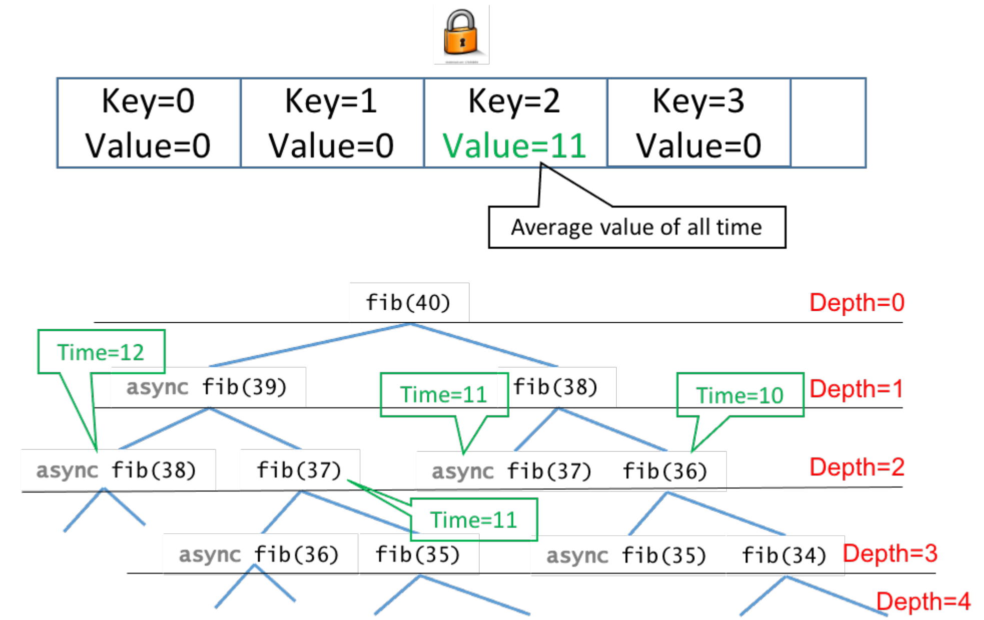
Lecture 26: End Semester Review

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

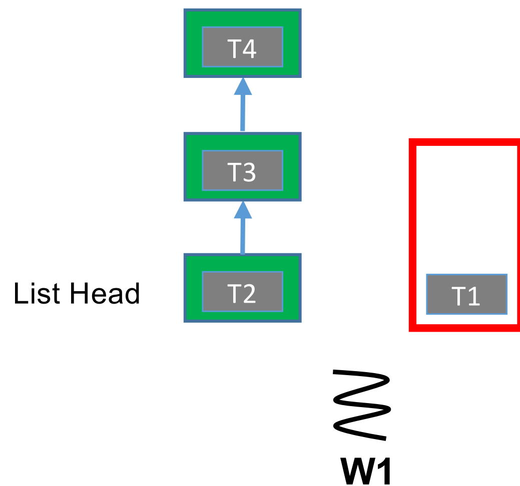
 © Vivek Kumar

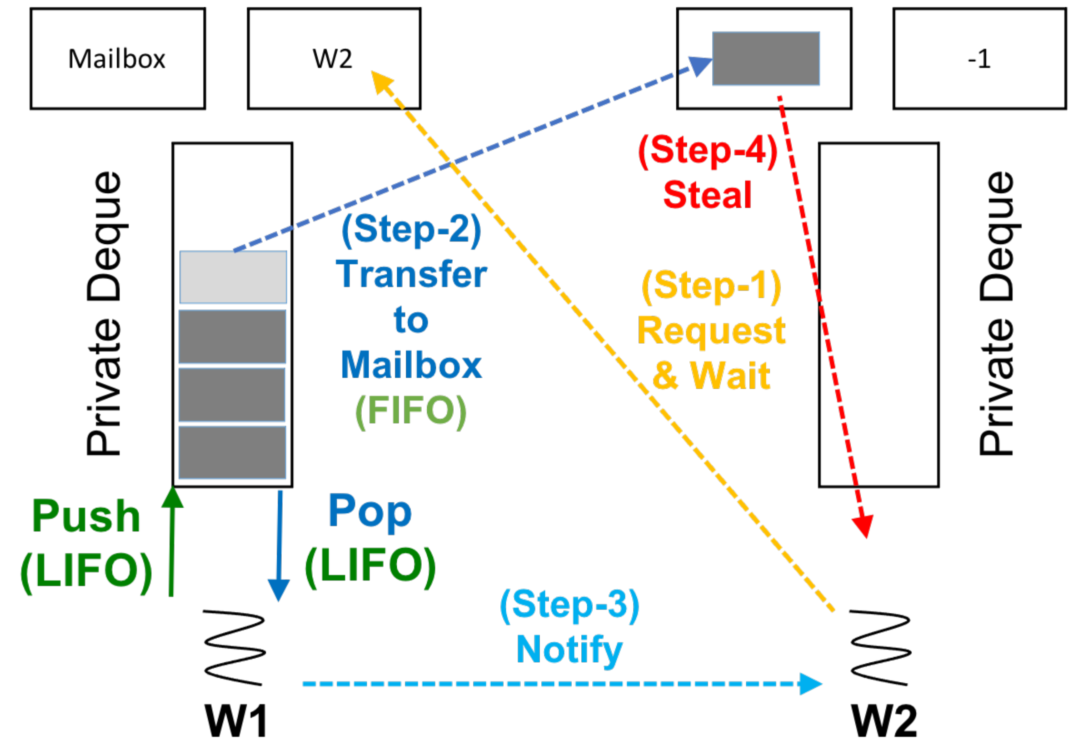
Sequential Overheads: Task Granularity

- Sequential overheads from fine granular task creation
  - Tasks near the bottom of tree are smaller computations
  - Deep procedure calling stack in thread due to recursion
- Automatically controlling task granularity in recursive task decomposition
  - Assumption is that the tree (computation graph) is well balanced
    - Dynamic task aggregation
    - Each task records its depth and the execution time at that depth
    - Above information is used to decide if any more tasks at certain depth has to be created or should be executed serially



