Lecture 06: Sequential Overheads from Task Granularity

Vivek Kumar

Computer Science and Engineering

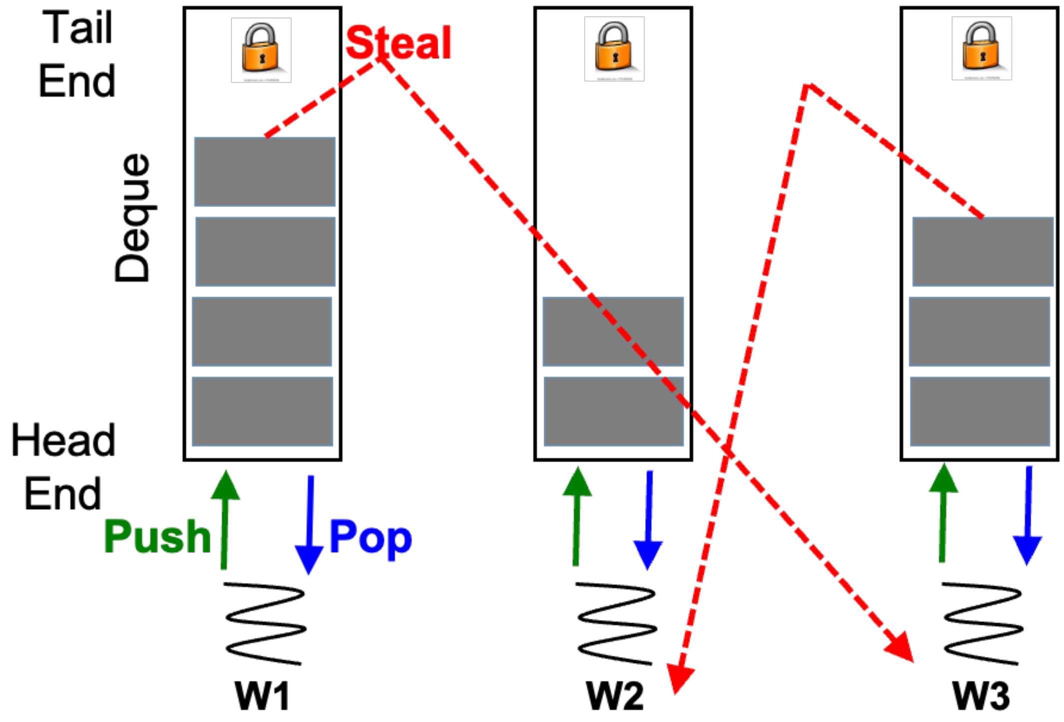
IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture (Recap)

```
void async(task) {
  lock_finish();
  finish_counter++;//concurrent access
  unlock_finish();
  // copy task on heap
  void* p = malloc(task_size);
  memcpy(p, task, task_size);
  //thread-safe push_task_to_runtime
  push_task_to_runtime(&p);
  return;
}
```

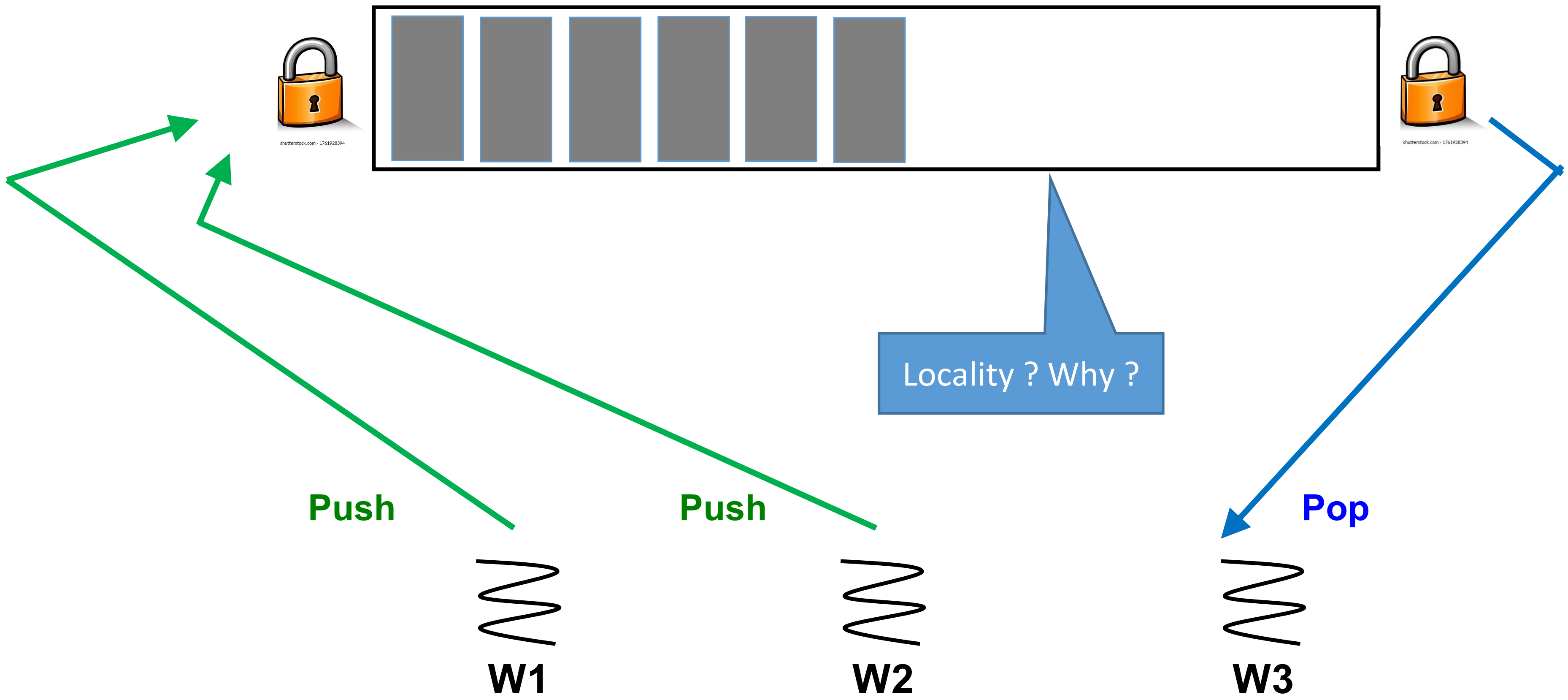
```
void find_and_execute_task() {
   //pop_from_runtime is thread-safe
   task = pop_task_from_runtime();
   if(task != NULL) {
      execute_task(task);
      free(task);
      lock_finish();
      finish_counter--;
      unlock_finish();
   }
}
```



Today's Class

- Performance in task based parallel programming models (contd.)
  - Work-sharing
- Sequential overheads of work-stealing
  - Controlling task granularity

