Lecture 13: Mid Semester Review

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

Introduction to Parallel Programming (1/2)

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

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

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);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

$$\text{Speedup using P processes} = \frac{T_{S1} + T_{S2}}{T_{S1} + T_P}$$

Introduction to Parallel Programming (2/2)

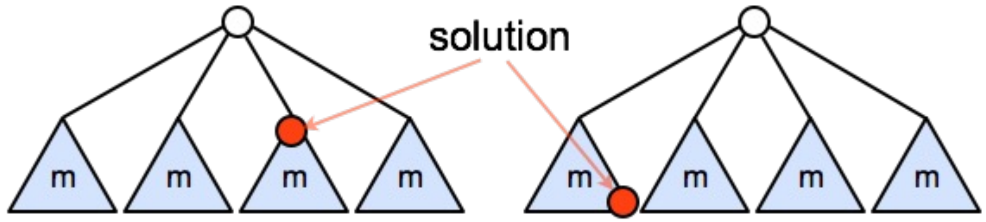
Mutex lock

```
pthread_mutex_lock(&mutex);
while(task_queue_size() == 0)
  pthread_cond_wait(&cond, &mutex);
}
task = pop_task_queue();
pthread_mutex_unlock(&mutex);
execute_task (task);
```

```
pthread_mutex_lock(&mutex);
int queue_size = task_queue_size();
push_task_queue(&task);
if(queue_size == 0) {
  pthread_cond_broadcast(&cond);
}
pthread_mutex_unlock(&mutex);
```

Concurrency Decomposition

- How should one decompose a task into various subtasks?
  - No single universal recipe
    - Recursive decomposition
    - Data decomposition
    - Exploratory decomposition
    - Speculative decomposition



- Serial execution time = 7 T
- Parallel execution time using 4 threads to compute each triangle in parallel = T
- Speedup (4 threads) = 7T/T = 7
- **Super**-linear speedup

- Serial execution time = 3 T
- Parallel execution time using 4 threads to compute each triangle in parallel = 3T
- Speedup (4 threads) = 3T/3T = 1
- **Sub**-linear speedup

```
int val = T1 //compute intensive
  switch(val) {
    case 0: T2; break;
    case 1: T3; break;
    …..
    case n: Tn; break;
}
```

Async and Finish Statements for Task Creation and Termination (Pseudocode)

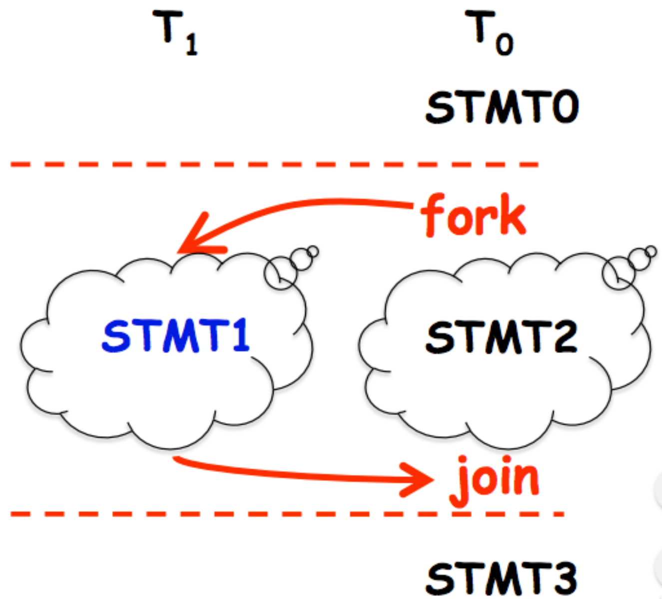
async S

- Creates a new child task that executes statement S

finish S

- Execute S but wait until all async in S's scope have terminated

```
// T0(Parent task)
STMT0;
finish {    //Begin finish
  async {
    STMT1; //T1(Child task)
  }
  STMT2;    //Continue in T0
            //Wait for T1
}           //End finish
STMT3;      //Continue in T0
```

