

Lecture 10

Midterm review

Announcements

- The midterm is on Tue Feb 9th in class
 - Bring photo ID
 - You may bring a single sheet of notebook sized paper “8x10 inches” with notes on both sides (A4 OK)
 - You may not bring a magnifying glass or other reading aid unless authorized by me
- Review session in section Friday
- Practice questions posted here:
<https://goo.gl/MtlUXh>
Post answers to Piazza, I will collect and edit into the review document

Practice questions Q1-4

1. What is false sharing and why can it be detrimental to performance?
2. What is a critical section and what do we use to implement it?
3. What is the consequence of Amdahl's Law and how can we overcome it?
4. We run a program on a parallel computer and observe a superlinear speedup. Explain what a superlinear speedup is, and give 1 explanation for why we are observing it

Q5-7

5. A certain parallel program completes in 10 seconds on 8 processors, and in 60 seconds on 1 processor. What is the parallel speedup and efficiency? Show your work. Be sure to show your work to get full credit
6. We take a single core program and parallelize with threads. The fraction of time that the serial code spends in code that won't parallelize is 0.2. What is the speedup on 7 processors? Be sure to show your work to get full credit.
7. Name 2 ways to synchronize a multithreaded program.

Q 8-10

8. Name the 3Cs of cache misses
9. Briefly explain the differences between shared variables, thread local variables (automatic), and ordinary local variables (i.e. within main() or any user-defined function), both in terms of where they appear in the source code, and any data races or race conditions that may arise in a multithreaded program
10. Why is memory consistency a necessary but not sufficient condition to ensure program correctness?

Worked problems

1. There are two synchronization errors in this code. Point out which lines(s) of code are involved and what is causing the errors. Do not fix the code.

There are no syntax errors

We will never intentionally introduce syntax errors

```
(1) int N_odds = 0;
(2) void Odds(std::vector<int>& x, int NT){
(3)     int N = x.size();
(4)     int i0 = $TID * N / $NT, i1 = i0 + N/$NT;
(5)     int local_N_odds = 0;
(6)     for i = i0 to i1-1
(7)         if ((x[i] % 2) == 1)
(8)             local_N_odds++;
(9)     N_odds += local_N_odds;
(10)    if ($TID==0) print N_odds;
(11)}
```

There is a synchronization error at line 9: a data race

There is also a race condition at line 10: we need to wait for everyone to update N_odds before printing out N_odds

Worked problem #2

2. What are the possible outcomes of the following program where `Mtx0` and `Mtx1` are C++ mutex variables and `X` is a global variable that has been initialized to zero? Give an interleaving of relevant statements for every possible outcome.

Thread 0	Thread 1
(1) <code>Mtx0.lock();</code>	(5) <code>Mtx1.lock();</code>
(2) <code>X++;</code>	(6) <code>X++;</code>
(3) <code>Mtx1.unlock();</code>	(7) <code>Mtx0.unlock();</code>
(4) <code>cout << "x=" << X << endl;</code>	(8) <code>cout << "x = " << X << endl;</code>

There is a data race at lines 2 and 6
`X` will either be 1 or 2 depending on the ordering of the instructions that increment `X`

Worked problem #3

3. Bang's clovertown processor has 32KB of L1 cache per core. When an integer array $a[]$ is much larger than L1, what types of L1 cache misses are likely to be the most numerous in the second loop?

```
for (i=0; i<n; i++)  
    a[i] = a[i]+2;  
for (i=0; i<n; i++)  
    a[i] = a[i]*3
```

Capacity misses

Worked problem #3

3. Bang's clovertown processor has 32KB of L1 cache per core. When an integer array $a[]$ is much larger than L1, what types of L1 cache misses are likely to be the most numerous in the second loop? $J[]$ is assumed to contain legal subscripts for $a[]$

Conflict misses

```
for (i=0; i<n; i++)
```

```
    a[i] = a[i]+2;
```

```
for (i=0; i<n; i++)
```

```
    a[i] = a[J[i]]*3
```

Worked problem #4

4. You are processing a set of strings that are N characters long, & each character is an *unsigned int* from 0 to 255.

Compute the *histogram*, a table counting the number of occurrences of each possible character appearing in the input. We run on multiple threads by giving each thread its own contiguous piece of the input: from *mymin* to *mymax*.

