CSE 160
Lecture 10

MIDTERM Review

Announcements

- The Midterm: Tuesday Nov 5th in this room
  - Covers everything in course through today
  - Closed book; may bring a 8x11" sheet of paper
  - No Blue Book needed
  - Midterm review session (special section time)
    Friday Nov 1st at 4pm in CSB 002
- Midterm review today
  - Some readings relevant to the review
  - Q2 return and questions
  - **Be prepared** to ask and answer questions

- SDSC Tour on Friday at 1.15 PM

Quiz highlights

S=2 (shared)
All other variables are local (0)
What value will thread 1 assign to b?

Thread 0	Thread 1
a = S;	b = 4 * S;
S = 7;	

What are the possible outcomes of the following program ?

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

Today's lecture

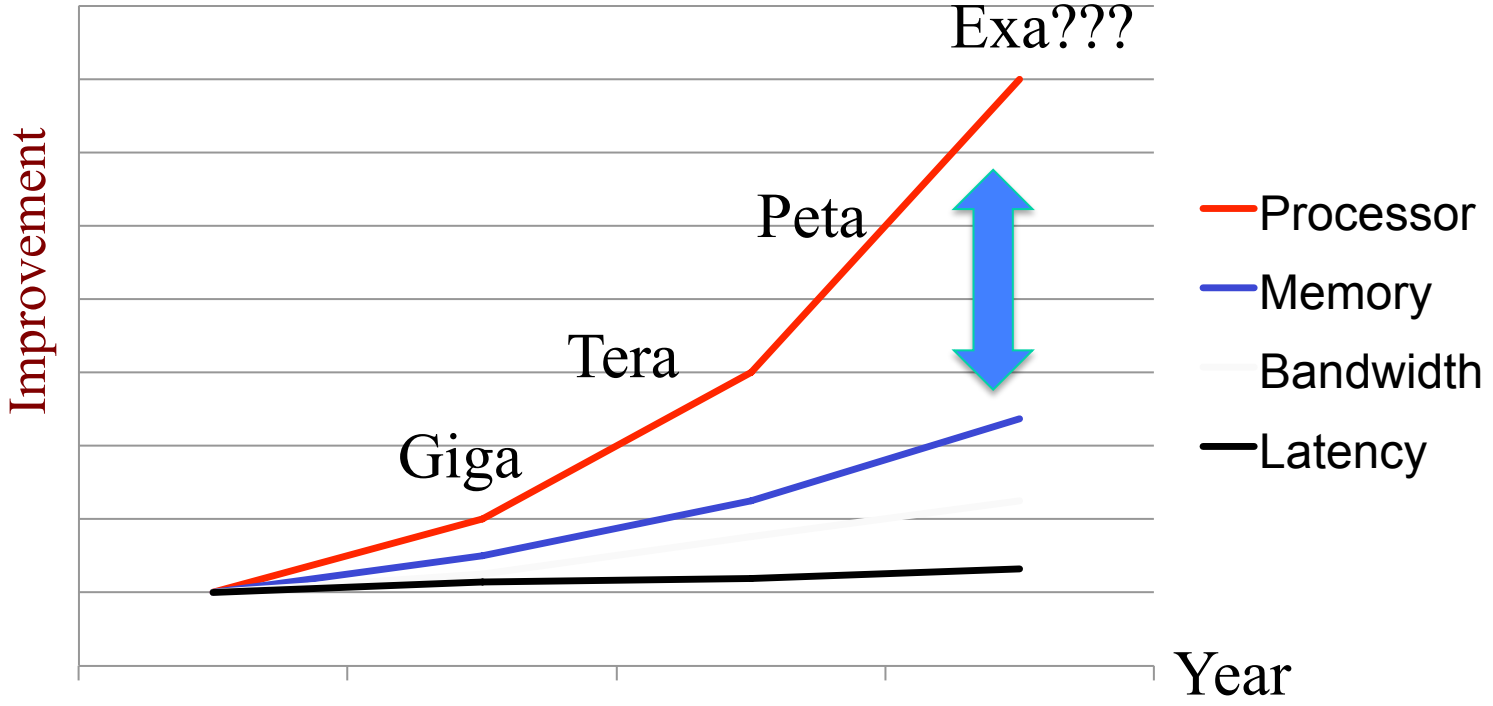
- Technology
- Threads Programming
  - Correctness
  - Performance
- Algorithms (applications)
- List of Keywords and Topics (cross cutting)

Technology: Terms and concepts

- Processor Memory Gap
- Caches (What are the 3 C's of Cache Misses?)
  - Cache coherence and consistency
  - Snooping
  - False sharing
- MIMD
- Multiprocessors: NUMAs and SMPs

Trend: data motion costs are rising

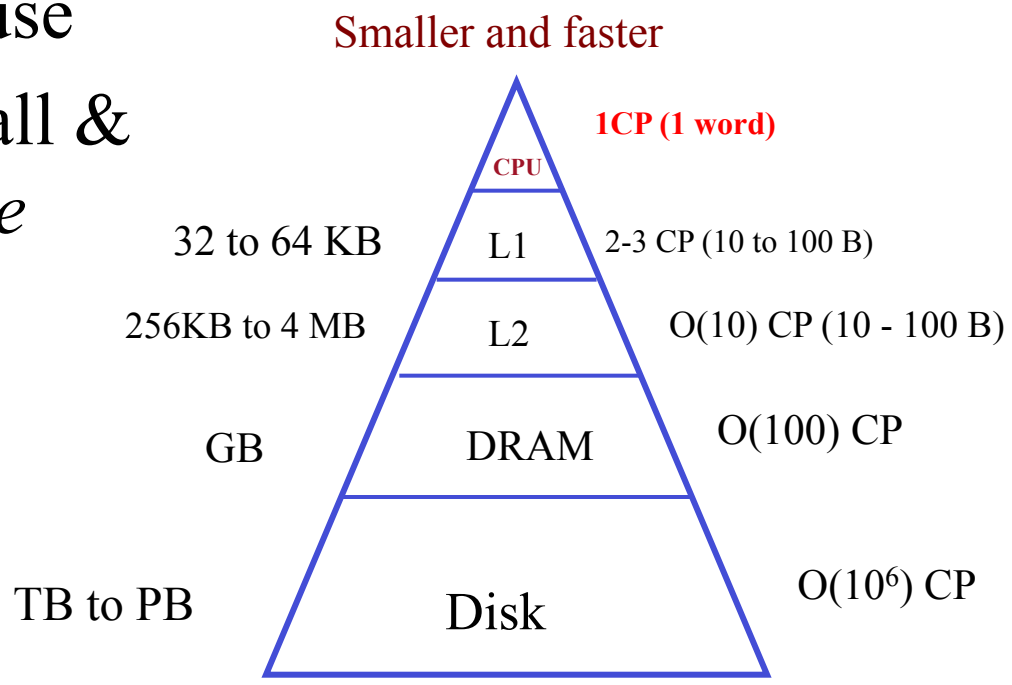
- Increase amount of computation performed per unit of communication
  - Conserve locality, tolerate or avoid communication
- Many threads



