Lecture 07: Controlling Task Granularity

Vivek Kumar

Computer Science and Engineering

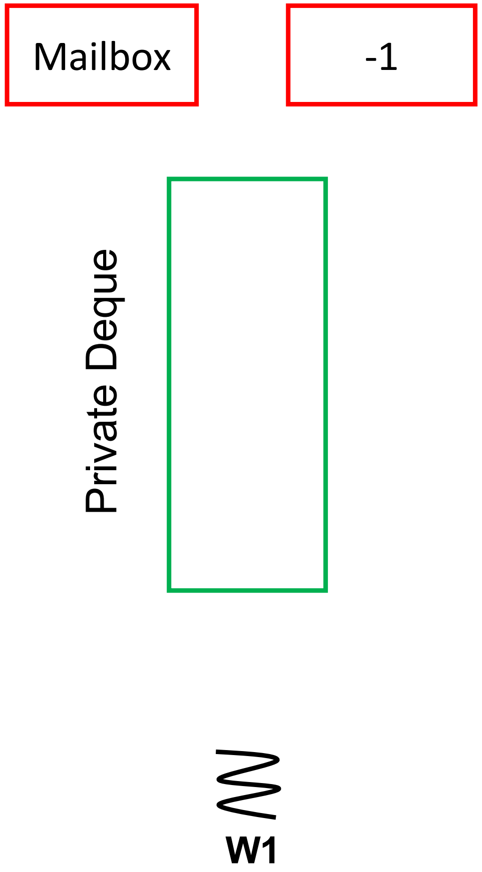
IIIT Delhi

vivekk@iiitd.ac.in

Today's Class

- Alternative deques (contd.)
- Automatic task granularity control
- Quiz-1

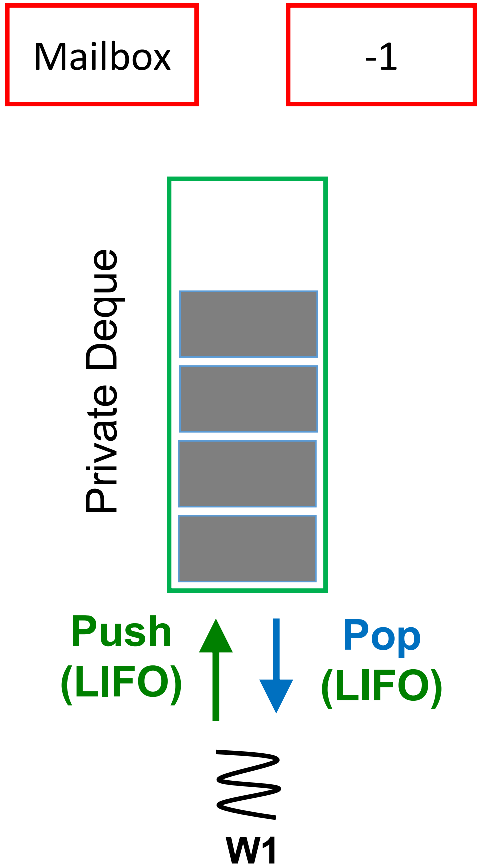
Reducing Concurrent Access: Using Private Deque



- Every worker maintains three data structures
  - A non-concurrent private deque
    - Same as the default deque, but without the support for concurrent (thread-safe) accesses
  - One mailbox
    - That can store one or more tasks
    - Contains a counter indicating total number of stored tasks
  - One shared counter

Paper: https://hal.inria.fr/hal-00863028/document

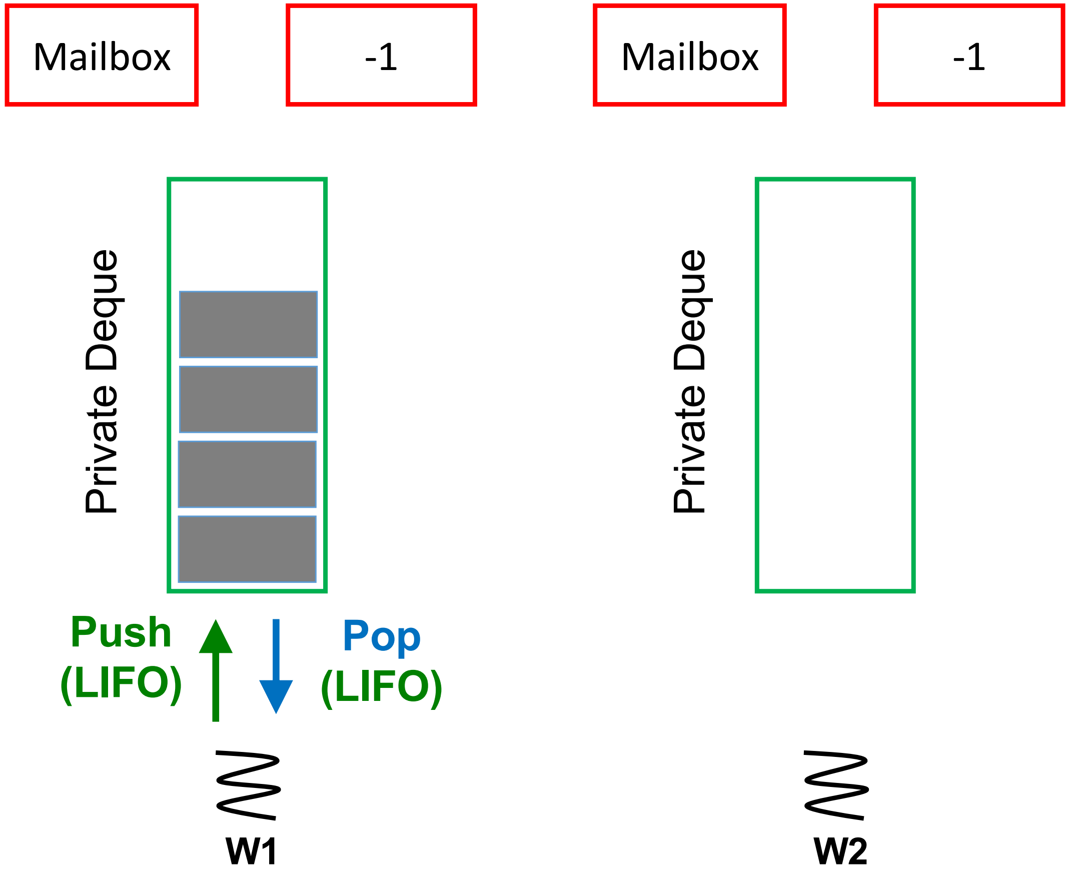
Reducing Deque Access: Using Private Deque