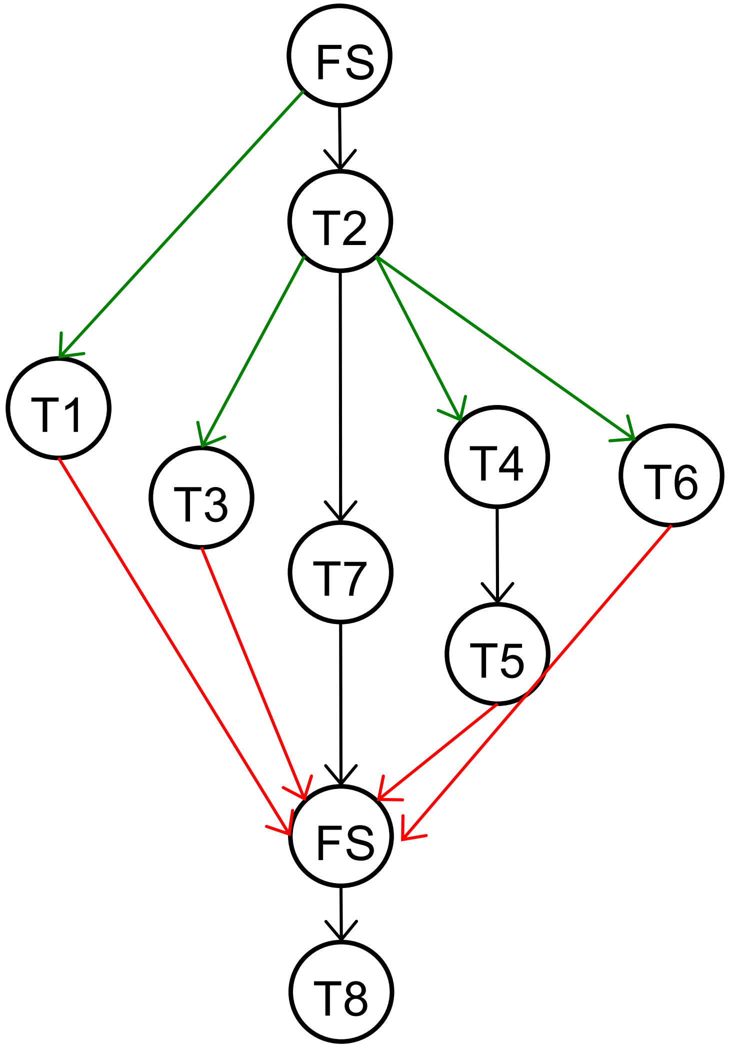


Source: https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s16-lec1-slides.pdf?version=1&modificationDate=1452732285045&api=v2

Computation Graph

- Granularity of task decomposition
  - Fine and coarse granular
- Computation graph

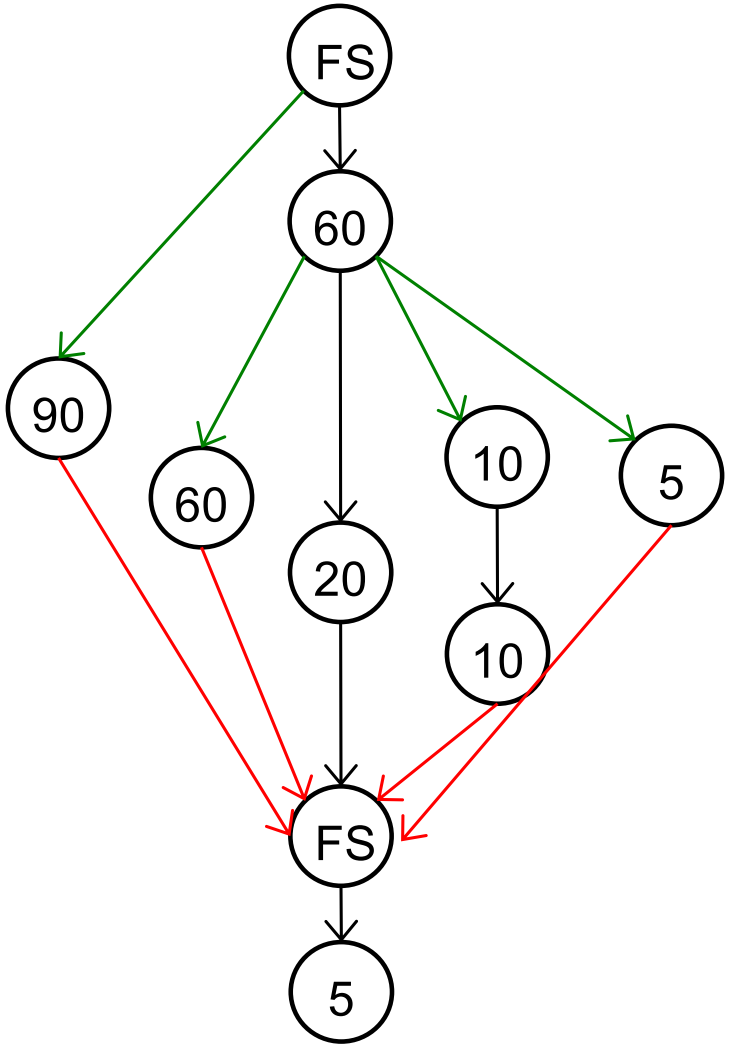
```
finish {                                                          FS
        async { Wash your clothes in washing machine }            T1
                Complete your PRMP project deadline               T2
        async { Watch movies on laptop }                          T3
        async { Talk to father                                    T4
                Talk to mother }                                  T5
        async { Buy fruits online using your smartphone }         T6
                Make your bed                                     T7
}                                                                 FE
Post on Facebook that you are done with all your tasks!           T8
```