An important principle: locality

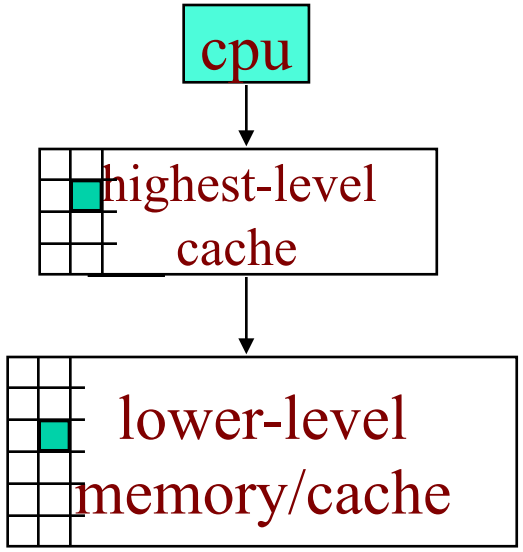
- Memory accesses exhibit two forms of locality
  - Temporal locality (time)
  - Spatial locality (space)
- Often involves loops
- Opportunities for reuse
- Idea: construct a small & fast memory to *cache* re-used data

```
for t=0 to T-1
   for i = 1 to N-2
      u[i]=(u[i-1] +  u[i+1])/2
```



Cache – a summary

- *cache hit:* an access that finds the data in cache
- *cache miss:* an access that's not
- *hit time:* time to access the higher cache
- *miss penalty:* time to move data from lower level to upper, then to cpu

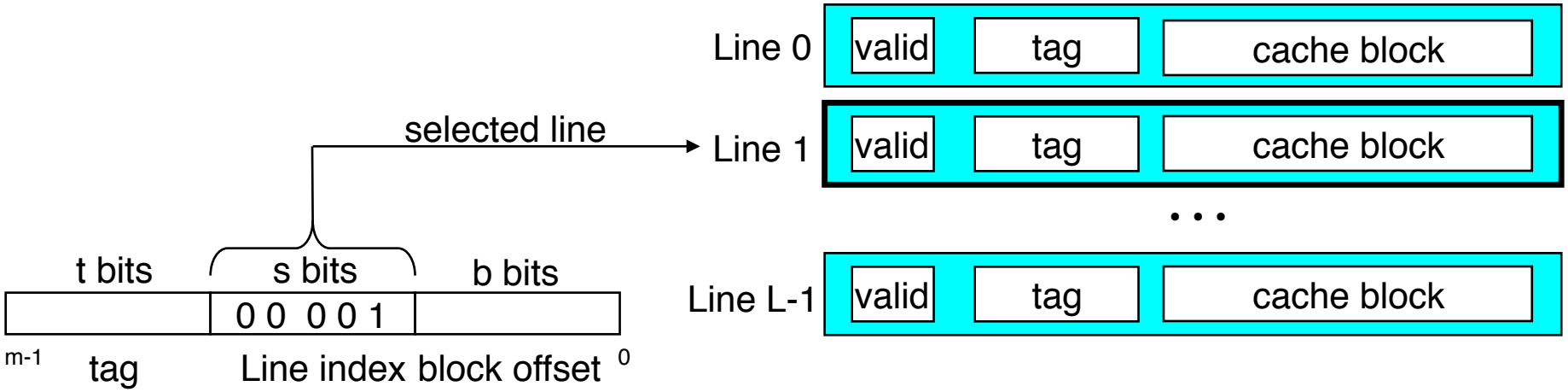


- *hit rate:* percentage of time the data is found in the higher cache [*miss rate*: (1 - hit rate)]
- *cache block size* or *cache line size*: the amount of data that gets transferred on a cache miss
- *Instruction* (*data*) cache: a cache that holds only instructions (data); unified cache holds both I+D

Michael Taylor

Direct mapped cache

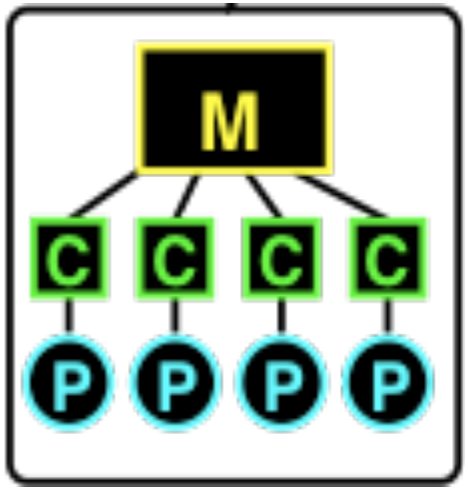
- Simplest cache
- Look up the line indexed by the line index
- Match the stored tag against the higher order address bits

