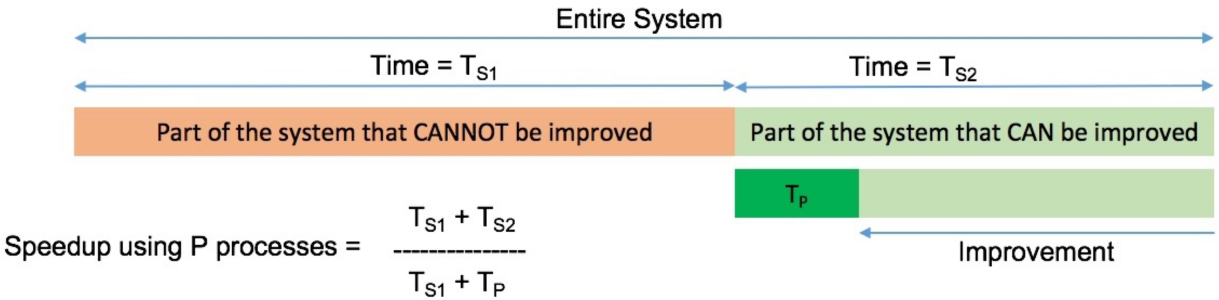
Lecture 14: Mid Semester Review

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

Introduction to Parallel Programming



- Free lunch is now over!
  - Multicore processors everywhere
- Amdahl's law
- Explicit multithreading
- Thread synchronization

```
int main(int argc, char *argv[]) {
  uint64_t result;
  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    uint64_t x, y;
    std::thread T1([&]() { x = fib(n-1); });
    // main can continue executing
    y = fib(n-2);
    // Wait for the thread to terminate.
    T1.join();
    result = x + y;
  }
  cout<<"Fibonacci of ”<<n<<" is "<<result<<endl;
  return 0;
}
```

```
std::mutex _mtx;
std::condition_variable _cv;
bool available=false;

L1: std::unique_lock lck(_mtx);
L2: _cv.wait(lck, [ ]() { return available;});
L3: consume_data();
L4: available=false;
```

