CSE 160
Lecture 4

Synchronization in applications
Performance Characterization

Scott B. Baden

Announcements

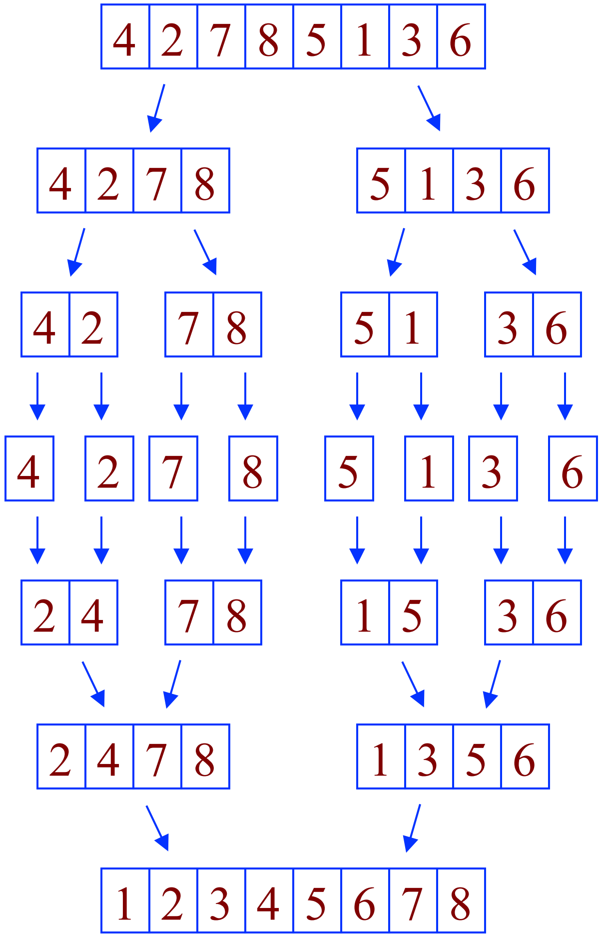
- Quiz will be held in section on Wednesday

Today's lecture

- Revisiting Parallel Merge
- More on synchronization
- Performance Characterization

Recall merge sort

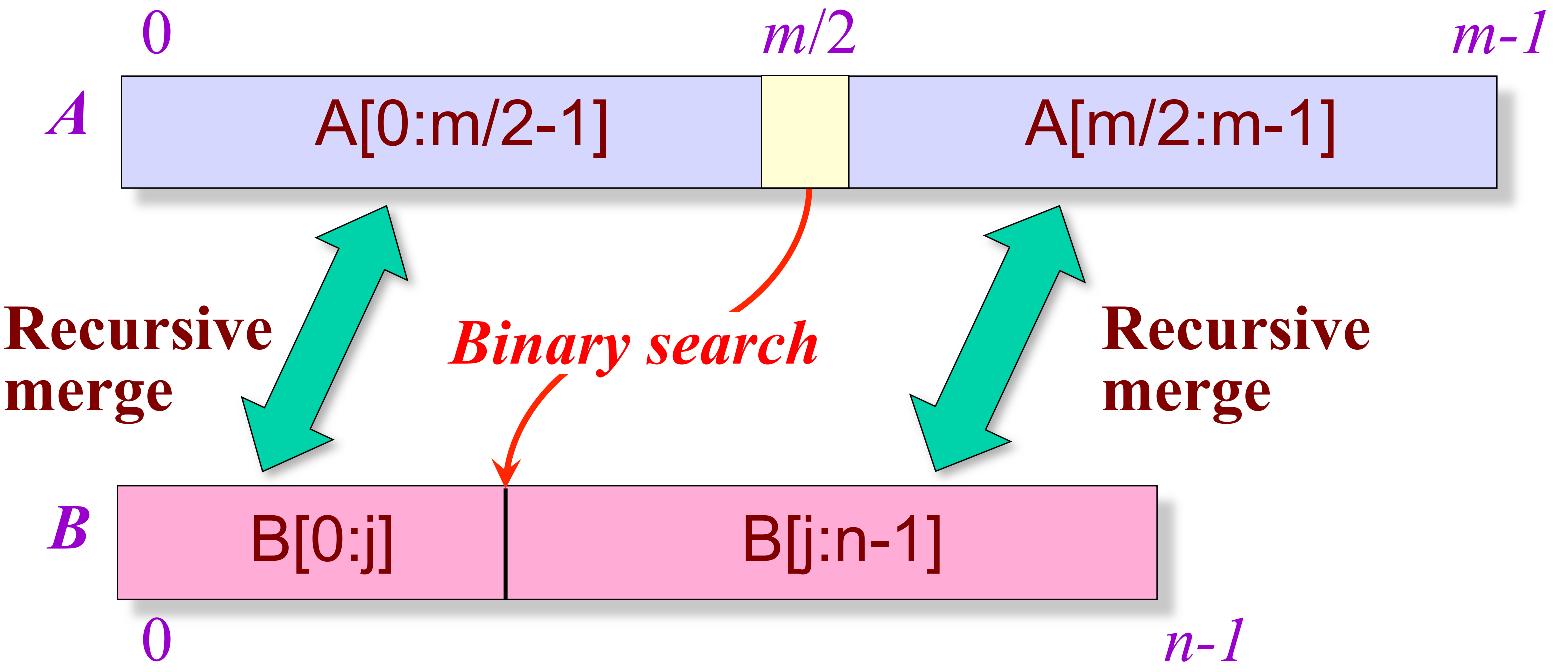
- Divide and conquer algorithm
- Running time O(N lg N)
- Traditional algorithm uses sequential merge, running in time O(m+n), 2 vectors of size m & n
- We can partition the merges into smaller ones to reduce the running time