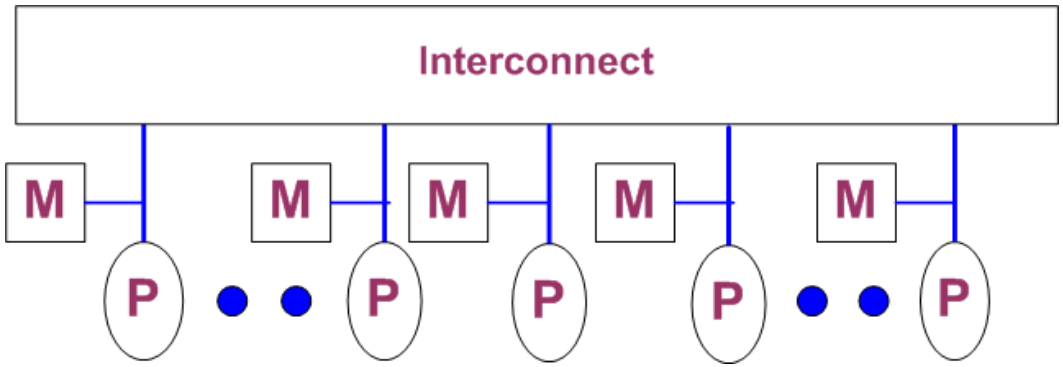


Randal E. Bryant and David R. O

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



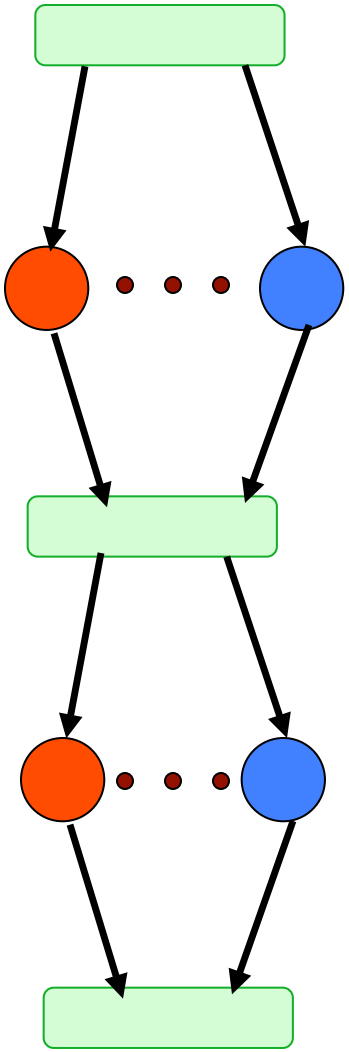


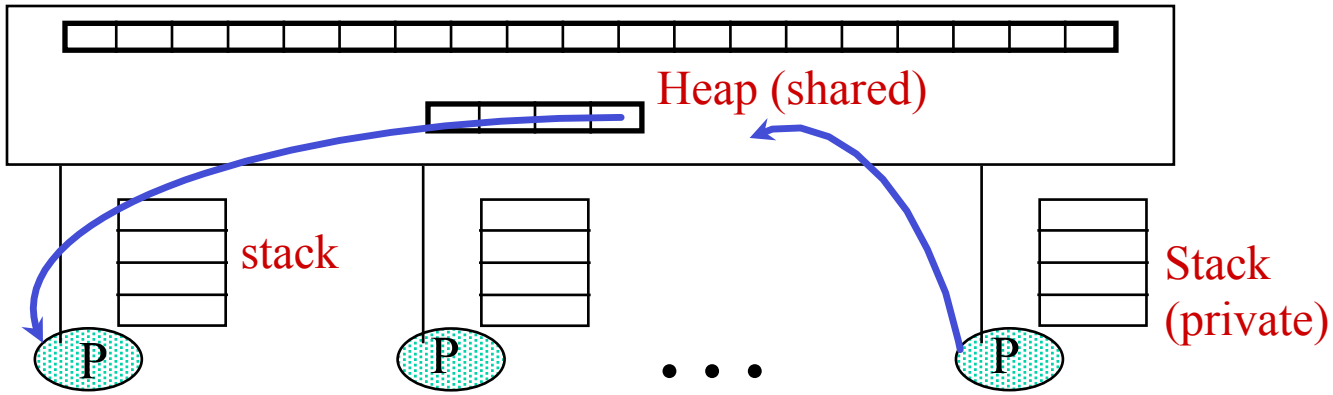
Today's lecture

- Technology
- Threads Programming
  - Correctness
  - Performance
- Algorithms (applications)
- List of Keywords and Topics (cross cutting)

Threads Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads communicate via shared memory
- 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
    - Race conditions
    - Deadlock

Implementation & techniques

- SPMD
  - Threads API
  - OpenMP
- Correctness: critical sections, race conditions
  - Mutexes and barriers
  - Atomic
  - Memory fences
- Performance: Data Partitioning
  - Block and cyclic decompositions
  - Dynamic scheduling
- Cross cutting issues (Performance & Correctness)
  - Cache coherenece and conistency
  - Cache locality
- Data dependencies, loop carried dependence

Today's lecture

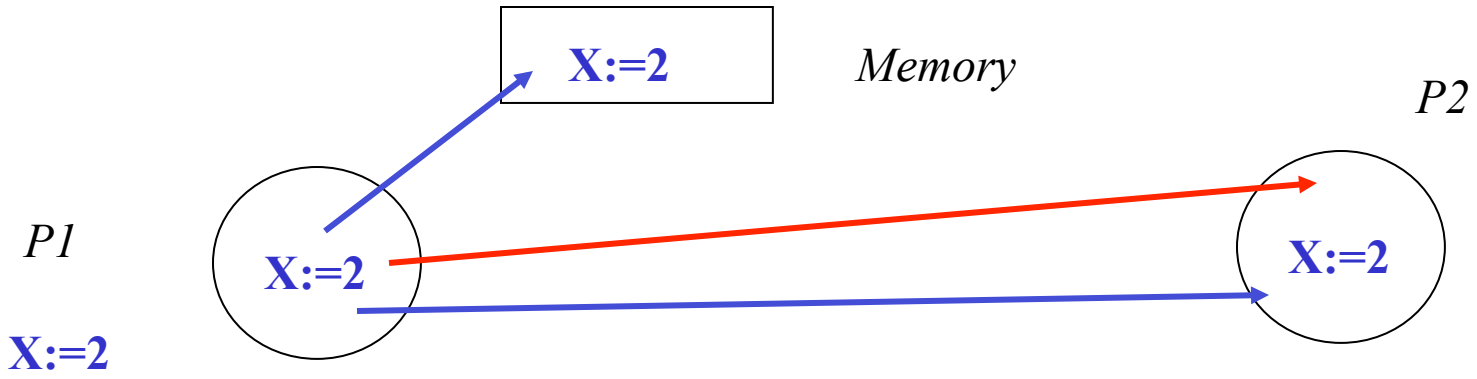
- Technology
- Threads Programming
  - Correctness
  - Performance
- Algorithms (applications)
- List of Keywords and Topics (cross cutting)