The program sometimes produces erroneous output. There are also 1 or more performance bugs in the program.

•

Worked problem #4 –continued

Rewrite the code to ensure that it is correct and efficient. To receive full credit, your solution must be both correct & efficient and you must demonstrate why your code design ensures both efficiency and correctness. The thread function is below.

- *Input* and *histogram* are global (shared) arrays
- The number of threads NT divides N exactly,
- All threads execute the loop is executed by all threads
- The histogram has been previously initialized to zero

```
for const int N = a large number
unsigned char input[N]; unsigned int histogram[256];
void histo_Thread(int NT){
    int mymin = $TID*(N/NT), mymax = mymin + (N/NT);
    for (int k = mymin; k < mymax; ++k)
        histogram[(int) input[k]]++;
}
```

Updates to the histogram array cause a data race, since different threads can update the same shared values simultaneously

A simple solution, to protect the update with a critical section incurs a high overhead, as lock operations are expensive. We should never put a critical section into a tight loop.

To avoid this performance “bug,” we use thread private histogram arrays and then we combine them into a single global array

Topics for Midterm

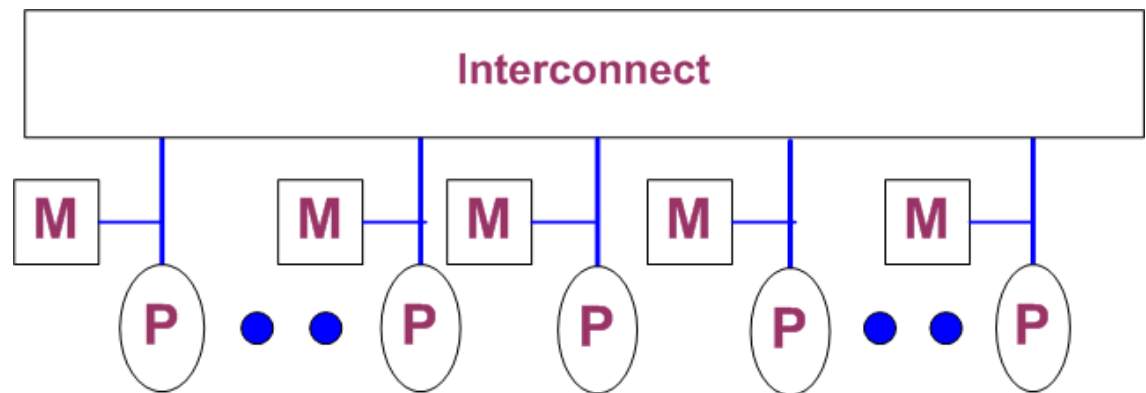
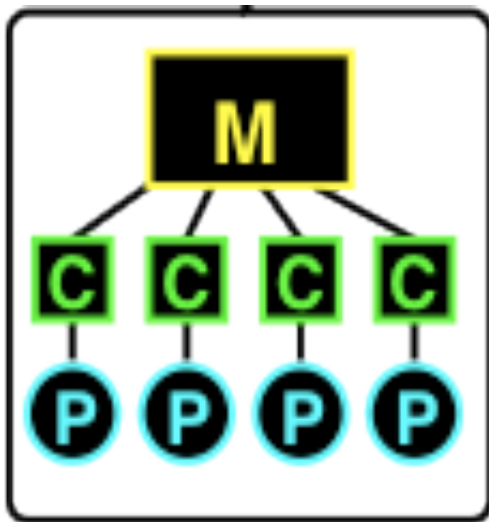
- Technology
- Threads Programming

