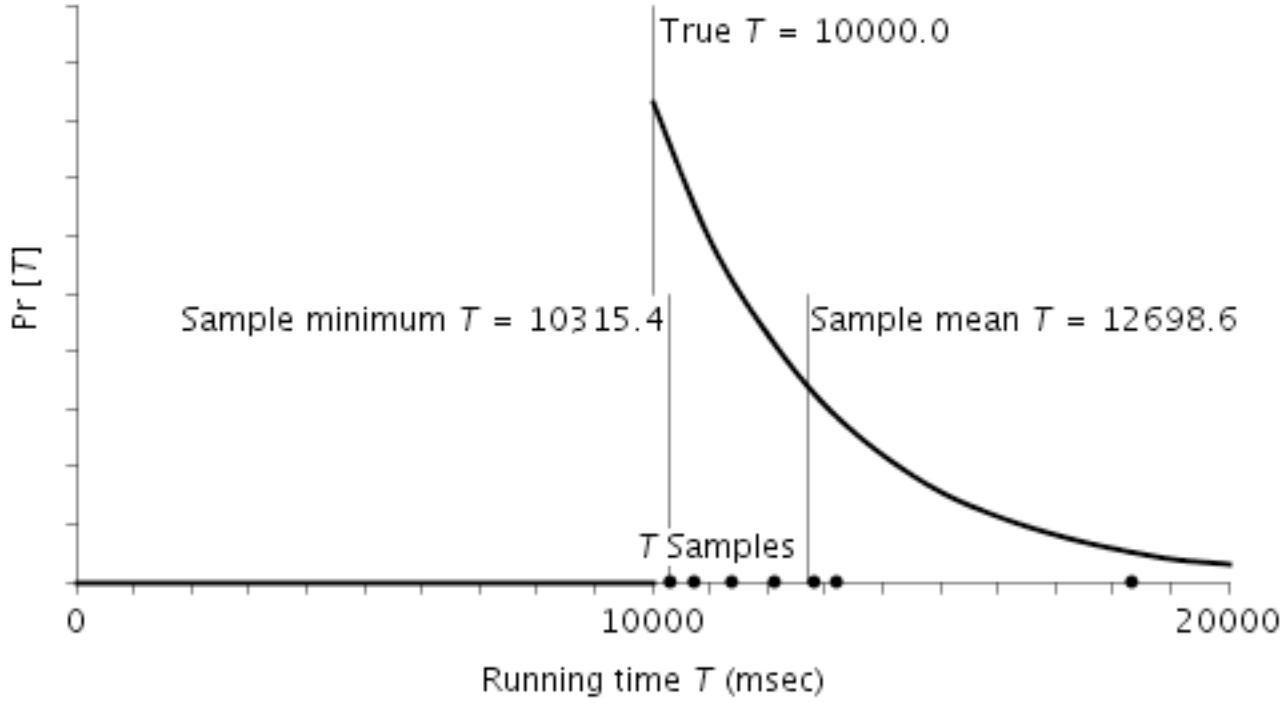


Why do we take the minimum time?



Alan Kaminsky. *Building Parallel Programs: SMPs, Clusters, and Java.* Copyright © 2010 Course Technology

Measurement errors are not distributed symmetrically



Alan Kaminsky. *Building Parallel Programs: SMPs, Clusters, and Java.* Copyright © 2010 Course Technology

Timing collection

- Measures of time
  - Elapsed, or “wall clock” time
  - CPU time = system + user time
  - Overhead, resolution, and quantization effects
- Measurement tools
  - Cam be platform dependent, especially library routines
  - Unix time command does a reasonable job for long-running programs
  - gettimeofday()

Enable others to reproduce your results

- Builds confidence within a community
- Report where you ran, software versions, processor, etc.
  - uname -a
    - Linux ieng6-203.ucsd.edu 2.6.18-194.26.1.el5PAE #1 SMP Tue Nov 9 13:34:42 EST 2010 i686 i686 i386 GNU/Linux
  - gcc --version

    gcc version 4.1.2 20080704 (Red Hat 4.1.2-48)

Fin