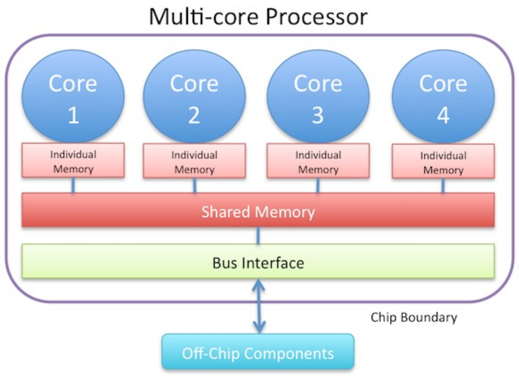
Work-Sharing Runtime System

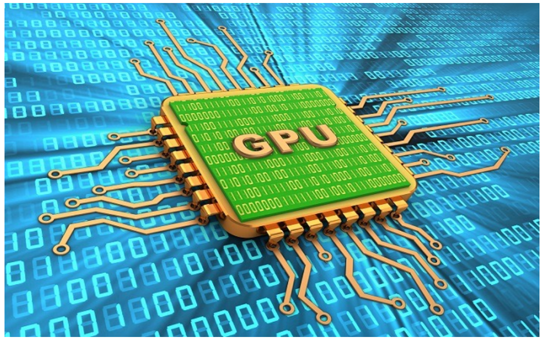


Work-Sharing v/s Work-Stealing

- Work-sharing
  - Busy worker re-distributes the task eagerly
  - Easy implementation through global task pool
  - Access to the global pool needs to be synchronized: scalability bottleneck
- Work-stealing
  - Busy worker pays little overhead to enable stealing
    - A lock is required for pop and steal only in case single task remaining on deque (only feasible by using atomic operations)
    - Idle worker steals the tasks from busy workers
  - Distributed task pools
  - Better scalability

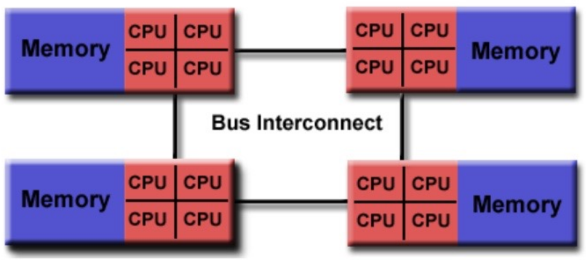
Supported on Wide Range of Architectures

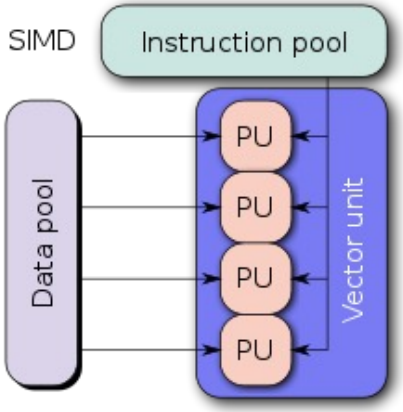




Multiprocessor System-on-Chip

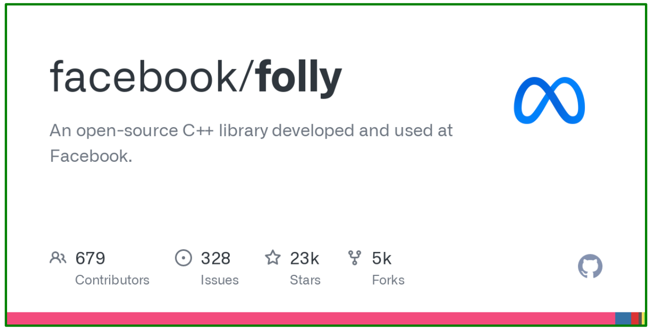
Shared Memory (NUMA)