Dan Harvey, S. Oregon Univ.

Parallel Merge Strategy

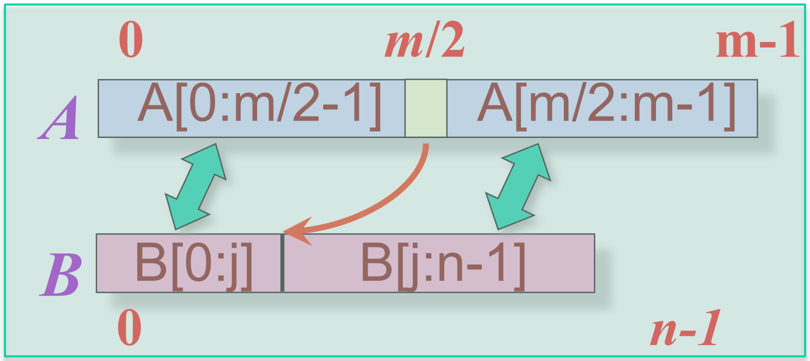
- We saw that if there are N = m+n elements, then the larger of the recursive merges processes ¾N elements
- Parallelism of merge sort, serial merge : $\Theta(\lg n)$
- Parallelism of parallel merge $\Theta(n/\lg^2 n)$



Charles Leiserson

Parallel Merge Algorithm

```
void P_Merge(int *C, int *A, int *B, int m, int n) {
  if (m < n) {
      P_Merge(C,B,A,n,m);
  } else if we can't recurse) {
    Serial(Merge)
    }
  } else {
    int m2 = m/2;
    int j = BinarySearch(A[m2], B, n);
    … "thread"(P_Merge,C, A, B, m2, j));
    … thread(P_Merge,C+m2+j, A+m2, B+j, m-m2, nb-j);
  }
}
```



Today's lecture

- Revisiting Parallel Merge
- More on synchronization
- Performance Characterization

