

Lecture 20

Computing with GPUs

Supercomputing

Final Exam Review

Announcements

- The Final is on
Tue March 15th from 3pm to 6pm
 - ▶ 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
- Don't forget to do the Peer Review Survey, which is worth 1.5% of your final exam grade
https://www.surveymonkey.com/r/Baden_CSE160_Wi16

Today's Lecture

- Computing with GPUs
- Logarithmic barrier strategy
- Supercomputers
- Review

Experiments - increment benchmark

- Total time: timing taken from the host, includes copying data to the device
- Device only: time taken on device only
- Loop repeats the computation inside the kernel – 1 kernel launch and 1 set of data transfers in and out of device

N = 8388480 (8M ints), block size = 128, times in *milliseconds*,

Repetitions				
10	100	1000	10 ⁴	
1.88	14.7	144	1.44s	Device time
19.4	32.3	162	1.46s	Kernel launch + data xfer

What is the cost of moving the data and launching the kernel?

- A. About 1.75 ms $((19.4-1.88)/10)$
- B. About 0.176 ms $(32.3-14.7)/100$
- C. About 0.018 ms $((162-144)/1000)$
- D. About 17.5 ms $(19.4-1.88)$

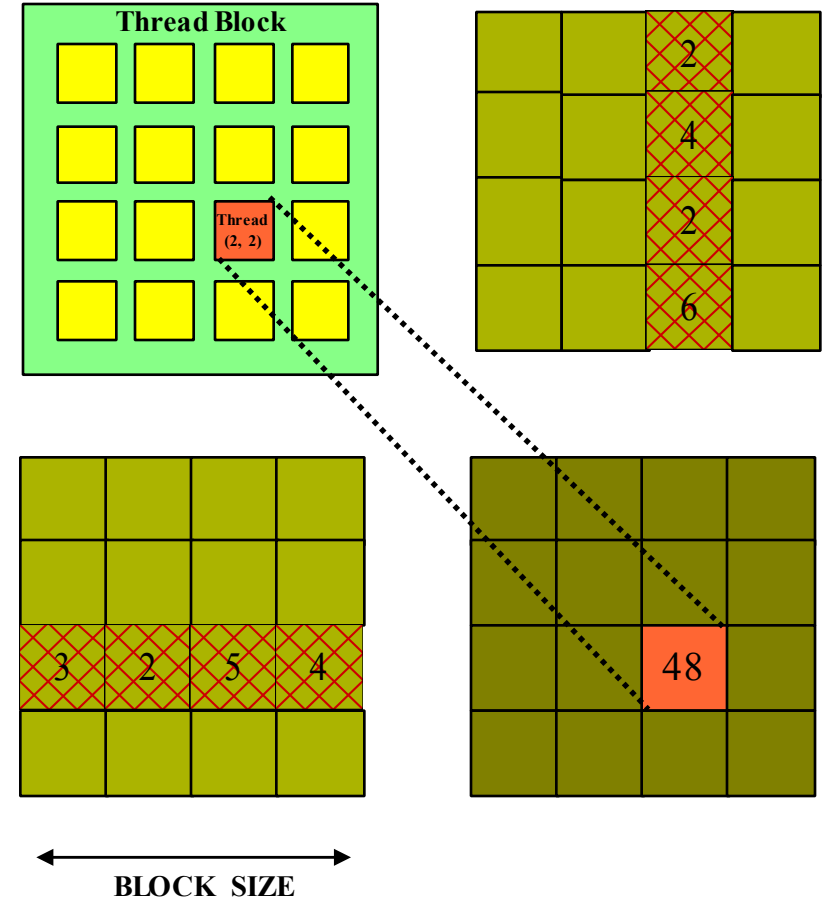
N = 8 M block size = 128, times in *milliseconds*

Repetitions

10	100	1000	10 ⁴	
1.88	14.7	144	1.44s	Device time
19.4	32.3	162	1.46s	Kernel launch + data xfer

Matrix Multiply on the GPU

- Naïve algorithm
 - Each thread loads all the data it needs, independently loads a row and column of input
 - Each matrix element loaded multiple times
- Tiled algorithm with shared memory
 - Divide the matrices into *tiles*, similar to blocking for cache
 - Threads cooperate to load a tile of A&B into on-chip shared memory
 - Each tile in the result matrix C corresponds to a thread block
 - Each thread performs:
 b mpy-adds + 1 load + 1 store



What is the floating point intensity of the tiled algorithm?