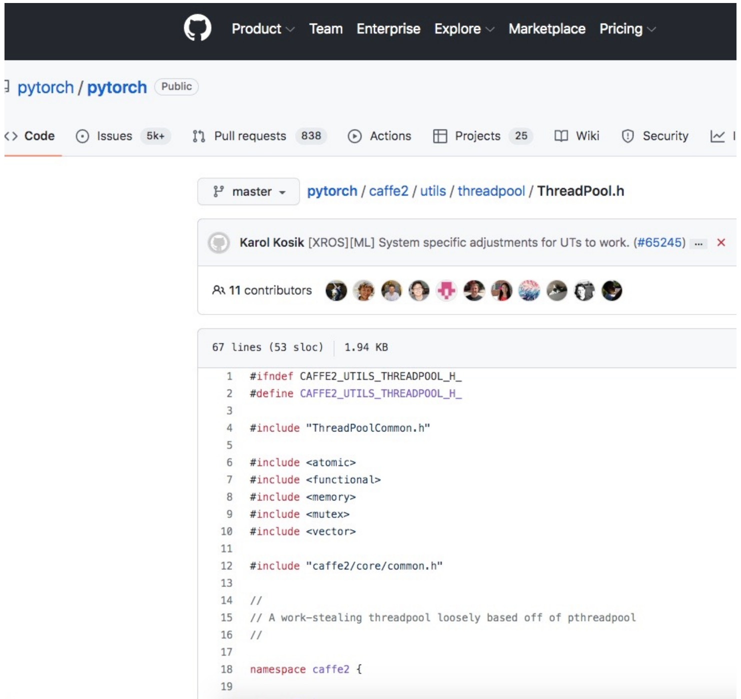


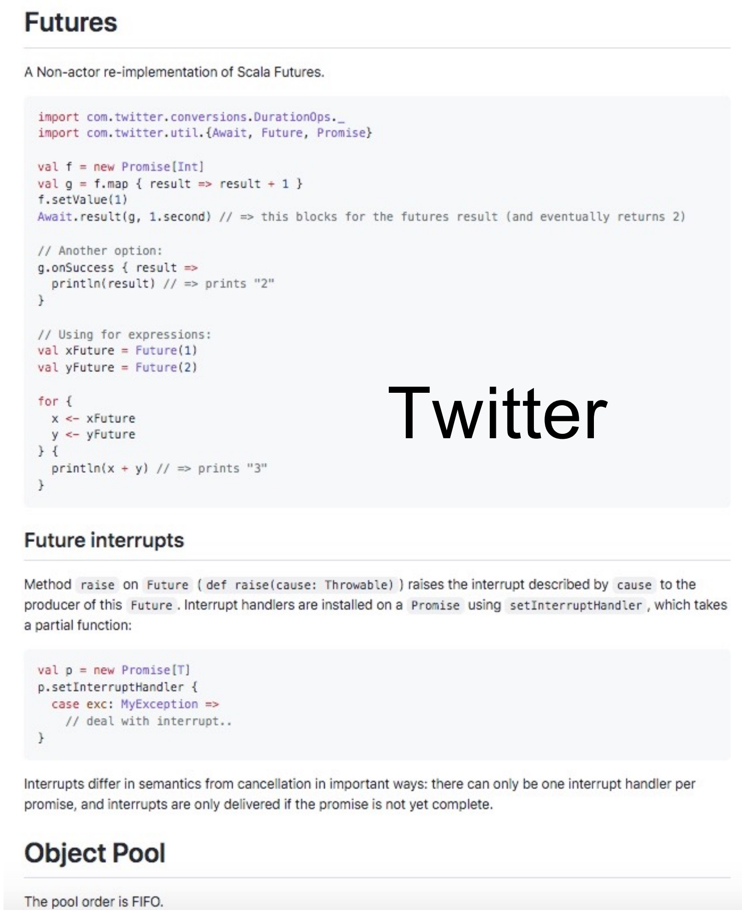


Supercomputers

Supported/Used by Several Companies/Projects







Twitter

PYTORCH

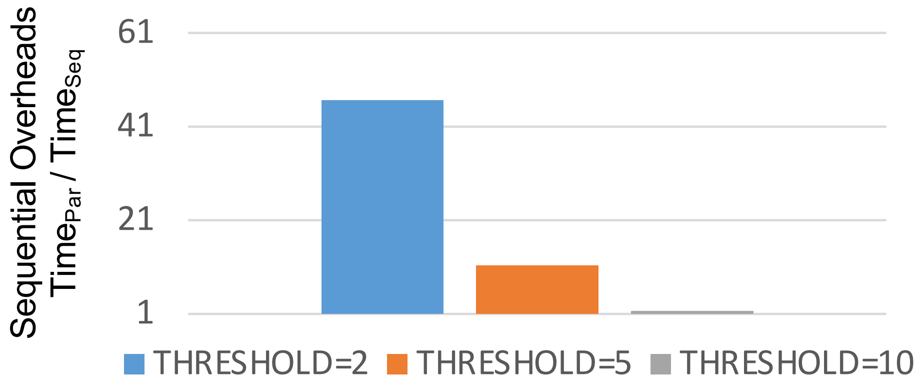
Caffe

Today’s Class

- Performance in task based parallel programming models
  - Work-sharing
- Sequential overheads of work-stealing
  - Controlling task granularity

Sequential Overheads (1/2)

```
uint64_t fib(uint64_t n) {
  if (n < THRESHOLD) {
    return fib_sequential(n);
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      async([&y]() { y = fib(n-2); }
    }
    return (x + y);
  }
}
```



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

- Sequential overhead = Ratio $Time_{seq}$ / $Time_{Par}$
  - $Time_{seq}$ is time for Fibonacci with serial elision
  - $Time_{seq}$ is for the corresponding parallel version, but by only using a single thread (sequential execution)
- **Observation**
  - Overheads can be controlled using **optimal 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?
     - What depth in a recursion tree is optimal to execute sequentially?
  3. Use memoization
     - Saving and reusing previously computed values of a function rather than recomputing them

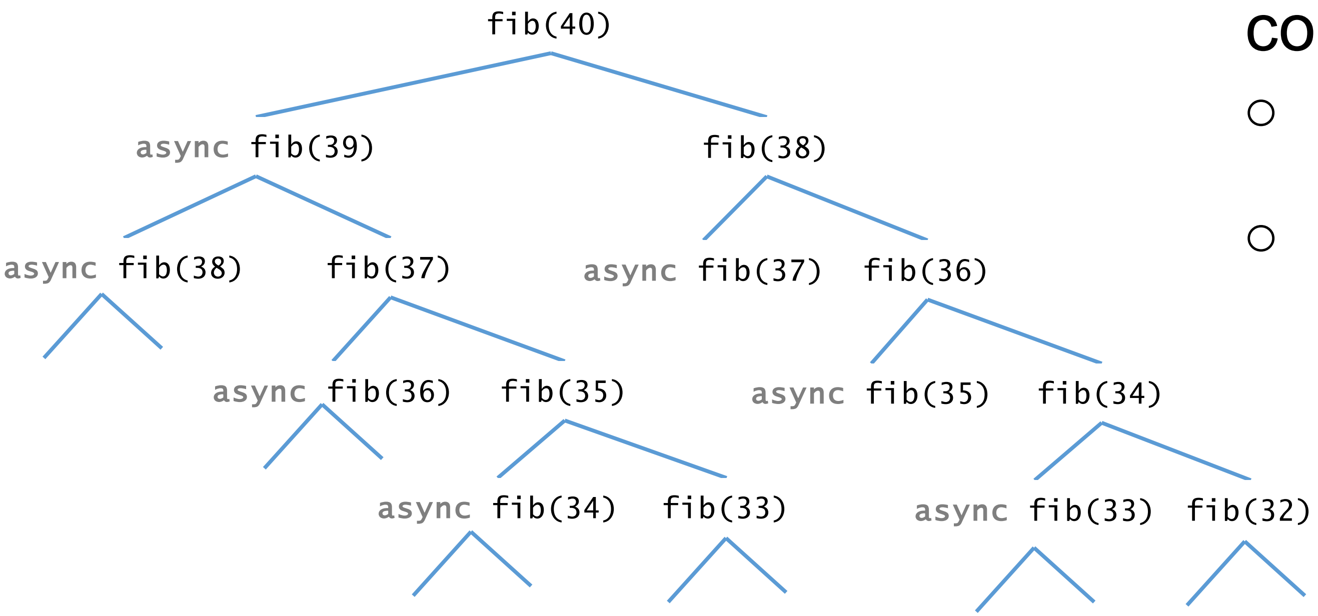
Sequential Overheads (2/2)

- Creating an async is not same as executing it sequentially
  - Each async has some metadata associated with it
    - Recall, coping user lambda on heap is important so that it can be used later even if the function that created that task has gone out of scope
  - **It is important to control task granularity**
    - We will discuss three different solutions in this lecture

