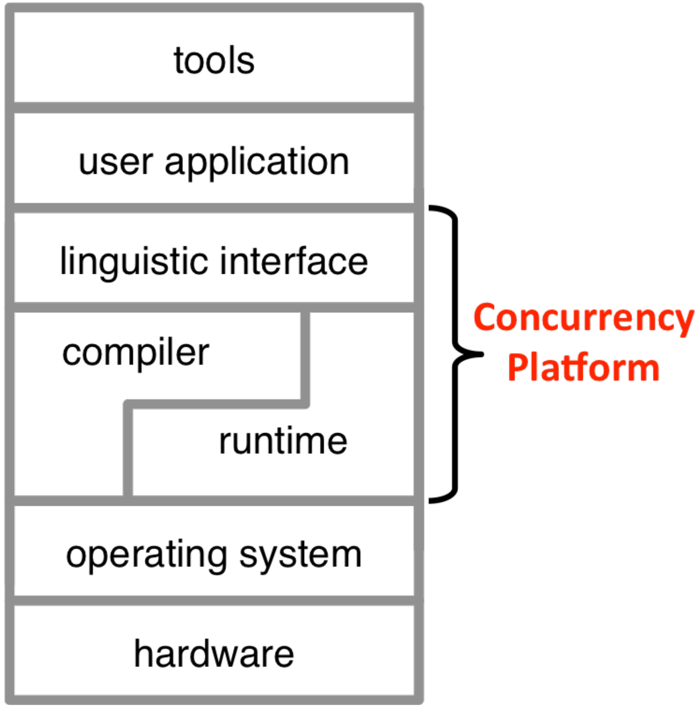
CSE502: Foundations of Parallel Programming

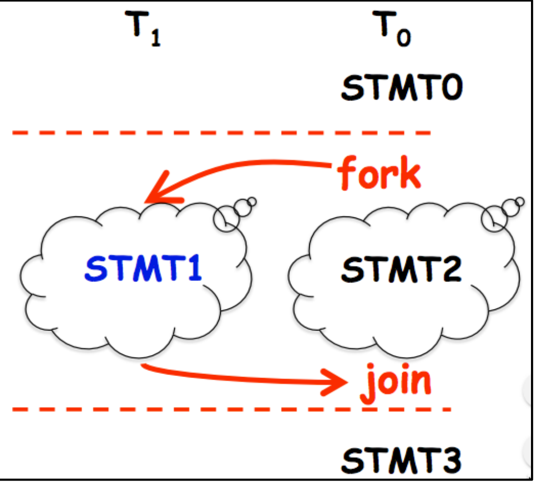
Lecture 06: Introduction to HClib, Computation Graphs and Ideal Parallelism

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

Last Class




```
// 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
```



finish {

async { Complete your FPP assignment **}**

~~**async**~~ **{** Wash your clothes in washing machine **}**

}

finish {

async { Watch movies on laptop **}**

async { Talk to father

Talk to mother **}**

async { Buy fruits online using your smartphone **}**

~~**async**~~ **{** Make your bed **}**

}

Post on Facebook that you are done with all your tasks!

Today's Class

- Introduction to Habanero-C concurrency platform
- Computation graph
  - Ideal parallelism
  - Introduction to data races

Installing Habanero-C library (1/3)

- Concurrency platform from Rice University that supports async-finish based parallel programming model
  - Supported on Linux and MacOS
  - Short name: **HClib**
  - Result of hard work from several researchers at Rice University
    - https://wiki.rice.edu/confluence/display/HABANERO/People
- Prerequisites
  - libxml2 and libxml2-devel
  - Check if its already installed on your Linux OS
    - Default installation location in one my Ubuntu machine
      - Headers in: /usr/include/libxml2
      - Libraries in: /usr/lib/x86_64-linux-gnu
  - gcc >= 4.9.0 (C++11 complaint)
  - On Mac OS you may install using brew
    - brew install libxml2   (installs everything)

Installing Habanero-C library (2/3)

- Installation (one time thing)
  - git clone https://github.com/vivkumar/cse502.git
    - This version is copied from the main hclib repository hosted at https://github.com/habanero-rice/hclib
  - cd cse532/hclib
  - ./install.sh
  - source <absolute path to cse502 dir>/hclib/hclib-install/bin/hclib_setup_env.sh
  - This source is always required once in a new terminal for both compiling and running programs that uses hclib

Installing Habanero-C library (3/3)