A. 1

B. 2

C. N

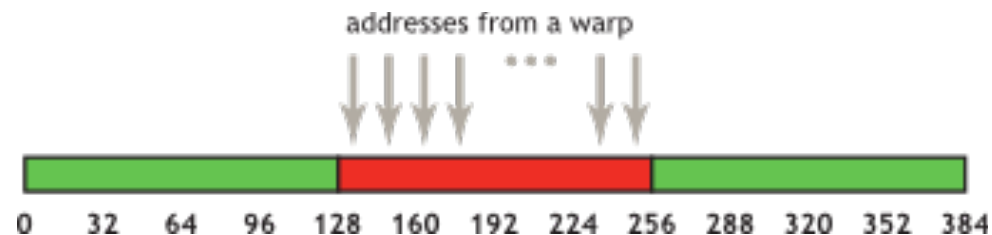
D. b

E. b^2

The analysis is the same as for blocked matrix multiplication

Results with shared memory

- N=512, double precision
- Last fall, CSE 260 students got up to **468 Gflops** (Naïve implementation: 116 GF)
- Compare with about 7.6 Gflops/core on Bang 19.1 GF per core on Intel Sandy Bridge [2.7GHz, 256 bit SSE4, peak speed 21.6 GF]
- What happened?
 - Reduced global memory accesses, and accessed in contiguous regions (*coalesced* memory accesses)
 - Blocking involves both shared memory and registers



GPU performance highlights

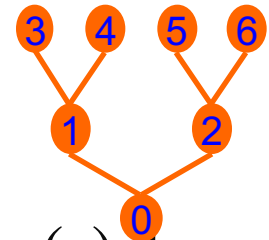
- Simplified processor design, but more user control over the hardware resources
- Use or lose the available parallelism
- Avoid algorithms that present intrinsic barriers to utilizing the hardware
- **Rethink the problem solving technique primarily to cut data motion costs**
 - Minimize serial sections
 - Avoid host $\leftrightarrow$ device memory transfers
 - Global memory accesses $\rightarrow$ fast on-chip accesses
 - Hide device memory transfers behind computation
 - Coalesced memory transfers
 - Avoid costly branches, or render them harmless

Today's Lecture

- Computing with GPUs
- **Logarithmic barrier strategy**
- Superscomputers
- Review

An improved barrier

- Replacing mutexes with atomics improves performance dramatically, but still doesn't scale
- Use a log time barrier, relying on a combining tree
- Each tree node on a separate cache line
- Each processor begins at its leaf node
 - Arrival signal(s) sent to parent
 - Last arriving core continues the process, other(s) drop out
- 1 processor left at root, starts the continue process, signals move in the opposite direction
- See code in \$PUB/Examples/Threads/CTBarrier.h
- More efficient variations based sense reversal, see Mellor-Crummey's lecture
cs.anu.edu.au/courses/comp8320/lectures/aux/comp422-Lecture21-Barriers.pdf

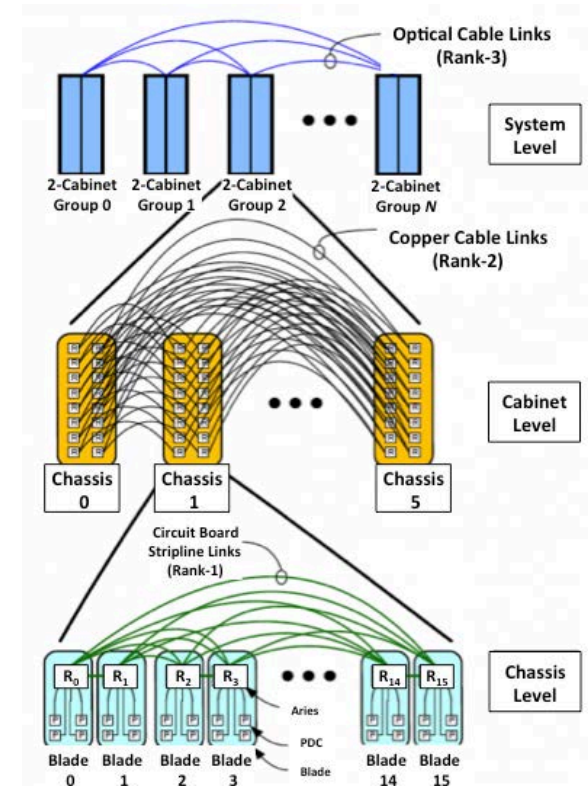
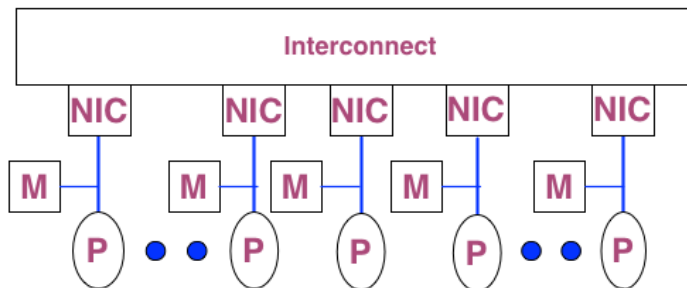