Technology

- Processor Memory Gap
- Caches
 - ▶ Cache coherence and consistency
 - ▶ Snooping
 - ▶ False sharing
 - ▶ 3 C's of Cache Misses
- Multiprocessors: NUMAs and SMPs

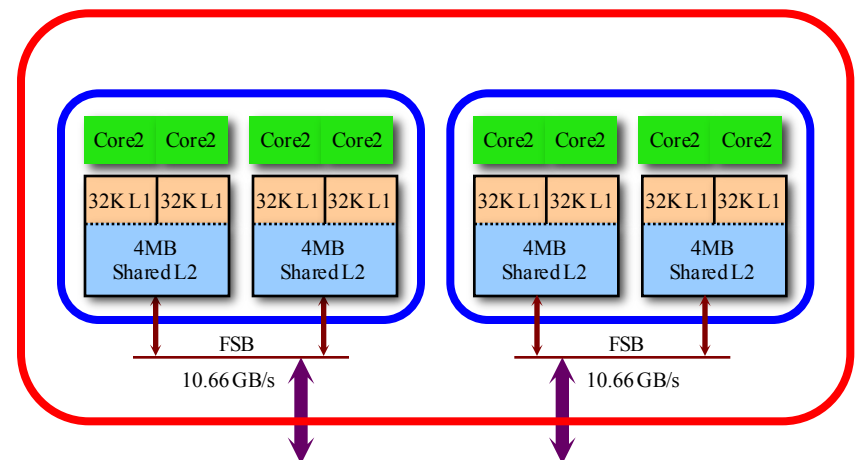
Address Space Organization

- Multiprocessors and multicomputers
- Shared memory, message passing
- With shared memory hardware automatically performs the global to local mapping using address translation mechanisms
 - **UMA:** *Uniform* Memory Access time
Also called a Symmetric Multiprocessor (SMP)
 - **NUMA:** *Non-Uniform* Memory Access time



Different types of caches

- Caches take advantage of locality by re-using instructions and data (space and time)
- Separate / unified Instruction (I) /Data (D)
- Direct mapped / Set associative
- Write Through / Write Back
- Allocate on Write / No Allocate on Write
- Last Level Cache (LLC)
- Translation Lookaside Buffer (TLB)
- Hit rate, miss penalty, etc..



What type of multiprocessor is on a Bang node?

A. NUMA

B. SMP



Which of these do we want to reduce to increase cache performance?

A. Hitrate

B. Miss penalty

C. Both

In a direct-mapped cache, how many possible lines within the cache can a line in main memory get mapped to?

A. Multiple

[This is a set associative cache]

B. Single

Memory consistency

- A memory system is consistent if the following 3 conditions hold
 - Program order (you read what you wrote)
 - Definition of a coherent view of memory (“eventually”)
 - Serialization of writes (a single frame of reference)
- Sequential and weak consistency models

Is a consistent memory system a necessary or sufficient condition for writing correct programs?