- Victim
  - Push/pop tasks into its private deque

Paper: https://hal.inria.fr/hal-00863028/document

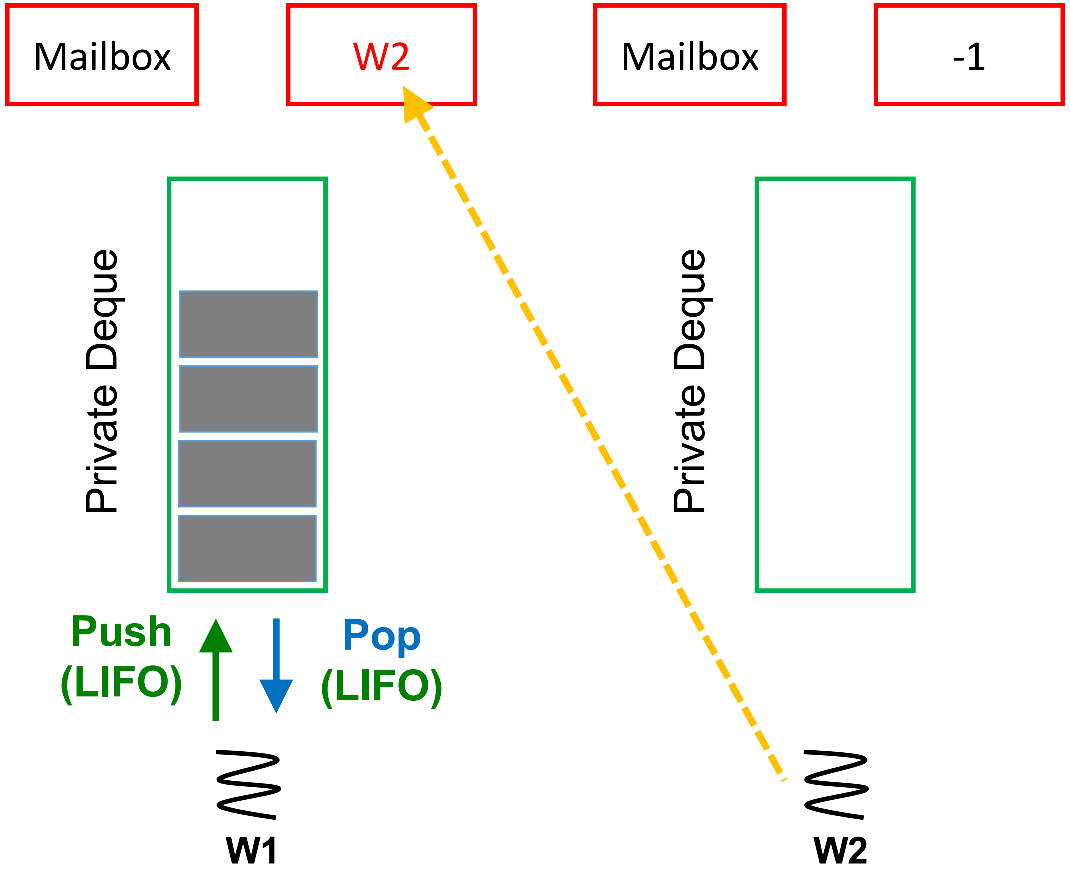
Reducing Deque Access: Using Private Deque



- Thief
  - Selects a random victim (W1) who has items on its deque
  - Checks victim deque size without any locks

Paper: https://hal.inria.fr/hal-00863028/document

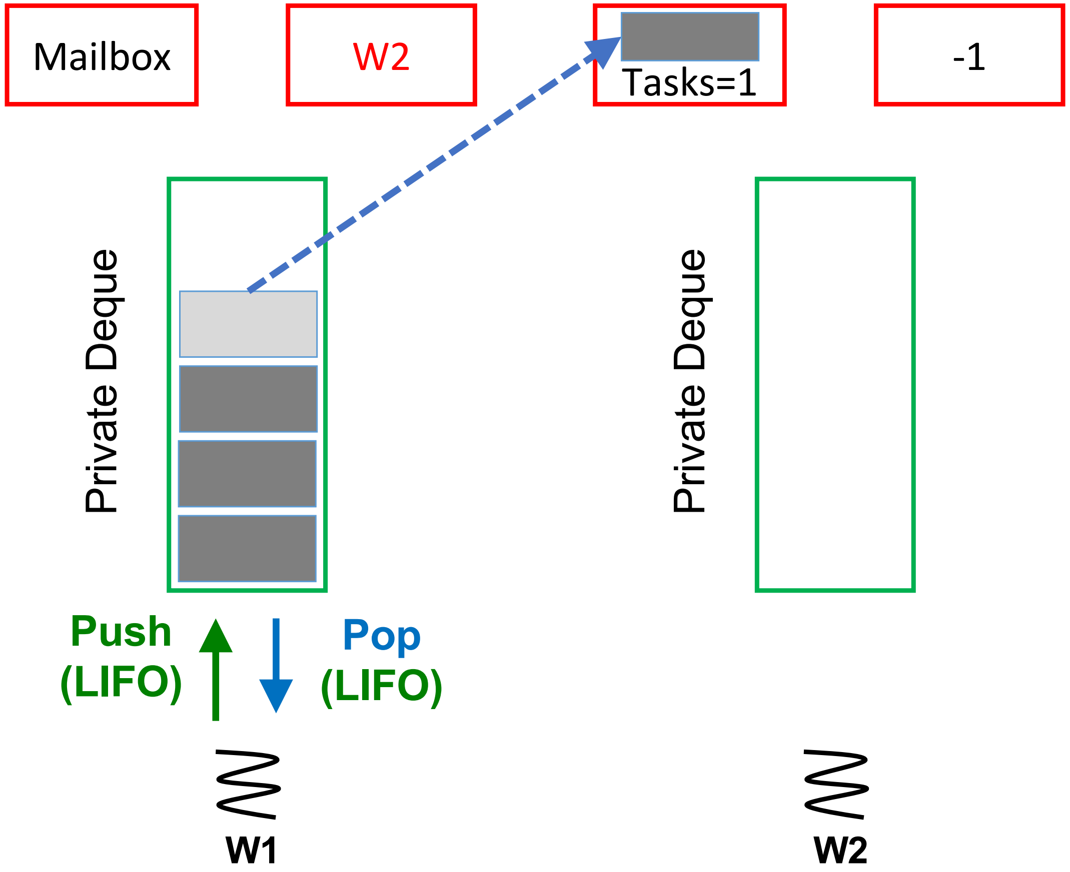
Reducing Deque Access: Using Private Deque



- Thief
  - Record its own id inside the request box at W1 (critical section), and goes inside condition wait
  - Only one thief at a time

Paper: https://hal.inria.fr/hal-00863028/document

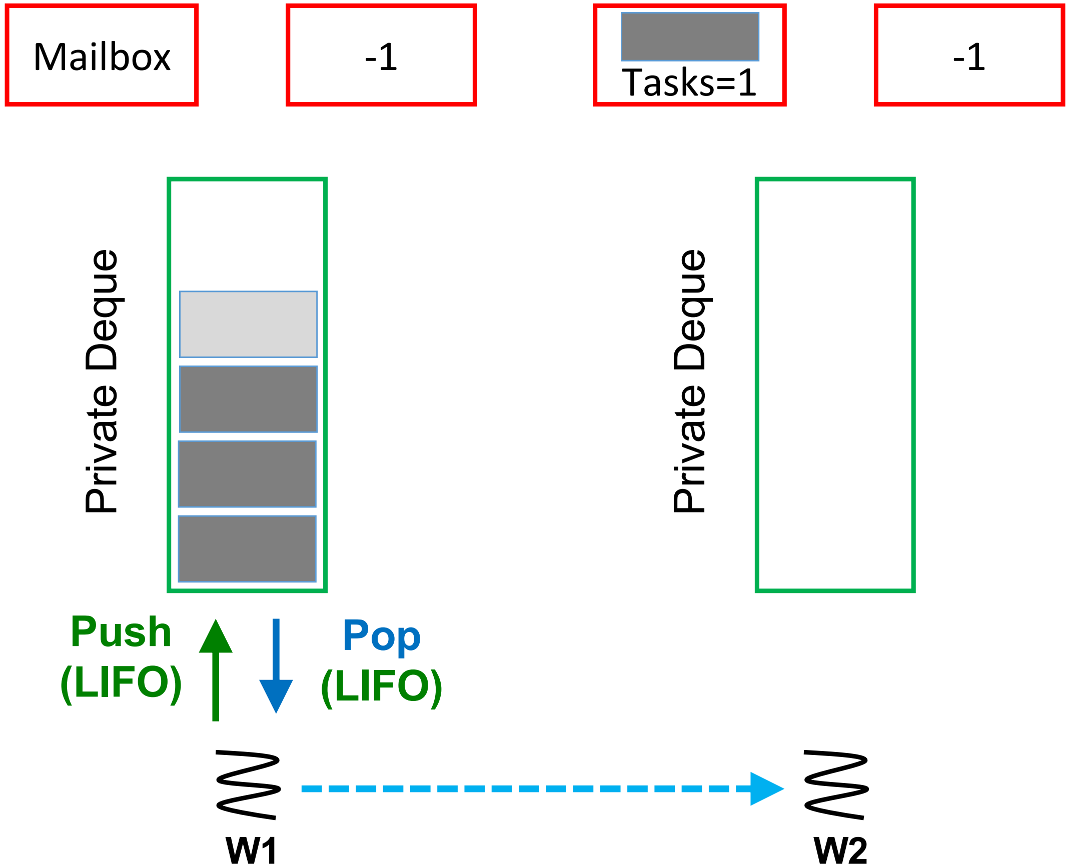
Reducing Deque Access: Using Private Deque