Sequential Overheads: Concurrent Deque





- Minimizing deque overheads
  - Using a mix of list and deque
  - Using private deque

Boost Context C++11 Library: Example

```
void A() {
  cout<< “IN-A” << endl;
  /* Do something */
  cout<< “OUT-A” << endl;
}
void B() {
  cout<< “IN-B” << endl;
  /* Do something */
  cout<< “OUT-B” << endl;
}
void C() {
  cout<< “IN-C” << endl;
  /* Do something */
  cout<< “OUT-C” << endl;
}
int main() {
  A();
  B();
  C();
}
```

Figure-1

```
#include <boost/context/all.hpp>
ctx::continuation A(ctx::continuation cont) {
  cout<< “IN-A” << endl;
  cont = cont.resume();
  /* Do something */
  cout<< “OUT-A” << endl;
  return std::move(cont);
}
/* Methods B & C rewritten as A above */
int main() {
  ctx::continuation a = ctx::callcc(A);
  ctx::continuation b = ctx::callcc(B);
  ctx::continuation c = ctx::callcc(C);
  a.resume();
  b.resume();
  c.resume();
}
```

Figure-2

Used to switch across different continuations

Call with current continuation. Captures current continuation and triggers a context switch

- Figure-1

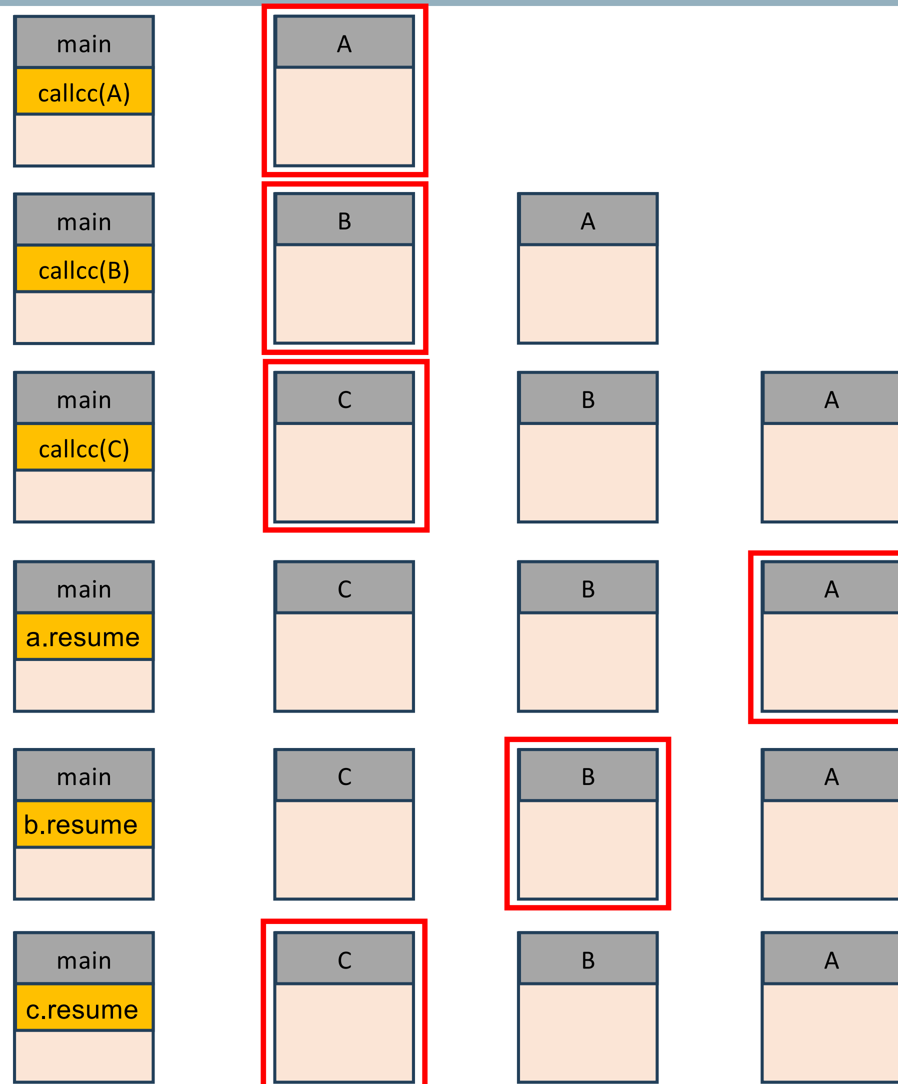
IN-A
OUT-A
IN-B
OUT-B
IN-C
OUT-C

- Figure-2

IN-A
IN-B
IN-C
OUT-A
OUT-B
OUT-C

Boost Context C++11 Library: Stack Switch

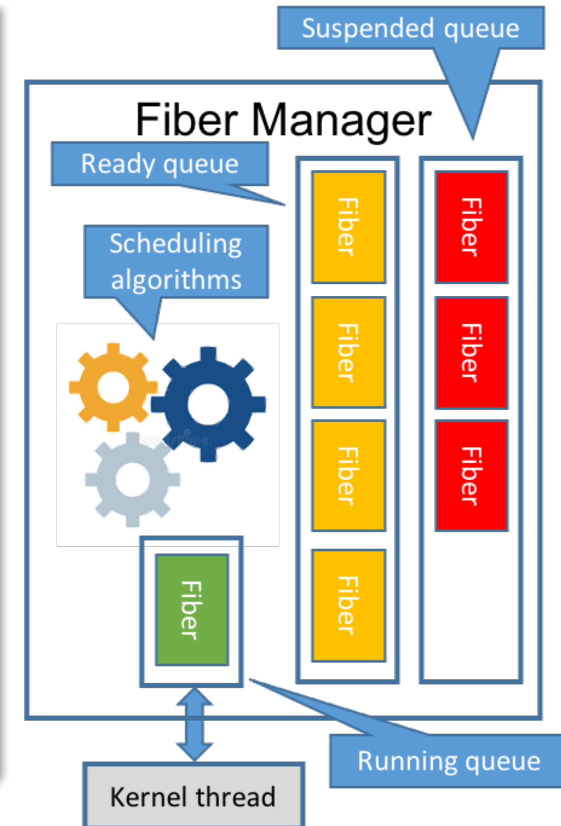
```