A. Necessary [otherwise, shared variables like locks could have different values on different processors]

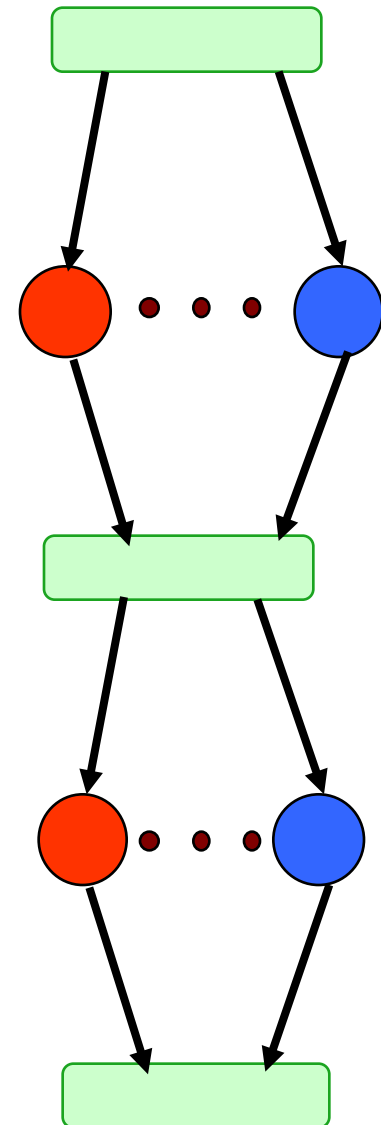
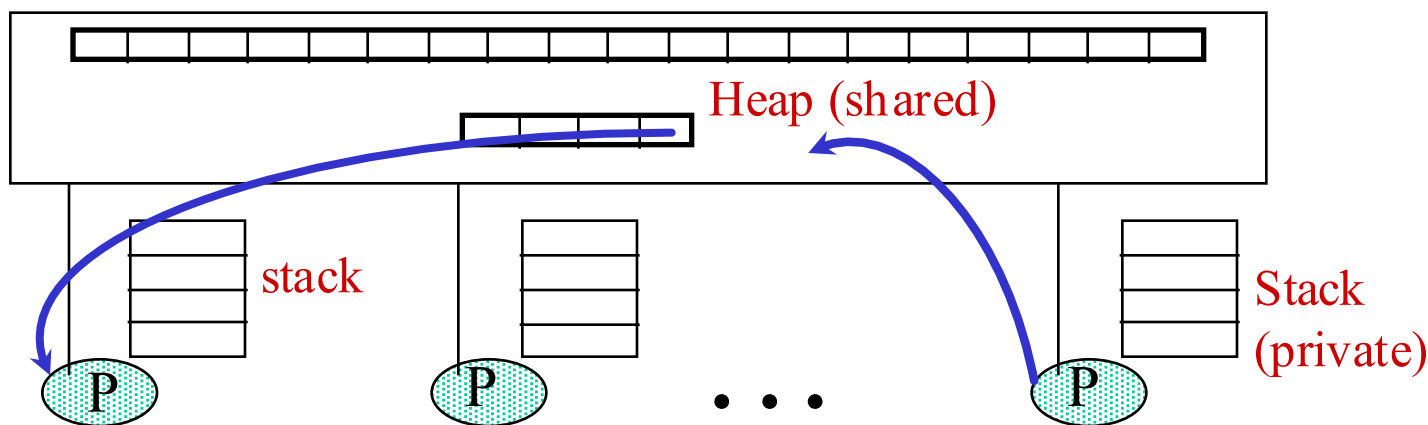
B. Sufficient

Today's lecture

- Technology
- **Threads Programming**

Threads Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads communicate via shared memory, also have private storage
- A spawned thread executes asynchronously until it completes
- Threads may or may not execute on different processors



Multithreading in perspective

- Benefits
 - Harness parallelism to improve performance
 - Ability to multitask to realize concurrency, e.g. display
- Pitfalls
 - Program complexity
 - Partitioning, synchronization, parallel control flow
 - Data dependencies
 - Shared vs. local state (globals like errno)
 - Thread-safety
 - New aspects of debugging
 - Data races
 - Race conditions
 - Deadlock
 - Livelock

Implementation & techniques

- SPMD
 - Threads API
- Correctness: critical sections, race conditions
 - Mutexes and barriers
- Performance: data partitioning
 - Block and cyclic decompositions
- Cross cutting issues (Performance & Correctness)
 - Cache coherence and consistency
 - Cache locality
- Data dependencies, loop carried dependence

RAII and Lock_guard

- The lock_guard constructor acquires (locks) the provided lock constructor argument
- When a lock_guard destructor is called, it releases (unlocks) the lock
- How can we improve this code?



```
int val;  
std::mutex valMutex;  
...
```