Today's Class

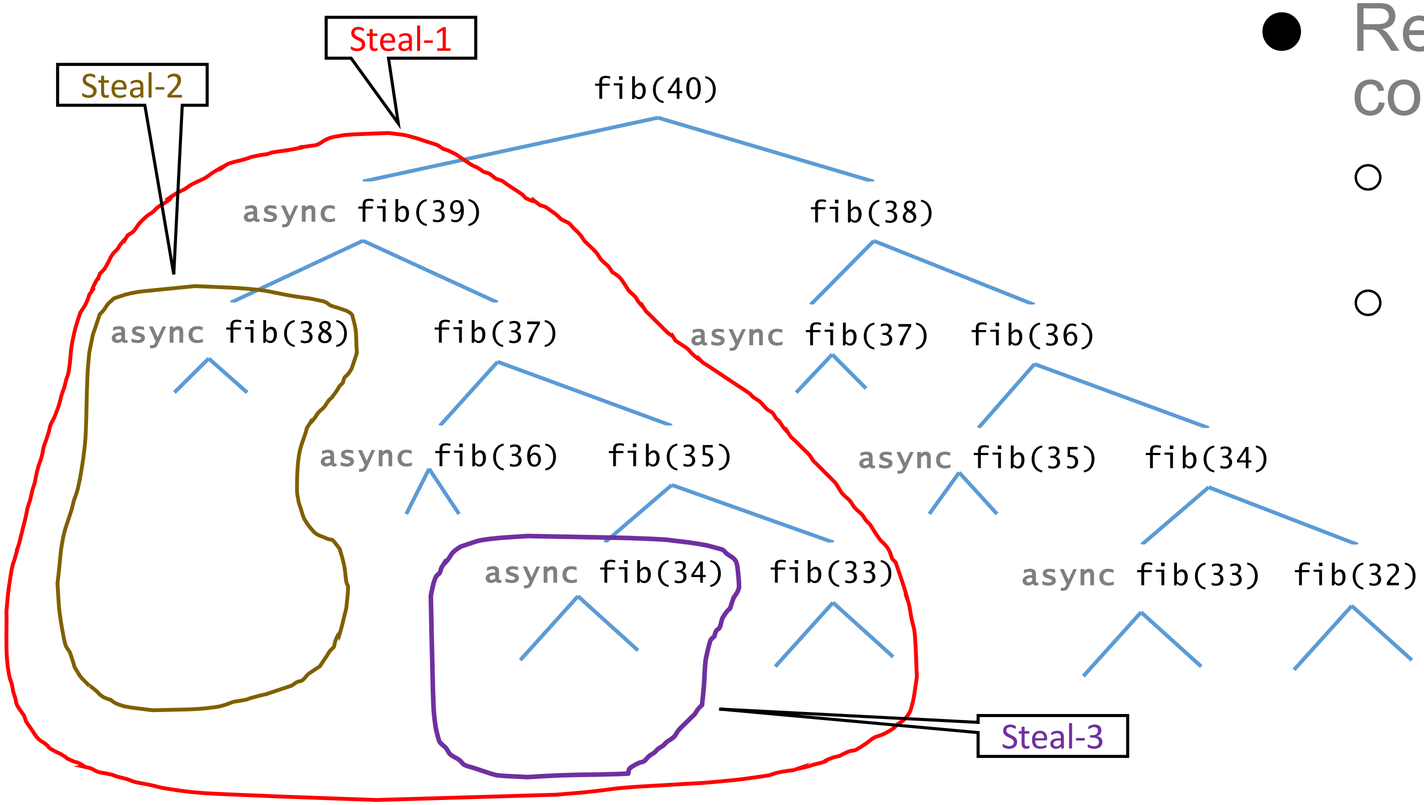
- Performance in task based parallel programming models
  - Work-sharing
- Sequential overheads of work-stealing
  - Controlling task granularity

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



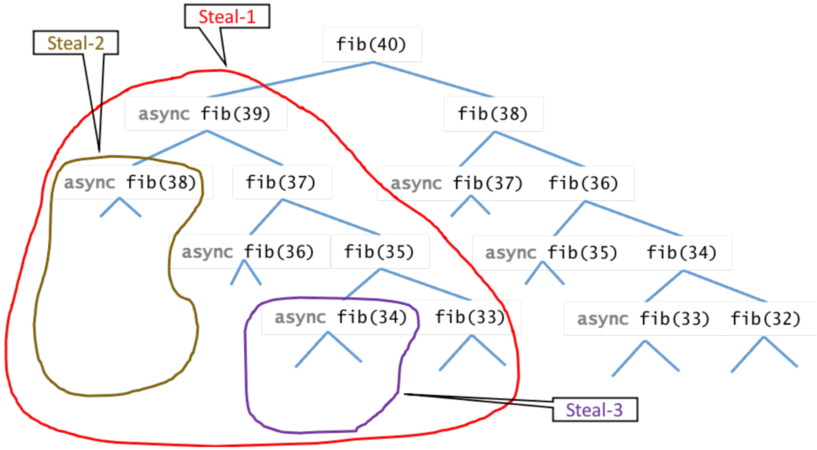
- Recursive divide-and-conquer style
  - Leads to fine granular task creation
  - How its helpful?
    1. Nested task creation

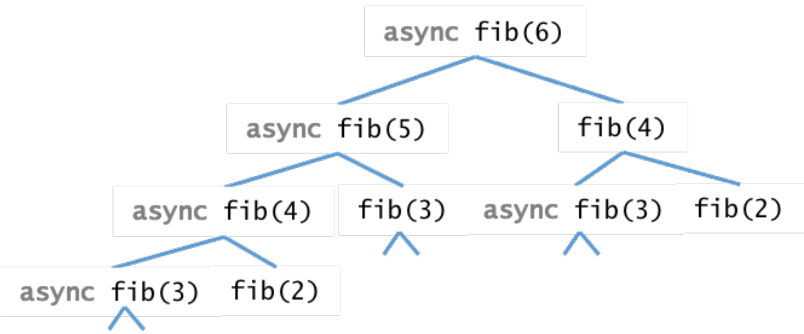
First and Foremost, Work-Stealing is Best Suited for What Kind of Applications?



- 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

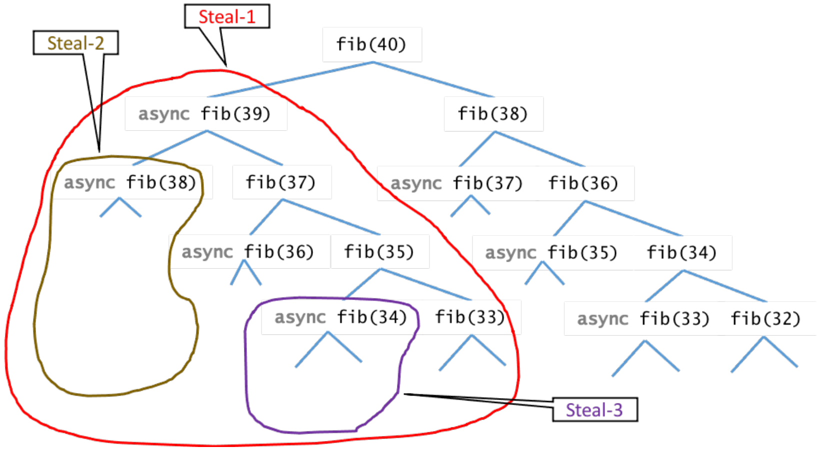
First and Foremost, Work-Stealing is Best Suited for What Kind of Applications?

