Lecture 14: Mid Semester Review

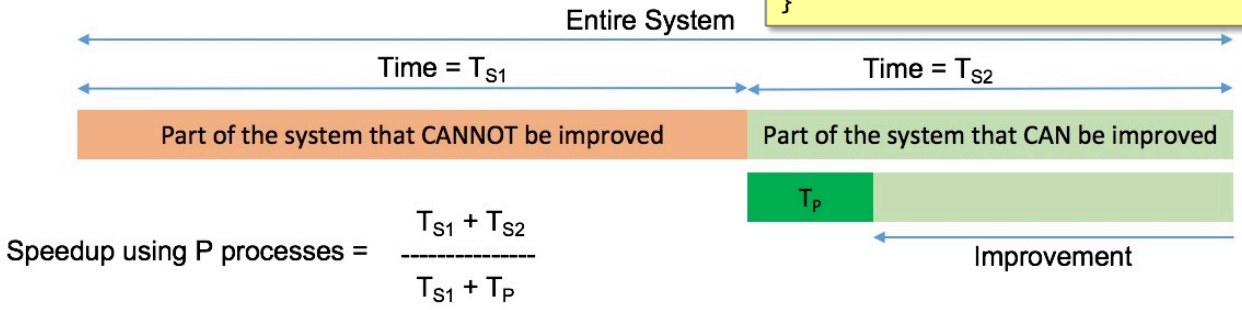
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

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;
}
```

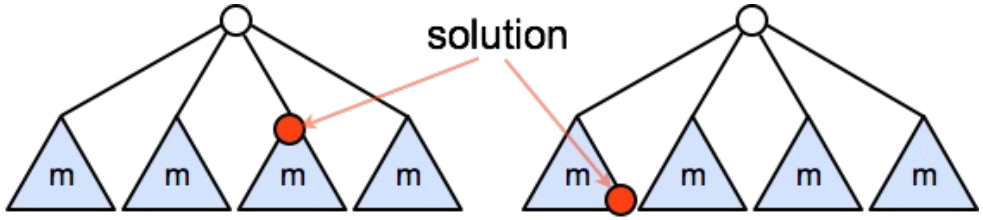
```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  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;
}
```



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

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)

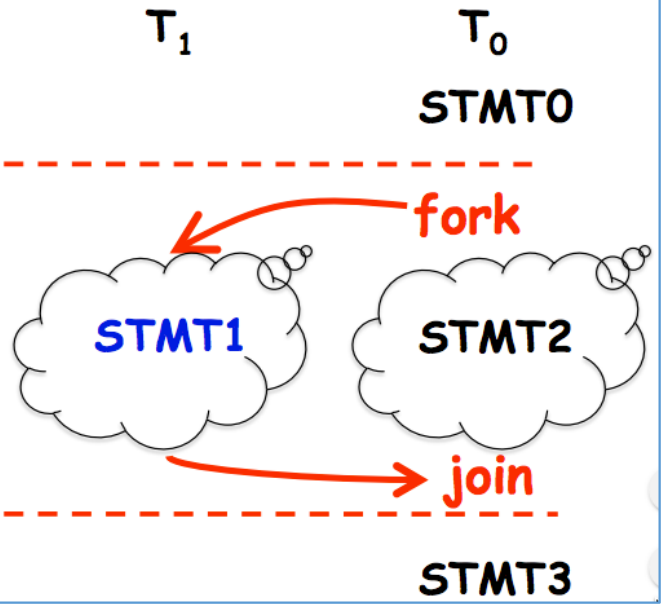
async S

- Creates a new child task that executes statement S

finish S

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

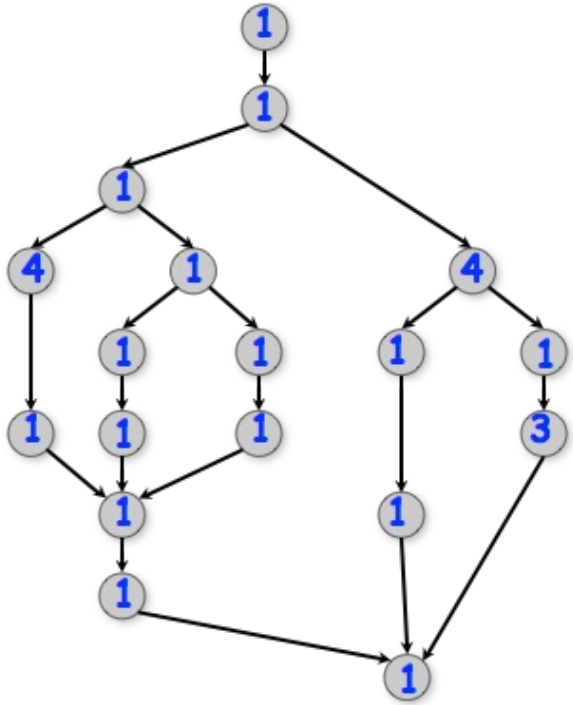
```
// T0 (Parent task)
STMT0;
finish {     //Begin finish
  async {
    STMT1;   //T1 (Child task)
  }
  STMT2;     //Continue in T0
             //Wait for T1
}            //End finish
STMT3;       //Continue in T0
```



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

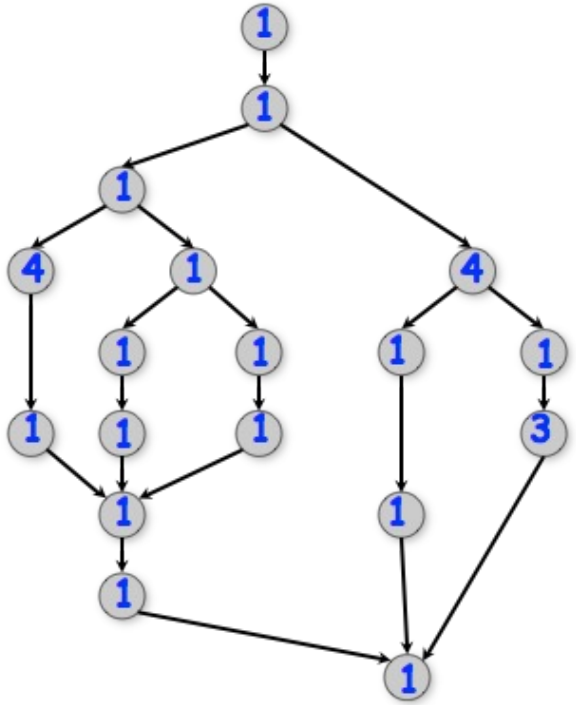
Critical Path