- Building an app test.cpp that uses hclib
  - Fibonacci code can be found in following directory:
    - cd cse502/hclib/test/lec05
  - Must use the header file hclib_cpp.h
    - #include “hclib_cpp.h”
  - All hclib programming constructs should be used with “hclib::” namespace
    - E.g., hclib::async, hclib::finish, hclib::launch
    - All the above constructs accepts C++11 lambda function
      - Several online resources on C++11 lambda function. Go through it on your own
  - Makefile is available in test directories for building the applications
  - Executing the application
    - HCLIB_WORKERS=<number of workers> ./test
      - HCLIB_WORKERS sets the total number of Pthread helper threads you want to create to solve the application in parallel
    - Time the result using unix “time” command (or use timing APIs):
      - time HCLIB_WORKERS=<number of workers> ./test

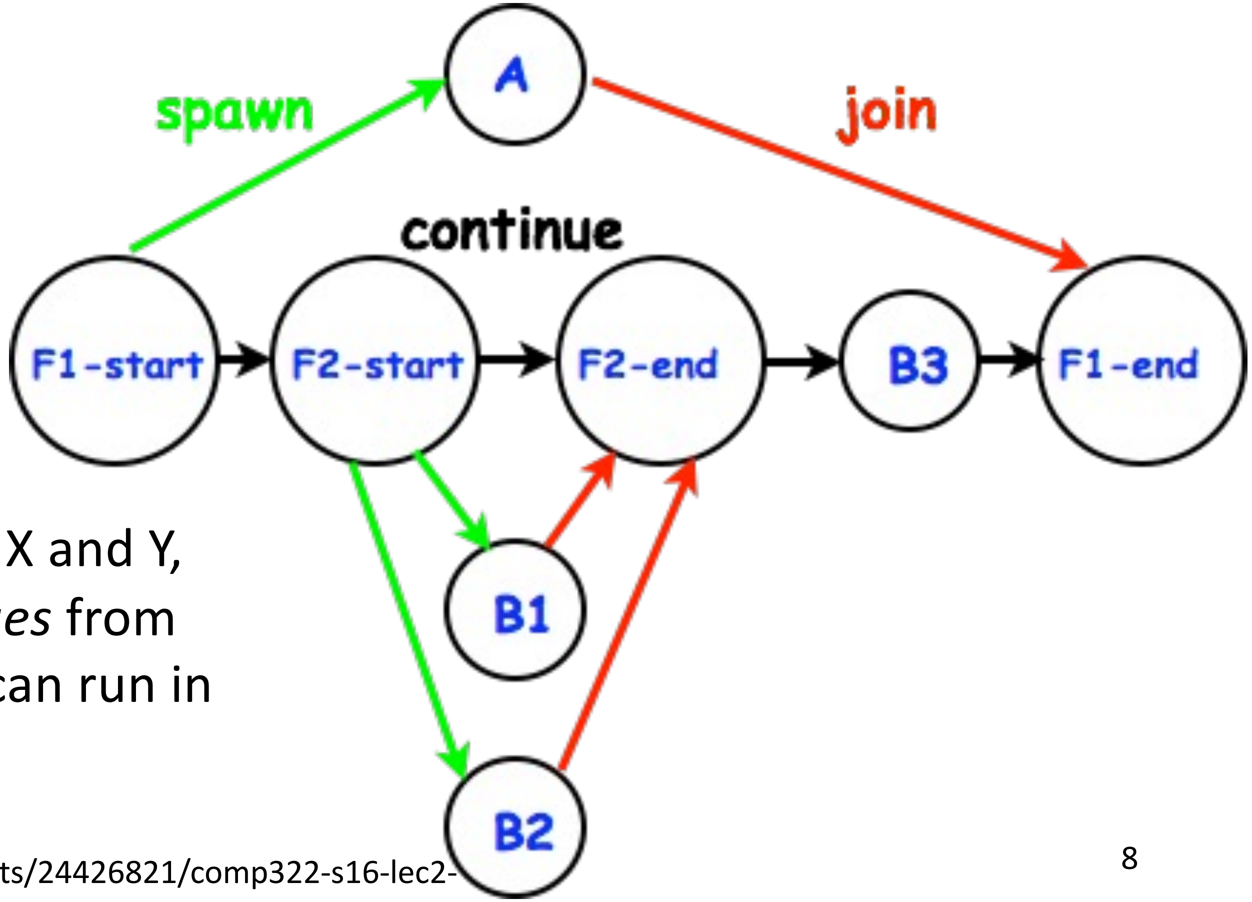
Today's Class

- Introduction to Habanero-C concurrency platform
- Computation graph
  - Ideal parallelism
  - Introduction to data races