Correctness

- System: necessary condition for user level correctness
  - Cache coherence and consistency
- User: avoid race conditions through appropriate program synchronization
  - Migrate shared updates into main
  - Critical sections
  - Barriers
  - Atomics
  - Fork/Join

Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write



Memory consistency

- Cache coherence tells us that memory will *eventually* be consistent
- The memory consistency policy tells us *when* this will happen
- 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)

Memory consistency models

- Should it be impossible for both **if** statements to evaluate to true?
- With sequential consistency the results should always be the same provide that
  - Each processor keeps its access in the order made
  - We can't say anything about the ordering across different processors: access are interleaved arbitrarily

Processor 1	Processor 2
A=0	B=0
…	…
A=1	B=1
if (B==0) …	**if** (A==0) …

Rules

- **3 rules** that mostly concern when values must be transferred between main memory and per-thread memory
- **Atomicity.** Which instructions must have indivisible effects? Only concerned with instance and static variables, including array elements, but **not** local variables inside methods
- **Visibility.** Under what conditions the effects of one thread are visible to another? The effects of interest are: writes to variables, as seen via reads of those variables
- **Ordering.** Under what conditions the effects of operations can appear out of order to any given thread? In particular, reads and writes associated with sequences of assignment statements
- All changes made in one synchronized variable or code block are atomic and visible with respect to other synchronized variables and blocks employing the same lock, and processing of synchronized methods or blocks within any given thread is in program-specified order

Data races

- We say that a program allows a *data race* on a particular set of inputs if there is a *sequentially consistent execution*, i.e. an interleaving of operations of the individual threads, in which two conflicting operations can be executed "simultaneously" (Boehm)
- We'll say that operations can be executed "simultaneously", if they occur next to each other in the interleaving, and correspond to different threads
- We can guarantee sequential consistency only when the program avoids data races
- This program has a data race (x = = y = = 0 initially)

Execution 3
x = 1; **r1 = y;** **y = 1;** r2 = x; **// r1 = 1 ∧ r2 ==1**

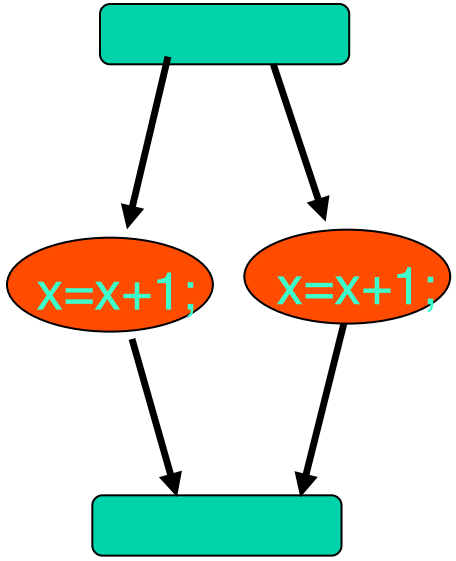
Thread 1	**Thread 2**
x = 1; r1 = y;	y = 1; r2 = x;

Under the hood of a data race

- Assume x is initially 0

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

- Generated assembly 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*

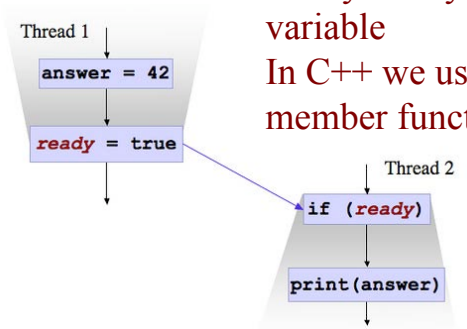
"Happens-before"

- An important fundamental concept in understanding the memory model
- Run on two separate threads, with counter = 0
  A: counter++;
  B: prints out counter
- Even if B occurs after A, no guarantee that B will see 0 …
- … unless we establish *happens-before relationship* between these two statements
- What guarantee is made by a *happens-before* relationship ?
  A guarantee that memory writes by one specific statement are visible to another specific statement

- Different ways of accomplishing this in C++: synchronization, atomics, variables, thread creation and completion

The C++11 Memory model

- The C++11 memory model makes minimal guarantees about semantics of memory access, bounding effects of optimizations on execution semantics
- Special mechanisms are needed to guarantee that **communication** happens between threads, that establish the "happens before" relationship
- Memory writes made by one thread can become visible, but no guarantee.
- *Without explicit communication, you can't guarantee which writes get seen by other threads, or even the order in which they get seen*
- C++ atomic variables (and the Java volatile modifier) are a special mechanism guaranteeing that communication happens between threads
- When one thread writes to a *synchronization variable*, and another thread sees that write, the first thread is telling the second about all of the contents of memory up until it performed the write to that variable



Ready is a synchronization variable
In C++ we use load and store member functions

All the memory contents seen by T1, before it wrote to ready, must be visible to T2,

after it reads the value true for ready.

http://jeremymanson.blogspot.com/2008/11/what-volatile-means-in-java.html

Avoiding Data races

- C++ and Java provide *synchronization* variables to communicate between threads, and are intended to be accessed concurrently
- Such concurrent accesses are not considered data races
- Thus, sequential consistency is guaranteed so long as the only conflicting concurrent accesses are to synchronization variables
- Any write to a synchronization variable establishes a *happens-before* relationship with subsequent reads of that same variable
- Declaring a variable as a synchronization variable
  - ensures that the variable is accessed indivisibly
  - prevents both the compiler and the hardware from reordering memory accesses in ways that are visible to the program.
- In C++ we have various kinds of synchronization variables, including
  - The *atomic* types
  - Mutexes

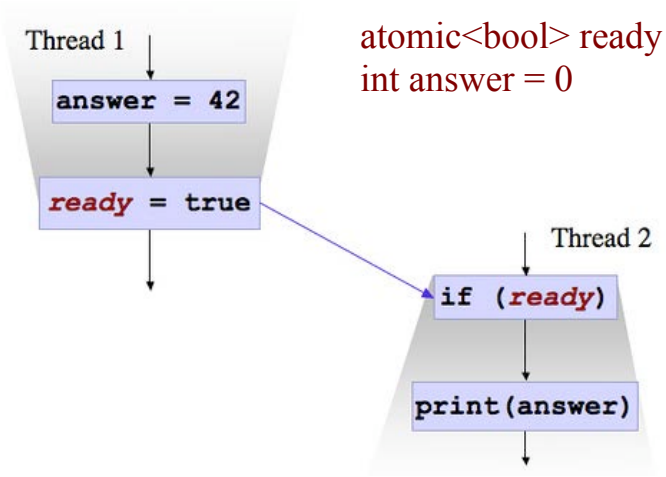
Using synchronization variables to ensure sequentially consistent execution