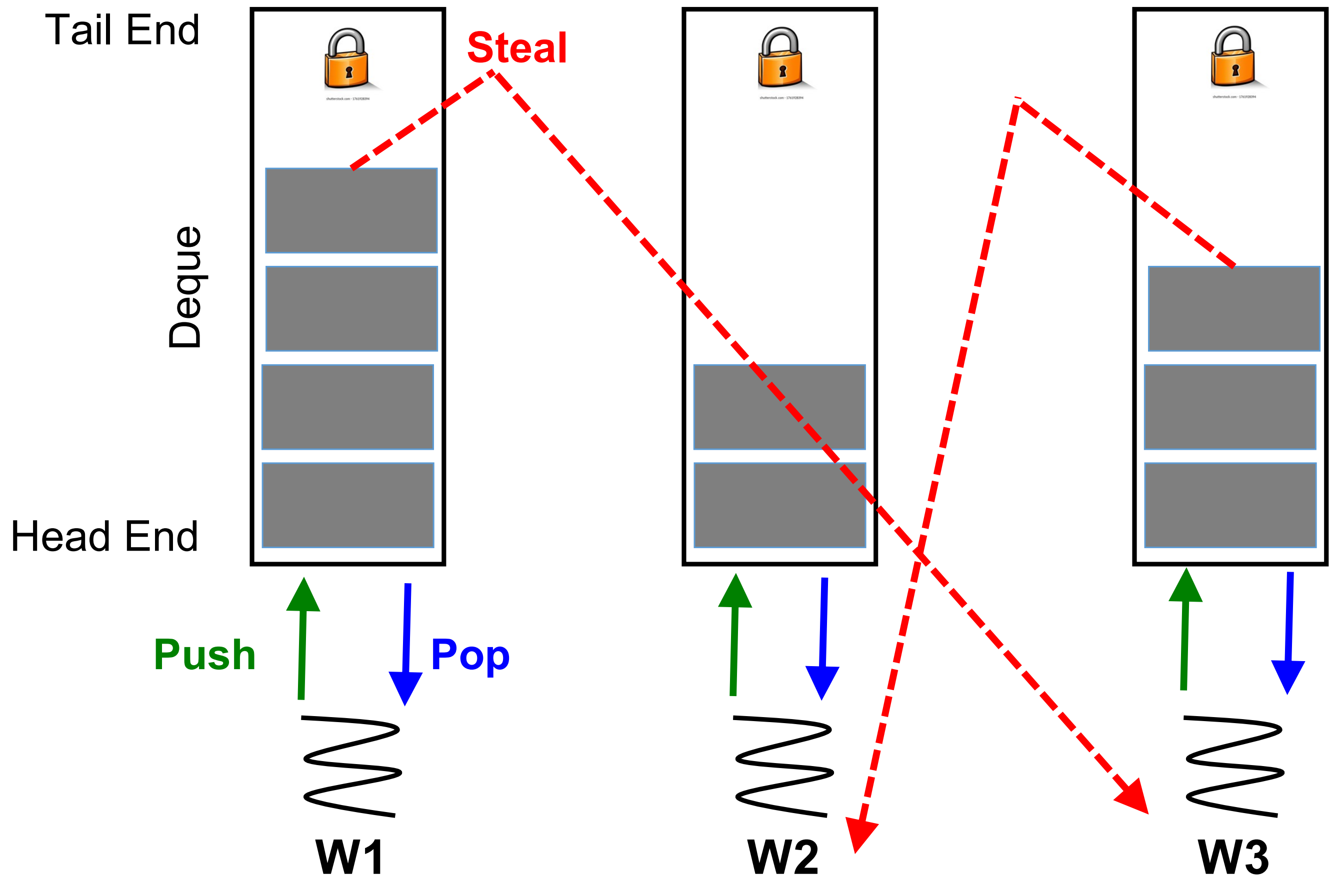
Critical Path

- Critical path is the longest weighted path in computation graph that represents task serialization

finish {	FS
async { Wash your clothes in washing machine }	T1
Complete your PRMP project deadline	T2
async { Watch movies on laptop }	T3
async { Talk to father	T4
Talk to mother }	T5
async { Buy fruits online using your smartphone }	T6
Make your bed	T7
}	FE
Post on Facebook that you are done with all your tasks!	T8