- Victim
  - Check its request box inside each push/pop/steal
  - If tasks are available on victim's private deque
    - Pop item(s) from the head and copies it into the waiting thief's mailbox (W2)
    - Update W2's mailbox with the total number of tasks copied

Paper: https://hal.inria.fr/hal-00863028/document

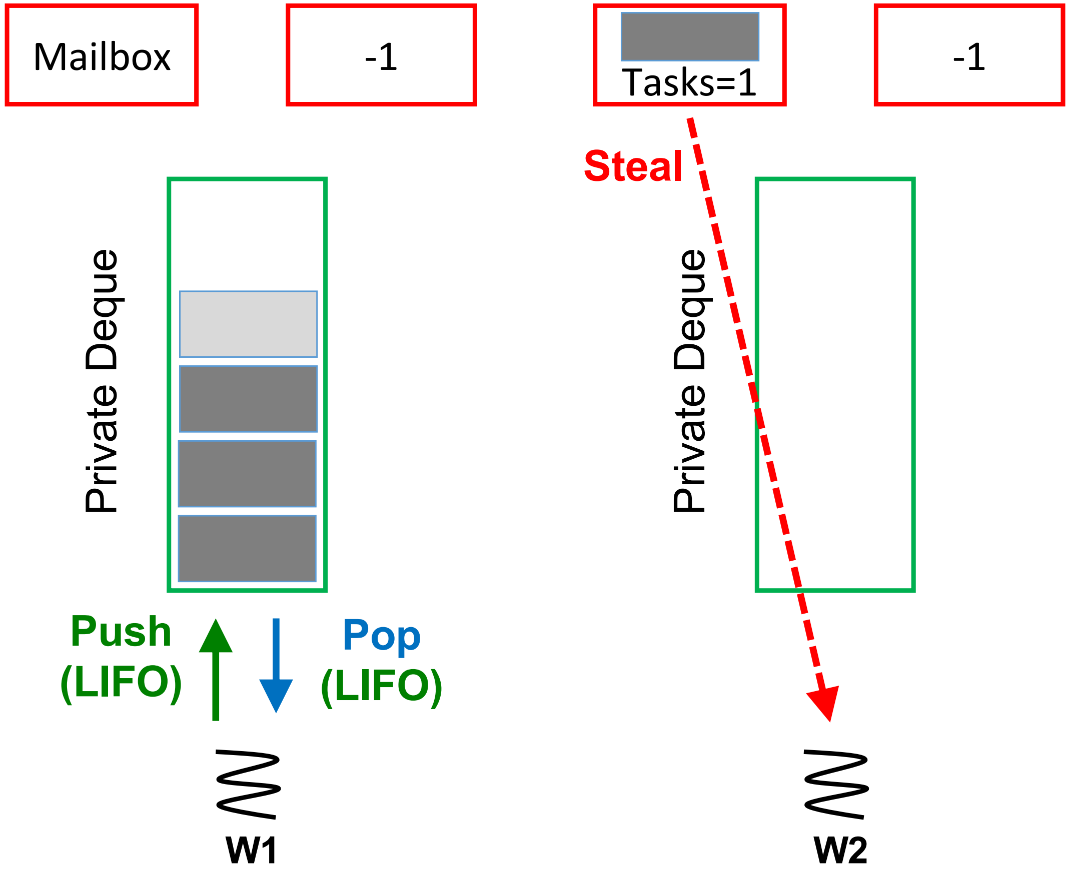
Reducing Deque Access: Using Private Deque



- Victim
  - Clears its request box
  - Signals the waiting thief W2

Paper: https://hal.inria.fr/hal-00863028/document

Reducing Deque Access: Using Private Deque



- Thief
  - Unblocks after being notified by W1
  - Steal tasks from its mailbox and start executing them
    - If more than one task received then extra tasks pushed to its private deque
  - Failed steal attempt if it did not receive any task (i.e., if W1 ran out of tasks)

Paper: https://hal.inria.fr/hal-00863028/document

Private Deque using Argobots?

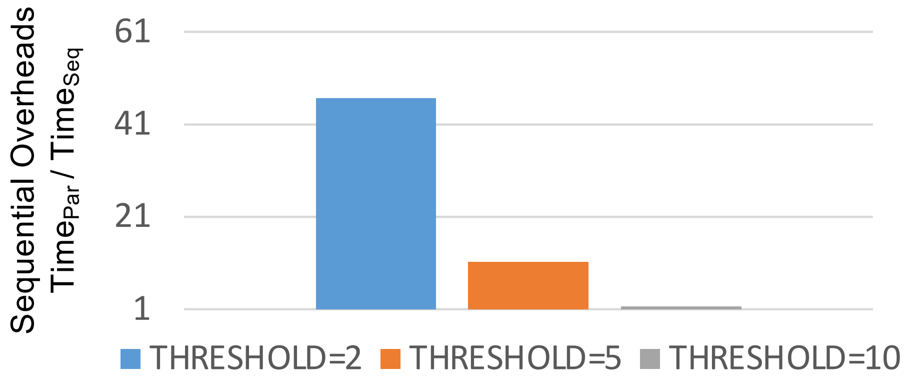
- You will have to create a custom scheduler instead of using the inbuilt work-stealing scheduler
  - See an example: https://github.com/pmodels/argobots/blob/main/examples/scheduling/sched_and_pool_user.c