Compare and exchange sorts

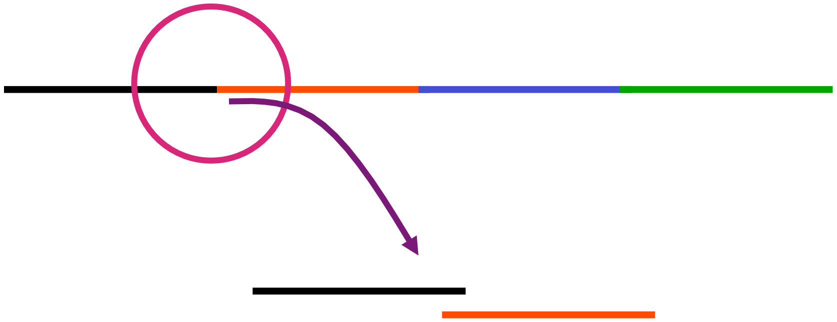
- Simplest sort, based on the bubble sort algorithm
- The fundamental operation is compare-exchange
- Compare-exchange(a[j] , a[j+1])
  - Swaps arguments if they are in decreasing order: (7,4) → (4, 7)
  - Satisfies the post-condition that $a[j] \leq a[j+1]$
  - Returns false if a swap was made

```
for i = 1 to  N-1 do
    done = true;
    for j = 0 to i-1 do // Compare-exchange(a[j] , a[j+1])
        if (a[i] < a[j])  { a[i] ↔ a[j];
                                done=false; }
    end do
    if (done) break;
end do
```

Loop carried dependencies

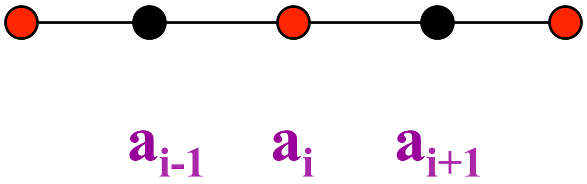
- We cannot parallelize bubble sort owing to the *loop carried dependence* in the inner loop
- The value of a[j] computed in iteration j *depends* on the a[i] computed in iterations 0, 1, …, j-1

```
for i = 1 to N-1 do
    done = true;
    for j = 0 to i-1 do
        done = Compare-exchange(a[j] , a[j+1])
    end do
    if (done) break;
end do
```

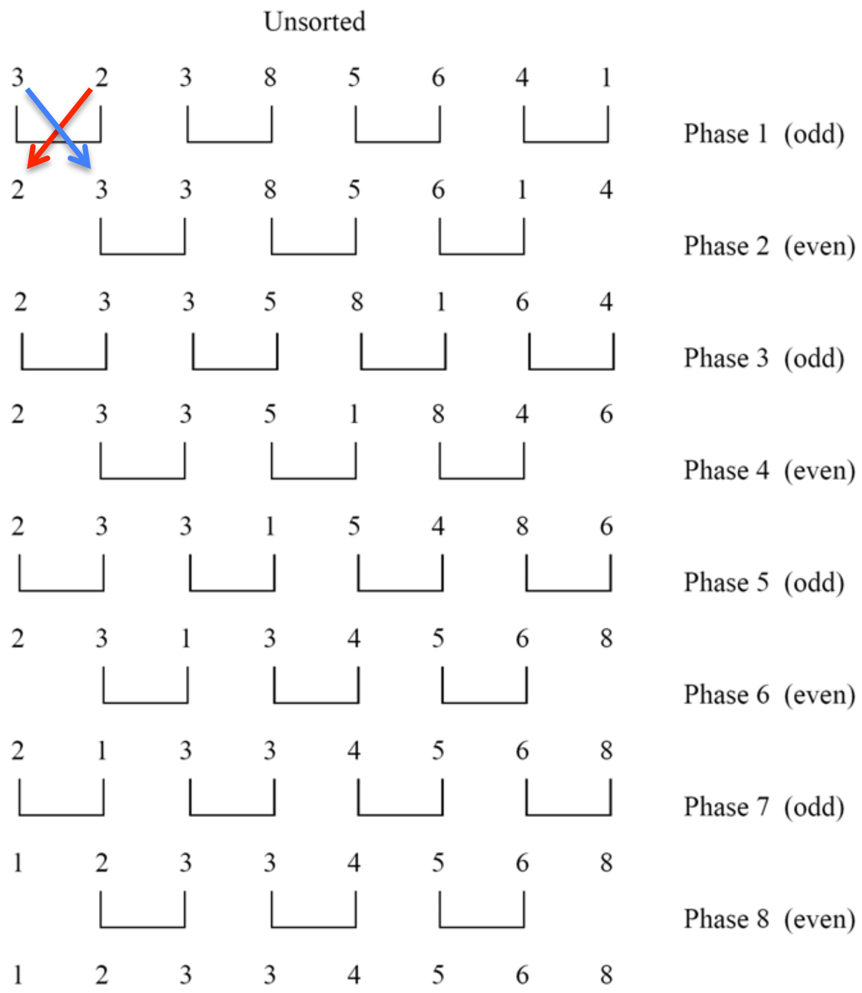


Odd/Even sort

- If we re-order the comparisons we can parallelize the algorithm
  - number the points as even and odd
  - alternate between sorting the odd and even points
- This algorithm parallelizes since there are no loop carried dependences
- All the odd (even) points are decoupled



