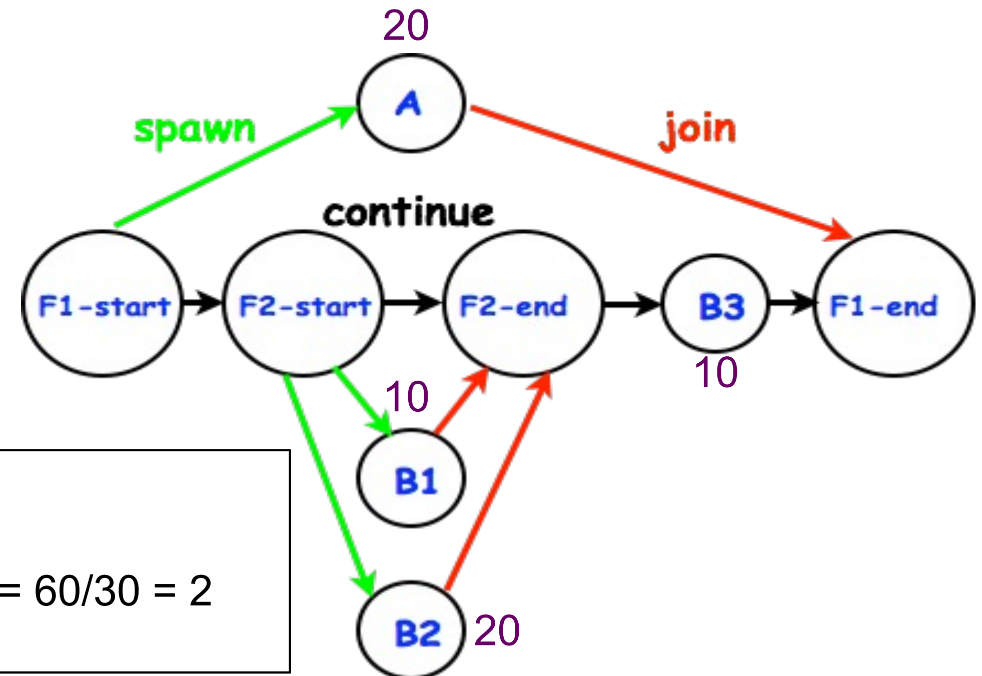
Lecture 06: Greedy Scheduler

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

 © Vivek Kumar

Last Lecture

- Computation graph
  - Ideal parallelism
  - Introduction to data races




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

Work = 60
CPL = 30
Ideal Parallelism = Work/CPL = 60/30 = 2

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.

Today's Lecture

- **Greedy scheduling of computation graph on fixed number of processors**
  - **Lower and upper bound on execution time**
- Thread pool

Greedy Schedule

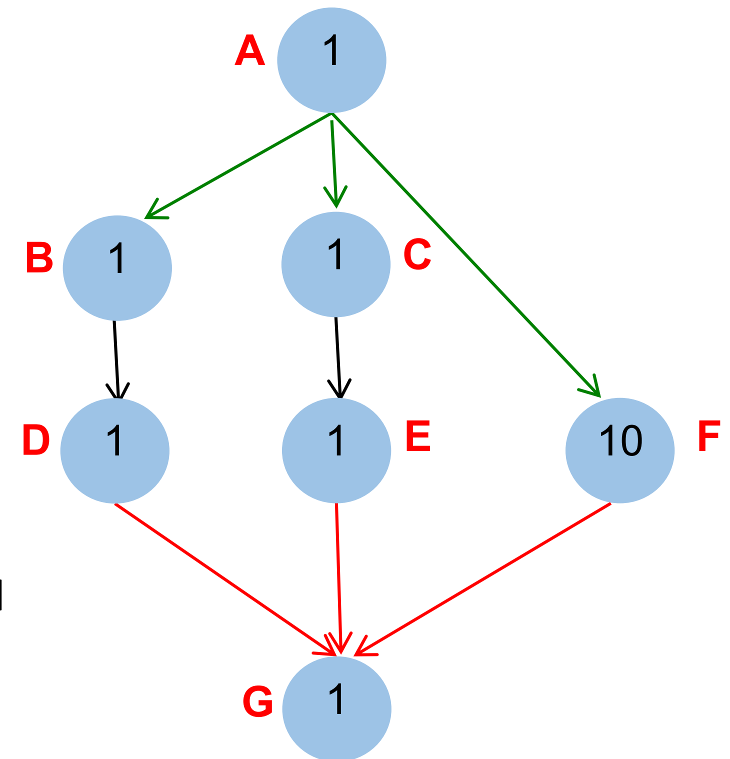
- A greedy schedule is one that never leaves a processor idle when one or more nodes in a CG are ready for execution
- A node is **ready** for execution if all its predecessors have been **executed**
- Observations
  - T_1 = WORK(G), for all greedy schedules
  - T_∞ = CPL(G), for all greedy schedules
- where T_P = execution time of a schedule for computation graph G on P processors