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



Computation Graph

- Granularity of task decomposition
  - Fine and coarse granular
- Ideal parallelism of Computation G Graph as the ratio, WORK(G)/CPL(G)
- Ideal Parallelism only depends on the computation graph, and is the speedup that you can obtain with an unbounded number of processors



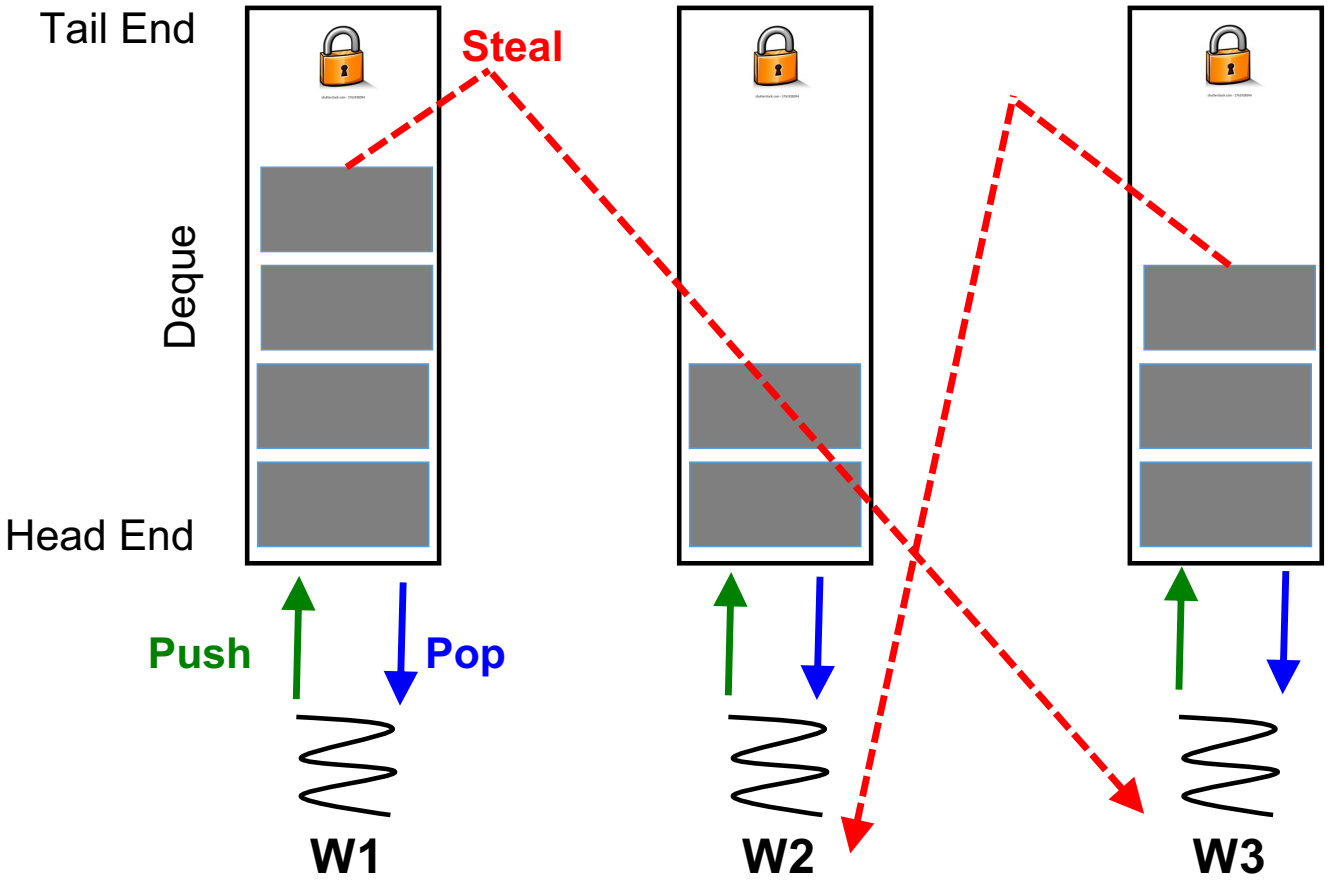
Example:

WORK(G) = 26

CPL(G) = 11

Ideal Parallelism = WORK(G)/CPL(G) = 26/11 ~ 2.36

Work-Stealing Runtime System



Work-Sharing v/s Work-Stealing

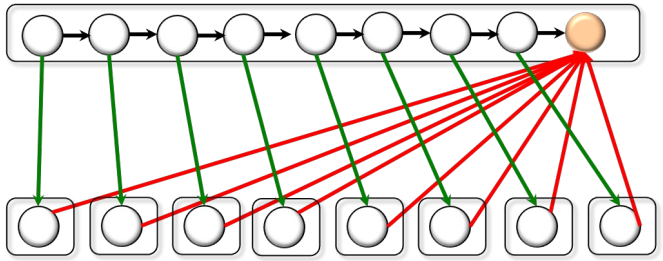
- Work-sharing
  - Busy worker re-distributes the task eagerly
  - Easy implementation through global task pool
  - Access to the global pool needs to be synchronized: scalability bottleneck

- 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
  - Distributed task pools
    - Idle worker steals the tasks from busy workers
  - Better scalability

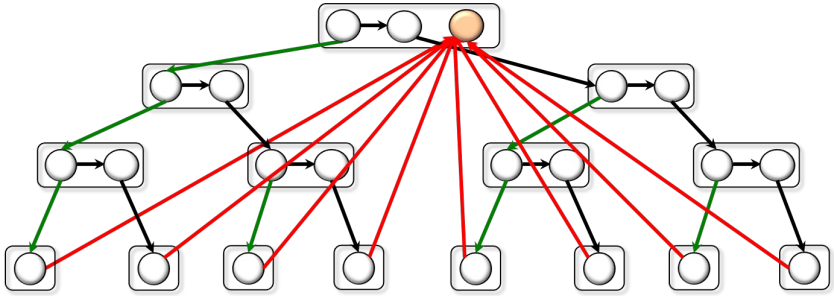
Loop Level Parallelism

```