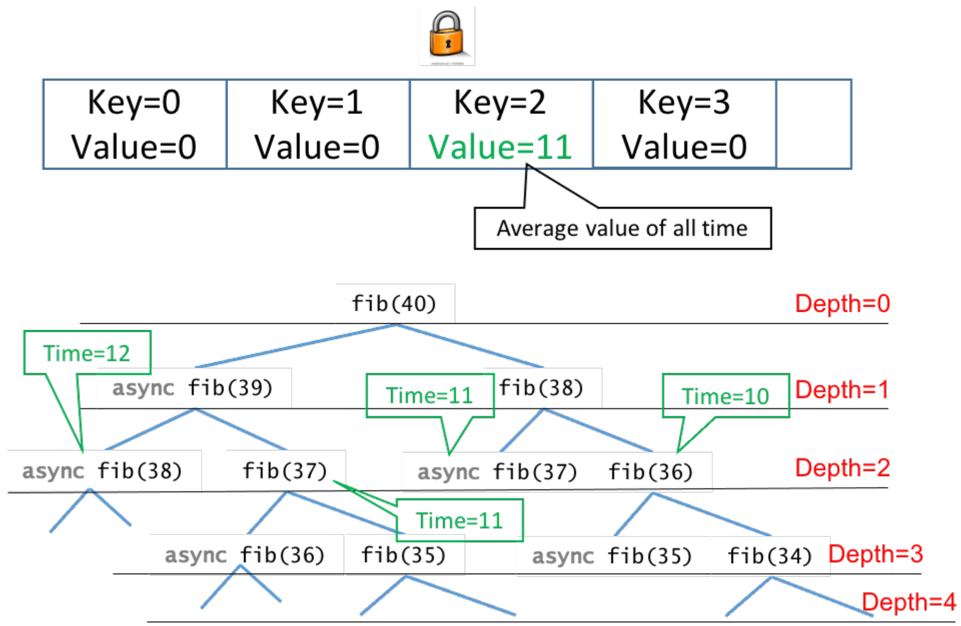


Work-Stealing Runtime System

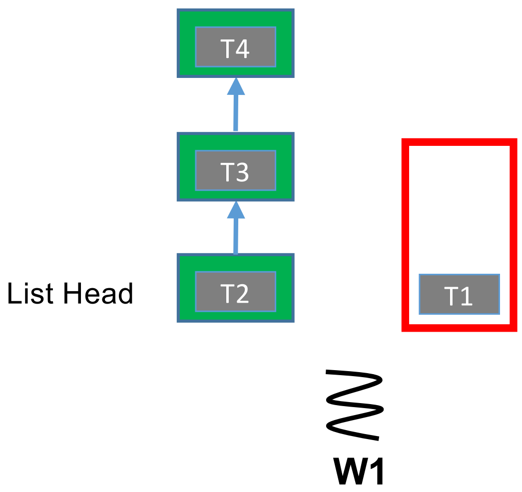


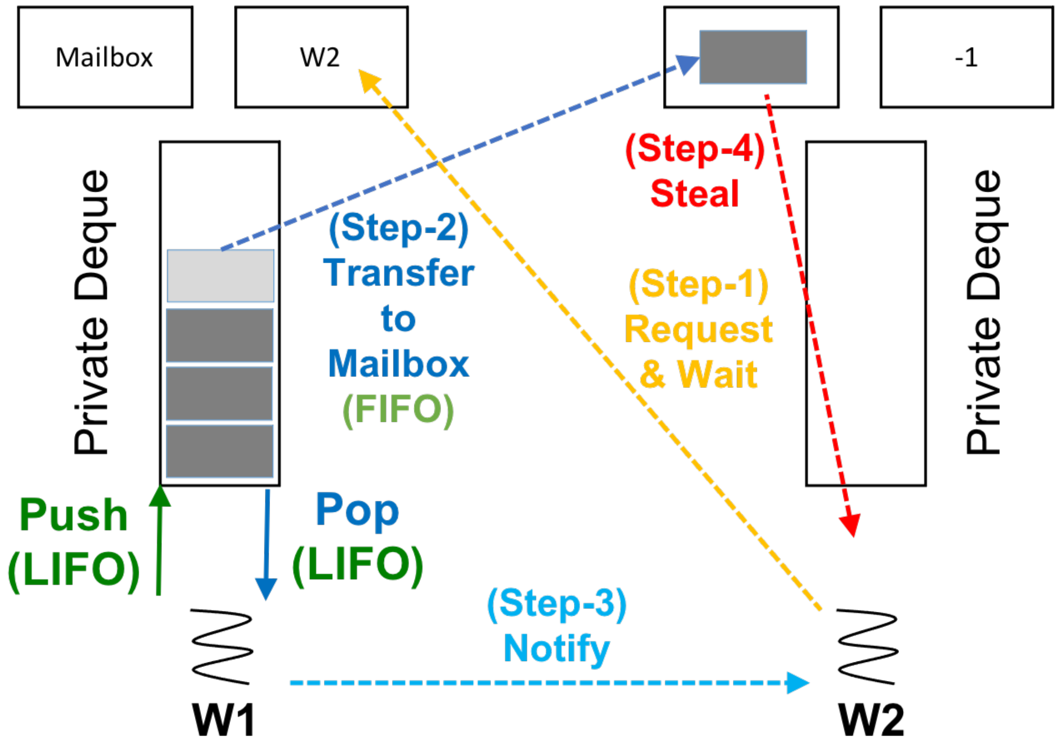
Sequential Overheads: Task Granularity