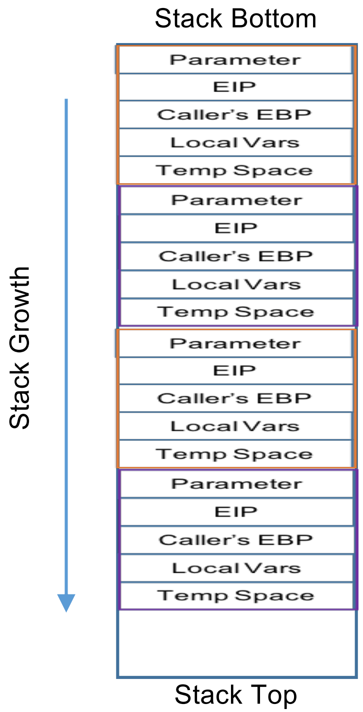




- Recursive divide-and-conquer style
  - Leads to fine granular task creation
  - **Disadvantages**?
    1. Tasks created near the bottom of the tree are too small in computation, and wouldn't be able to keep a thief busy once stolen

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





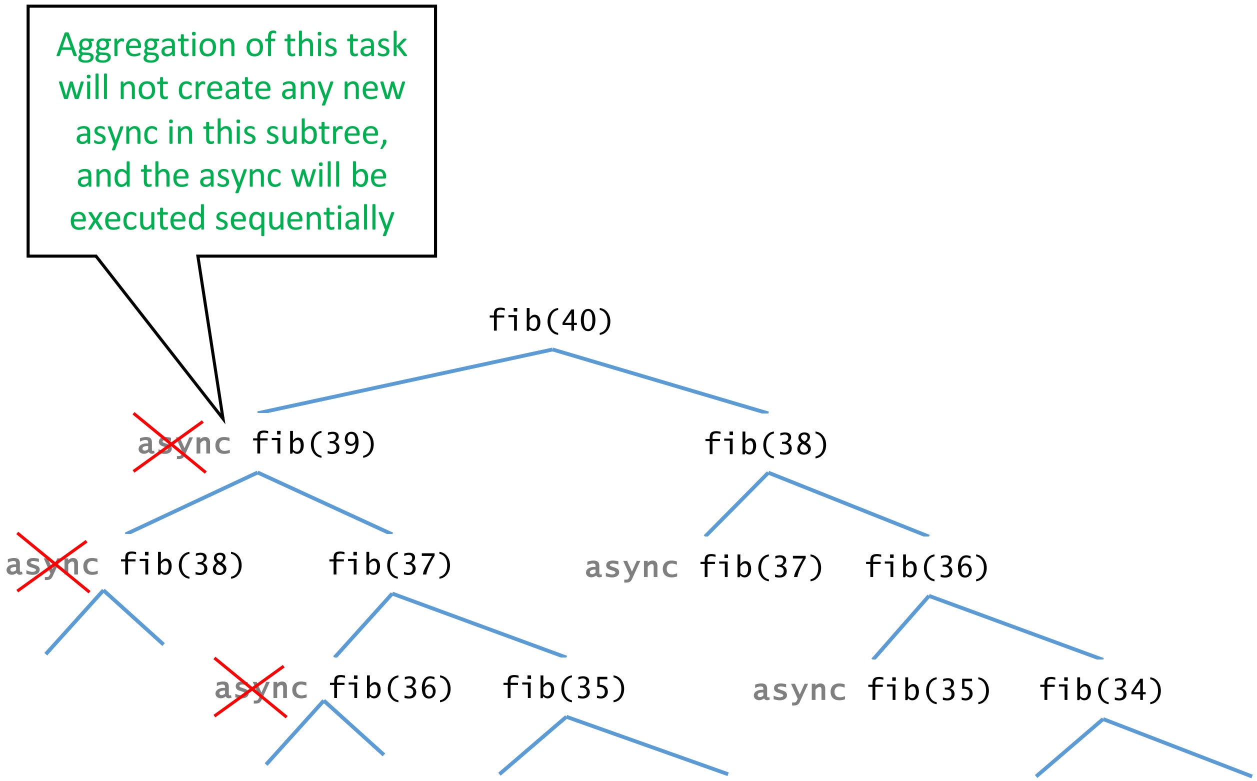
- Recursive divide-and-conquer style
  - Leads to fine granular task creation
  - **Disadvantages**?
    1. Tasks created near the bottom of the tree are too small in computation, and wouldn't be able to keep a thief busy once stolen
    2. Thread stack too deep
       - Too many context switches for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

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”

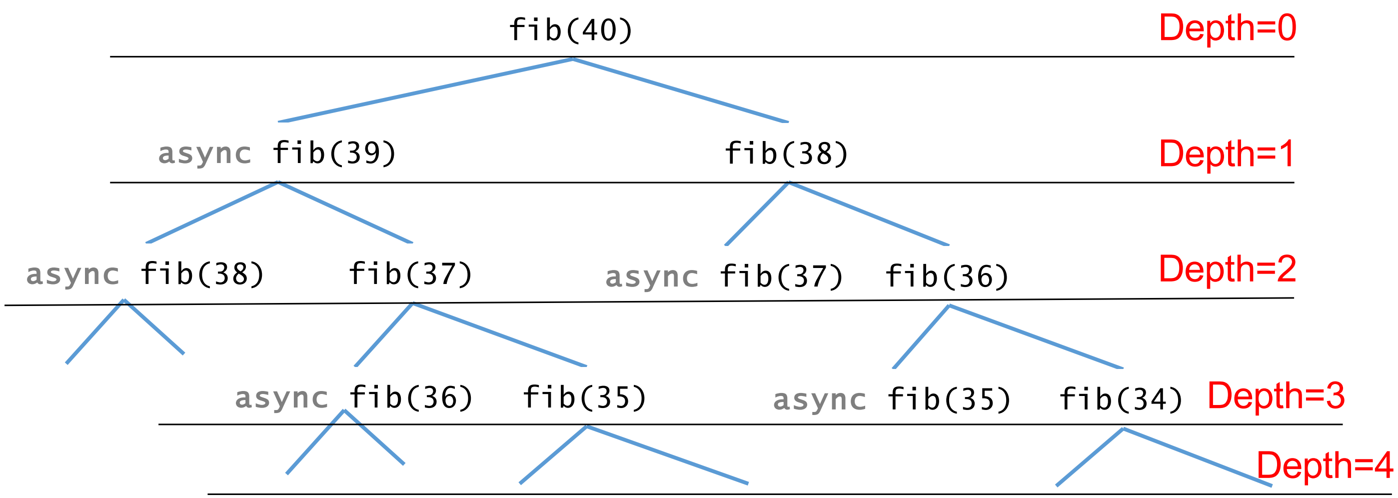
Solution-1: Automatic Granularity Control