- This program is free from data races
- There cannot be an interleaving of the steps in which the actions x = 42 and r1 = x are adjacent
- Sequentially consistent execution is guaranteed: r1 = 42
- The implementation of atomic ensures that
  - thread 1's assignments to x and x_init become visible to other threads in order
  - The assignment r1 = x operation in thread 2 cannot start until we have seen x_init set.
- The practical significance..
  - The compiler must obey extra constraints and generate special code …
  - to prevent potential hardware optimizations, such as thread 1 making the new value of x_init available before that of x because it happened to be faster to access x_init's memory

Thread 1	Thread 2
atomic_bool x_ready; int x; x = 42; x_ready = true;	atomic_bool x_ready; int x; while (!x_ready) {} r1 = x;

Visibility

- Changes to variables made by one thread are guaranteed to be visible to other threads only under certain conditions



atomic<bool> ready = false;
int answer = 0

All the memory contents seen by T1, before it wrote to ready, must be visible to T2,

after it reads the value true for ready.

http://jeremymanson.blogspot.com/2008/11/what-volatile-means-in-java.html

Visibility

- Changes to fields made by one thread are guaranteed to be visible to other threads only under the following conditions
- A writing thread *releases* a synchronization lock and a reading thread subsequently *acquires* that same
  - Release flushes all writes from the thread's working memory, acquire forces a (re)load of the values of accessible variables
  - While lock actions provide exclusion only for the operations performed within a synchronized block, the memory effects cover all variables used by the thread performing the action
- If a variable is declared as atomic
  - Any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation.
  - Readers must reload the values of volatile fields upon each access.
- As a thread terminates, all written variables are flushed to main memory. Thus join, guarantees visibility of all thread's writes

```
Lock;
  x++;
Unlock;
```

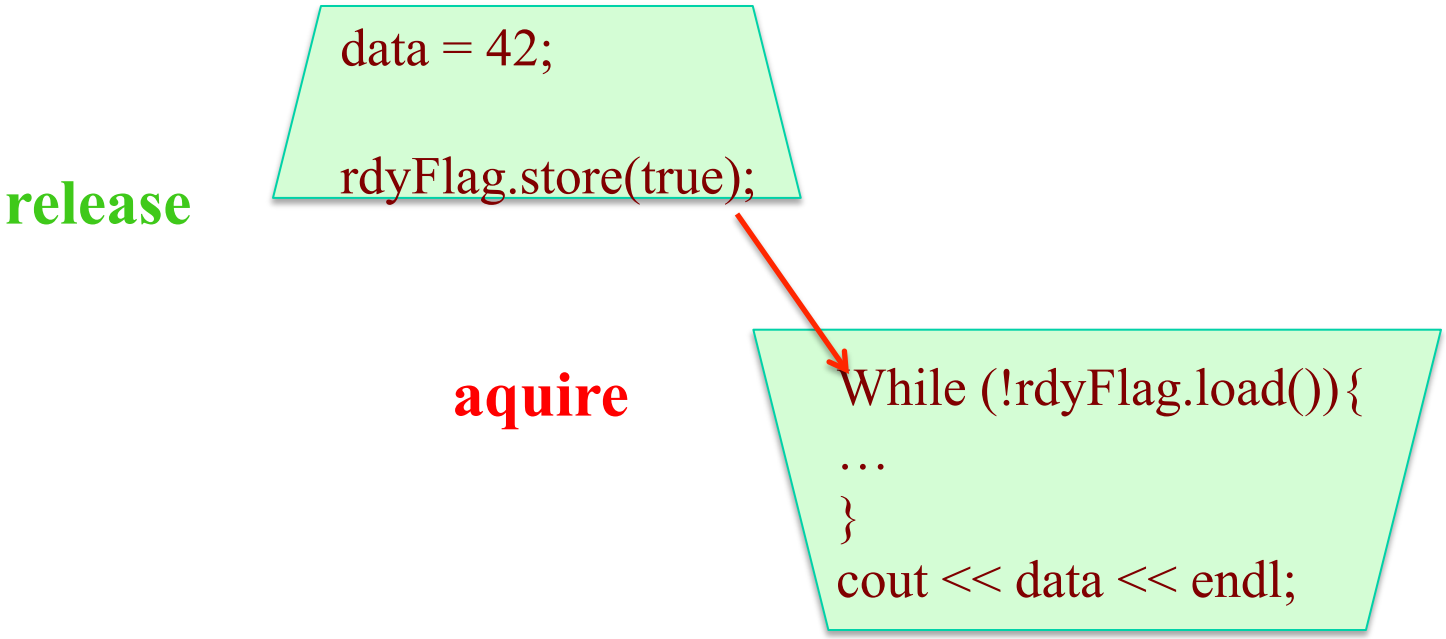
Acquire and release

- Why can the program tolerate non-atomic reads and writes? (Listing 5.2, *Williams*, p. 120)
- Howare the *happens-before* relationships established

```
std::vector<int> data;
std::atomic<bool> data_ready(false);
void reader_thread() {
    while(!data_ready.load())
        std::this_thread::sleep(std::milliseconds(1));
    std::cout<<"The answer="<< data[0]<<"\n";
}
void writer_thread() {
    data.push_back(42);
    data_ready=true;
}
```

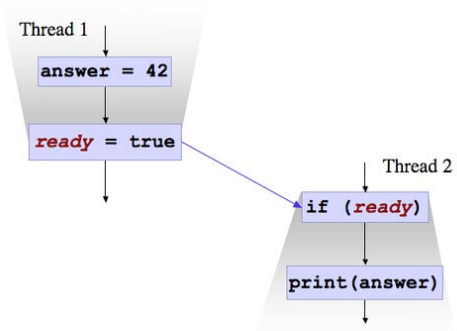
Acquire and release with atomics

- What can do wrong in these thread functions if there is no synchronization ?