#include <boost/context/all.hpp>
ctx::continuation A(ctx::continuation cont) {
  cout<< "IN-A" << endl;
  cont = cont.resume();
  /* Do something */
  cout<< "OUT-A" << endl;
  return std::move(cont);
}
/* Methods B & C rewritten as A above */
int main() {
  ctx::continuation a = ctx::callcc(A);
  ctx::continuation b = ctx::callcc(B);
  ctx::continuation c = ctx::callcc(C);
  a.resume();
  b.resume();
  c.resume();
}
```



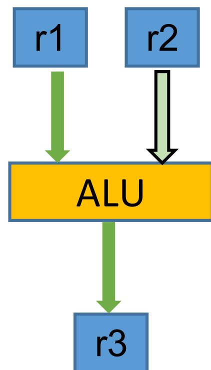
User Level Threads: Fibers

```
boost::fibers::fiber f1([=]() {
  cout << “A ”;
  boost::this_fiber::yield();
  cout << “B ”;
  boost::this_fiber::yield();
  cout << “C ”;
});

boost::fibers::fiber f2([=]() {
  cout << “D ”;
  boost::this_fiber::yield();
  cout << “E ”;
  boost::this_fiber::yield();
  cout << “F ”;
});

f1.join();
f2.join();
```

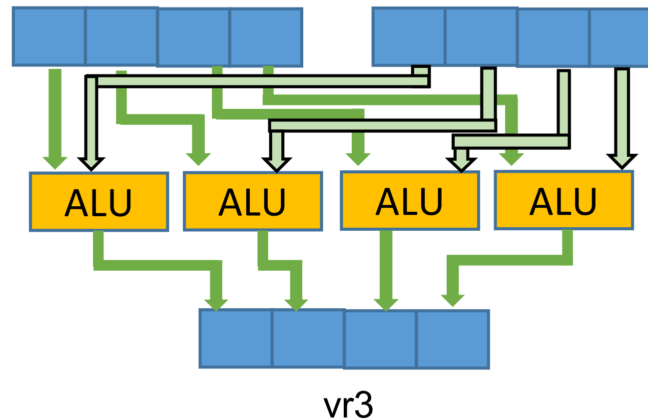
```
std::mutex mtx;
std::condition_variable cnd;
std::string str;

boost::fibers::fiber f1([=]() {
  std::unique_lock<std::mutex> lck(mtx);
  if(str.size() == 0) {
    cnd.wait(lck);
  }
  cout << str << endl;
});

boost::fibers::fiber f2([=]() {
  std::unique_lock<std::mutex> lck(mtx);
  str = “Hello Fiber”;
  cnd.notify_one();
});

f1.join();
f2.join();
```



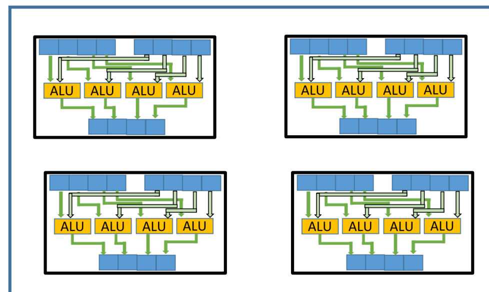
SIMD Vector Units (1/3)