- Runtime can perform dynamic task aggregations

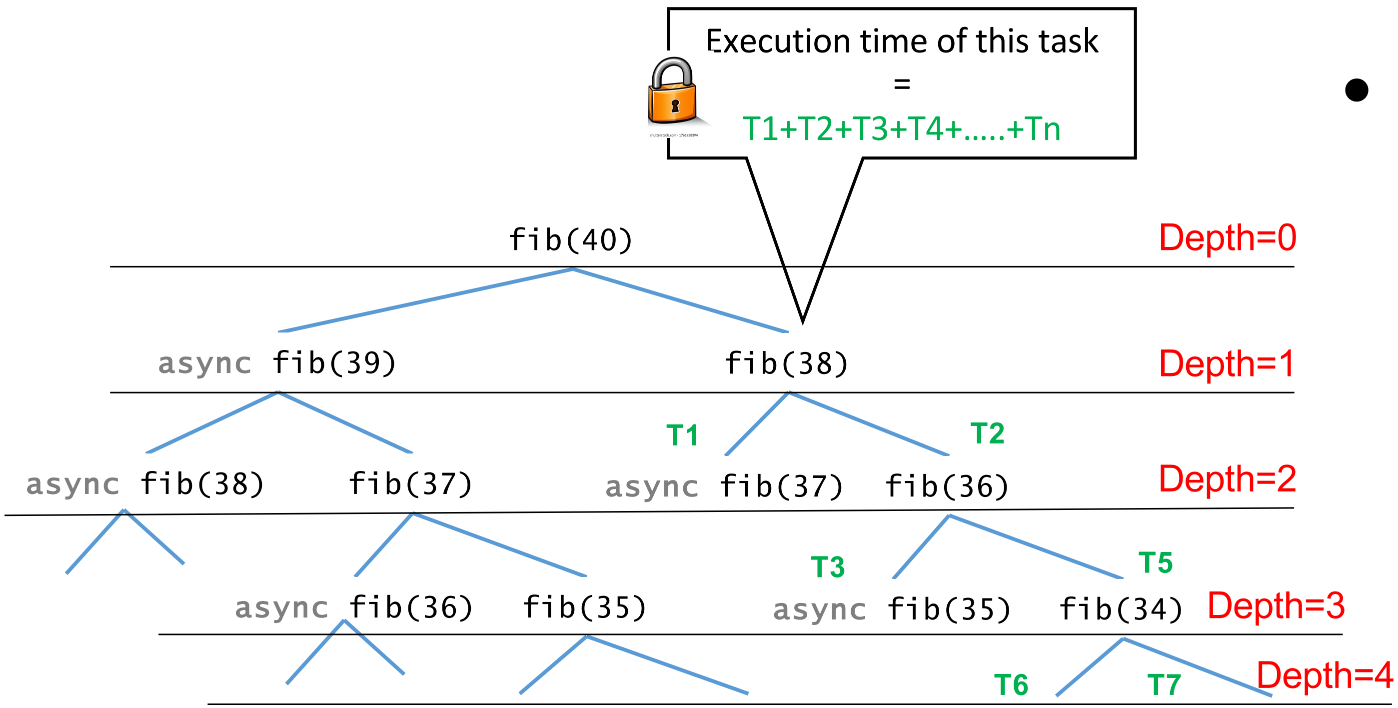


Solution-1: Automatic Granularity Control

- 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



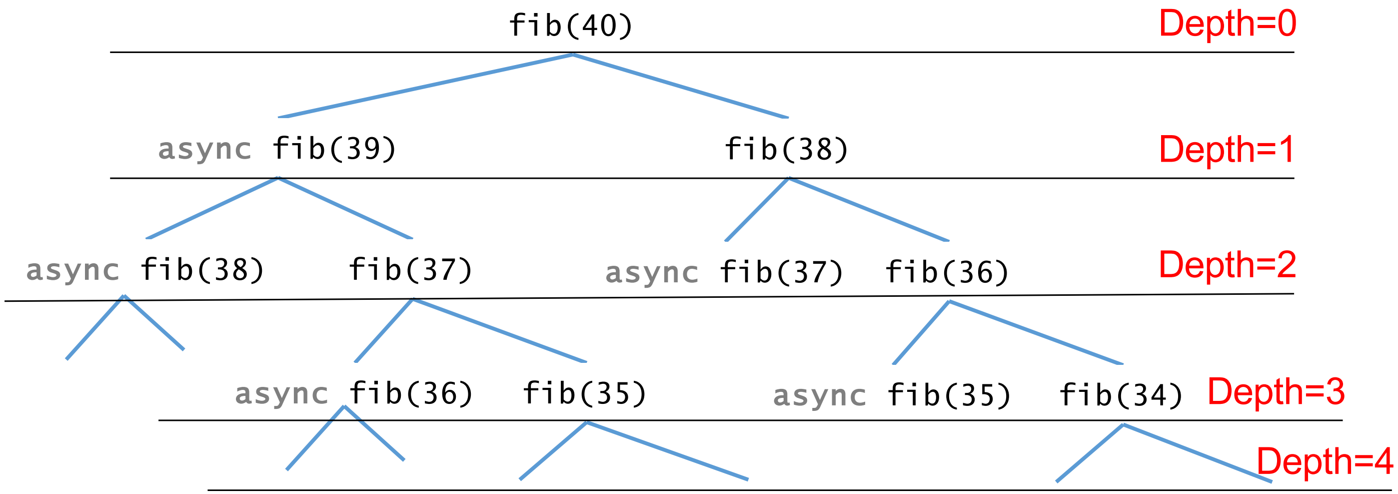
Solution-1: Automatic Granularity Control