Today's Class

- Alternative deques (contd.)
- Automatic task granularity control
- Quiz-1

Task Granularity Affects Execution

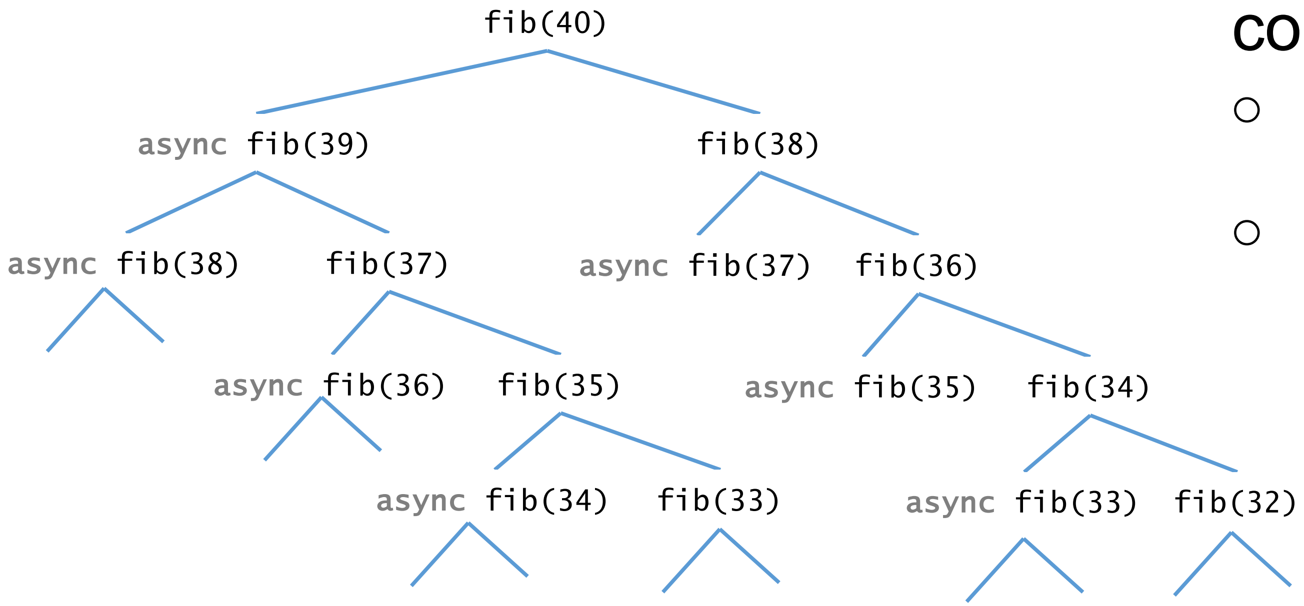
```
uint64_t fib_seq(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    return fib_seq(n-1) + fib_seq(n-2);
  }
}

uint64_t fib(uint64_t n) {
  if(n<THRESHOLD) {
    return fib_seq(n);
  } else {
    std::future<uint64_t> f1 = std::async([=](){ return fib(n-1); });
    std::future<uint64_t> f2 = std::async([=](){ return fib(n-2); });
    //get will block until result is ready
    return f1.get() + f2.get();
  }
}
```



- We know concurrent deques have overheads, but if we want to continue using the concurrent deques, then how can we avoid sequential overheads?
  - By controlling task granularity
    - Neither too many tasks, nor too few!
- Options to control task granularity?
  1. Calculate Task-2 (fib of n-2) sequentially
  2. Don't create async tasks when N is less than certain threshold
     - What threshold is optimal?
  3. Use memoization
     - Is it possible for all parallel programs?

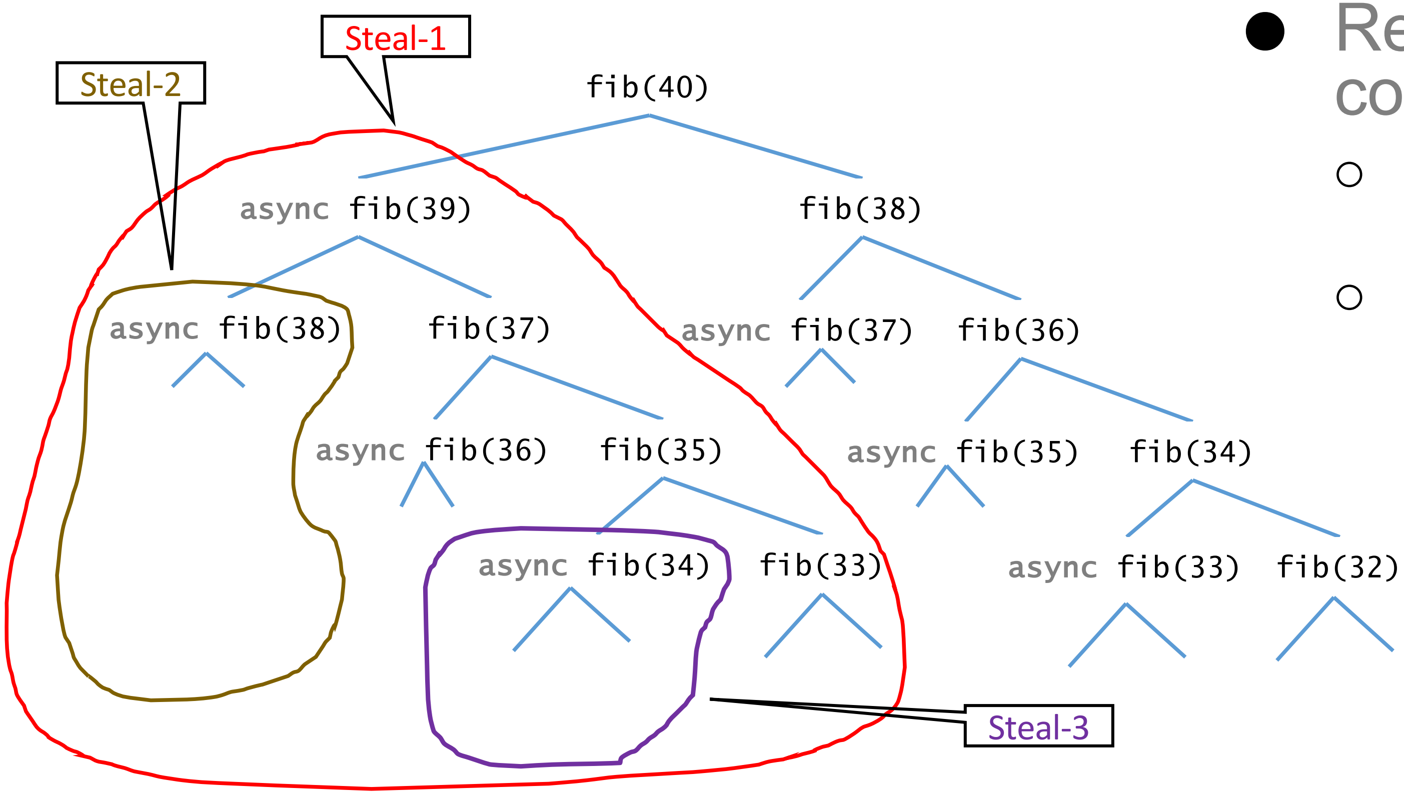
Running parallel recursive parallel Fib(40) using HClib as its async won't launch thread unlike std::async

First and Foremost, Work-Stealing is Best Suited for What Kind of Applications?



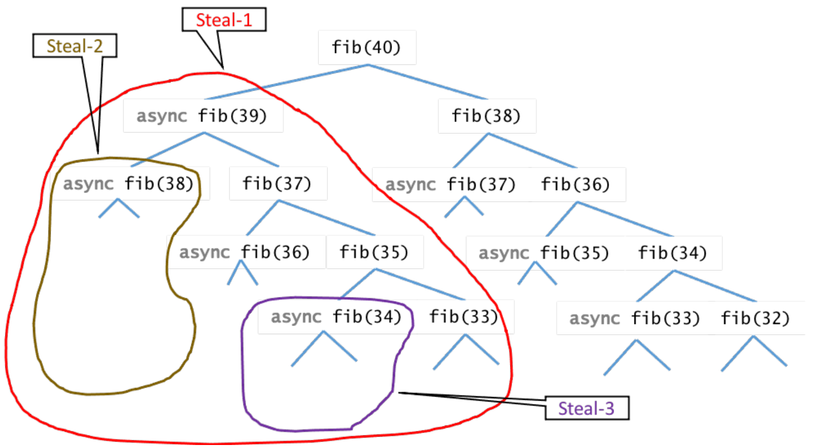
- Recursive divide-and-conquer style
  - Leads to fine granular task creation
  - How its helpful?
    1. Nested task creation

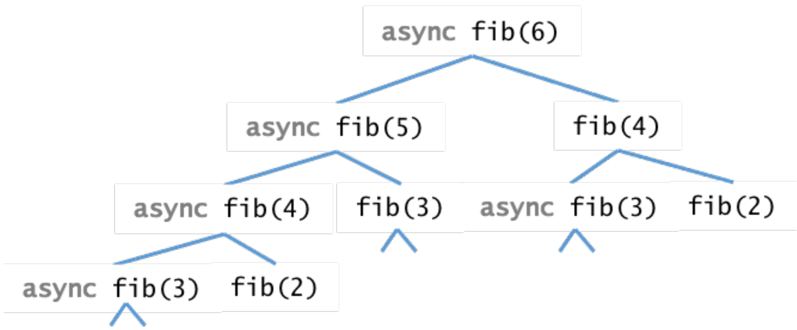
First and Foremost, Work-Stealing is Best Suited for What Kind of Applications?