Which Statements can Potentially Execute in Parallel with Each Other?

```
finish {            // F1
  async A;
  finish {          // F2
    async B1;
    async B2;
  }                 // F2
  B3;
}                   // F1
```

Computation Graph

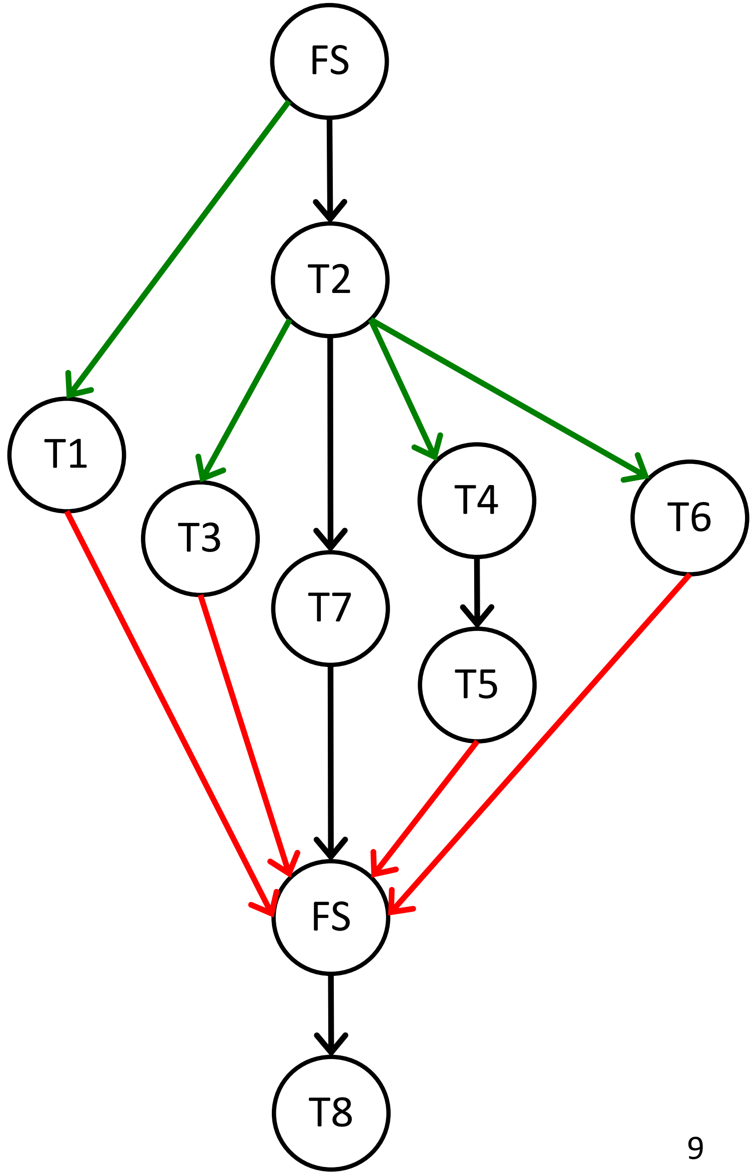


Key idea: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec2-slides-v1.pdf?version=1&modificationDate=1483206145211&api=v2

Question: Draw Computation Graph for this Parallel Computation

```
finish {                                                        // FS
    async {Wash your clothes in washing machine }               // T1
            Complete your FPP assignment                        // 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
```



Computation Graphs

- A Computation Graph (CG) captures the dynamic execution of a parallel program, for a specific input
- CG nodes are "steps" in the program's execution
  - A step is a sequential sub-computation without any async, begin-finish and end-finish operations
- CG edges represent ordering constraints
  - "Continue" edges define sequencing of steps within a task
  - "Spawn" edges connect parent tasks to child async tasks
  - "Join" edges connect the end of each async task to its IEF's end-finish operations
- All computation graphs must be acyclic
  - It is not possible for a node to depend on itself
- Computation graphs are examples of "directed acyclic graphs" (dags)

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec2-slides-v1.pdf?version=1&modificationDate=1483206145211&api=v2