Today's Lecture

- Computing with GPUs
- Logarithmic barrier strategy
- **Supercomputers**
- Review

What does a supercomputer look like?

- Hierarchically organized parallelism
- Hybrid communication
 - Threads within each server
 - Pass messages between servers (or among groups of cores)
“shared nothing architectures”

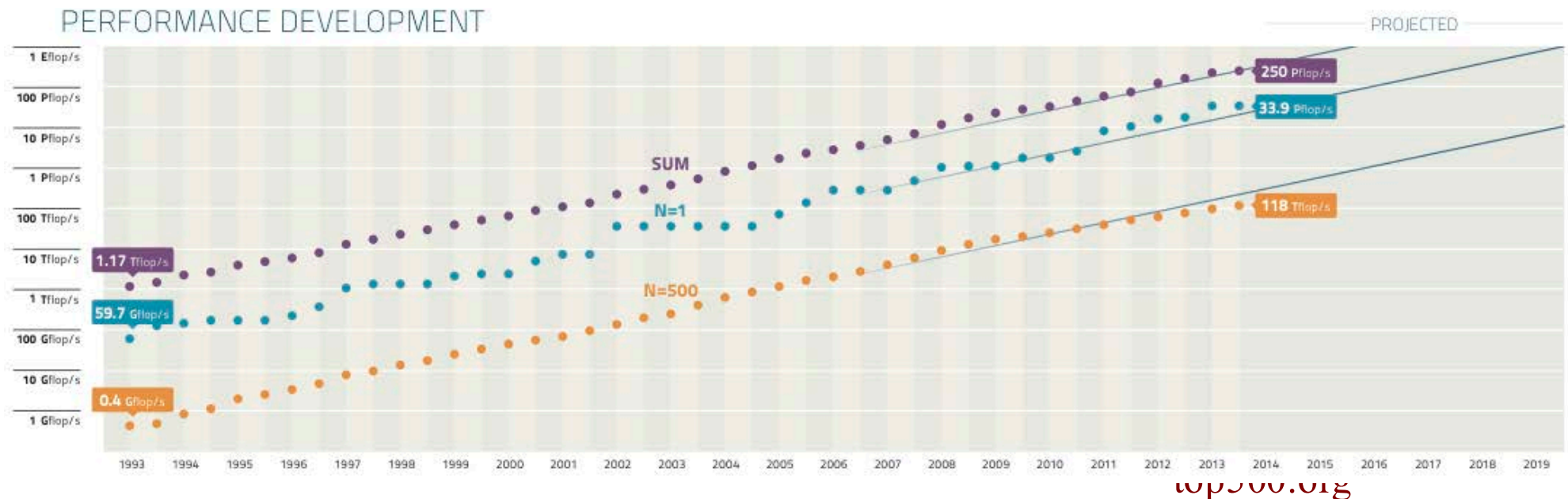


Edison @ nersc.gov

conferences.computer.org/sc/2012/papers/1000a079.pdf

What is the worlds fastest supercomputer?