The need for synchronization

- What can do wrong in these thread functions if there is no synchronization ?



Other kinds of data race errors

```
int64_t   global_sum = 0;

void sumIt(int TID) {
   mtx.lock();
   sum += (TID+1);
   mtx.unlock();
   if (TID == 0)
        cout << "Sum of 1 : " << NT << " = " << sum << endl;
}
```

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

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?

Correctness and Fairness

Each thread increments a shared variable until reaching a given maximum value

Using an atomic doesn't avoid the race condition

```
atomic<int> sharedVar;
int* howMany;
void summer(int TID) {
   while(sharedVar<MAX_VAL){
      sharedVar++;
      howMany[TID]++;
   } // $PUB/Examples/Threads/whoDunnit.cpp
}
```

Spawning 2 threads
shared Var: 1024
Thread 1 made 1024 updates
Spawning 2 threads
shared Var: 1025
Thread 0 made 531 updates
Thread 1 made 494 updates
Spawning 2 threads
shared Var: 1024
Updates by thread
Thread 1 made 1024 updates

- If we run with 2 threads, both try to update the shared object What happens to the shared var?
- If we spawn 100 threads, will all spawned threads get to update the shared variable?
- If one of not, we call that effect "starvation.
- How can we avoid starvation to ensure fairness?

Spawning 100 threads
shared Var: 1024
Thread 0 made 269 updates
Thread 1 made 484 updates
Thread 2 made 190 updates
Thread 3 made 81 updates
shared Var: 1025
Thread 0 made 4 updates
Thread 1 made 735 updates
Thread 2 made 286 updates
Spawning 100 threads
Thread 1 made 1024 updates

Synchronization in Perspective

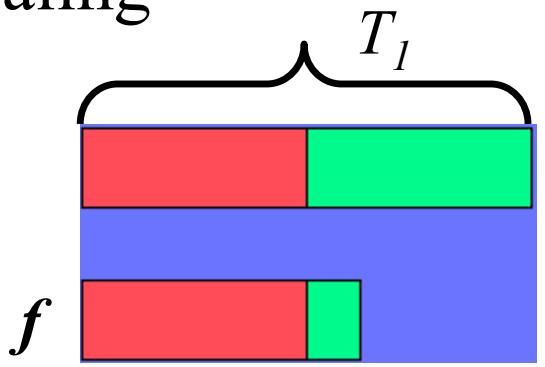
- Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness
- We need to take steps to avoid race conditions through appropriate program synchronization
  - Critical sections
  - Barriers
  - Atomics

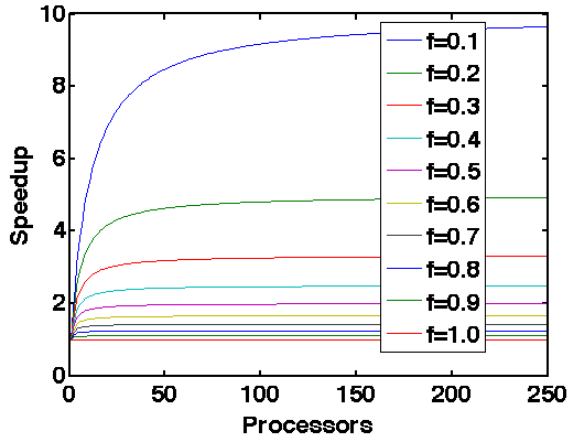
Today's lecture

- Technology
- Threads Programming
  - Correctness
  - Performance
- Algorithms (applications)
- List of Keywords and Topics (cross cutting)

Performance Terms and concepts

- Know the definition and significance of ….
- Parallel speedup and efficiency, super-linear speedup, strong scaling, weak scaling
- Amdahl's law, Gustafson's law, serial bottlenecks
- Strong and Weak Scaling





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

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, β = 1000 time units
- Constraint: $T(512,N) \leq 10^4$

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

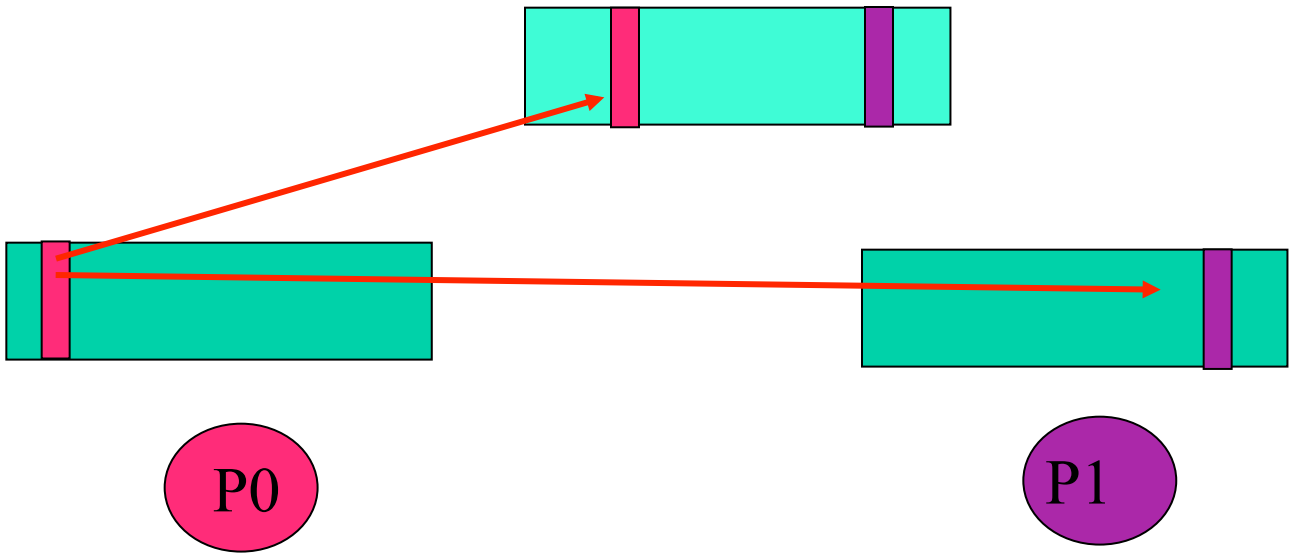
  $\Rightarrow \quad N \leq 1 \times 10^6$ (approximately)