Execution Time Analysis for Computation Graphs

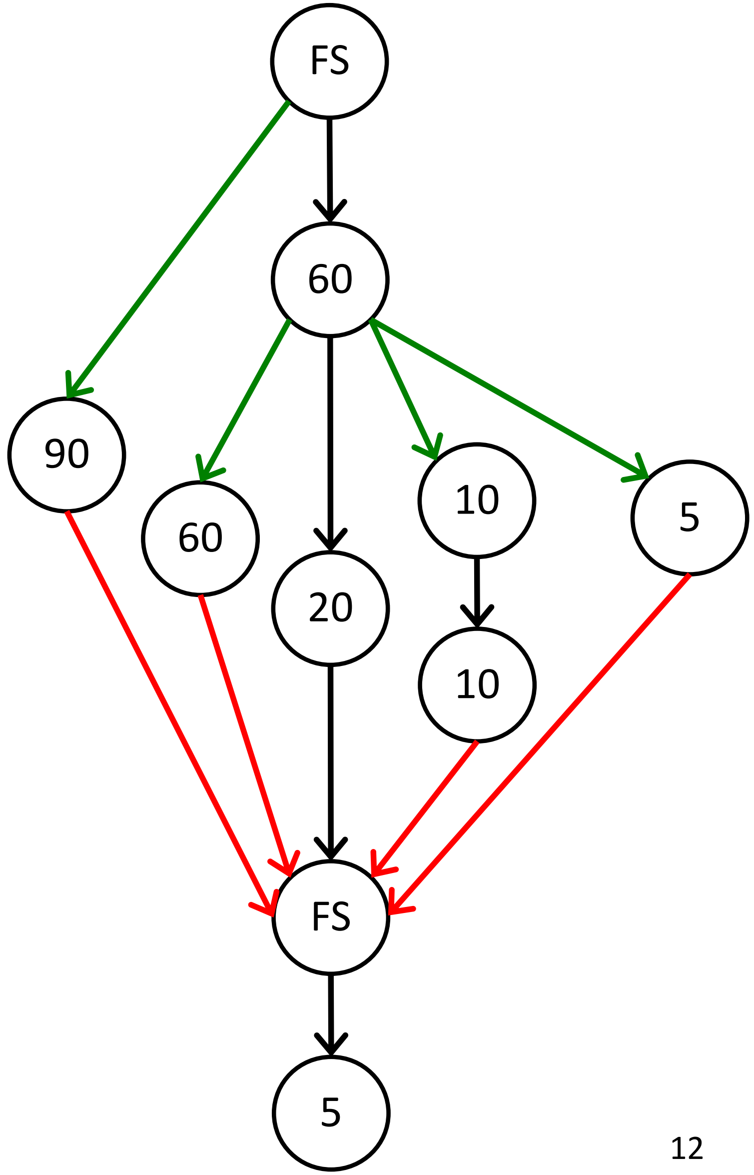
Define

- TIME(N) = execution time of node N
- WORK(G) = sum of TIME(N), for all nodes N in CG G
  - WORK(G) is the total work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up execution times of all nodes in the path
  - Such paths are called *critical paths*
  - CPL(G) is the length of these paths (critical path length, also referred to as the ***span*** of the graph)
  - CPL(G) is also the smallest possible execution time for the computation graph

Source:
https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec2-slides-v1.pdf?version=1&modificationDate=1483206145211&api=v2

Question: What is the Critical Path Length of this Parallel Computation?

```
finish {                                                        // FS
    async {Wash your clothes in washing machine }               // T1
            Complete your FPP assignment                        // 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
```