- Recursive divide-and-conquer style
  - Leads to fine granular task creation
  - How its helpful?
    1. Nested task creation
    2. Stealing an async will eventually give birth to several new asyncs at the thief
       - It will keep the thief busy and reduce steal attempts

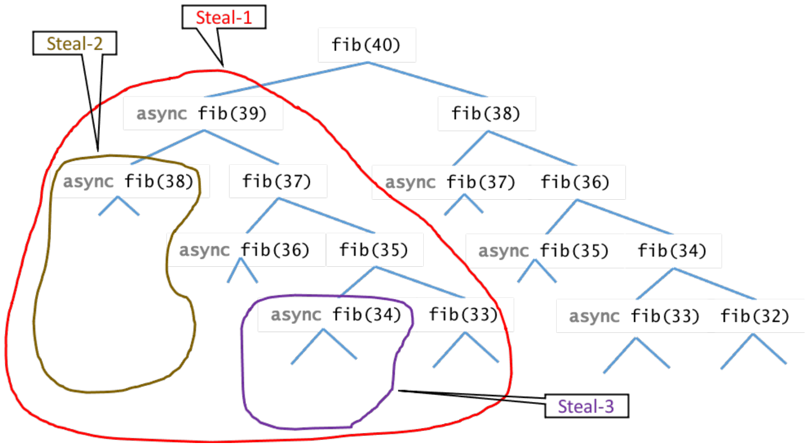
First and Foremost, Work-Stealing is Best Suited for What Kind of Applications?

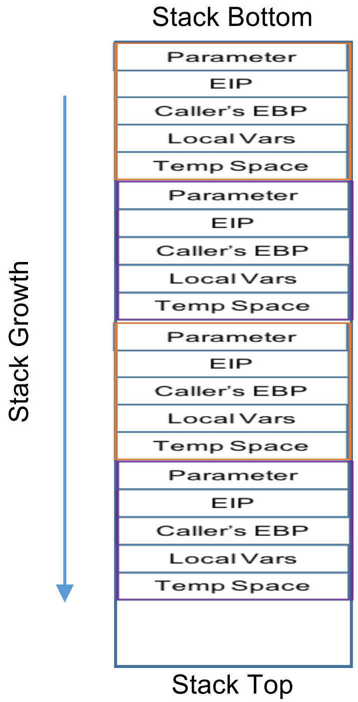




- Recursive divide-and-conquer style
  - Leads to fine granular task creation
  - **Disadvantages**?
    1. Tasks created near the bottom of the tree are too small in computation, and wouldn't be able to keep a thief busy once stolen

First and Foremost, Work-Stealing is Best Suited for What Kind of Applications?





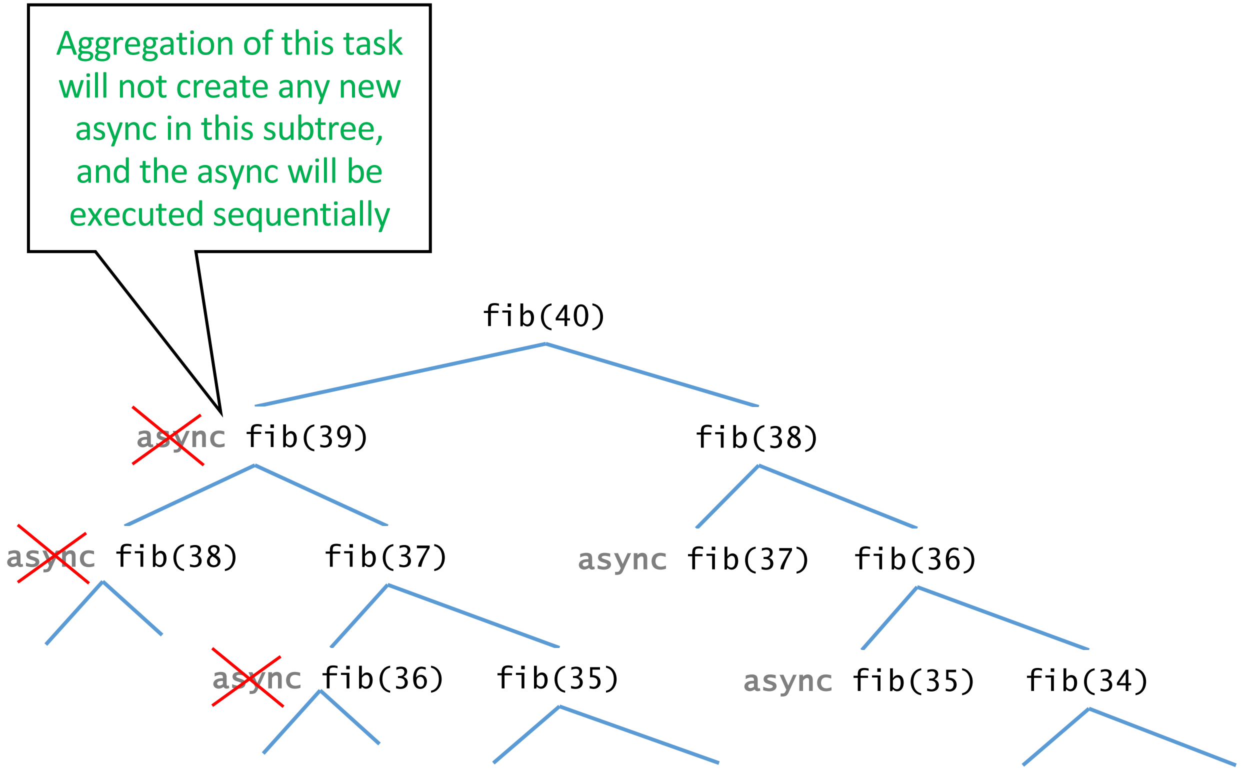
- Recursive divide-and-conquer style
  - Leads to fine granular task creation
  - **Disadvantages**?
    1. Tasks created near the bottom of the tree are too small in computation, and wouldn't be able to keep a thief busy once stolen
    2. Thread stack too deep
       - Too many context switches for moving back and forth between caller and callee stack frames (although in user space)
       - Too many context switches for moving back and forth between caller and callee stack frames (although in user space)

How to Avoid Those Disadvantages

1. Tasks near the bottom of the tree are small computations
   - Automatic granularity control
     - Stop creating new async after some "**depth**" is reached
     - Async created after that "depth" is executed sequentially
2. Deep thread stack due to recursion
   - Using two versions of the parallel code
     - Convert recursion into iterative call after appropriate "depth"