void foo() {
  loop_domain_t loop = {0, 8, 1, 1};
  finish([&]() {
    forasync1D (&loop, [=](int i) {
      S(i); // can execute in parallel for all i
    }, MODE);
  });
}
```

MODE= FORASYNC_MODE_FLAT
Work = O(n)
CPL = O(n)



MODE= FORASYNC_MODE_RECURSIVE
Work = O(n)
CPL = O(log n)



Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec02_parallelism.pdf

Properties of a Good Locking Algorithm

- Safety guarantees
  - ***Mutual exclusion***
- Progress guarantees
  - ***Deadlock freedom:*** *system as a whole makes progress.*
    If some thread calls **lock()** and never returns, then other threads must complete **lock()** and **unlock()** calls infinitely often
  - ***Starvation freedom:*** A thread should not indefinitely hold the lock for doing some big computation while other threads keep waiting to get this lock

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Object Based Isolation for Avoiding Deadlock in async-finish Program

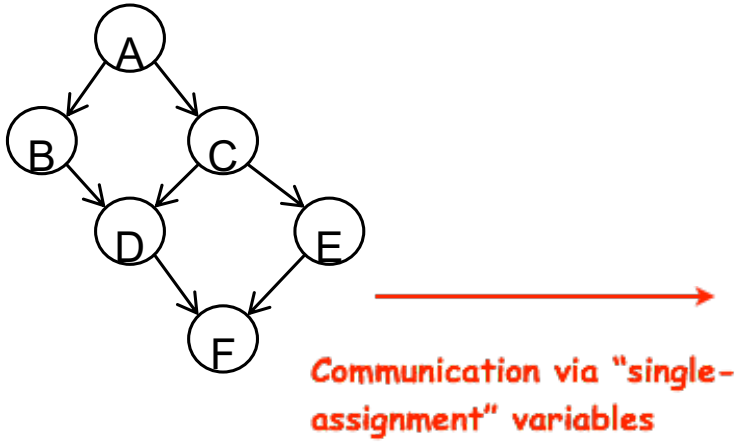