SISD operation on scalars



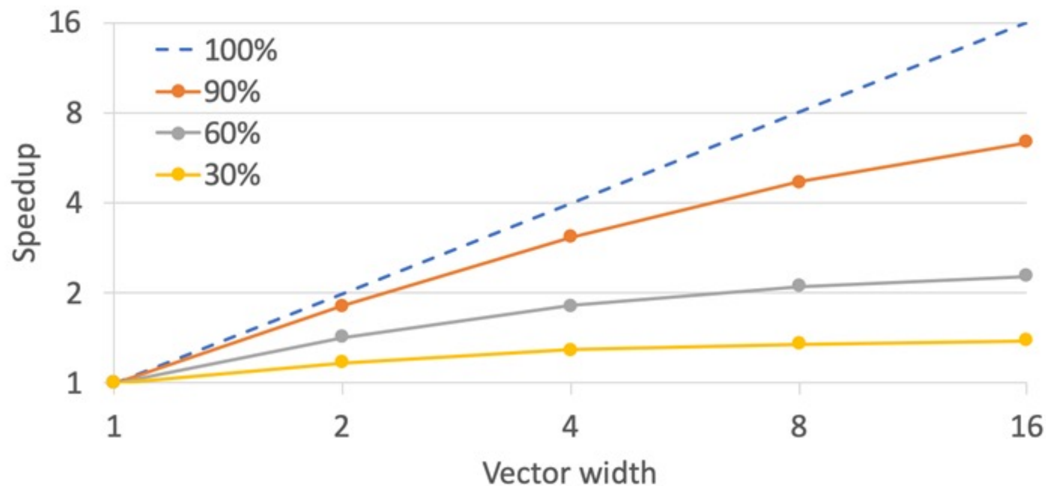
SIMD operation on vectors



Multicore processor supporting SIMD operations

- Special registers that support instructions to operate upon **vectors** than **scalar** values
- SIMD operation is supported on processors by adding more ALUs to each core, and by using wide registers (greater than 32 bit)
- Each core can operate on more than one 32-bit value in each cycle
  - Each core has its own SIMD unit
- Increasing vector register width require adding new instructions

SIMD Vector Units (2/3)



- Assume some work takes “W” time on a scalar CPU
- Time taken on a CPU with vector width “N” for total vectorized fraction “f” available in that work
  - $Time_{scalar} + Time_{vector}$ => (1-f)W + fW/N
- Hence, maximum possible speedup
  - W / {(1-f)W + fW/N} => 1 / { (1-f) + f/N }

Picture source: https://cvw.cac.cornell.edu/vector/performance_amdahl

- Linear speedup is possible only for perfectly parallel code
- The exact upper bound depends significantly on the percentage of code that is vectorized
  - At a vector width of 16, code that is 60% vectorized performs only twice as fast as non-vectorized code
- Sequential or scalar code would limit the performance
  - What about memory access pattern?

SIMD Vector Units (3/3)

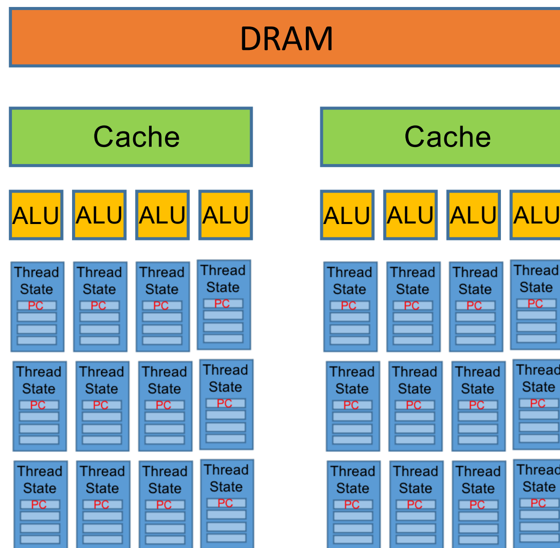
```
double A[1024];

void sum() {
  for (int i=0; i<1024; i++) {
    A[i] = 1.0;
  }
}
```

AVX512

```
#include "vectorclass.h"
int A[1024];
void sum() {
  Vec8d Av(1.0);
  for (int i=0; i<1024; i+=8) {
    Av.store(A+i);
  }
}
```

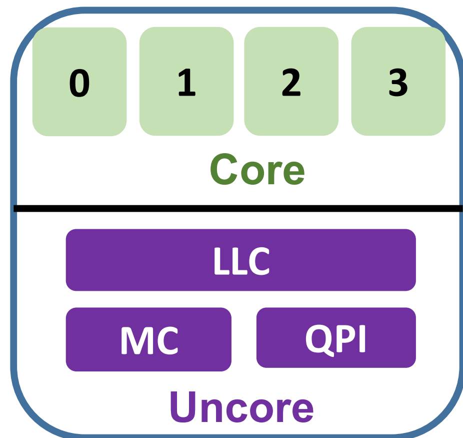
- VCL program compilation
  g++ -std=c++17 -O3 -msse4 -fopt-info-vec -I/path_to/VCL/version2 sum.cpp

Design Goals for a GPU



- Increase the number of hardware threads supported on each core
  - CPU stalls are significantly reduced
  - Improves the performance as the hardware schedule the threads instead of the OS
- Improve the memory bandwidth
  - As large chunks of memory addresses are being fetched from DRAM due to large number of threads
- But, won't these enhancements increase the complexity and cost of the multicore processor?