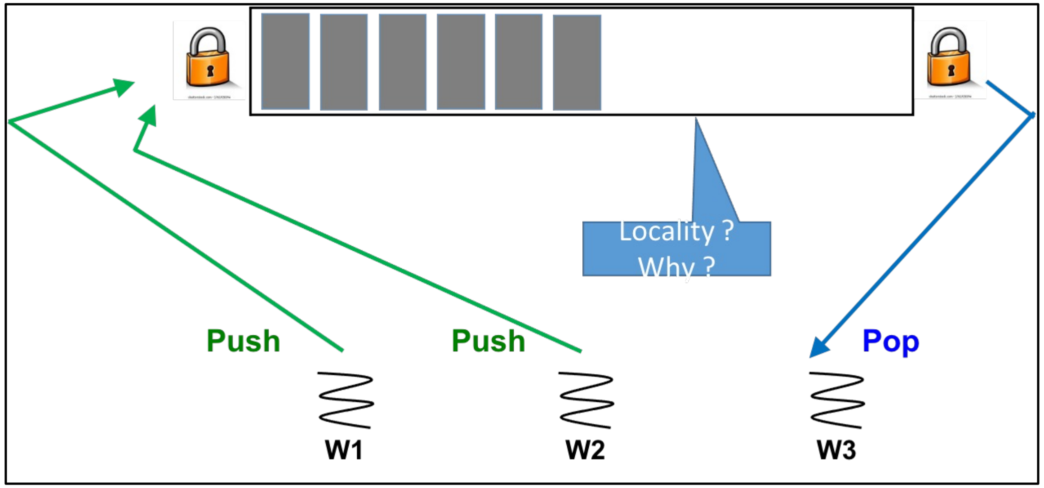
Consumer

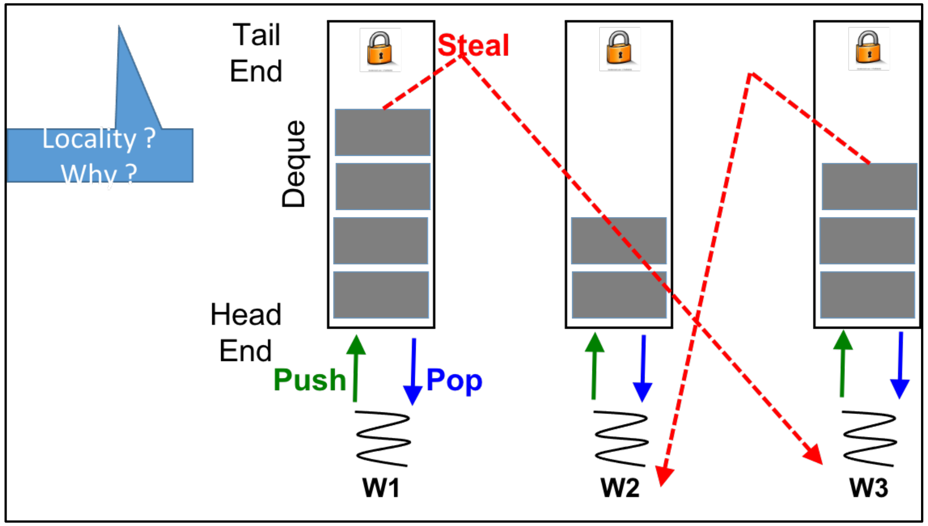
```
L5: std::unique_lock lck(_mtx);
L6: if(!available) {
L7:     create_data();
L8:     available=true;
L9:     _cv.notify_one(lck);
L10: }
```

Producer

Parallel Runtime Systems

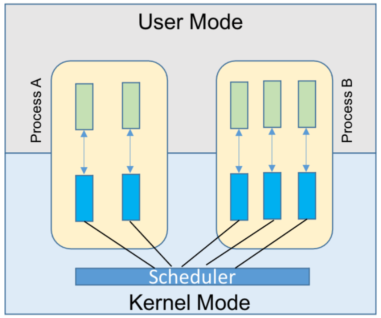
- Parallel runtime system for task-scheduling
  - Work-sharing
  - Work-stealing

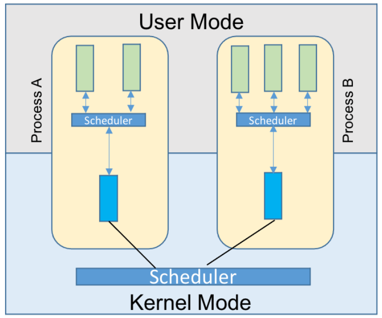


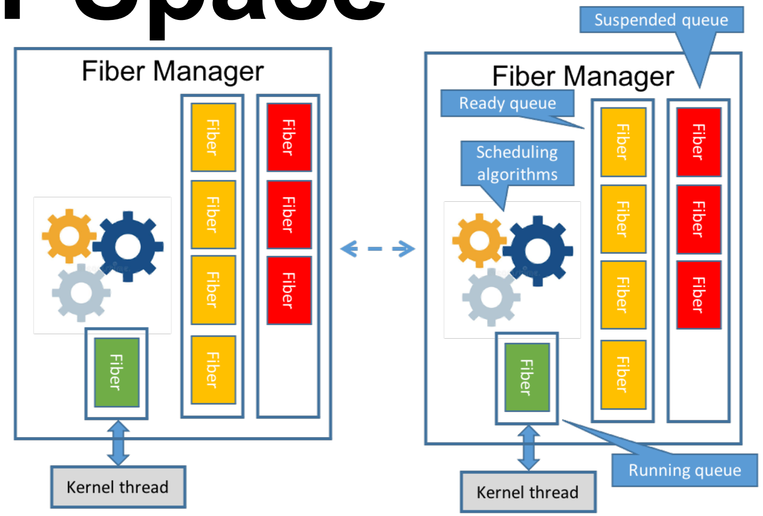


Context Switching in User Space

- User Level Threads v/s Kernel Level Threads
- Boost fibers emulates std::thread operations, but as a ULT instead of a KLT








```
#include <boost/fiber/all.hpp>
#define millisleep(x) boost::this_fiber::sleep_for(std::chrono::milliseconds(a))
.....
boost::fibers::fiber f1 ([=]() { millisleep(500); }); // Fiber F1 launched
boost::fibers::fiber f2 ([=]() { millisleep(100); }); // Fiber F2 launched
f1.join(); // Wait for termination of F1
f2.join(); // Wait for termination of F2
```