isolated(obj1, obj2, …, lambda_function)

- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated constructs that have a common object in their object lists
  - Standard isolated is equivalent to “isolated(*)” by default i.e., isolation across all objects
- Experimental implementation exists in HClib (some APIs might change in future but not the concepts)

Pros and Cons of Object Based Isolation

- Pros
  - Productivity: simpler approach than “locks”
  - Deadlock-freedom property is guaranteed
- Cons
  - Programmer needs to worry about getting the object list right
  - Objects in object list can only be specified at start of the isolated construct (new objects cannot be added later on)

Dataflow Programming (1/4)




```
future_t<void> *A = async_future([=](){...; return;});
future_t<void> *B = async_future([=](){A->get(); ...; return;});
future_t<void> *C = async_future([=](){A->get(); ...; return;});
future_t<void> *D = async_future([=](){B->get(); C->get(); ...; return;});
future_t<void> *E = async_future([=](){C->get(); ...; return;});
future_t<void> *F = async_future([=](){D->get(); E->get(); ...; return;});
F->get();
```

The above program will force workers executing an async to block until the future.get() is ready

- General idea: build an arbitrary task graph, but restrict all inter-task communications to single-assignment variables
- Semantic guarantees: race-freedom, determinism
- Deadlocks are possible due to unavailable inputs (but they are deterministic)

Source:https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec5-slides.pdf?version=1&modificationDate=1483206145961&api=v2

Dataflow Programming (2/4)

```
for(int i=0; i<N i++) A[0][i] = 1;
for(int i=0; i<N i++) A[i][0] = 1;

for(int i=1; i<N; i++) {
  for(int j=1; j<N; j++) {
    A[i][j] = F(A[i-1][j], A[i][j-1]);
  }
}
```

1	1	1	1
1			
1			
1			

Dataflow Programming (3/4)

```
for(int i=0; i<N i++) A[0][i] = 1;
for(int i=0; i<N i++) A[i][0] = 1;

for(int d=2; d<=2*(N-1); d++) {
  finish {
    forasync1D(int i=1; i<N; i++) {