- 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

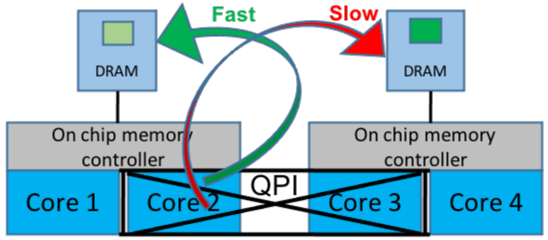


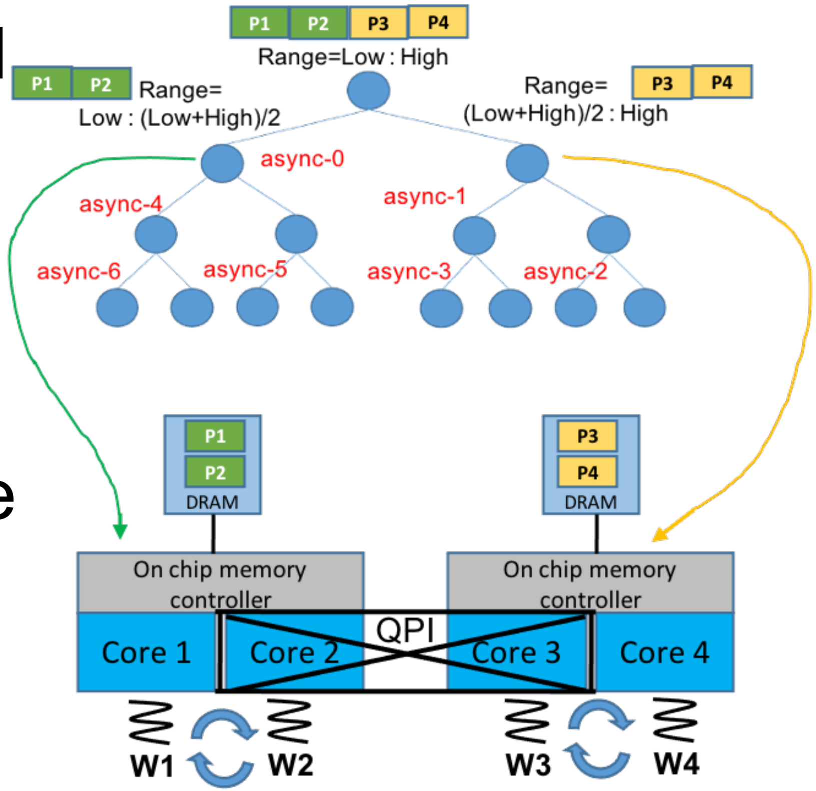


- Minimizing deque overheads
  - Using a mix of list and deque
  - Using private deque

