CSE502: Foundations of Parallel Programming

Lecture 14: Futures, Promises and Data-Driven Tasks

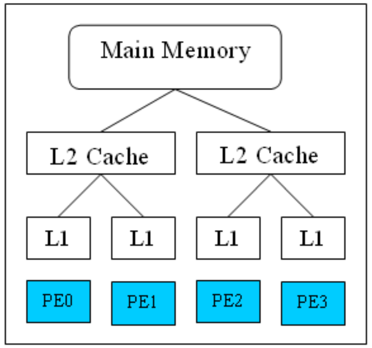
Vivek Kumar

Computer Science and Engineering

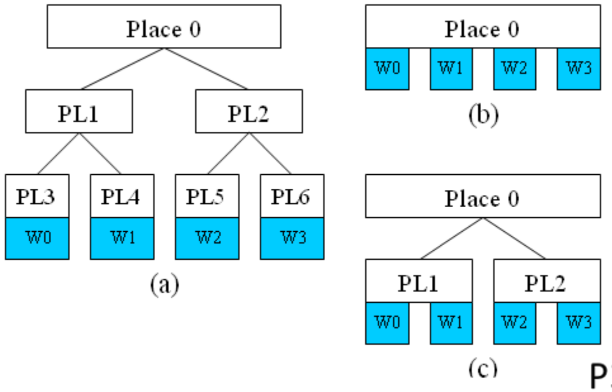
IIIT Delhi

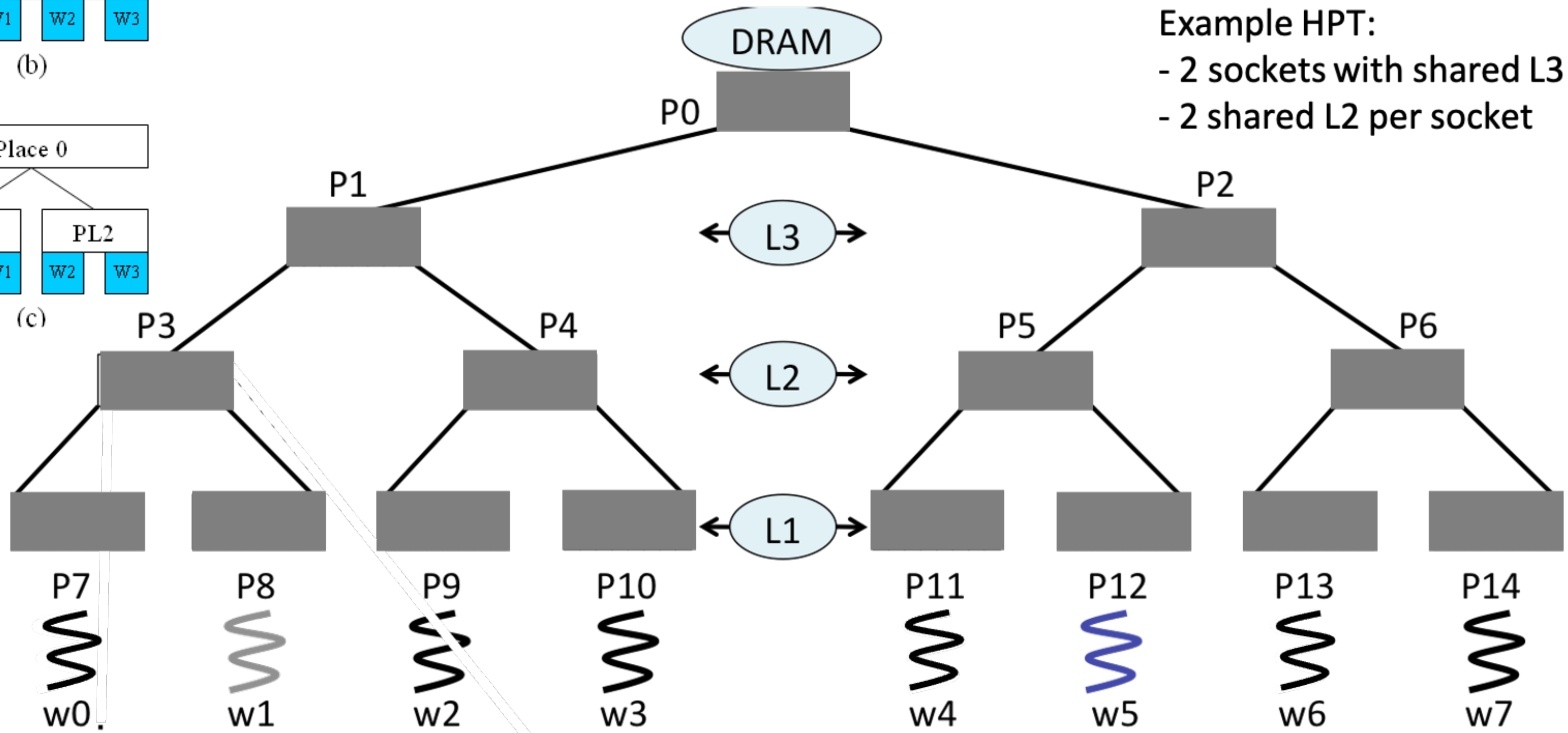
vivekk@iiitd.ac.in

Last Lecture (HPT)



A Quad-core workstation





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

asyncAtHpt(P7, S1); asyncAtHpt(P9, S2); asyncAtHpt(P12, S3);
asyncAtHpt(P14, S4);
asyncAtHpt(P2, S5);
asyncAtHpt(P4, S6);
asyncAtHpt(P6, S7);
asyncAtHpt(P0, S8);

Today’s Class

- Futures
- Promises
- Data-driven tasks

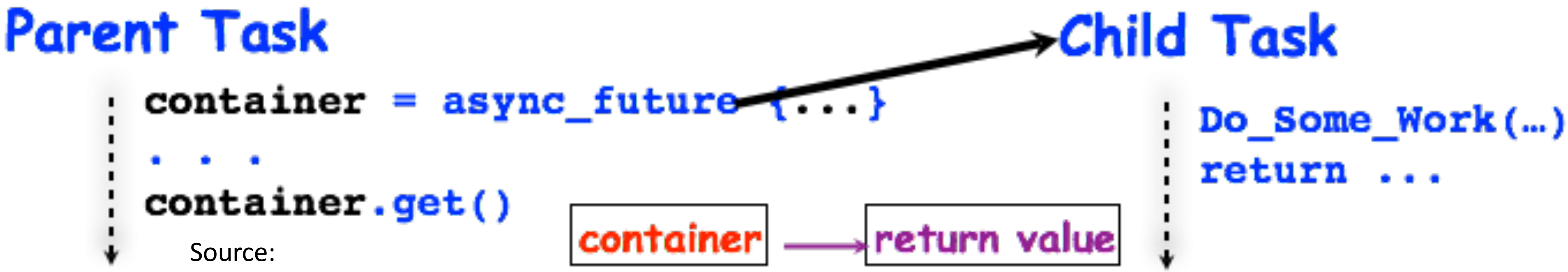
Functional Parallelism: Adding Return Values to Async Tasks

```
int main(int argc, char** argv) {
  launch([&]() {
    hclib::future_t<int> *part1 = hclib::async_future([=]() {        // Task-T1
                                      int res = DO_SOME_WORK();
                                      return res;
                                  });
    int part2 = DO_SOME_OTHER_WORK();
    //get will block until result is ready                            // Task-T2
    int total = part1->get() + part2;
  });
}
```

Two issues to be addressed:

1) Distinction between container and value in container (future)

2) Synchronization to avoid race condition in container accesses



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

HClib Futures: Tasks with Return Values

future_t<T> *f = async_future { S }

- Creates a new child task that executes S, which must terminate with a return statement and return value
- Async expression returns a pointer to a container of type future_t

T result = f.get();

- get() evaluates f and blocks if f's value is unavailable
- Unlike finish which waits for all tasks in the finish scope, a get operation only waits for the specified async_future

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

Two-Way Parallel ArraySum

```
int main(int argc, char** argv) {
  launch([&]() {
    double array[SIZE]; // initialized with random numbers

    hclib::future_t<double> *sum1 = async_future([=]() {          // Task-T1
      double sum = 0;
      for(int i=0; i<SIZE/2; i++) sum += array[i];
      return sum;
    });

    hclib::future_t<double> *sum2 = async_future([=]() {          // Task-T2
      double sum=0;
      for(int i=SIZE/2; i<SIZE; i++) sum += array[i];
      return sum;
    });

    //get will block until result is ready
    double total = sum1->get() + sum2->get();
  });
}
```

Is there a data-race?