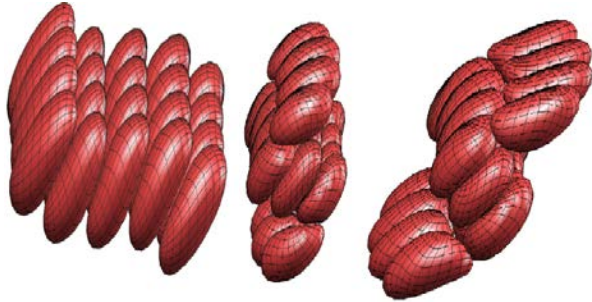
- Top500 #1, Tianhe-2 @ NUDT (China)
 - 3.12 Million cores
 - 54.9 Tflop/sec peak
 - 17.8 MW power (+6MW for cooling)
 - 1 PB memory (2^{50} Bytes)



State-of-the-art applications

Blood Simulation on Jaguar

Gatech team



p	48	384	3072	24576
Time (sec)	899.8	116.7	16.7	4.9
Efficiency	1.00	0.96	0.84	0.35

Strong scaling

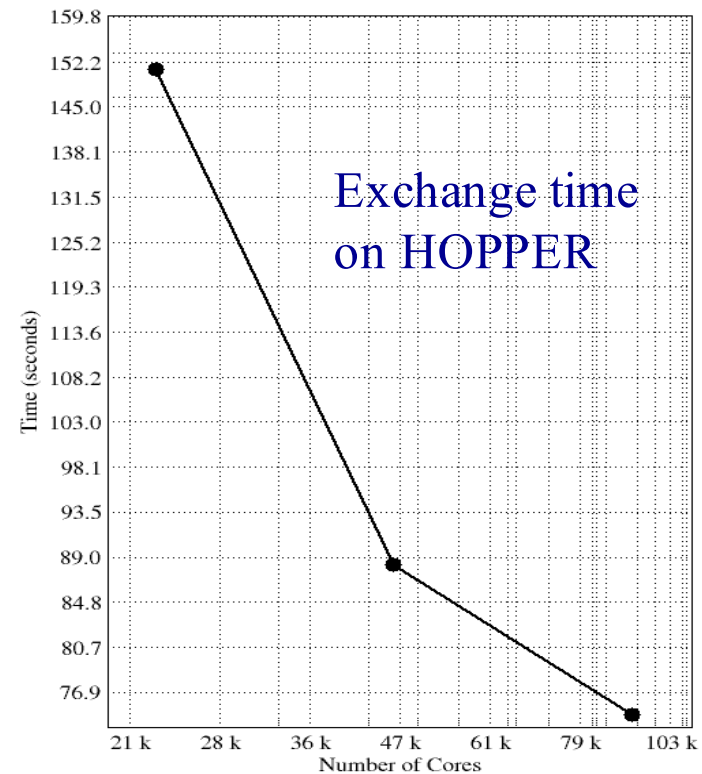
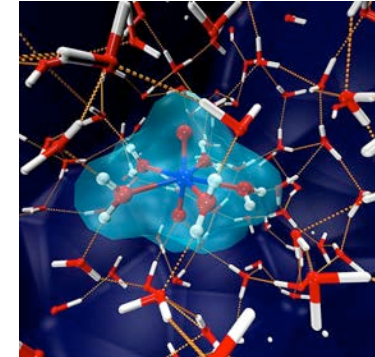
p	24576	98304	196608
Time (sec)	228.3	258	304.9
Efficiency	1.00	0.88	0.75

Weak scaling

Slide courtesy
Tan Nguyen, UCSD

Ab Initio Molecular Dynamics (AIMD) using Plane Waves Density Functional Theory

Eric Bylaska (PNNL)



Have you ever seen a supercomputer in real life?

A. Yes

B. No

C. Not sure

Up and beyond to Exascale

- In 1961, President Kennedy mandated a Moon landing by decade's end
- July 20, 1969 at tranquility base
“The Eagle has landed”
- The US Govt set an ambitious schedule to reach 10^{18} flops by 2023, x100 performance increase.
- DOE is taking the lead in the US, China and the EU also engaged
- Massive technical challenges esp software, resilience and power consumption



Why numerically intensive applications?