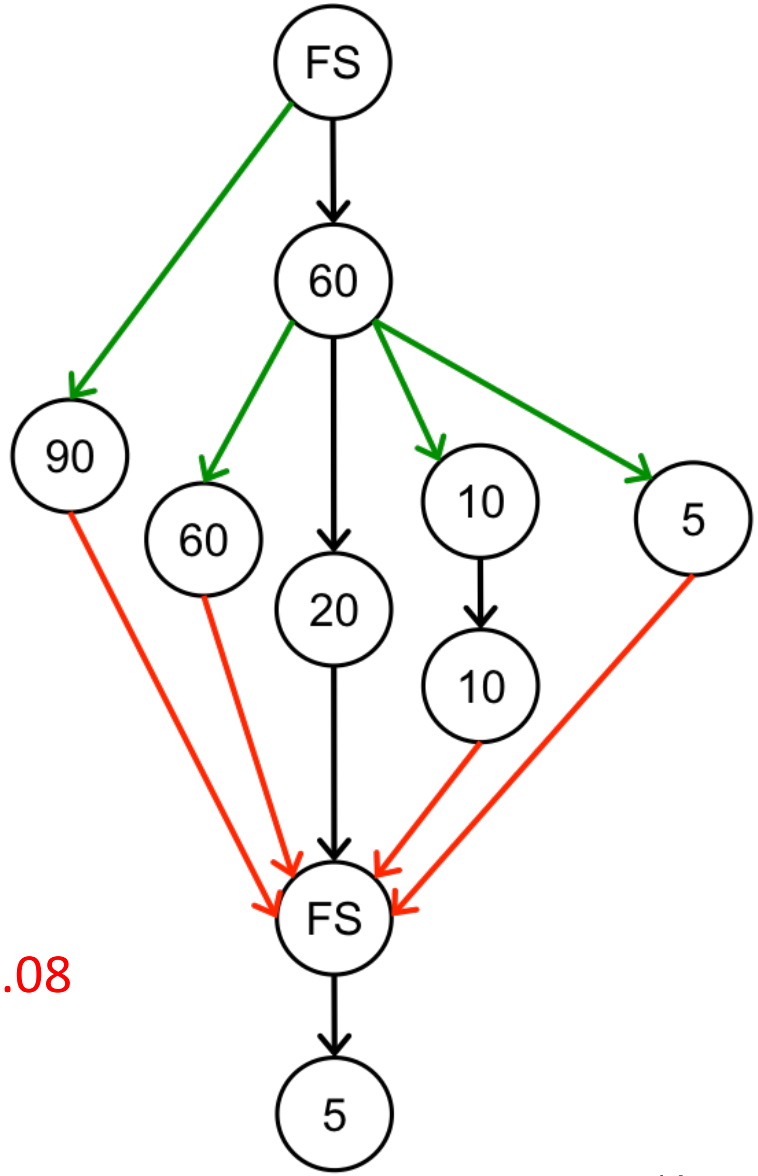


Today's Class

- Introduction to Habanero-C concurrency platform
- Computation graph
  - Ideal parallelism
  - Introduction to data races

Ideal Parallelism

- Define 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) = 260

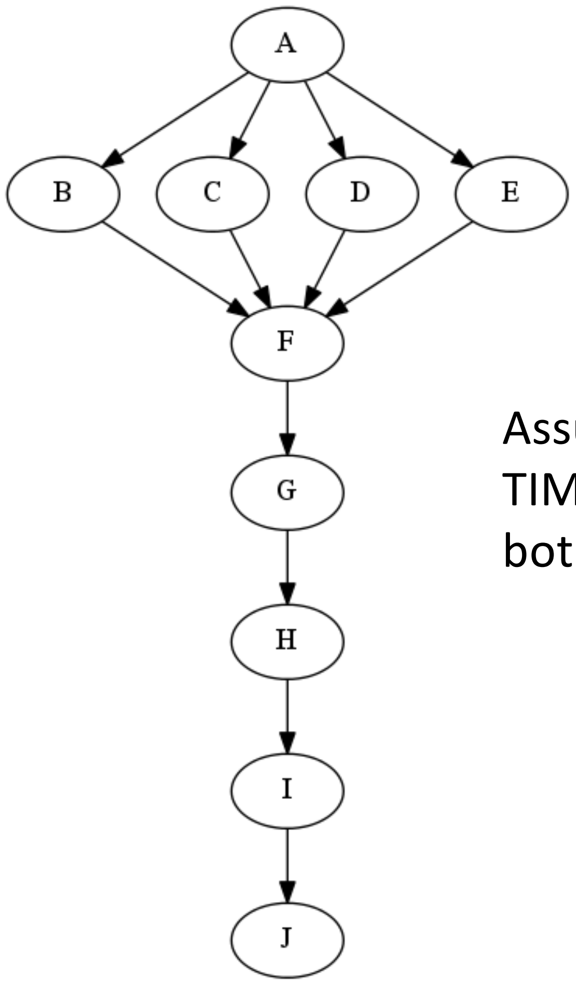
CPL(G) = 125

Ideal Parallelism = WORK(G)/CPL(G) = 260/125 ~ 2.08

Source:
https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec2-slides-v1.pdf?version=1&modificationDate=1483206145211&api=v2