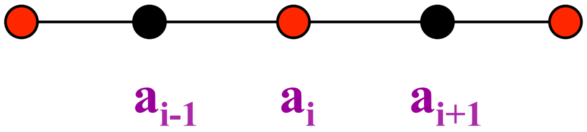
Odd/Even sort in action



Introduction to Parallel Computing, Grama et al, 2nd Ed.

The algorithm

```
for i = 0 to N-1 do
    done = true;
    for j = 0 to N-2 by 2 do   // Even
        done &= Compare-exchange(a[j] , a[j+1]);
    end do

    for j = 1 to N-2 by 2 do // Odd
        done &= Compare-exchange(a[j] , a[j+1]);
    end do
    if (done) break;

end do
```



```
// Bubble sort
for i = 1 to N-1 do
    done = true;
    for j = 0 to i-1 do
        done = Compare-Exchange(a[j] , a[j+1])
    end do
    if (done) break;
end do
```

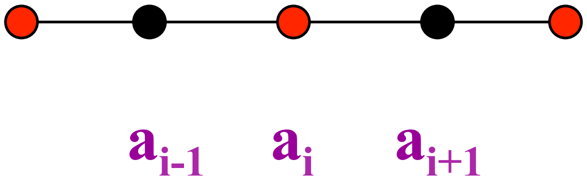
Odd/Even Sort Code

- Where do we need synchronization?

```
Global bool AllDone;
int OE = lo % 2;
for (s = 0; s < MaxIter; s++) {
     int done = Sweep(Keys, OE, lo, hi);    /* Odd phase */
     done &=  Sweep(Keys, 1-OE, lo, hi);  /* Even phase */

    AllDone &= done;

      if (AllDone)
          break;

} // End For
```

```
bool Sweep(int *Keys, int OE, int lo, int hi){
int Hi = hi;
if (TID == (NT-1)
   Hi – –;
bool myDone = true;
for (int i = OE+lo; i <= Hi; i+=2) {
      if (Keys[i] > Keys[i+1]){
         Keys[i] ↔ Keys[i+1];
         myDone = false;
      }
    }
   return myDone ;
}
```

Odd/Even Sort Code – with synchronization



```
Global bool AllDone;
int OE = lo % 2;
for (s = 0; s < MaxIter; s++) {
    barr.sync();
    if (!TID)
       AllDone = true;
    barr.sync();

    int done = Sweep(Keys, OE, lo, hi);    /* Odd phase */
    barr.sync();
    done &= Sweep(Keys, 1-OE, lo, hi);  /* Even phase */
    mtx.lock();
    AllDone &= done;
    mtx.lock();
    barr.sync();

    if (allDone)
        break;
}
```

Today's lecture

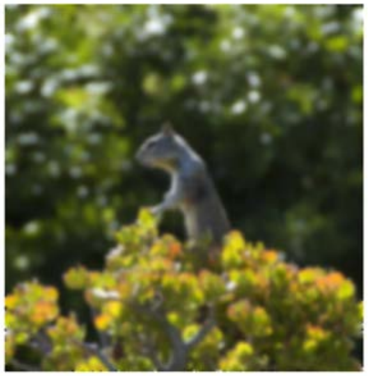
- Revisiting Parallel Merge
- More on synchronization
- Performance Characterization

Image smoothing algorithm

- Repeat as many times as necessary

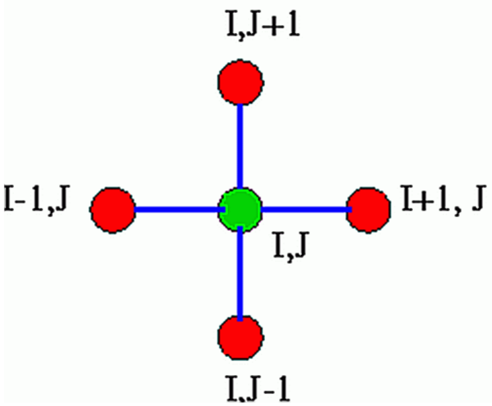
```
for (i,j) in 0:N-1 x 0:N-1
  $I^{new}[i,j] = ( I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i, j+1])/4$
$I = I^{new}$
```