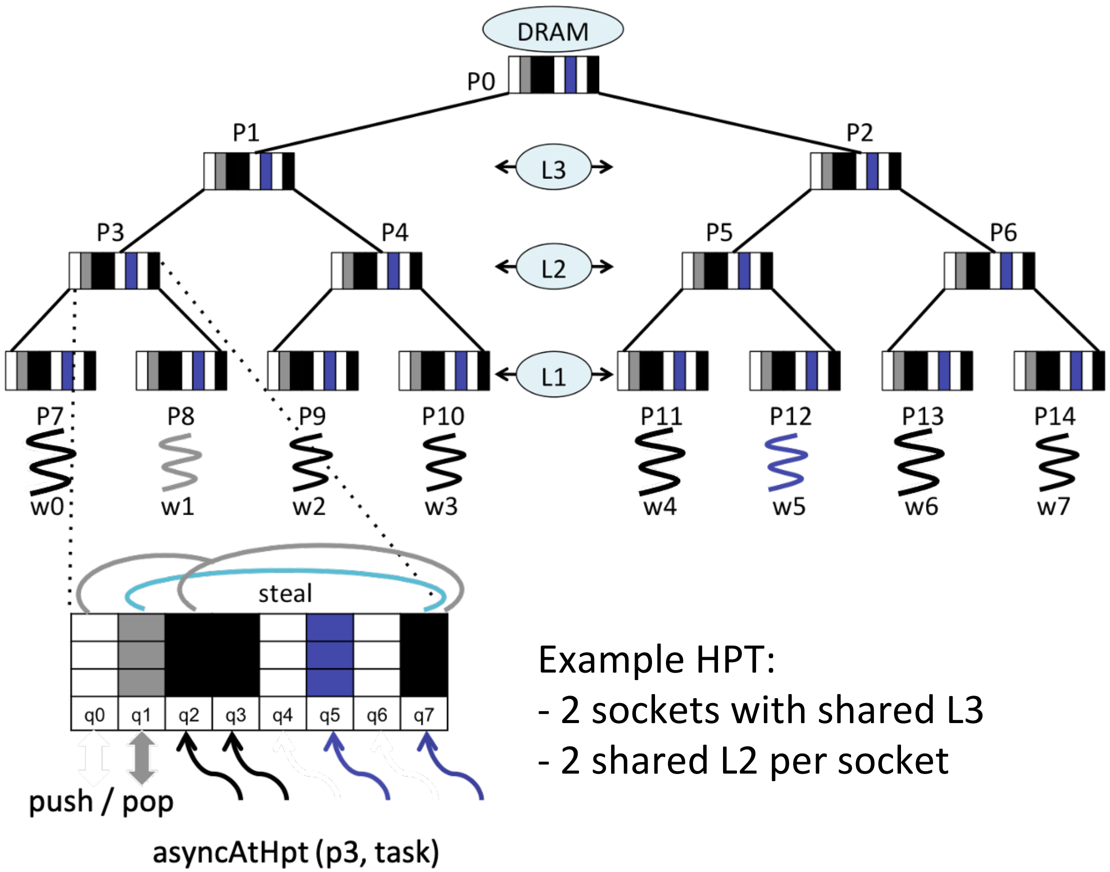
NUMA Aware Work-Stealing

- 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





NUMA Aware Work-Stealing Using HPT



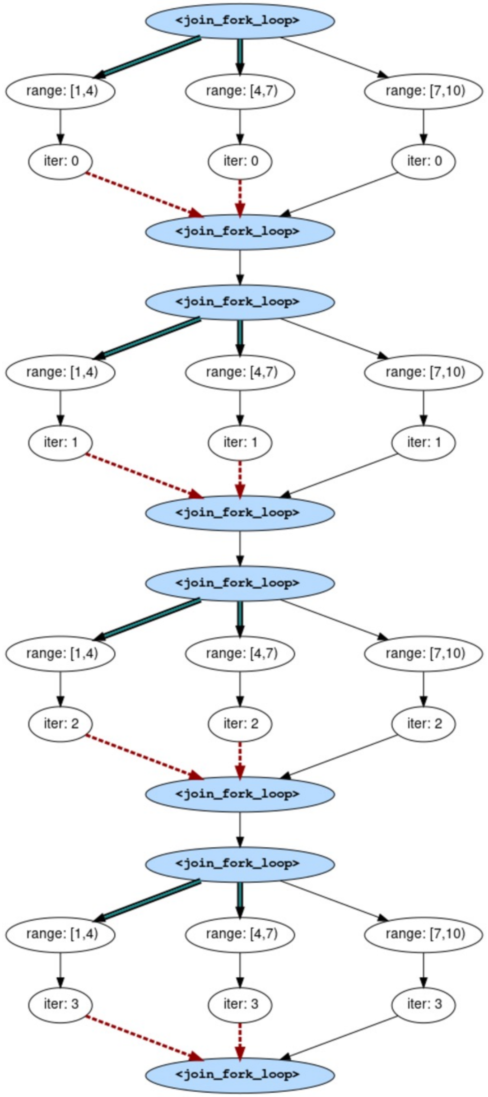
Example HPT:
- 2 sockets with shared L3
- 2 shared L2 per socket

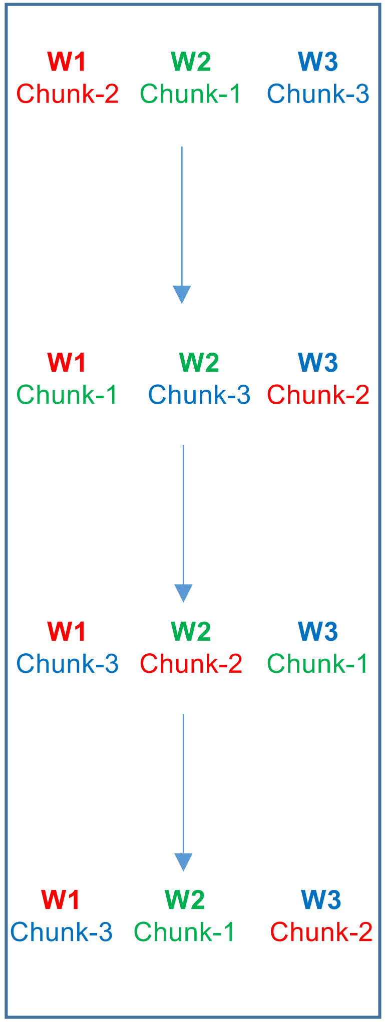
- Round-robin steals instead of random work-stealing
- Workers attach to (own) leaf places
- Each place has one queue per worker
  - Ensures non-synchronized push and pop
- Any worker can push a task at any place
- Pop / steal access permitted to subtree workers
- Workers traverse path from leaf to root
- Tries to pop, then steal, at every place
- After successful pop / steal worker returns to leaf
- Worker threads are bound to cores

Picture credit: Runtime Systems for Extreme Scale Platforms, PhD thesis, Sanjay Chatterjee, Rice University, 2013