Call f() outside the critical section
We'll need a unique_lock for this purpose,
since it can be cleared unlike a lock_guard

```
{ std::lock_guard<std::mutex> lg(valMutex); // lock and automatically unlock  
  if (val >= 0)  
    f(val);  
  else  
    f(-val); // pass negated negative val  
}
```

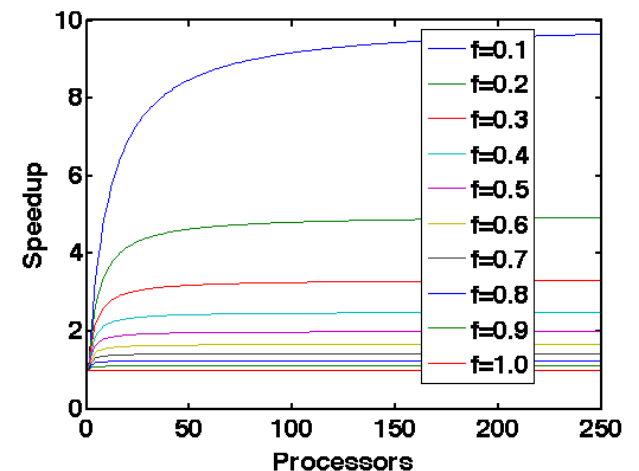
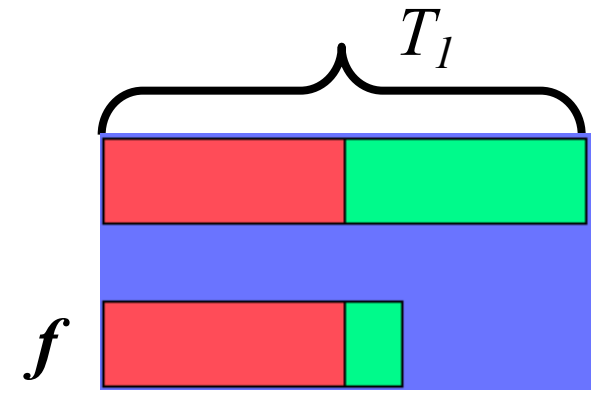
// ensure lock gets released here

Today's lecture

- Technology
- **Threads Programming**
 - Correctness
 - **Performance**

Performance Terms and concepts

- Parallel speedup and efficiency
- Super-linear speedup
- Strong scaling, weak scaling
- Amdahl's law, Gustafson's law, serial bottlenecks
- Strong scaling and weak scaling

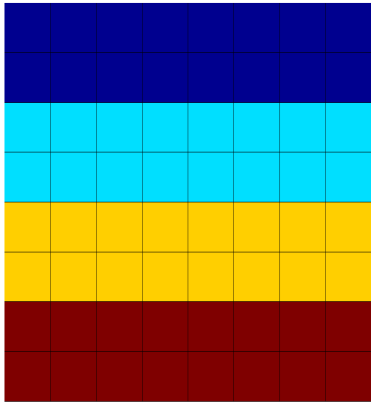


Which is strong scaling?

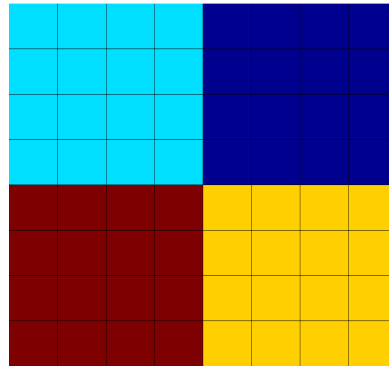
- A. Constant work/core [This is *weak scaling*]
- B. Constant work regardless of the number of cores
- C. Growing work/processor
- D. Shrinking work/core
- E. Total work is exponential in the number of cores

Workload Decomposition

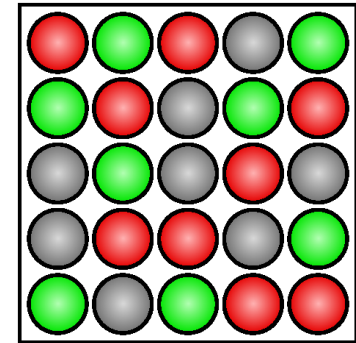
- Block vs. Cyclic
- Static vs. Dynamic Decomposition