- Highly repetitive computations are prime candidates for parallel implementation
- Improve quality of life, economically and technologically important
 - Data Mining
 - Image processing
 - Simulations – financial modeling, weather, biomedical

Courtesy of Randy Bank

Classifying the application domains

- ▶ Patterns of *communication* and computation that persist over time and across implementations

- ▶ Structured grids

- Panfilov method

- ▶ Dense linear algebra

- Matrix multiply, Vector-Mtx Mpy
Gaussian elimination

- ▶ N-body methods

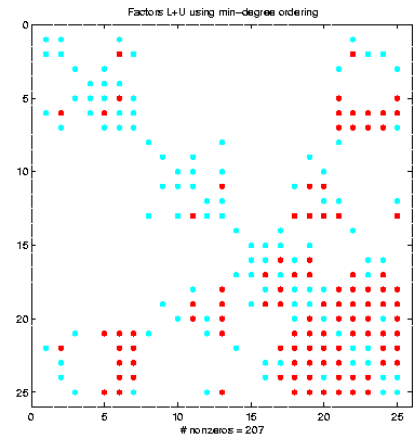
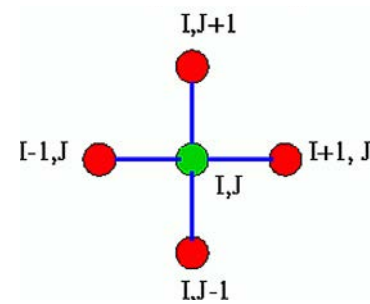
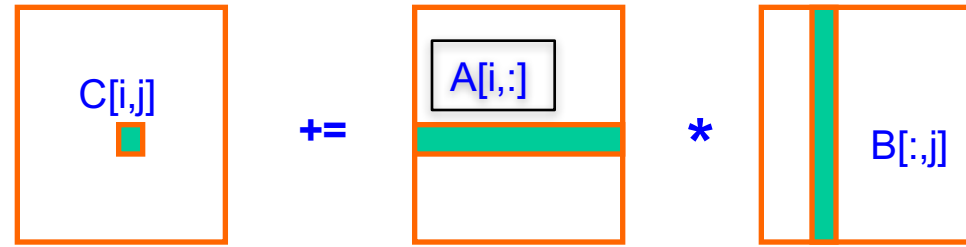
- ▶ Sparse linear algebra

- In a sparse matrix, we take advantage of knowledge about the locations of non-zeros, improving some aspect of performance

- ▶ Unstructured Grids

- ▶ Spectral methods (FFT)

- ▶ Monte Carlo



Courtesy of Randy Bank

I increased performance — so what's the catch?

- Currently there exists no tool that can convert a serial program into an efficient parallel program
 - ... for all applications ... all of the time ... on all hardware*
- The **more we know** about the application...
 - ... specific problem ... math/physics ... initial data ...
 - ... context for analyzing the output...
 - ... the more we can improve performance**
- We can classify applications according to Patterns of communication and computation that persist over time and across implementations - Phillip Colella's 7 Dwarfs
- Performance Programming Issues
 - Data motion and locality
 - Load balancing
 - Serial sections

What you learned in this class

- How to solve computationally intensive problems on parallel computers effectively
 - Theory and practice
 - Software techniques
 - Performance tradeoffs
- Emphasized multi-core implementations, threads programming, but also the memory hierarchy and SSE vector instructions
- Developed technique customized to different application classes
- We built on what you learned earlier in your career about programming, algorithm design & analysis and generalize them

Do you have an application in mind for multithreading?

A. Yes

B. No

C. Maybe

How about SSE?

- A. Yes
- B. No
- C. Maybe

Today's Lecture

- Computing with GPUs
- Logarithmic barrier strategy
- Supercomputers
- Review

What are the main issues in implementing multithreaded applications?

- Conserve locality: cache, registers, minimize use of shared memory
- Maximize concurrency: avoid serial sections, take advantage of ILP, SSE
- Ensure correctness
- Avoid overheads: serial sections, load imbalance, excessive thread spawning, false sharing, contention on shared resources, including synchronization variables