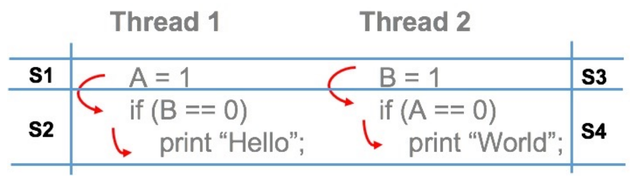
```
boost::fibers::mutex mtx;
boost::fibers::condition_variable cnd;
std::string str;

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

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

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

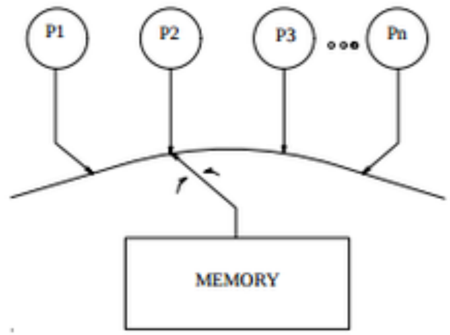
Parallel Runtime Overheads

- Task granularity affects overheads
  - Each async is heap allocated, and also has some metadata associated with it
    - User should choose appropriate task granularity
    - Tasks near the bottom of recursion tree are small computation, where task aggregation could be performed by switching to an iterative version from recursive
- Deque operations are costly
  - For implementing any thread-safe (concurrent) data structure we always have to use some sort of mutual exclusion that avoids the race condition
  - Reducing overheads?
    - Steals are rare
      - Majority of the tasks produced by the victim are consumed by itself
      - Each victim should minimize accessing its "concurrent" deque for push/pop by using a mix of private and shared task pools
        - Push/pop from private pool, but ensure task(s) availability in shared pool to support stealing

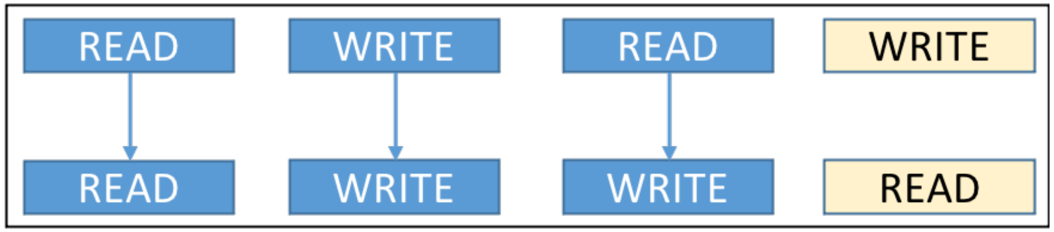
Memory Consistency

- 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
- Sequential consistency is the most primitive form of memory consistency that basically says memory access to any location always happens atomically, and the effect is visible to each and every core
  - Modern programming languages supports sequential consistency only for code block within a mutex lock/unlock operation (Data Race Free)



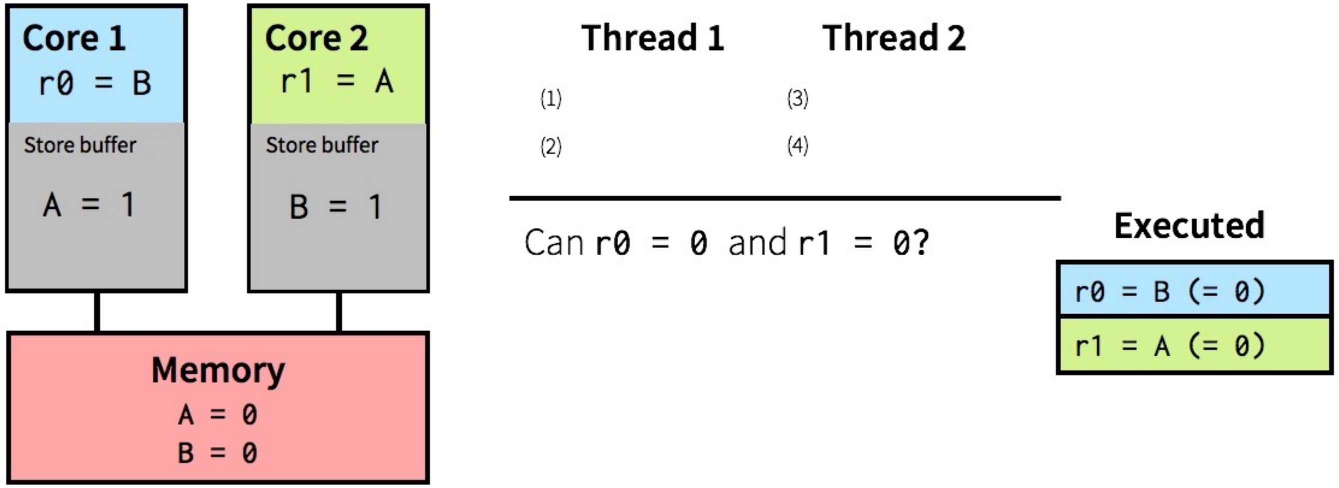


Hardware Memory Model



- x86-TSO memory model (Intel/AMD)
- Store buffer

- **Rule-5**: Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other



Language Memory Model