Original



100 iter

1000 iter



Multithreaded Smoother()

```
Global Change, I[:,:], $I^{new}$[:,:]
Local mymin = 1 + ($TID * n/$NT),
      mymax = mymin + n/$NT-1;
Local  done = FALSE;

while (!done) do
    Local myChange = 0;
    Change= 0;
    update $I^{new}$ and myChange
    Change+= myChange;
    if (Change< Tolerance) done = TRUE;
    Swap pointers:  I ↔ $I^{new}$
end while
```

```
update $I^{new}$ and myChange:
for i = mymin to mymax do
    for j = 1 to n do
        $I^{new}$[i,j] = …
        myChange+= ($I^{new}$[i,j]-I[i,j]) $^2$
    end for
end for
```

Is this code correct?

Correctness

```
Global Change, I[:,:], $I^{new}$[:,:]
Local mymin = 1 + ($TID * n/$NT),
        mymax = mymin + n/$NT-1;
Local  done = FALSE;
while (!done) do
     Local myChange= 0;
     BARRIER
     Only on thread 0:  Change= 0;    // PRODUCE
     BARRIER
     update $I^{new}$ and myChange
     CRITICAL SEC: Change += myChange // PRODUCE + CONSUME
     BARRIER
     if (Change< Tolerance) done = TRUE; // CONSUMER
    Only on thread 0: Swap pointers:  I ↔ $I^{new}$
end while
```

Does this code use minimal synchronization?