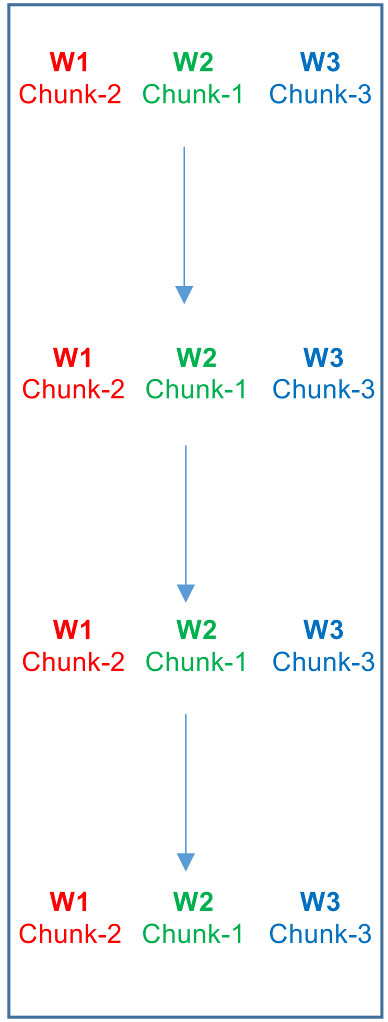
Trace/Replay

- 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

Context Switch Inside Userspace

```
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();
}
```

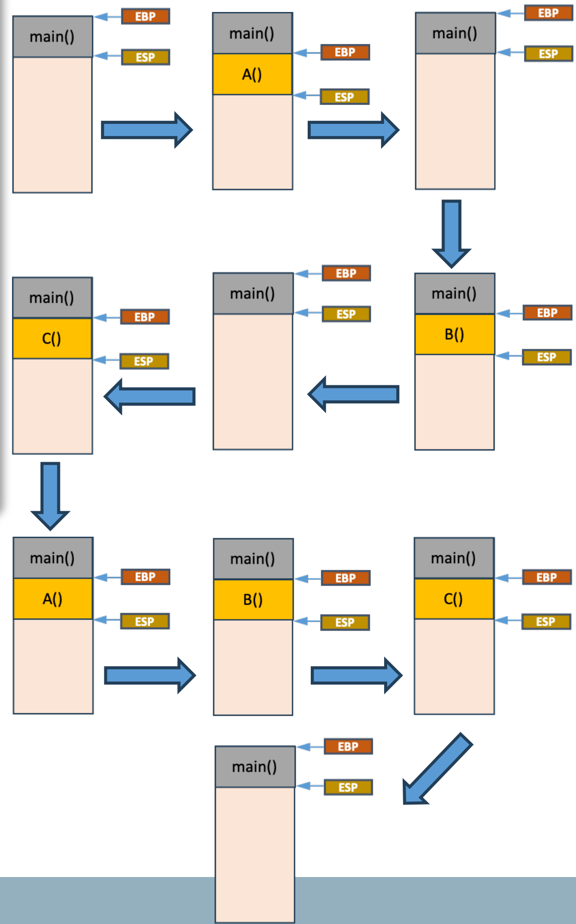
Figure-1

```
#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

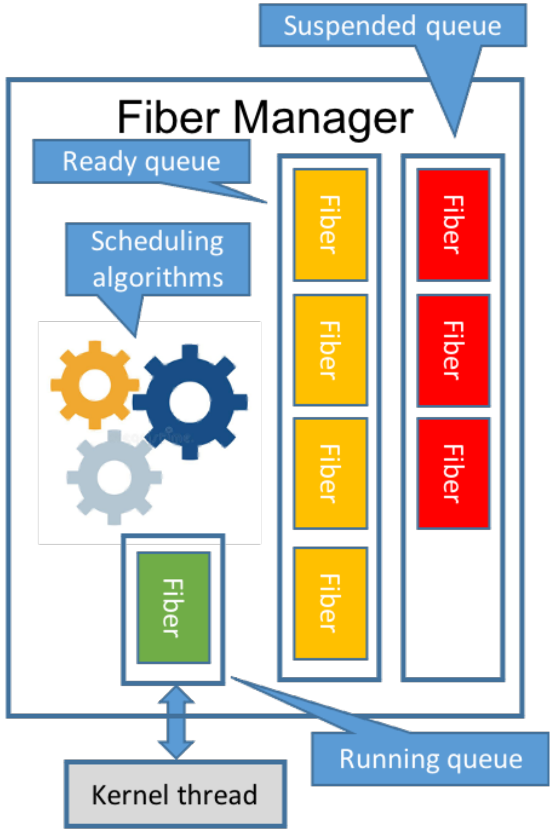
- Figure-2

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

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();
```



Midterm Exams

- Midterm exam will be held on 24/02/25 (Monday) 3pm—4pm
  - Total weightage is 20%
  - 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