No false sharing

Code available on github: https://github.com/vivkumar/cse502/blob/master/hclib/test/lec13/

Comparison of Future Task and Regular Async Versions of Two-Way Parallel ArraySum

- Future task version initializes two pointers to future objects, sum1 and sum2
  - No finish construct needed in this example
  - Instead parent task waits for child tasks by performing sum1->get() and sum2->get()
- Easier to guarantee absence of race conditions in Future Task version
  - No race on sum because it is declared as a local variable in both tasks T1 and T2
  - No race on future variables, sum1 and sum2, because of blocking-read semantics

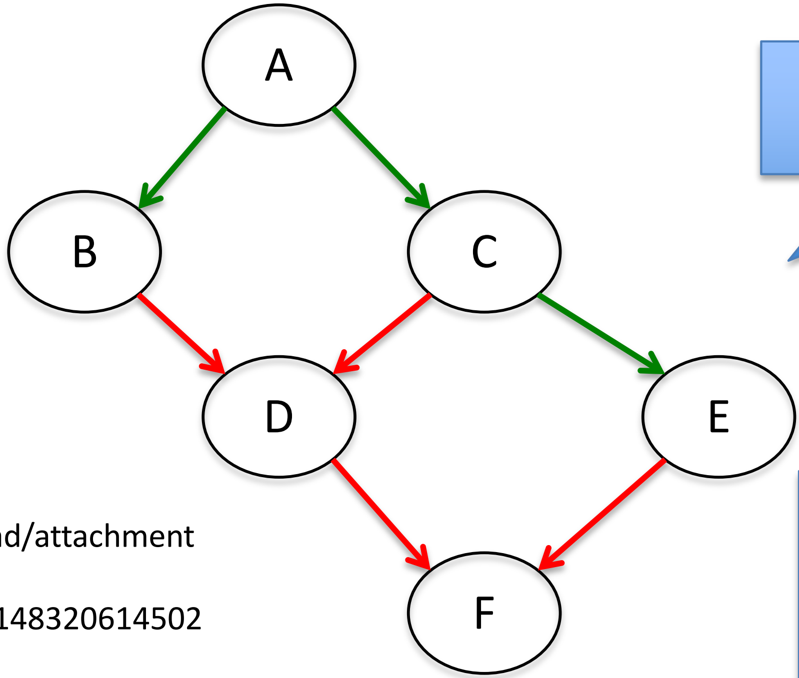
Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec5-slides.pdf?version=1&modificationDate=1483206145961&api=v2

Fibonacci with Future Tasks

```
uint64_t fib(uint64_t n) {
  if(n<THRESHOLD) return fib_serial(n);
  else {
    hclib::future_t<uint64_t> *f1 = hclib::async_future([=](){ return fib(n-1); });
    hclib::future_t<uint64_t> *f2 = hclib::async_future([=](){ return fib(n-2); });
    //get will block until result is ready
    return f1->get() + f2->get();
  }
}
```

Code available on github: https://github.com/vivkumar/cse502/blob/master/hclib/test/lec13/

Task Parallel Code with Futures to Generate Following Computation Graph



Can we use async-finish to draw this C.G??

No, Finish cannot be used to ensure that D waits for both B & C, while E waits only for C

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec14-slides-v1.key.pdf?version=1&modificationDate=1483206145028&api=v2

```
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();
```

Code available on github: https://github.com/vivkumar/cse502/blob/master/hclib/test/lec13/