Building a linear time barrier with locks

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

 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
```

Today's lecture

- Revisiting Parallel Merge
- More on synchronization
- Performance Characterization

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 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 is the speedup and efficiency on 8 processors?
- What will the running time be on 4 processors?
- What is the maximum possible speedup on an infinite number of processors?
- What fraction of the total running time on 1 processor corresponds to the serial section?
- What fraction of the total running time on 2 processors corresponds to the serial section?

NT	Time
1	10000
2	6000
8	3000

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

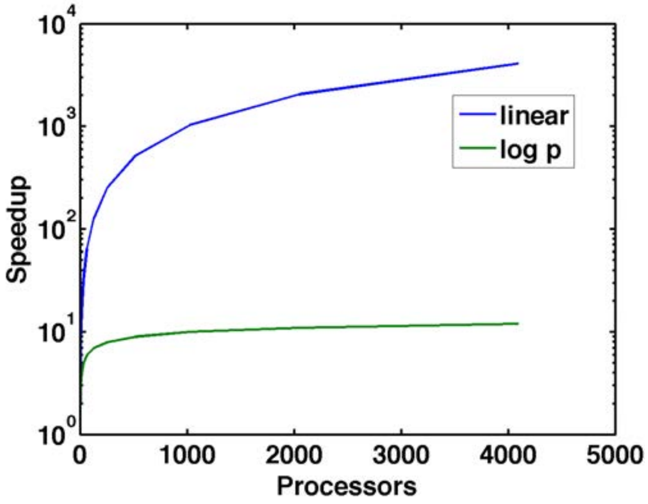
$$E_P > 1 \;\Rightarrow\; S_P > P$$

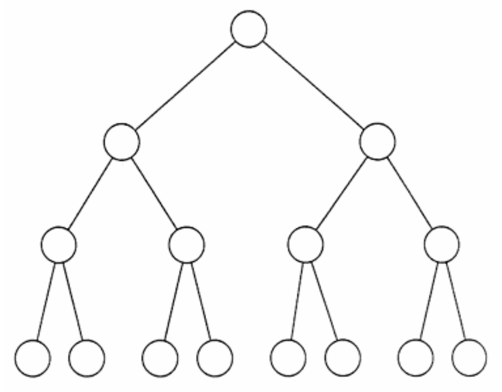
- Is it real?
  - Super-linear speedups are often an artifact of inappropriate measurement technique
  - Where there is a super-linear speedup, a better serial algorithm may be lurking

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. 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]