- 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

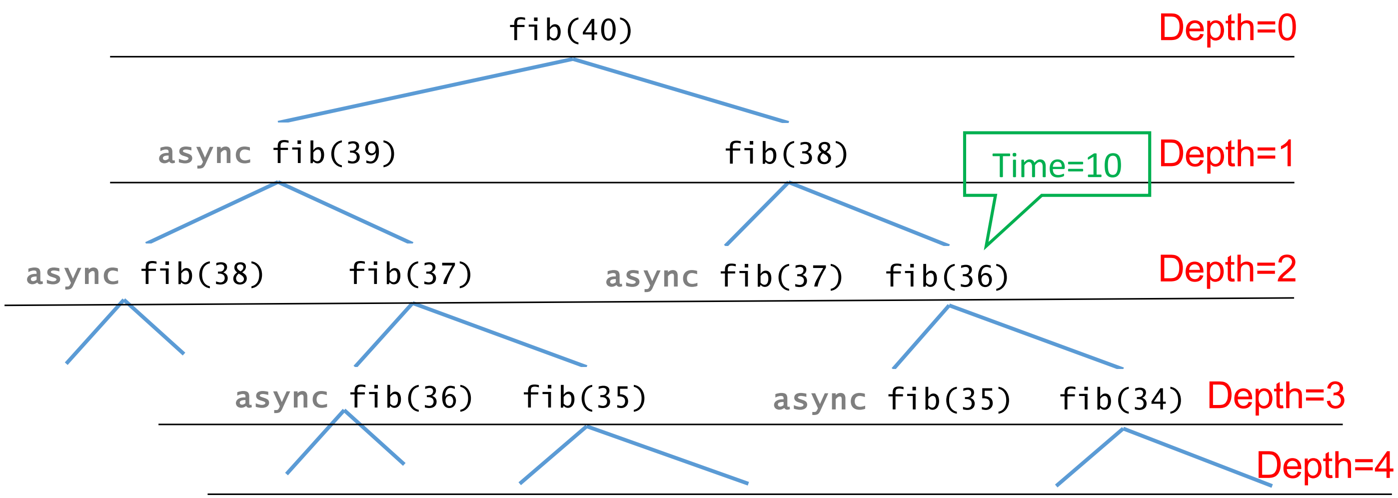
Depth=0 Time=0	Depth=1 Time=0	Depth=2 Time=0	Depth=3 Time=0	



- 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

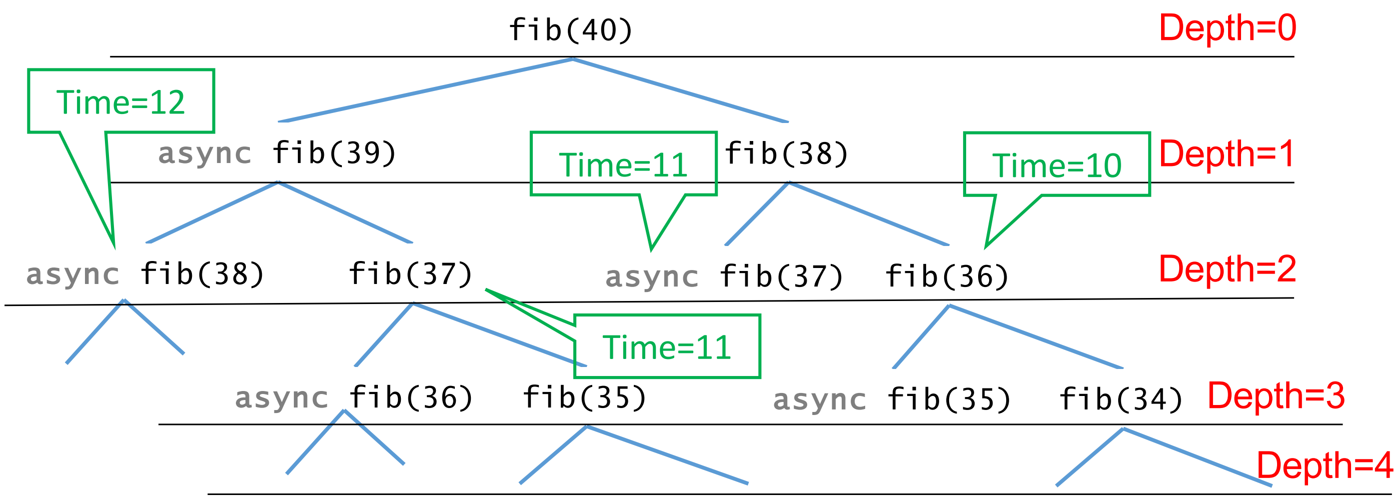
Depth=0 Time=0	Depth=1 Time=0	Depth=2 Time=10	Depth=3 Time=0	



- 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)

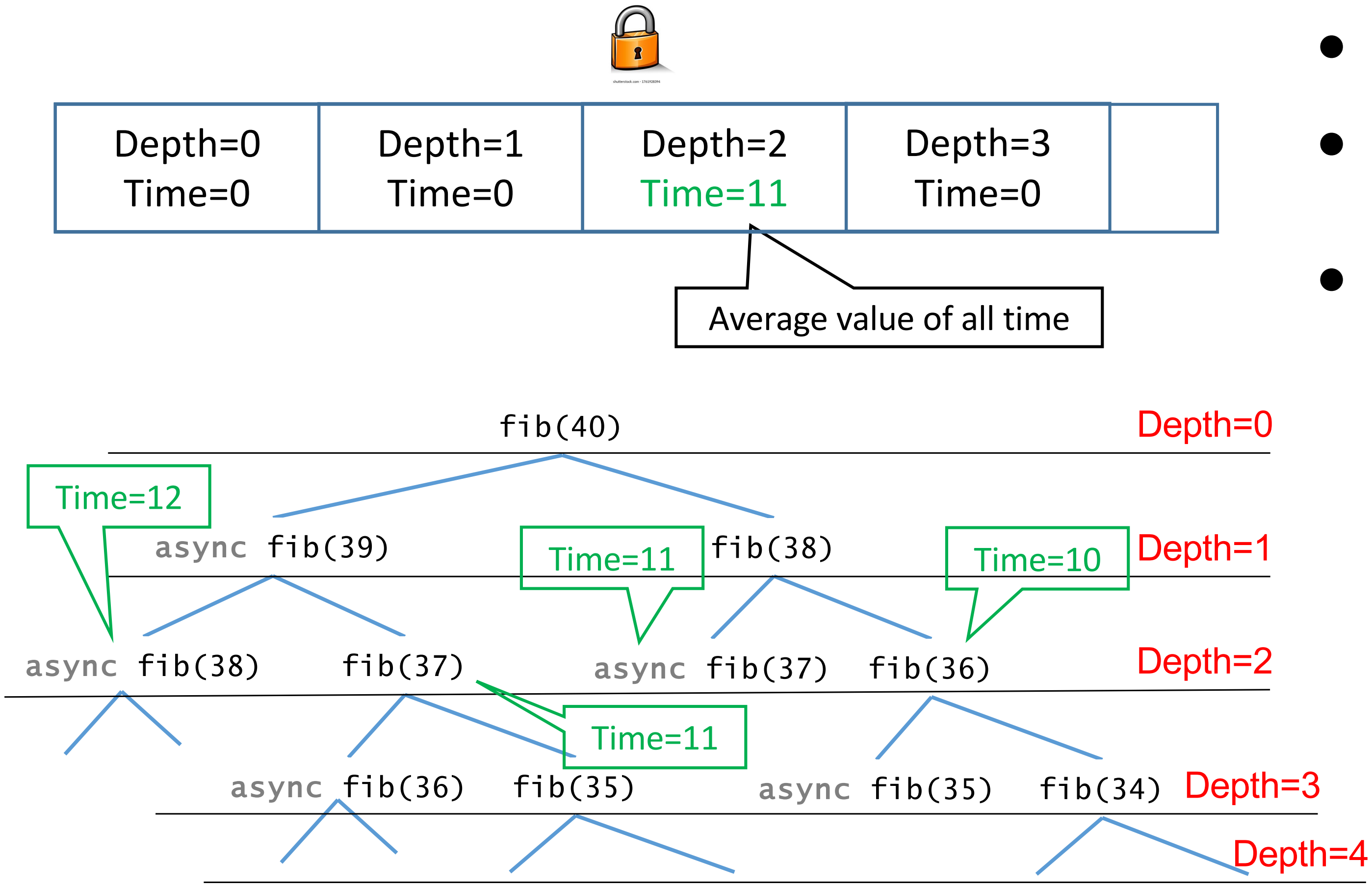
Solution-1: Automatic Granularity Control