Today's Class

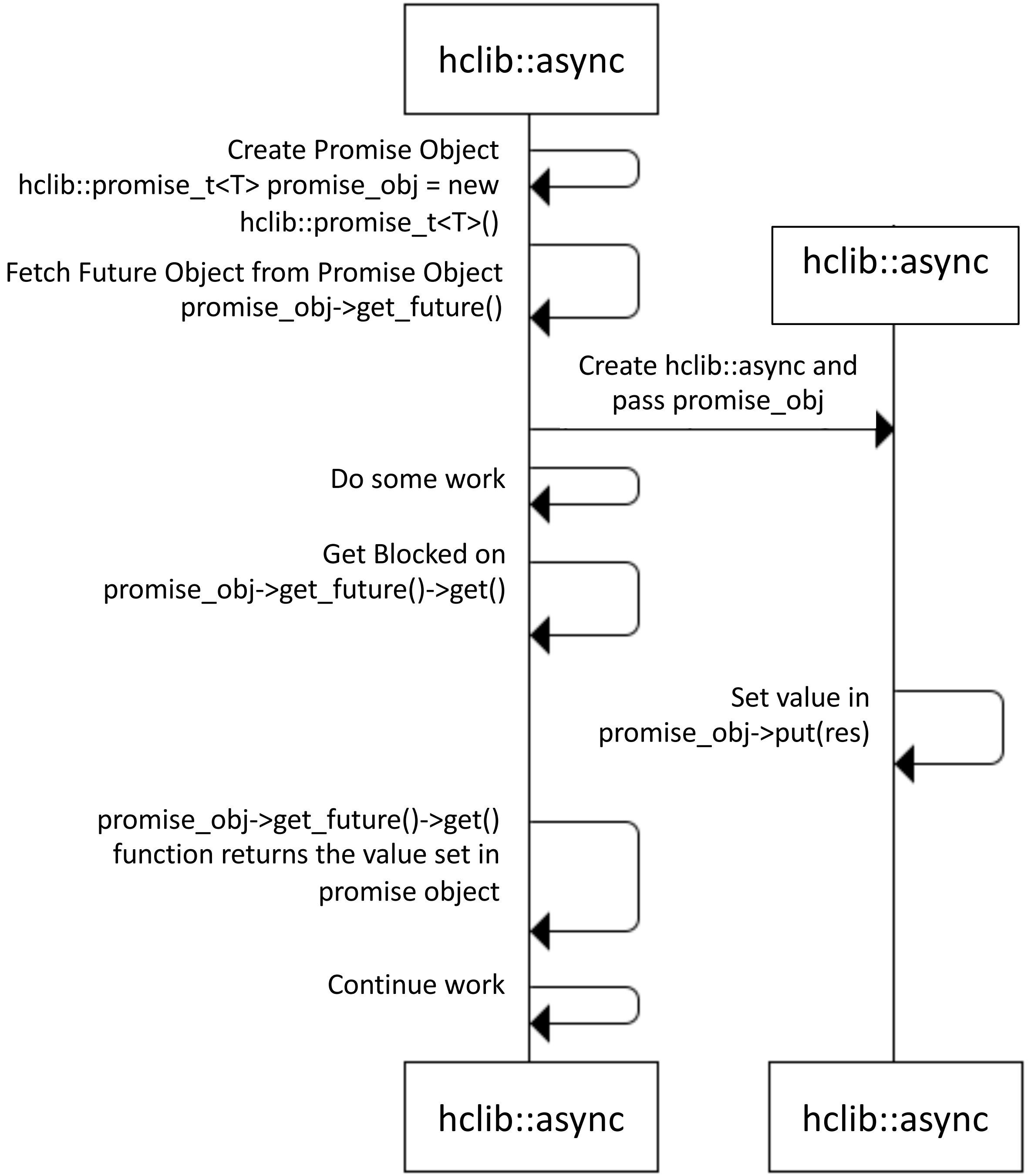
- Futures
- Promises
- Data-driven tasks

hclib::promise v/s hclib::future

- *"A promise is an object that can store a value of type T to be retrieved by a future object (possibly in another thread), offering a synchronization point"*
  - Writable end of an object
- *"A future is an object that can retrieve a value from some provider object or function, properly synchronizing this access if in different threads"*
  - Readable end of an object

Source: http://www.cplusplus.com/reference

hclib::promise_t and hclib::future_t workflow



Modification of Source: http://thispointer.com/c11-multithreading-part-8-stdfuture-stdpromise-and-returning-values-from-thread/

Rules for Read and Write

- put(...)
  - Only allowed on a promise_t object
  - Single-assignment only
  - Runtime assertions to guard against multiple put on same promise_t object
- get()
  - Only allowed on a future_t object
  - Can be called multiple times
    - Simply blocks unless put has been done
  - Never returns without a matching put performed by any **other** task

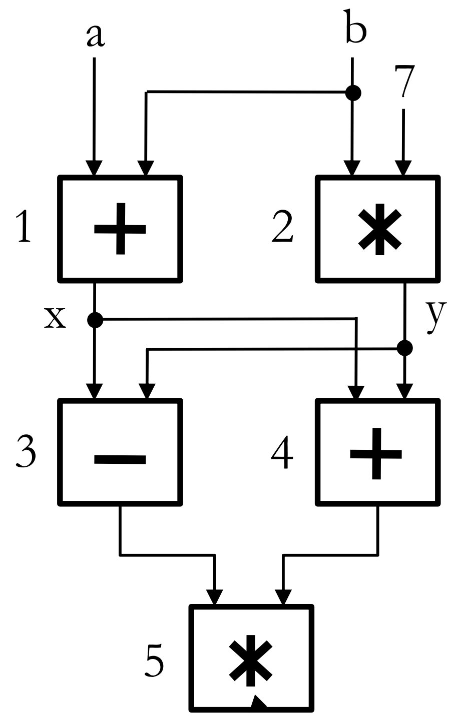
Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec5-slides.pdf?version=1&modificationDate=1483206145961&api=v2