Performance

- We run on 16 processors
- An application with 2 phases
  - Phase 1: perfectly parallelizable, $E_{16} = 100\%$
  - Phase 2: $E_{16} = 25\%$ efficiency on 16 processors
    Serial section: f of the overall running time on *2* processors: fT_2
- Express T_{16} in terms of f, as a fraction of the form x/y

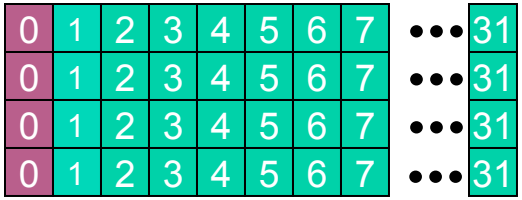
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another's cache



Eliminating false sharing

- Put each counter in its own cache line




```
static int counts[];
for (int k = 0; k<reps; k++)
        for (int r = first; r <= last; ++ r)
                if ((values[r] % 2) == 1)
                        counts[TID]++;
```

```
static int counts[][LINE_SIZE];
for (int k = 0; k<reps; k++)
        for (int r = first; r <= last; ++ r)
                if ((values[r] % 2) == 1)
                counts[TID][0]++;
```

	NT=1	NT=2	NT=4	NT=8
Unoptimized	4.7 sec	6.3	7.9	10.4
Optimized	4.7	5.3	1.2	1.3

Load Balancing

- We have an int array of length 1024, and share the computation evenly among the processors. The workload consists of updating the array
- All the updates take the same amount of time, 1 second

a. What is the running time on 4 processors?

b. What is the running time on 5 processors?

c. What is the running time on 16 processors?

Today's lecture

- Technology
- Threads Programming
  - Correctness
  - Performance
- Algorithms (applications)
- List of Keywords and Topics (cross cutting)

Algorithms

- Image smoother (stencil method)
- Sort
  - Odd-even sort
  - Merge
- Particle method
- Barrier
- Tree thread
- Processor Mapping with OpenMP

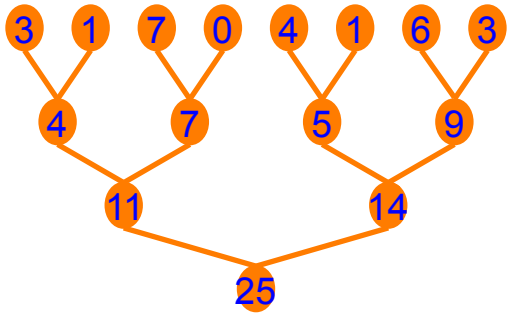
5. Tree Summation

- Input: an array x[], length N >> P
- Output: Sum of the elements of x[]
- Goal: Compute the sum in lg P time

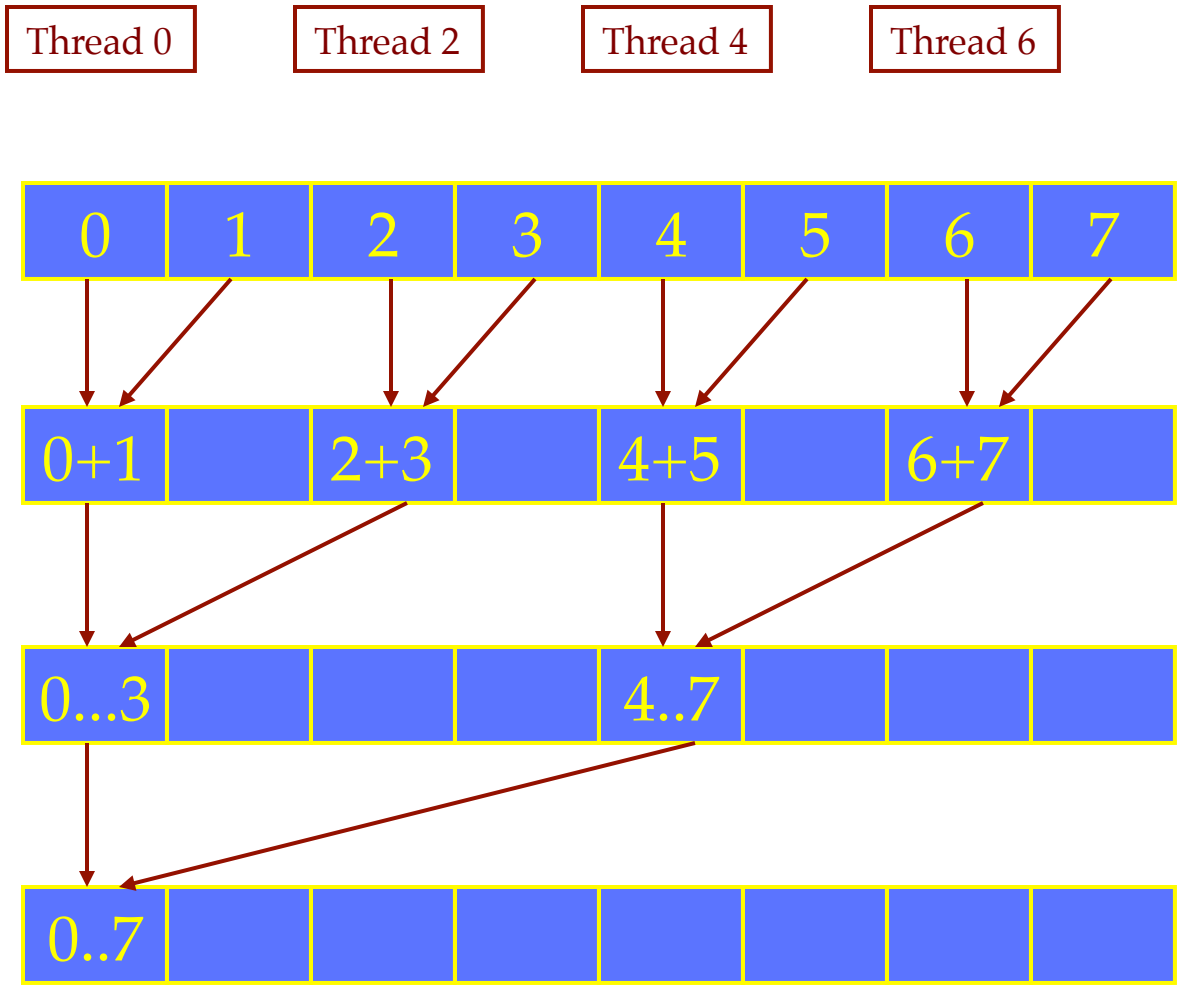
```
sum = 0;
for i=0 to N-1
    sum += x[i]
```