      int j = d – i;
      if(j>=1 && j<N) {
        A[i][j] = F(A[i-1][j], A[i][j-1]);
      } //end-if
    } //end forasync1D
  } //end finish
} //end for-d
```

Pseudocode implementation

1	1	1	1
1	d=2	d=3	d=4
1	d=3	d=4	d=5
1	d=4	d=5	d=6

2D wavefront

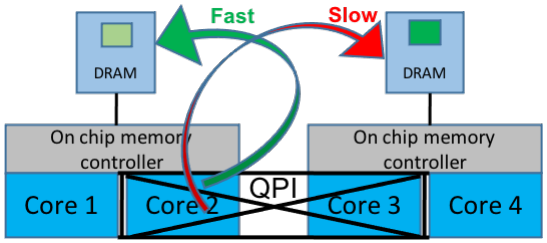
Dataflow Programming (4/4)

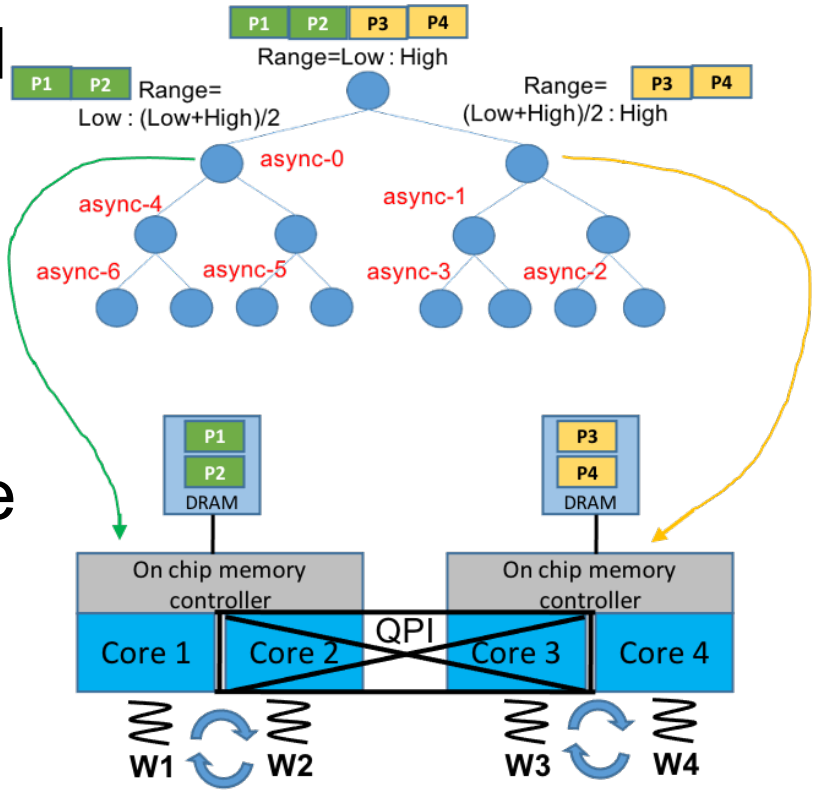
```
for(int i=0; i<N i++) promise[0][i].put(1);
for(int i=0; i<N i++) promise[i][0].put(1);
finish {
  for(int i=1; i<N; i++) {
    for(int j=1; j<N; j++) {
      async_await([=]() {
        promise[i][j].put();
      }, future[i-1][j], future[i][j-1]);
    }
  }
}
```

Pseudocode implementation

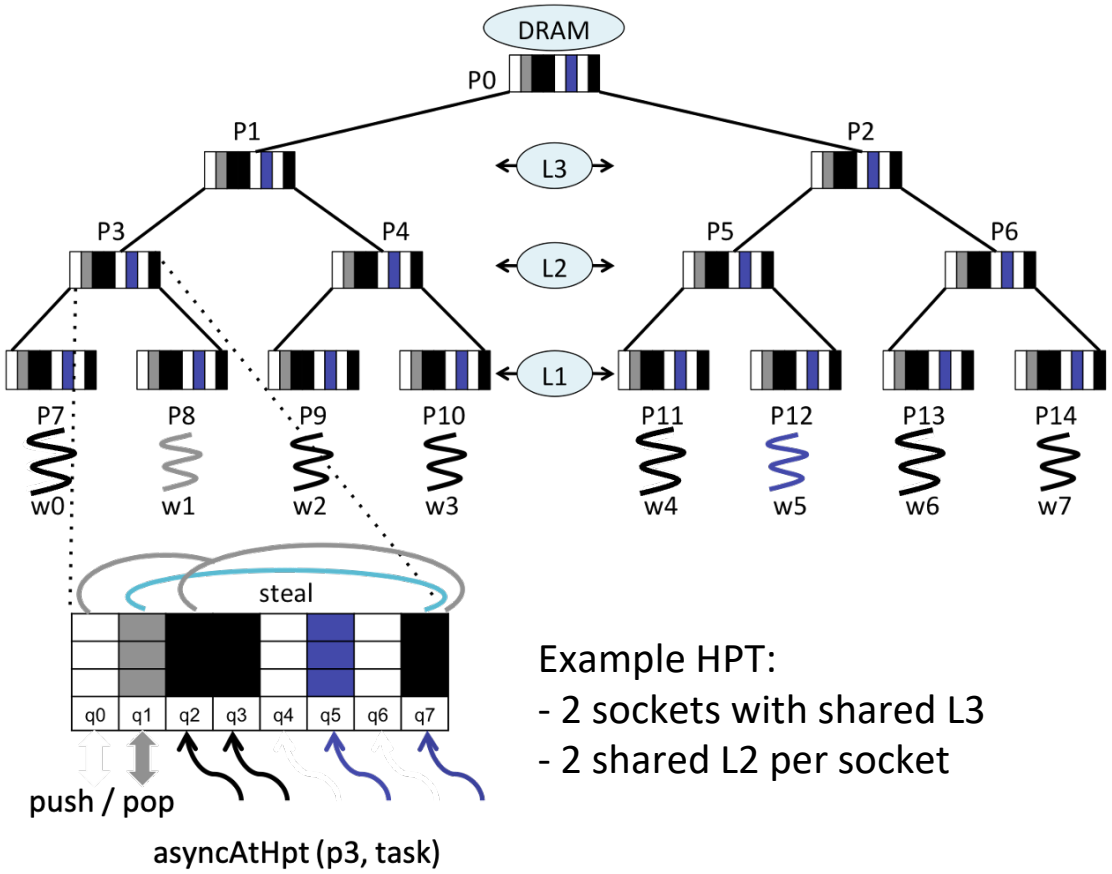
NUMA Aware Work-Stealing

- 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

Midterm Exams

- Midterm exam will be held on 01/03/26 (Sunday) 10am—11am (C12 Old academic block)
  - 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 1–13
  - 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