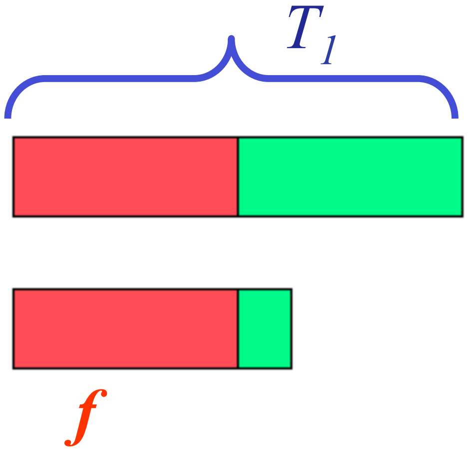




Serial Sections

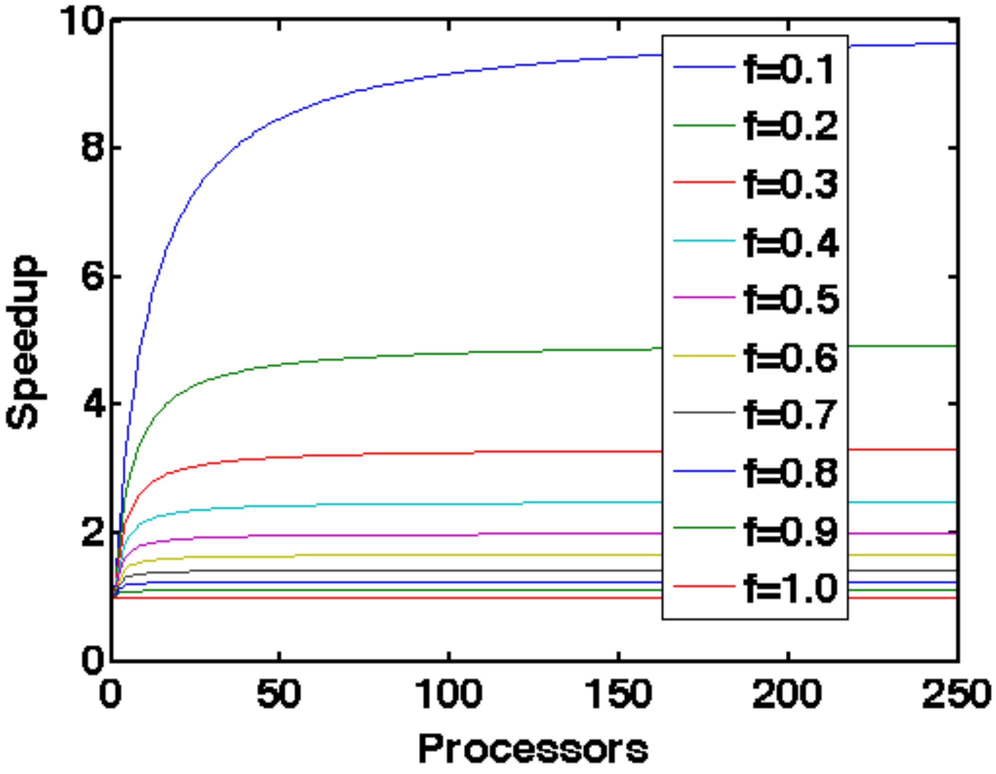
- Limit scalability
- Let f = the fraction of T_1 that runs serially
- $T_1 = f \times T_1 + (1\text{-}f) \times T_1$
- $T_P = f \times T_1 + (1\text{-}f) \times T_1 / P$

  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]

    http://en.wikipedia.org/wiki/Gustafson's_law

    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

Time constrained scaling

- Sum N numbers on P processors
- Let N >> P
- Determine the largest problem that can be solved in time $T=10^4$ time units on 512 processors
- Let time to perform one addition = 1 time unit
- Let β = time to add a value inside a critical section

Performance model

- Local additions: $N/P - 1$
- Reduction: β ($\lg P - 1$)
- Since $N >> P$

  $T(N,P) \sim (N/P\ ) + \beta\ (\lg P - 1)$
- Determine the largest problem that can be solved in time $T= 10^4$ time units on $P=512$ processors, $\beta = 1000$ time units
- Constraint: $T(512,N) \leq 10^4$

  $\Rightarrow (N/512) + 1000\ (\lg 512 - 1) = (N/512) + 1000*(8) \leq 10^4$

  $\Rightarrow\ \ N \leq 1\times10^6$ (approximately)

Challenges to measuring performance