Power Management



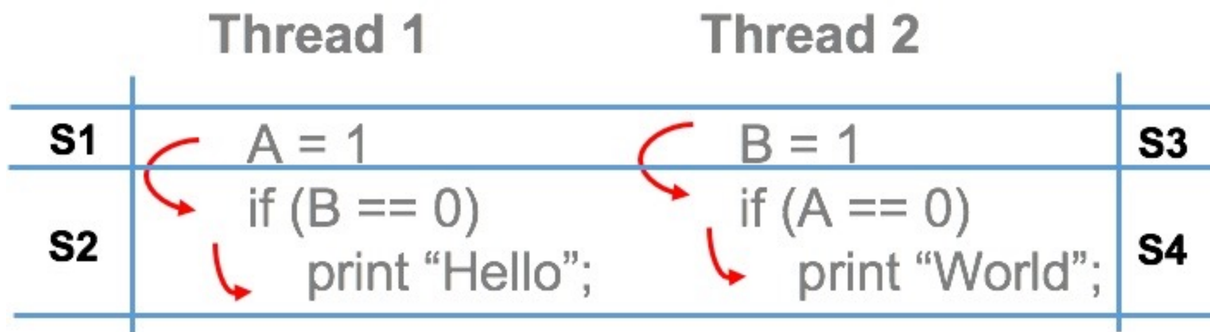
Multicore Processor

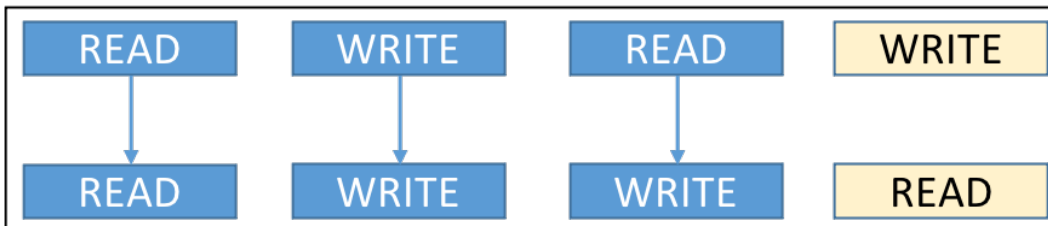
- P-states
  - Dynamic Voltage and Frequency Scaling (DVFS) is used by the processor to operate the core at a specific frequency and voltage
  - Each P-states have an associated frequency
- C-states
  - Power states used by the CPUs to reduce the power at **C**ore level or on a CPU **P**ackage **C**ore level (core, private caches, etc.)
  - Core goes to sleep
    - Used when some of the cores are not being used at all
- Uncore frequency scaling
  - Changes the frequency of the uncore elements in the processor
  - Can be set in the userspace similar to DVFS
  - Currently supported only by Intel processor

Hardware Memory Model (1/2)

- Memory latency continues to limit the performance of multicore processors
  - Several optimizations inside processors for hiding the load/store latency
    - As a side effect of these optimizations, load/store inside a program could be reordered, and hence may not happen in the source code order as expected by programmer
- Memory consistency model defines a set of rules for valid set of reordering of **two different memory accesses**
  - Both compiler and processor can perform reordering

Hardware Memory Model (2/2)

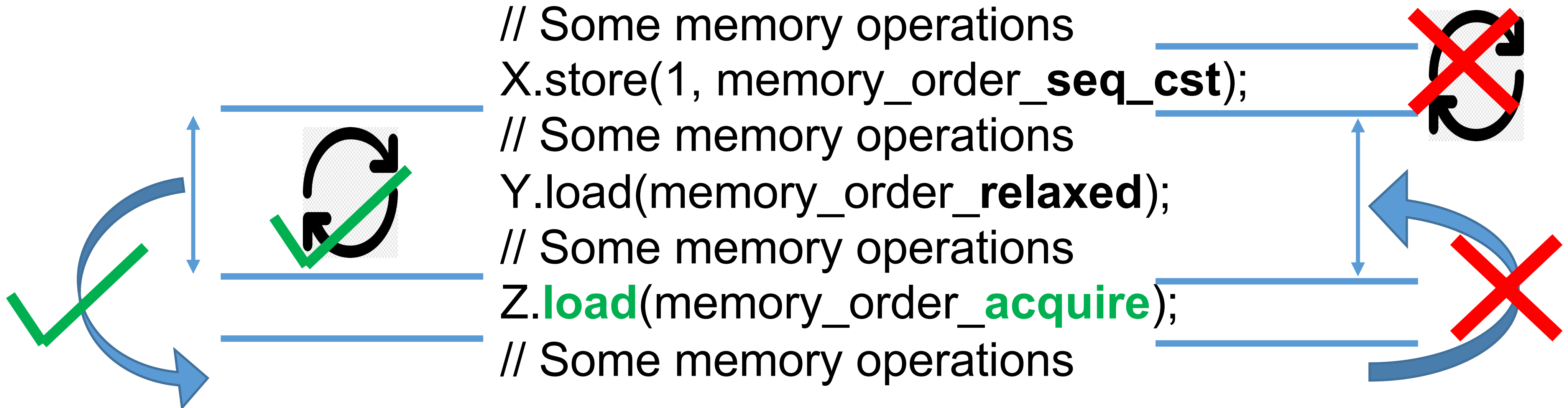




- x86-TSO memory model (Intel/AMD)
  - Write-Read reordering is only allowed
  - Due to presence of store buffers
- Store buffer
  - Temporarily holds write before they are committed to the cache
    - FIFO on x86
  - Size enough for ~50 stores on Intel processor (Skylake)