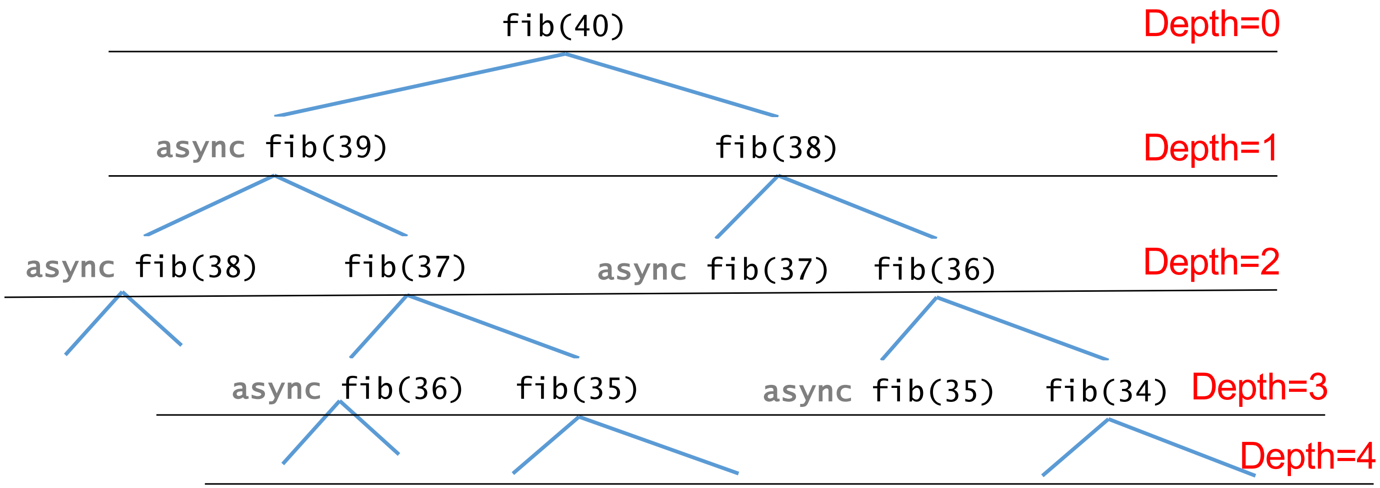
Solution-1: Automatic Granularity Control

- Runtime can perform dynamic task aggregations

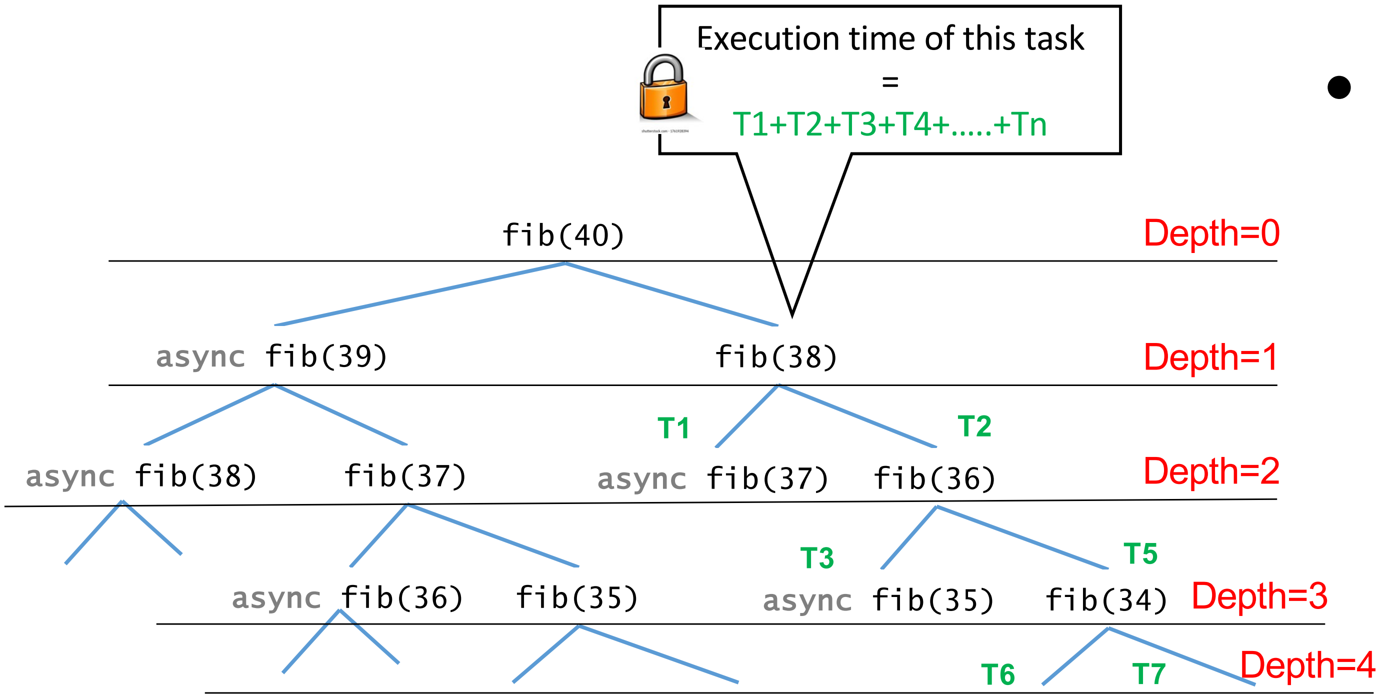


Solution-1: Automatic Granularity Control

- Runtime can perform dynamic task aggregations
- Each task keeps track of its depth in the recursion tree, and its execution time
  - Depth is stored locally inside the task



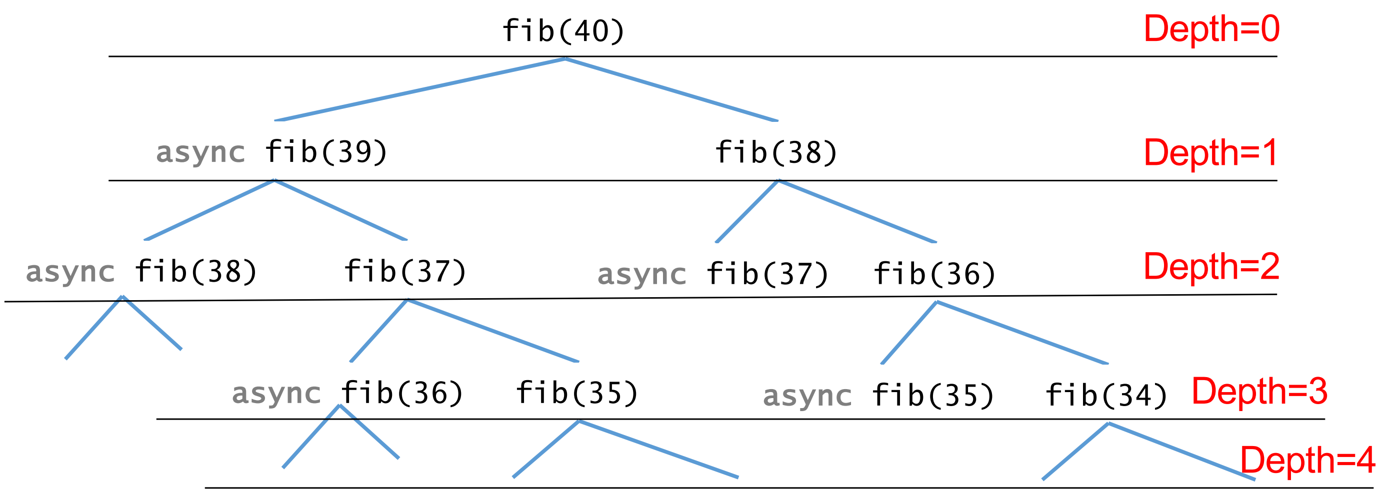
Solution-1: Automatic Granularity Control