Today's Class

- Futures
- Promises
- Data-driven tasks

Dataflow Computing Analogy

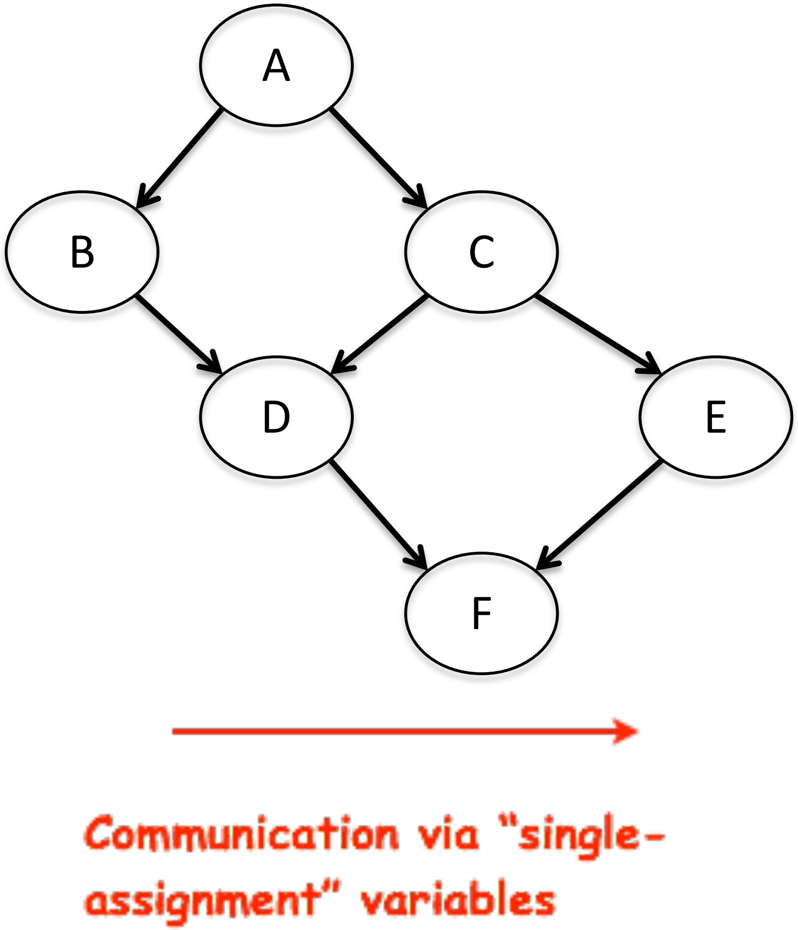
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



An operator executes when all its input values are present; copies of the result value are distributed to the destination operators.

No separate control flow

Dataflow Programming



- 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-lec14-slides-v1.key.pdf?version=1&modificationDate=1483206145028&api=v2