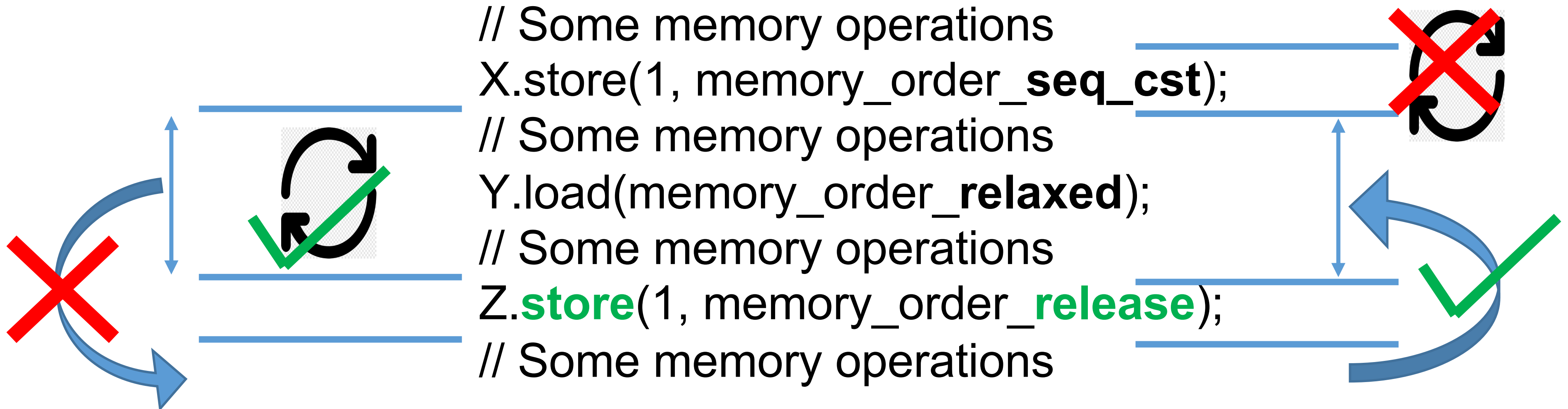
Language Memory Model (1/3)

```
std::atomic<int> X, Y, Z;

// Some memory operations
X.store(1, memory_order_seq_cst);
// Some memory operations
Y.load(memory_order_relaxed);
// Some memory operations
Z.load(memory_order_acquire);
// Some memory operations
```



Language Memory Model (2/3)

std::atomic<int> X, Y, Z;



Language Memory Model (3/3)

```
std::atomic<bool> A(false), B(false);
int non_atomic = 0;
```

Thread 1

```
non_atomic = 10;
A.store(true, memory_order_release);
```

Synchronizes-with

Thread 2

```
if(A.load(memory_order_acquire) == true) {
    B.store(true, memory_order_release);
}
```

Synchronizes-with

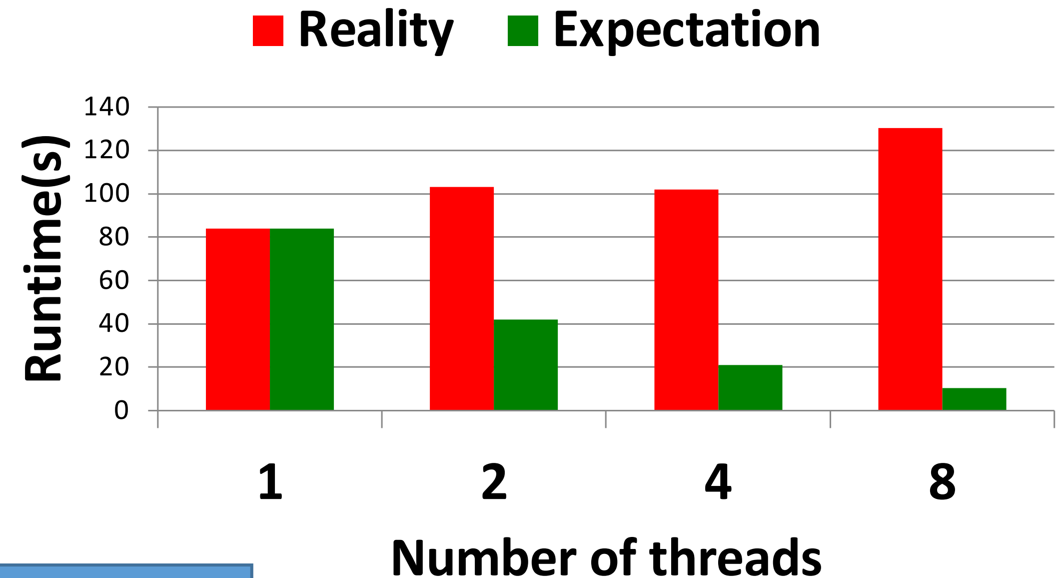
Thread 3

```
if(B.load(memory_order_acquire) == true) {
    assert(non_atomic == 10);
}
```

Acquire/Release ensures synchronization between threads that are storing and loading the same atomic object (also called as half-synchronization)

False Sharing (1/5)