- Runtime can perform dynamic task aggregations
- Each task keeps track of its depth in the recursion tree, and its execution time
  - Depth is stored locally inside the task
- Whenever a task complete its execution, it does two things
  - It add its execution time to the parent task's execution time
    - Mutual exclusion required

Solution-1: Automatic Granularity Control

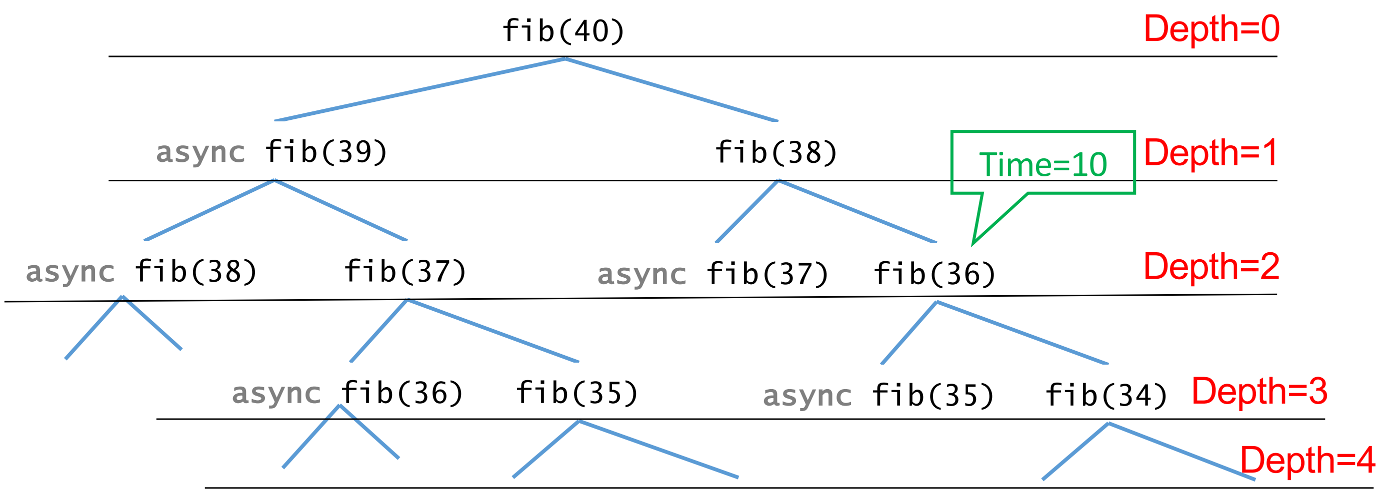
Key=0 Value=0	Key=1 Value=0	Key=2 Value=0	Key=3 Value=0	



- Runtime can perform dynamic task aggregations
- Each task keeps track of its depth in the recursion tree, and its execution time
  - Depth is stored locally inside the task
- Whenever a task complete its execution, it does two things
  - It add its execution time to the parent task's execution time
    - Mutual exclusion required
  - Update the execution time at its depth in a shared global hash map (key=depth, value=time)

Solution-1: Automatic Granularity Control

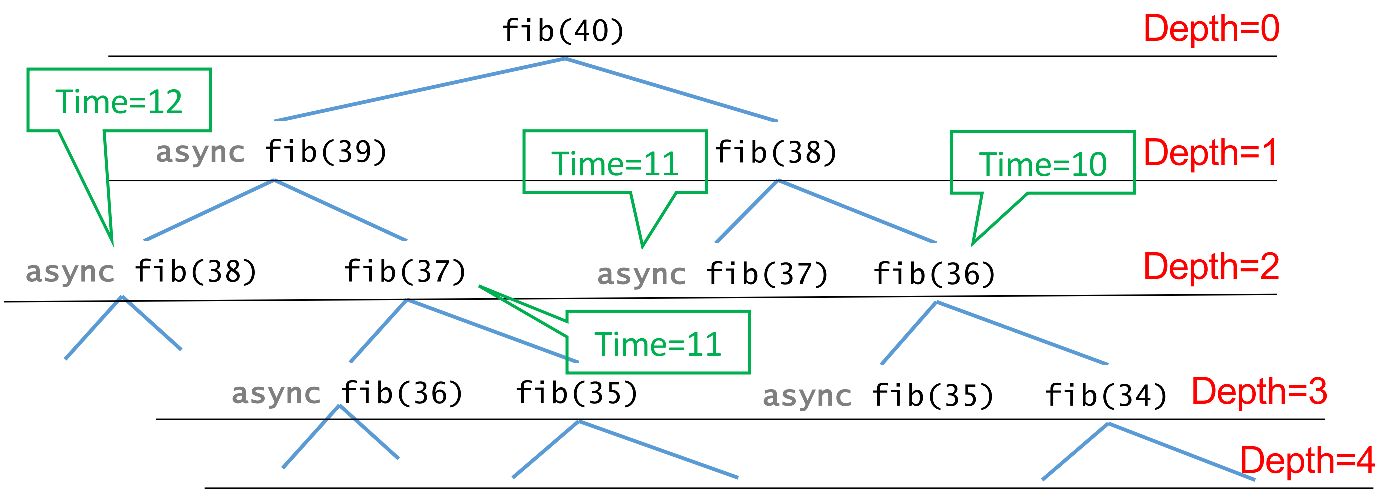
Key=0 Value=0	Key=1 Value=0	Key=2 Value=10	Key=3 Value=0	



- Runtime can perform dynamic task aggregations
- Each task keeps track of its depth in the recursion tree, and its execution time
  - Depth is stored locally inside the task
- Whenever a task complete its execution, it does two things
  - It add its execution time to the parent task's execution time
    - Mutual exclusion required
  - Update the execution time at its depth in a shared global hash map (key=depth, value=time)