Why we need a memory model

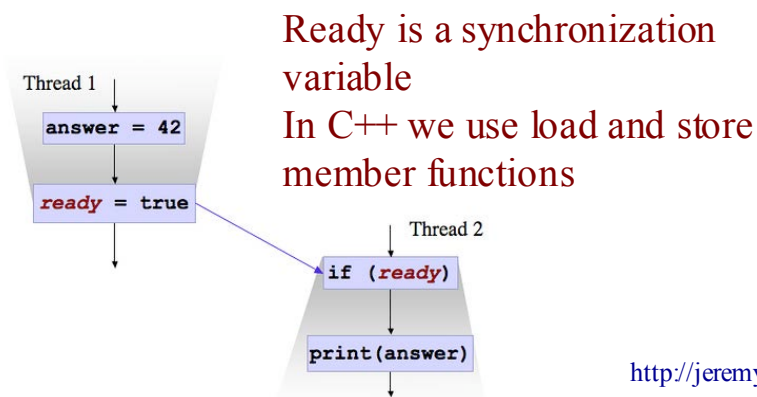
- When one thread changes memory, then there needs to be a definite order to those changes, as seen by other threads
- Ensure that multithreaded programs are *portable*: they will run correctly on different hardware
- Clarify which optimizations will or will not break our code
 - Compiler optimizations can move code and may need to obey extra constraints and to generate special code to prevent potential hardware optimizations that could re-order the time to access the variables in memory (e.g. cache)
 - Hardware scheduler executes instructions out of order
- The memory model makes certain guarantees that a particular update to a particular variable made by one thread will **eventually** be visible to another
- The programmer uses synchronization variables
 - To ensure that the variable is accessed indivisibly, that changes become visible and that there are strict orderings in how writes are seen by other threads
 - Prevent both the compiler and the hardware from reordering memory accesses in ways that are visible to the program and could break it
- The model does not require visibility failures across threads, they merely allow these failures to occur
- Not using synchronization in multithreaded code doesn't guarantee safety violations, it just allows them

Consistency vs coherence

- Cache coherence is a *mechanism*, a hardware protocol to ensure that memory updates propagate to other cores..
- Which will then be able to agree on the values of information stored in memory, *as if* there were no cache at all
- *Cache consistency* defines a *programming model*: **when** do memory writes become visible to other cores?
 - Defines the ordering of memory updates
 - A contract between the hardware and the programmer: if we follow the rules, the results of memory operations are guaranteed be predictable

The central role of synchronization variables

- The C++ *atomic* variable provides a special mechanism to guarantee that communication happens between threads
 - Which writes get seen by other threads
 - The order in which they will be seen
- The *happens-before* relationship provides the guarantee that memory writes by one specific statement are visible to another specific statement
- Different ways of accomplishing this: atomics, variables, thread creation and completion
- When one thread writes to a *synchronization variable* (e.g. an *atomic* or *mutex*) 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



*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>

Where are there happens-before relationships?

- A. (T1:1) -> (T2:1) D. A and B
B. (T1:3) -> (T2:1) E. B and C
C. (T1:2) -> (T2:6)

Thread 1

```
(1) lock();  
(2) NT++;  
(3) unlock();  
(4) BARRIER( );  
(5) if(TID==0)  
    cout<<NT<<endl;
```

Thread 2

```
(1) lock();  
(2) NT++;  
(3) unlock();  
(4) BARRIER( );  
(5) if(TID==0)  
(6)    cout<<NT<<endl;
```

Inter-thread happens-before relations are defined between pairs of synchronization operations. Atomics are one example another is the beginning and end of a critical section, as in this example

Synchronization: What is minimal?

(1)	void sweep(int TID, int myMin, int myMax, double ϵ , atomic<double>& err) {
(2)	for (int s = 0; s < 100; s++) {
(3)	double localErr = 0;
(4)	for (int i = myMin; i < myMax; i++) {
(5)	unew [i] = (u [i-1] + u [i+1])/2.0;
(6)	double δ = fabs(u[i] - unew[i]);
(7)	localErr += δ * δ ;
(8)	}
(9)	err += localErr;
(10)	if ((s > 0) && (err < ϵ))
(11)	break;
(12)	if (!TID){double *t = u; u = unew; unew = t;} // Swap u $\leftrightarrow$ unew
(13)	err = 0;
(14)	} // End of s loop
(15)	}

Which barriers can we remove?

(1)	void sweep(int TID, int myMin, int myMax, double ϵ , atomic<double>& err){
(2-8)	for (int s = 0; s < 100; s++) { ... unew [i] = (u [i-1] + u [i+1])/2.0
(9)	err += localErr;
(9a)	BARRIER()
(10)	if ((s > 0) && (err < ϵ))
(11)	break;
(11a)	BARRIER()
(12)	if (!TID){ Swap u $\leftrightarrow$ unew }
(12a)	BARRIER()
(13)	If (!TID) err = 0;
(13a)	BARRIER()
(14)	} // End of s loop
(15)	}

- A. 9a & 11a
- B. 11a & 12a
- C. 11a or 12a
- D. 3 of them
- E. Something else