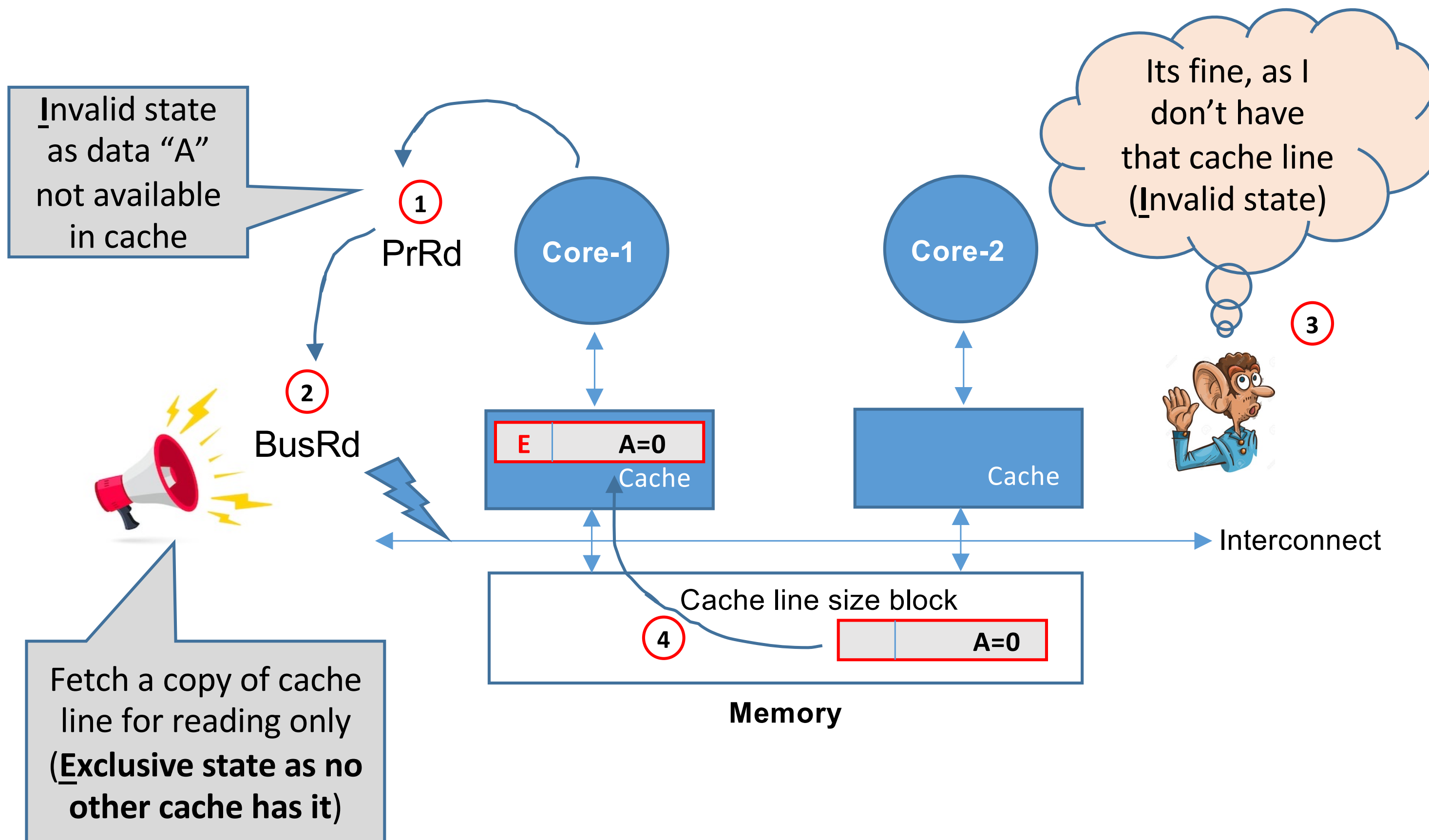
```
int A[8]; //Global array

thread_func(int id) {
   for(i = 0; i < M; i++)
      A[id]++; |++;
}
```



Let's try to understand the problem in this code using the MESI coherence protocol

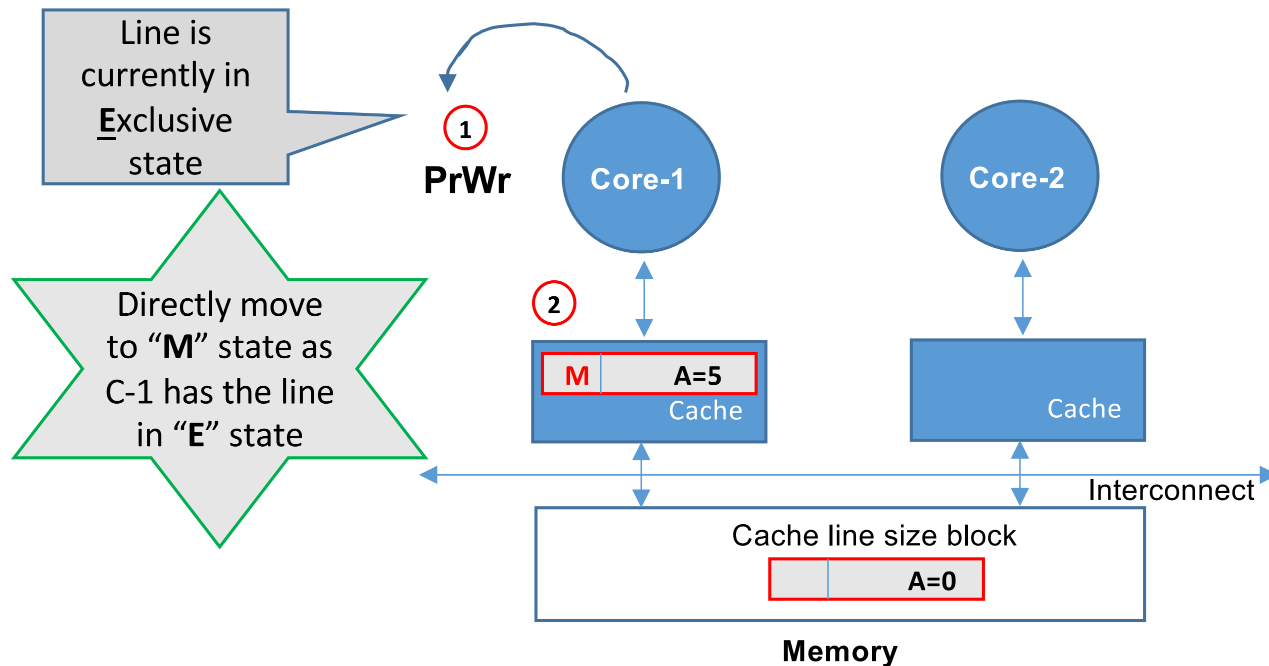
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing (2/5)