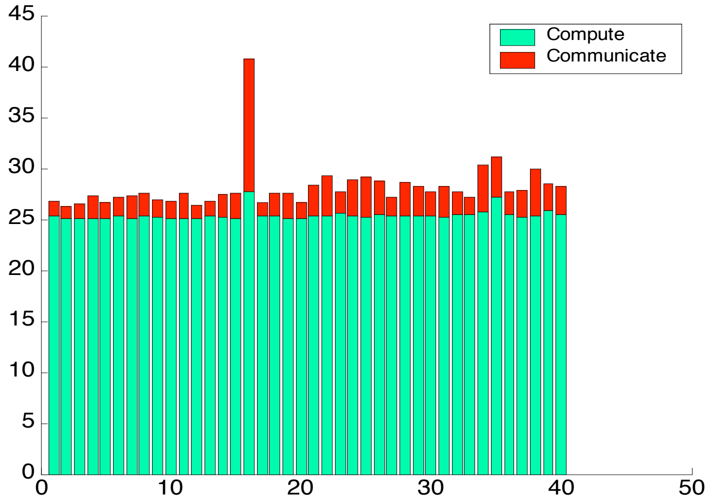
- Reproducibility
  - Transient system operating conditions
  - Differing systems or program configuration
- Measurements are imprecise
  - "Heisenberg uncertainty principle:" measurement technique may affect performance
  - Overheads and inaccuracy
- Explain anomalous behavior, but ignore anomalies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

Complications

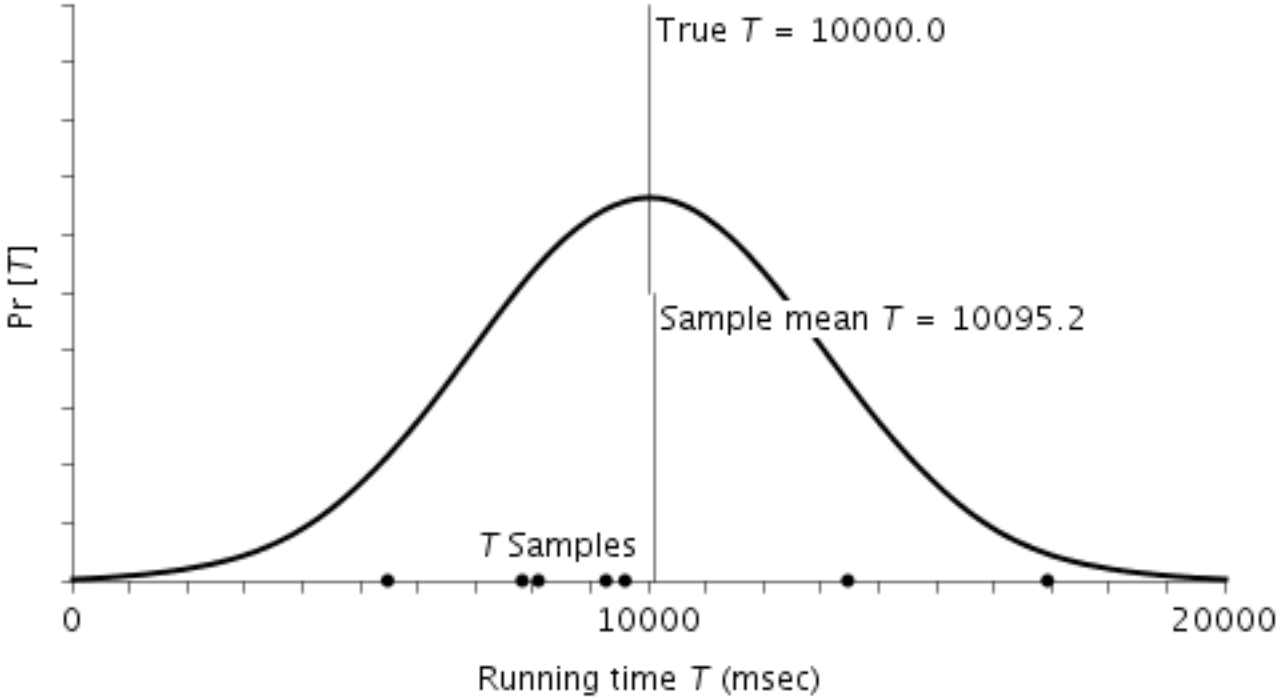
- Cost of measuring a full run is prohibitive
  - Ignore startup code if you plan to run for a much longer time in production
- Transient behavior
  - Repeat your measurements
  - "Warm up" the code before collecting measurements
  - Ignore outliers unless their behavior is important to you
  - Average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

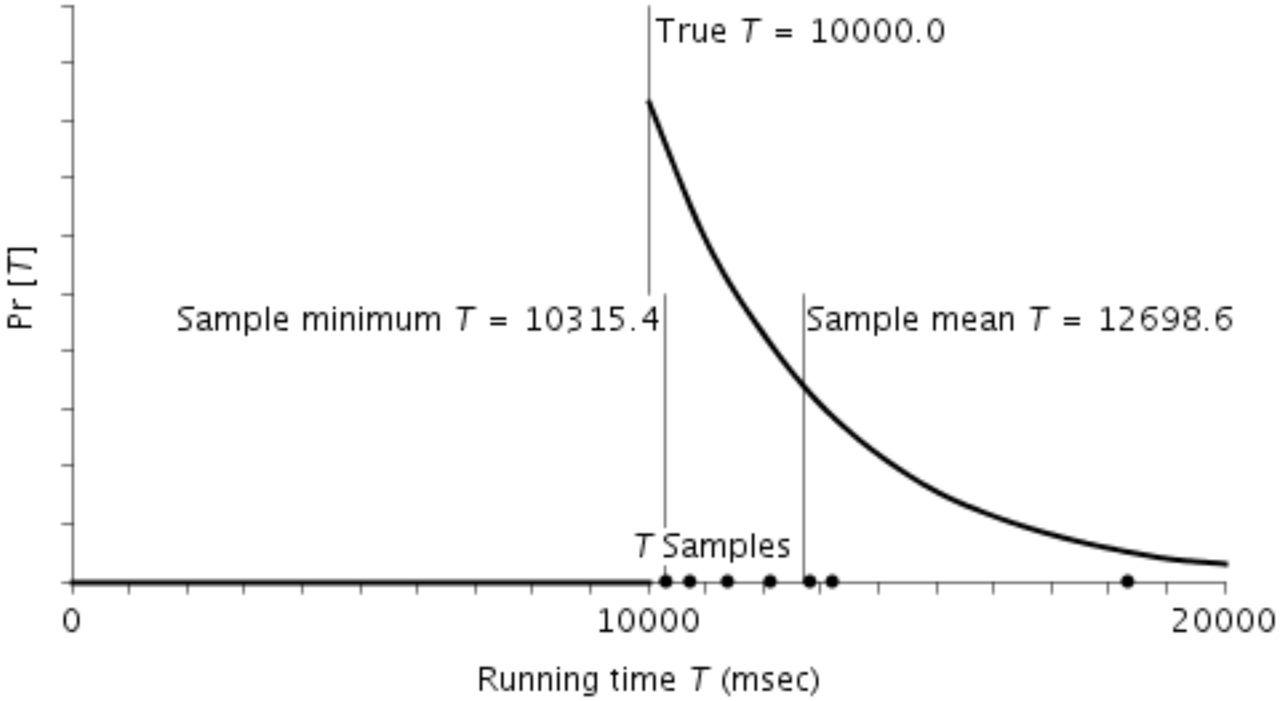


Why do we take the minimum time?



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 ccom-bang-login.local 2.6.32-358.18.1.el6.x86_64 #1 SMP Wed Aug 28
    17:19:38 UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
    ```

  - gcc –version

    ```
    gcc version 4.7.3 (GCC)
    ```

  - /proc/cpuinfo
  - /sys/devices/system/cpu/cpu0, cpu1, …. P-1

Fin