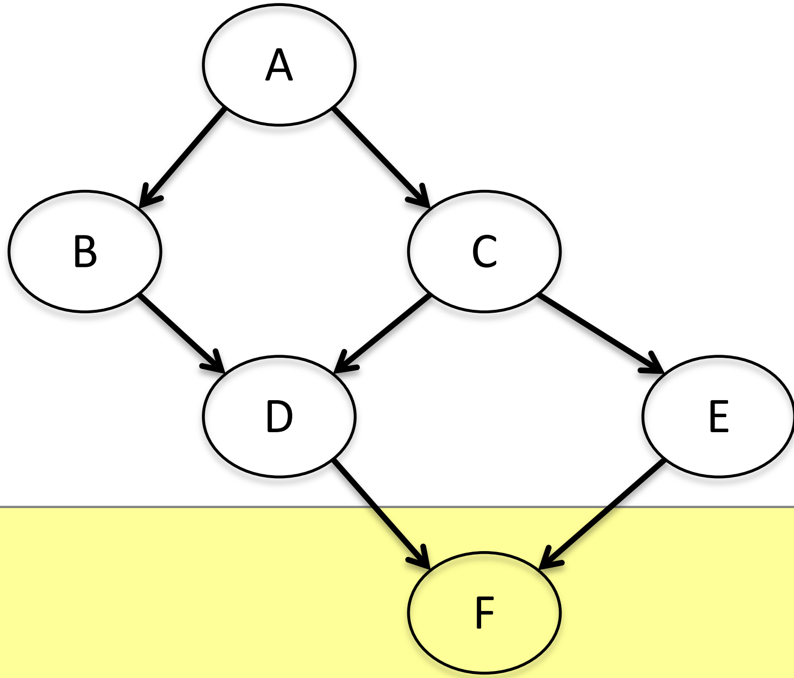
Data-Driven Task (DDT) in HClib

async_await(lambda, fObj_1, fObj_2,.....,fObj_n)

- Unlike any other async tasks that we have seen so far (async, asyncAtHpt, async_future), async_await task is pushed to the deque ONLY after all the future objects in the parameter list are ready with values inside them
  - i.e. the put has been performed on the promise end of each of the future objects

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec14-slides-v1.key.pdf?version=1&modificationDate=1483206145028&api=v2

Converting Previous Future Example to DDTs



Source:
https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec14-slides-v1.key.pdf?version=1&modificationDate=1483206145028&api=v2

```
promise_t<void>* prom_A = new promise_t<void>();
promise_t<void>* prom_B = new promise_t<void>();
promise_t<void>* prom_C = new promise_t<void>();
promise_t<void>* prom_D = new promise_t<void>();
promise_t<void>* prom_E = new promise_t<void>();
promise_t<void>* prom_F = new promise_t<void>();
finish([=]() {
  async([=]() { A(); prom_A->put(); });
  async_await([=]() { B(); prom_B->put(); }, prom_A->get_future());
  async_await([=]() { C(); prom_C->put(); }, prom_A->get_future());
  async_await([=]() { D(); prom_D->put(); }, prom_B->get_future(), prom_C->get_future());
  async_await([=]() { E(); prom_E->put(); }, prom_C->get_future());
  async_await([=]() { F(); prom_F->put(); }, prom_D->get_future(), prom_E->get_future());
});
```

Any possibility to remove this finish?

Code available on github: https://github.com/vivkumar/cse502/blob/master/hclib/test/lec13/

Fibonacci With DDTs

```
void fib(uint64_t n, hclib::promise_t<uint64_t> *prom) {
  if(n<THRESHOLD) {
    uint64_t res = fib_serial(n);
    prom->put(res);
  else {
    hclib::promise_t<uint64_t> *p1 = new hclib::promise_t<uint64_t>();
    hclib::async([=](){ fib(n-1, p1); }
    hclib::promise_t<uint64_t> *p2 = new hclib::promise_t<uint64_t>();
    hclib::async([=](){ fib(n-2, p2); }

    //wait for dependencies without using a blocking wait() operation
    hclib::async_await([=]() {
      uint64_t r = p1->get_future()->get() + p2->get_future()->get();
      prom->put(r);
      delete(p1); delete(p2);
    }, p1->get_future(), p2->get_future());
  }
}

main() {
  hclib::promise_t<uint64_t> *prom = new hclib::promise_t<uint64_t>();
  fib(n, prom); //NO finish
  uint64_t result = prom->get_future()->get();
  delete(prom);
}
```

Code available on github: https://github.com/vivkumar/cse502/blob/master/hclib/test/lec13/

Deadlock Example with DDT

```
promise_t<void>* left = new promise_t<void>();
promise_t<void>* right = new promise_t<void>();
finish([=]() {
  async_await([=]() { right->put(); }, left->get_future());
  async_await([=]() { left->put(); }, right->get_future());
});
```

Source:
https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec14-slides-v1.key.pdf?version=1&modificationDate=1483206145028&api=v2

Next Class

- Introduction to Cilk programming language

Acknowledgements

- Several of the slides used in this course are borrowed from the following online course materials:
  - Course COMP322, Prof. Vivek Sarkar, Rice University
  - Course COMP 422, Prof. John Mellor-Crummey, Rice University
  - Course CSE539S, Prof. I-Ting Angelina Lee, Washington University in St. Louis
- Contents are also borrowed from following sources:
  - “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003
  - https://computing.llnl.gov/tutorials/parallel_comp/
  - https://images.google.com/