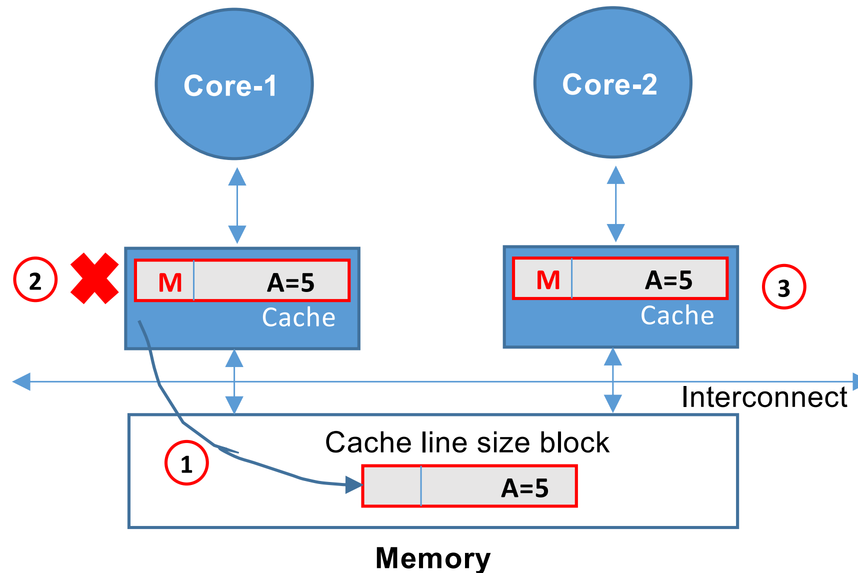
- An additional state **E**
  - Exclusive clean
  - Implies no other cache has a copy of this line

False Sharing (3/5)



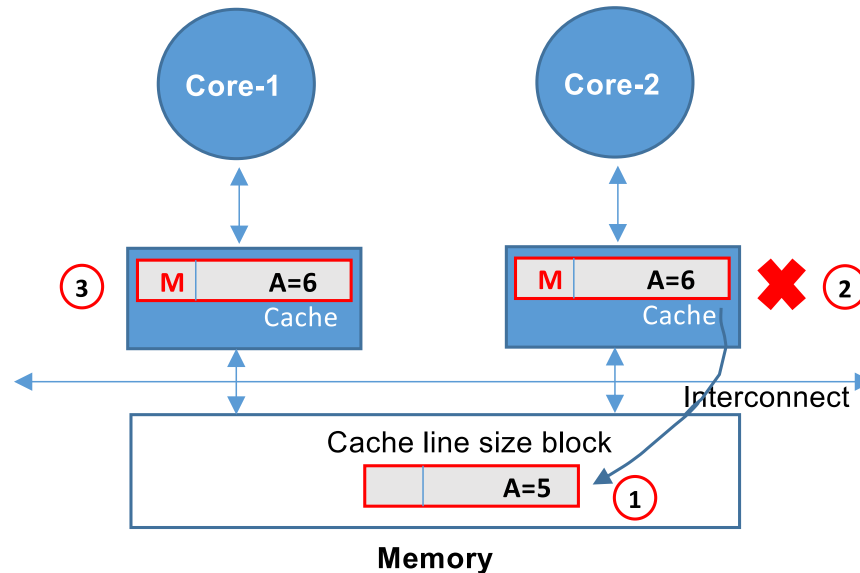
- Moving from **E** to **M** state
  - No action required to be performed on interconnect
  - Present **E** state implies the line is not in any other cache

False Sharing (4/5)



- Core-2 wants to read/write cache line A
  - Write back and invalidate cache line at Core-1

False Sharing (5/5)



- Core-1 wants to read/write cache line A
  - Write back and invalidate cache line at Core-2

End Semester Exam

- End term exam
  - Syllabus: Lectures 02-25
  - Exam pattern similar to midterm exam
    - 3rd May (C12 Old Academic Block)
    - Duration: 1 hour