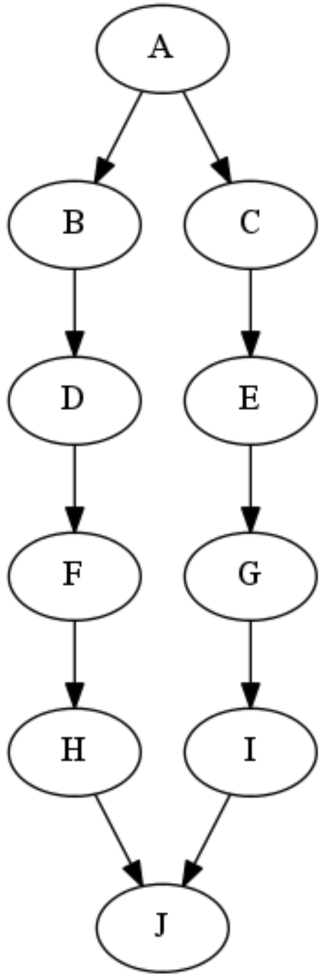
Question: Which Computation Graph has more Ideal Parallelism?

Computation Graph 1



Assume that all nodes have TIME = 1, so WORK = 10 for both graphs.

Computation Graph 2



Today's Class

- Introduction to Habanero-C concurrency platform
- Computation graph
  - Ideal parallelism
  - Introduction to data races

Data Races

- A data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:
  1. S1 does not depend on S2 and S2 does not depend on S1, i.e., S1 and S2 can potentially execute in parallel, and
  2. Both S1 and S2 read or write L, and at least one of the accesses is a write.
- A data-race is usually considered an error. The result of a read operation in a data race is undefined. The result of a write operation is undefined if there are two or more writes to the same location.
  - Note that our definition of data race includes the case that both S1 and S2 write the same value in location L, even if that may not be considered an error.
- Above definition includes all "**potential**" data races i.e., we consider it to be a data race even if S1 and S2 end up executing on the same processor.

Source:
https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec2-slides-v1.pdf?version=1&modificationDate=1483206145211&api=v2

Question: Locate the Data Race Bug

```
...
double a[SIZE];
sum1 = sum2 = 0;

   async { for(int i=0; i<SIZE/2; i++) sum1 += a[i]; }
   async { for(int i=SIZE/2; i<SIZE; i++) sum2+=a[i]; }

double sum = sum1 + sum2;
```

Data race bug! Reads and writes can occur in parallel on sum1 and sum2, in this example!

ArraySum Example

```
...
double a[SIZE];
sum1 = sum2 = 0;
finish {
   async { for(int i=0; i<SIZE/2; i++) sum1 += a[i]; }
   async { for(int i=SIZE/2; i<SIZE; i++) sum2+=a[i]; }
}
double sum = sum1 + sum2; // Now gives correct result
```

In this situation, finish was able to resolve the Data Race..

How to Parallelize Matrix Multiplication ?

```
for (int i = 0 ; i < N ; i++)
  for (int j = 0 ; j < N ; j++)

      for (int k = 0 ; k < N ; k++)

        C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Source:
https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec2-slides-v1.pdf?version=1&modificationDate=1483206145211&api=v2

Is this a Correct Solution ?

```
finish {
 for (int i = 0 ; i < N ; i++)
  for (int j = 0 ; j < N ; j++)

    for (int k = 0 ; k < N ; k++)
      async {
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
    } // async
} // finish
```

Data race bug! Reads and writes can occur in parallel on the same C[i][j] location, in this example!

One Possible Solution

```
finish {
 for (int i = 0 ; i < N ; i++)
  for (int j = 0 ; j < N ; j++)
   async {
    for (int k = 0 ; k < N ; k++)

     C[i][j] = C[i][j] + A[i][k] * B[k][j];
   } // async
} // finish
```

In this situation, by changing the position of async we are able to resolve the data race..

This program generates N2 parallel async tasks, one to compute each C[i][j] element of the output array.

What to do in Case of Valid Data Races that Cannot be Resolved with just async-finish ?

- We saw in Lecture 04 that we were able to resolve the data races between Pthreads by using pthread_mutex_locks/pthread_mutex_unlock
- It is perfectly legal to do this even in case of async-finish programs, but that's not productivity!
  - Highly prone to deadlocks and performance loss
  - In later lectures we will see a better way that provides both productivity and performance

Next Class

- Greedy scheduling of computation graphs
  - Lower bound
  - Upper bound
- Thread pools

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/