- Assume P is a power of 2, K = lg P
- Starter code

```
for m = 0 to K-1 {

}
```



Visualizing the Summation

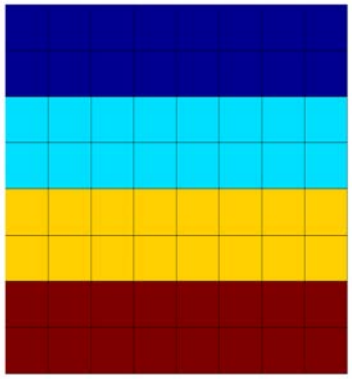


Building a linear time barrier with locks

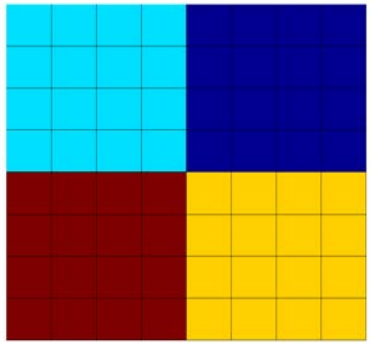
```
Mutex arrival=UNLOCKED, departure=LOCKED;  // Shared
 int count=0;                                // Shared

 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
```

Workload Decomposition

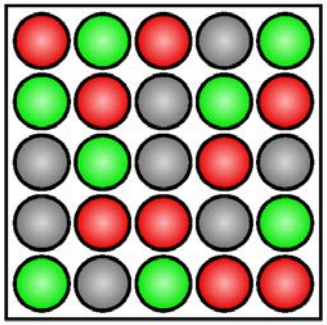
- Block vs. Cyclic
- Static vs. Dynamic Decomposition



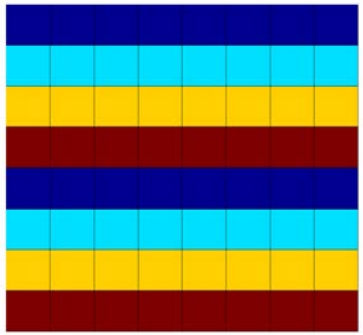
[Block, *]



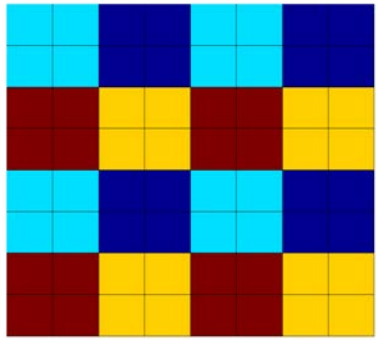
[Block, Block]



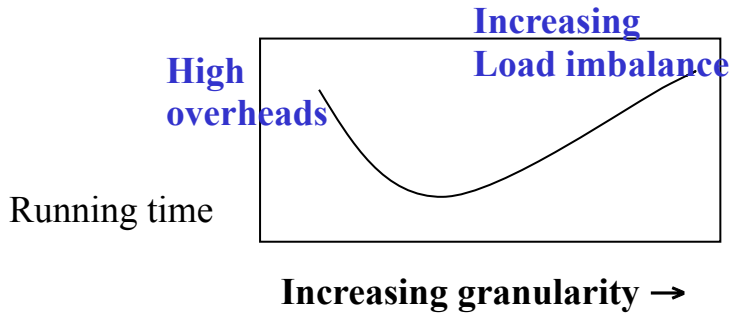
Dynamic



[Cyclic, *]



[Cyclic(2), Cyclic(2)]



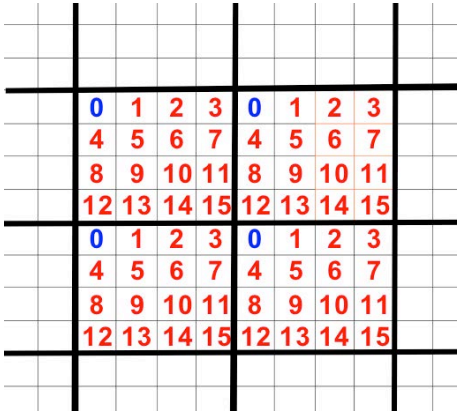
Tradeoffs in choosing the chunk size

- 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 ¼ 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

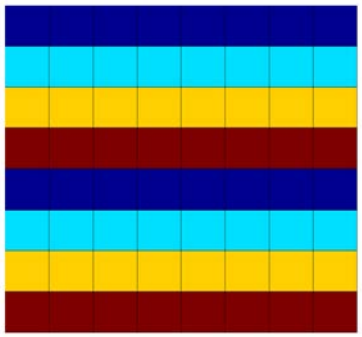
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3



Static approach: BLOCK CYCLIC

- Divide bins pieces of size CHUNK
- Core k gets chunks k, k+NT*CHUNK, k+2*NT*CHUNK, …
- Also called *round robin* or *block cyclic*

```
#pragma omp parallel for schedule(static, 2)
for( int i = 0; i < n; i++ ) {
    for (int j = 0; j < n; j++ ){
        doWork(i,j);
    }
}
```



Implementation of self-scheduling

- Critical section can be costly, but OMP atomic is restricted to simple update statements, e.g ++ +=
- C++ atomic<T> not guaranteed to be lock free, but probably more efficient
- OMP and C++ implementations

```
#include <openmp.h>

boolean getChunk(int& mymin){

 #pragma omp critical  // Inefficient
  {                    // Crit Sect

     k = _counter;
     _counter += _chunk;
  }
  if ( k > (_n – _chunk)
     return false;
  mymin = k;
  return true;
}
```

```
#include <atomic.h>
boolean getChunk(int& mymin){

  mymin =  _counter.fetch_add(_chunk)

  if (mymin > (_n – _chunk) // not past last chunk
       return false;

  else return true;
  }
```