```
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
Z.store(1, memory_order_release);
```

```
std::atomic<bool> X(false), Y(false);

Thread 1
X.store(true, memory_order_relaxed);
Y.store(true, memory_order_relaxed);

Thread 2
if(Y.load(memory_order_relaxed) == true) {
    assert(X.load(memory_order_relaxed))
}
```

~~Synchronizes-with~~

Not guaranteed

```
std::atomic<bool> Flag(false);

Thread 1
PushTask();
Flag.store(true);

Thread 2
if(Flag.load() == true) {
    PopTask();
}
```

Memory Operations-1 (MO1)

Synchronizes-with

Memory Operations-2 (MO2)

MO1 happens-before MO2

MO1 visible to Thread-2 before it performs MO2

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

Thread 1
non_atomic =10 // Memory Operations MO1
A.store(true, memory_order_release);

Thread 2
if(B.load(memory_order_acquire) == true) {
  // Memory Operations MO2
  assert(non_atomic == 10)
}
```

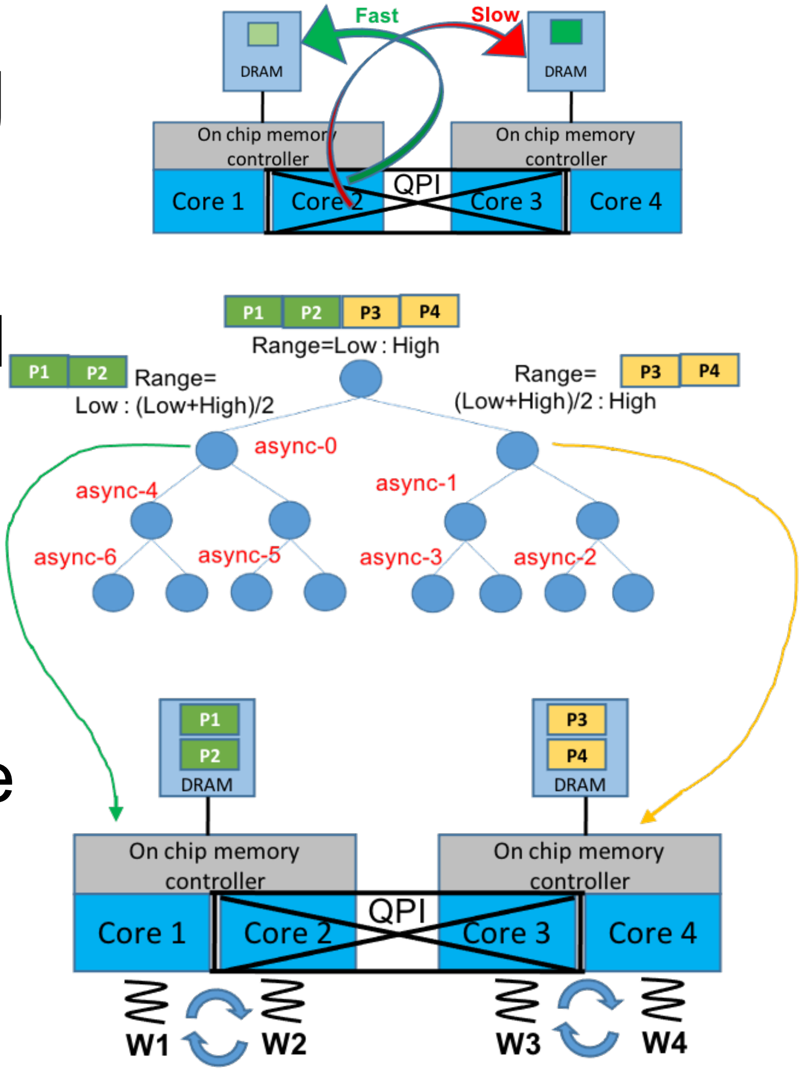
~~Synchronizes-with~~

~~MO1 happens-before MO2~~

~~MO1 visible to Thread-2 before it performs MO2~~

NUMA Aware Work-Stealing

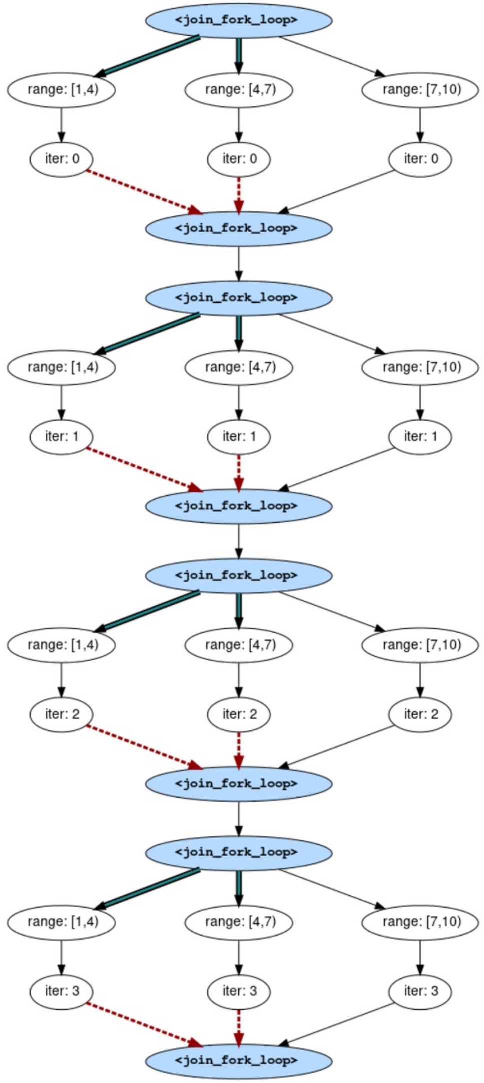
- High performance can be achieved on a NUMA architecture only if the task and its data are collocated, and is local to the worker executing that task
    - By default, Linux uses First-Touch policy for physical page allocation
- Random work-stealing would hurt the locality over NUMA machine due to random victim selection
    - Use hierarchical work-stealing