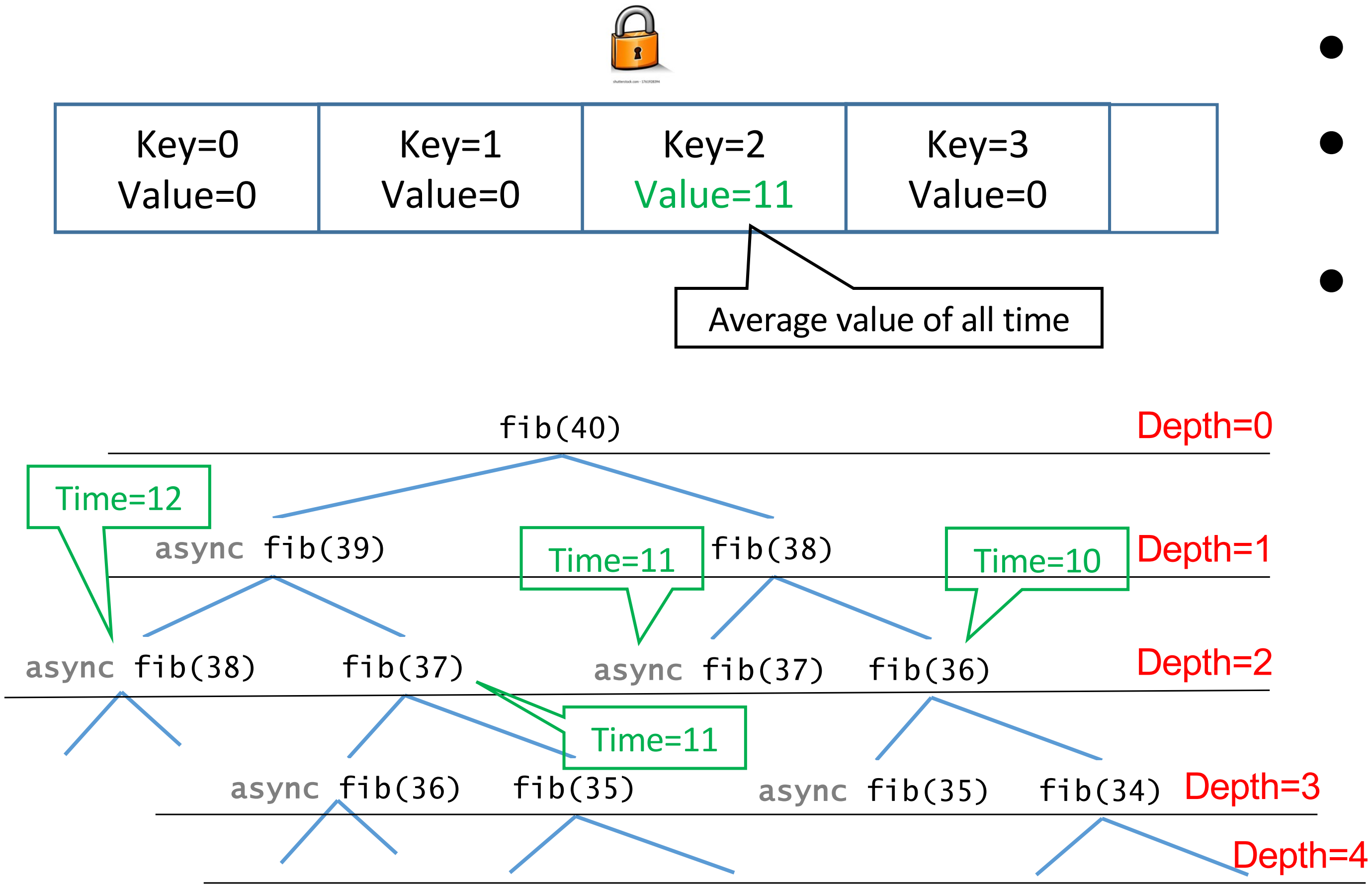
Solution-1: Automatic Granularity Control

Key=0	Key=1	Key=2	Key=3	
Value=0	Value=0	Value=10	Value=0	



- Runtime can perform dynamic task aggregations
- Each task keeps track of its depth in the recursion tree, and its execution time
  - Depth is stored locally inside the task
- Whenever a task complete its execution, it does two things
  - It add its execution time to the parent task's execution time
    - Mutual exclusion required
  - Update the execution time at its depth in a shared global hash map (key=depth, value=time)
    - Averaging of value (time) for a given key (depth) when more than one tasks complete its execution
    - Averaging would be stopped after enough samples collected at a depth

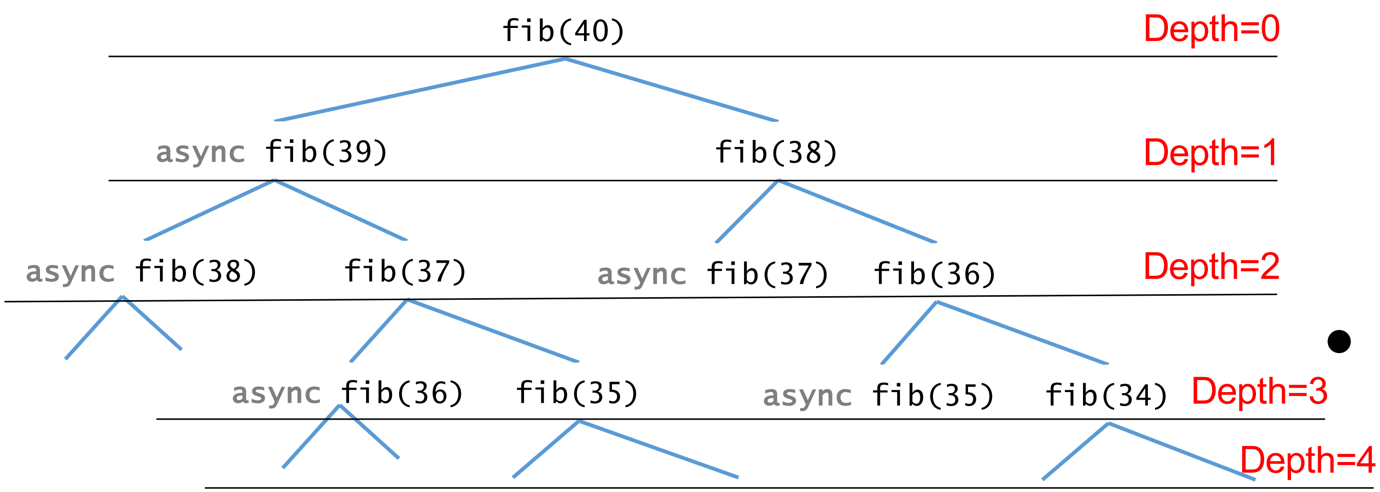
Solution-1: Automatic Granularity Control



- Runtime can perform dynamic task aggregations
- Each task keeps track of its depth in the recursion tree, and its execution time
  - Depth is stored locally inside the task
- Whenever a task complete its execution, it does two things
  - It add its execution time to the parent task's execution time
    - Mutual exclusion required
  - Update the execution time at its depth in a shared global hash map (key=depth, value=time)
    - Averaging of value (time) for a given key (depth) when more than one tasks complete its execution
    - Averaging would be stopped after enough samples collected at a depth

Solution-1: Automatic Granularity Control

Key=0 Value=14	Key=1 Value=12	Key=2 Value=11	Key=3 Value=9	



- Runtime can perform dynamic task aggregations
- Each task keeps track of its depth in the recursion tree, and its execution time
  - Depth is stored locally inside the task
- Whenever a task complete its execution, it does two things
  - It add its execution time to the parent task's execution time
    - Mutual exclusion required
  - Update the execution time at its depth in a shared global hash map (key=depth, value=time)
    - Averaging of value (time) for a given key (depth) when more than one tasks complete its execution
    - Averaging would be stopped after enough samples collected at a depth
- Depth threshold decided based on the execution time of tasks at each depth
  - Beyond this depth threshold tasks would be aggregated

Solution-2: Using Two Versions of the Code

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}
```

```
uint64_t fib(uint64_t n) {
  uint64_t f1=1;
  uint64_t f2=1;
  uint64_t m=2;
  while(m < n) {
    uint64_t temp = f2+f1;
    f1=f2;
    f2=temp;
    m=m+1;
  }
  return f2;
}
```

- When depth threshold is reached, switch to an iterative version of the recursive algorithm
  - Most of the recursive algorithms can be converted into iterative algorithm
  - Although, asking the user to provide an iterative version is breaking the support for serial elision

There is a general format for converting tail recursion into iterative version: https://www.baeldung.com/cs/convert-recursion-to-iteration

Reading Materials

- Private deques
  - https://hal.inria.fr/hal-00863028/document
- Automatic granularity control
  - An adaptive cut-off for task parallelism, SC 2008
    - https://www.academia.edu/download/35796885/1234120839604__a36-duran.pdf
- Using multiple versions of the code
  - A static cut-off for task parallel programs, PACT 2016
    - https://www.eidos.ic.i.u-tokyo.ac.jp/~iwasaki/files/PACT2016_slides.pdf
- You may only read the implementation section and skip theorem/proofs (if any)

Next Lecture (L #08)

- Memory consistency models
- Project milestone-1 will be announced over the weekend