Depth=0 Time=0	Depth=1 Time=0	Depth=2 Time=10	Depth=3 Time=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

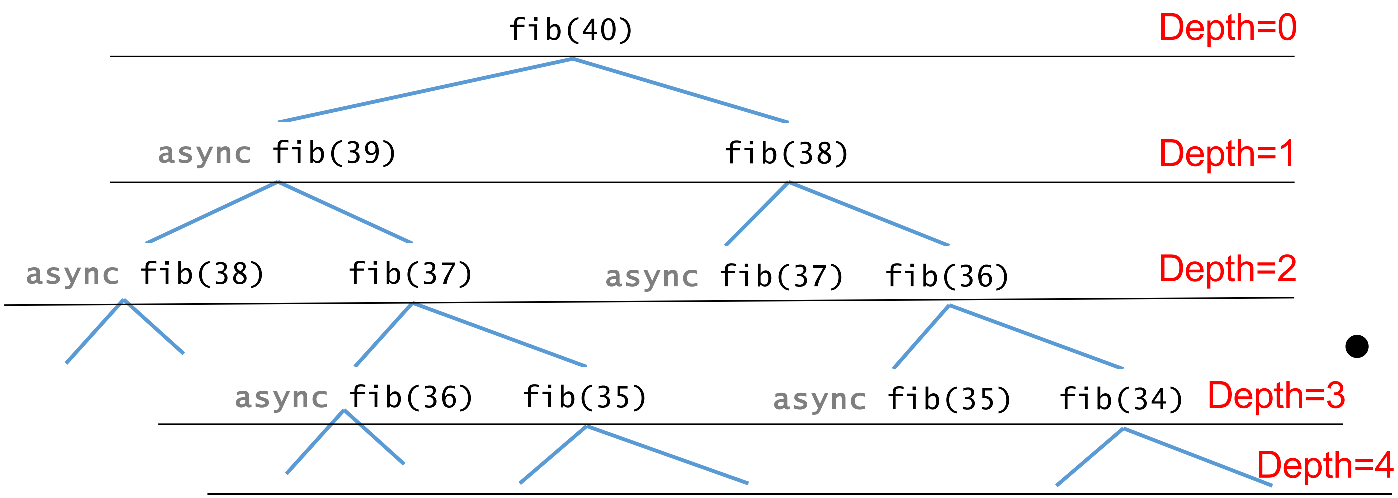
Solution-1: Automatic Granularity Control



- 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

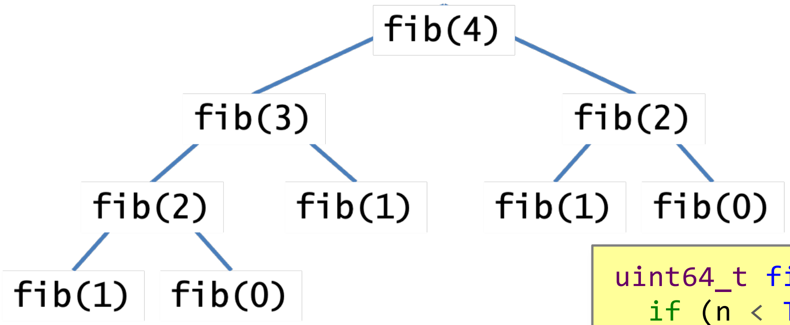
There is a general format for converting tail recursion into iterative version: https://www.baeldung.com/cs/convert-recursion-to-iteration

Solution-3: Using Memoization

- **Memoization** is about saving and reusing previously computed values of a function rather than recomputing them
- An optimization technique with space-time tradeoff

Memoization: Functional Programming

- Functional programming emphasizes functions whose results that depend only on their inputs and not on any other program state
- Revisiting recursive Fibonacci
    - Calling a function, fib(x), twice with the same value for the same argument ‘x’ will produce the same result both times




```
uint64_t fib(uint64_t n) {
  if (n < THRESHOLD) {
    return fib_sequential(n);
  } else {
    uint64_t x, y;
    finish([&]() {
      async([&x]() { x = fib(n-1); }
      async([&y]() { y = fib(n-2); }
    }
    return (x + y);
  }
}
```

Solution-3: Applying Memoization on Fib (1/2)

```
uint64_t fib(uint64_t n) {
  int value = getValue(n);
  if(value != -1) return value;
  else if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish ([&]( ) {
      async ([&]( ){x = fib(n-1);});
      y = fib(n-2);
    });
    int result = x + y;
    storeValue(n, result);
    return result;
  }
}
```

```
My_hashmap<int, int> my_cache;
int getValue(int key) {
           /* return value if available */
           /* else return -1 */
}
void storeValue(int key, value) { ..... }
```

- A function can only be memoized if it is functional
- Related to caching
  - memoized function "remembers" the results corresponding to some set of specific inputs
  - memoized function populates its cache of results transparently on the fly, as needed, rather than in advance

Solution-3: Applying Memoization on Fib (2/2)

```
uint64_t fib(uint64_t n) {
  int value = getValue(n);
  if(value != -1) return value;
  else if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    finish ([&]( ) {
      async ([&]( ){x = fib(n-1);});
      y = fib(n-2);
    });
    int result = x + y;
    lock(); {storeValue(n, result);} unlock();
    return result;
  }
}
```

```
My_hashmap<int, int> my_cache;
int getValue(int key) {
           /* return value if available */
           /* else return -1 */
}
void storeValue(int key, value) { ..... }
```

- A function can only be memoized if it is functional
- Related to caching
  - memoized function "remembers" the results corresponding to some set of specific inputs
  - memoized function populates its cache of results transparently on the fly, as needed, rather than in advance

Reading Materials

- 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 #07)

- Sequential overheads from concurrent deque