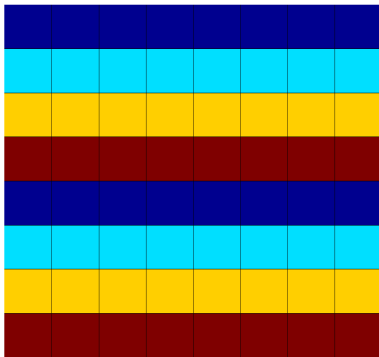
[Block, *]



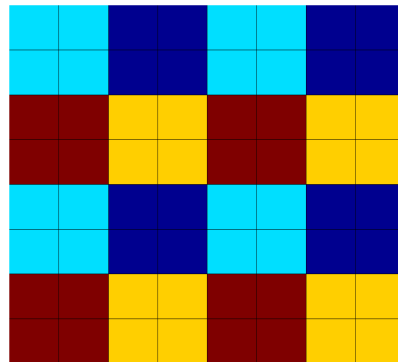
[Block, Block]



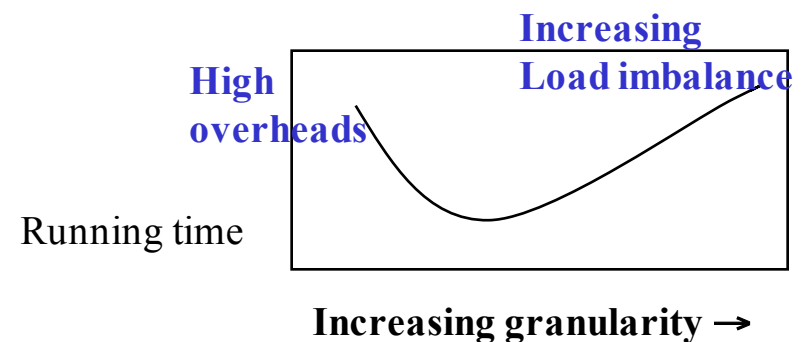
Dynamic



[Cyclic, *]



[Cyclic(2), Cyclic(2)]



Tradeoffs in choosing the chunk size

- **CHUNK=1:** each box needs data from all neighbors
 - Every processor loads all neighbor data into its cache!
 - Compare with [BLOCK,BLOCK]
- **CHUNK=2:** each box in a chunk of 4 boxes needs $\frac{1}{4}$ of the data from 3 neighboring chunks
 - Each processor loads 3 chunks of neighbor data into cache
- **CHUNK=4:** Only edge boxes in a chunk need neighbor data, 20 boxes: processor loads 1.25 chunks of neighbor data



0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3

Data parallelism

- We divide up the data, and the loops that operate on them
- Can we parallelize the loops as shown?

LOOP #1

for $j = 0$ to $n-2$
 $A[j+1] = A[j];$

LOOP #2

for $j = 1$ to $n-1$
 $A[j-1] = A[j];$

- A. Loop #1 only
- B. Loop #2 only
- C. Loop #1 and Loop #2
- D. Neither

Data parallelism



- How do we structure Loop #1 to get it to parallelize?
- How do we restructure Loop #2 to get it to parallelize?

LOOP #1

```
for j = 0 to n-2  
    A[j+1] = A[j];
```

LOOP #2

```
for j = 1 to n-1  
    A[j-1] = A[j];
```

```
for j = 1 to n-2  
    a[j] = a[0]
```

```
for j = 1 to n-2 {  
    b[j-1] = a[j];  
    a[j-1] = b[j-1];  
}
```

Correctness

- Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness
- User: avoid race conditions through appropriate program synchronization
 - ▶ Migrate shared updates out of the thread function
 - ▶ Critical sections
 - ▶ Barriers
 - ▶ Fork/Join