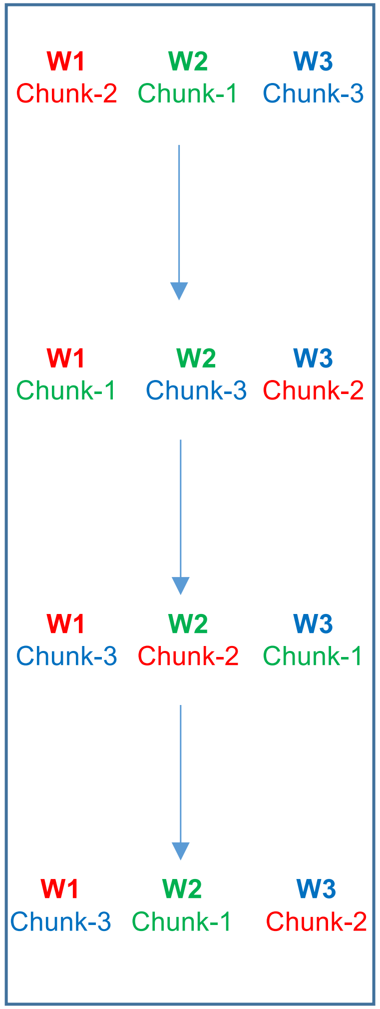


Trace/Replay

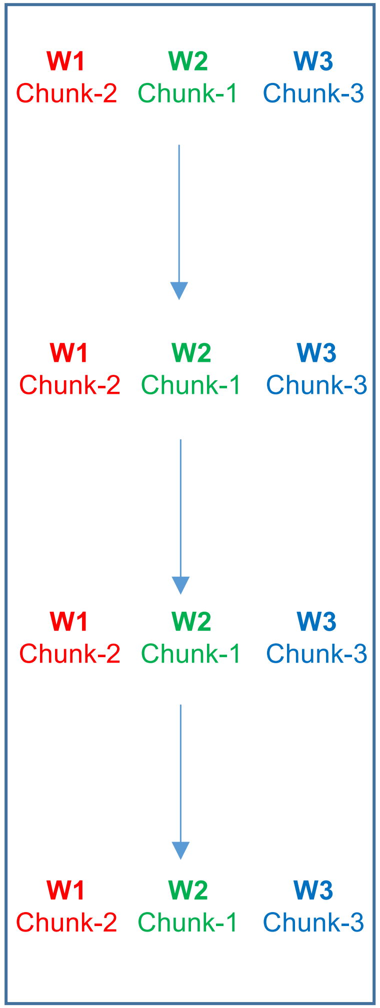
```
double A[SIZE+2], A_shadow[SIZE+2];

void recurse(int low, int high) {
  if((high - low) > THRESHOLD) {
    int mid = (high+low)/2;
    future<void> f1 = async([=]() { recurse(low, mid); });
    recurse(mid, high);
    f1.get();
  } else {
    for(int j=low; j<high; j++) {
      A_shadow[j] = (A[j-1] + A[j+1])/2.0;
    }
  }
}

void compute(int MAX_ITERS) {
  for(int i=0; i<MAX_ITERS; i++) {
    recurse(1, SIZE+1);
    double* temp = A_shadow;
    A_shadow = A;
    A = temp;
  }
}
```

- Improved locality if each workers executes the exact same set of tasks in each for loop iteration of compute
- Trace/Replay for improving locality
  - Trace (i.e., record) the tasks executed by each worker during the first iteration of for loop inside compute
  - For the rest of iterations of the above for loop of compute, disable random work-stealing and use the information gathered during the Trace (i.e., record) phase to replay the exact set of tasks at each worker





Random work-stealing



Trace/Replay

Midterm Exams

- Midterm exam will be held on 16/10/22 (Sunday) in Room no. C21 old-academic block from 10am—11am
  - Total weightage is 10%
  - It is your responsibility to arrive on time. No extra time if you arrive late
  - Closed-notes, closed-book, closed-laptop written exam
  - Syllabus includes Lectures 2–13
  - No penalty for **minor** syntax errors in programming related questions. Minor syntax errors only include **missing semicolon, missing braces, and spell mistakes**.
    - However, you must ensure that your program is: a) clear to understand, and b) has proper indentation. If these two perquisites are not met, then the marks allocated will be final and reevaluation requests will not be entertained