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec3-slides-v1.pdf?version=1&modificationDate=1483206145596&api=v2

Scheduling of a Computation Graph on a fixed number of processors: Example

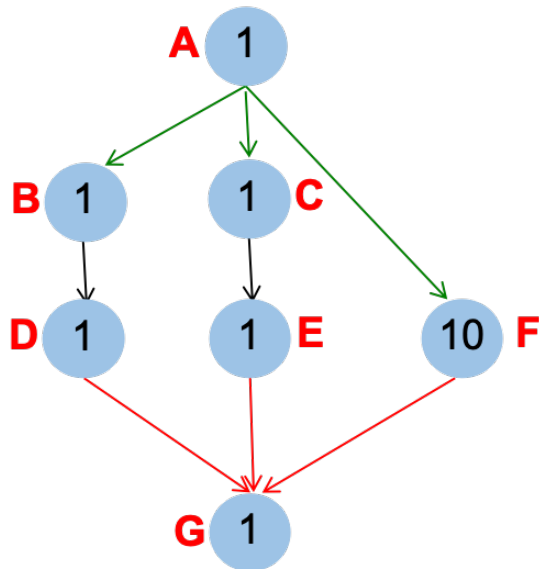
```
A();                    // 1 units
finish {
    async {
        B();            // 1 units
        D();            // 1 units
    }
    async {
        C();            // 1 units
        E();            // 1 units
    }
    async F();          // 10 units
}
G();                    // 1 units
```

- Spawn edge
- Continue edge
- Join edge
- Node label = time(N), for all nodes N in the graph
- CPL (Graph) = 12
- Work (Graph) = 16
- Ideal Parallelism = 16/12 = 1.33



Lower Bounds on Execution Time of Greedy Schedules

- **Best** possible execution time of this computation graph on **two** processors



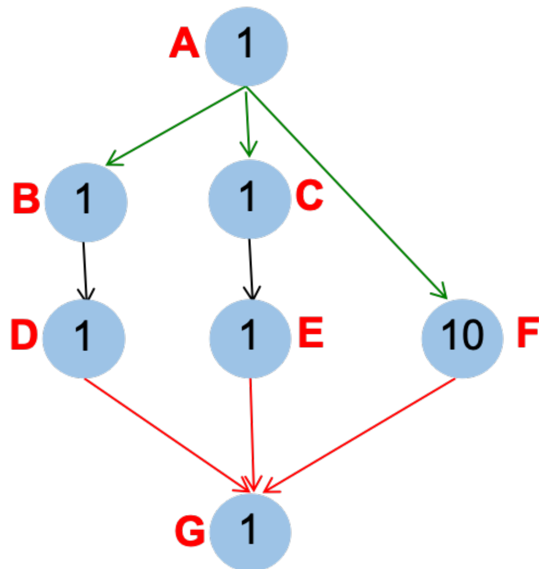
Start Time	Proc1	Proc2
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

Lower Bounds on Execution Time of Greedy Schedules

- Let T_P = execution time of a schedule for computation graph G on P processors
  - Can be different for different schedules
- Lower bounds for all greedy schedules
  - Capacity bound: $T_P \geq WORK(G)/P$
  - Critical path bound: $T_P \geq CPL(G)$
- Putting them together
  - $T_P \geq \max(WORK(G)/P, CPL(G))$

Upper Bound on Execution Time of Greedy Schedules

- **Worst** possible execution time of this computation graph on **two** processors



Start Time	Proc1	Proc2
0	A	
1	B	C
2	D	E
3	F	
4	F	
5	F	
6	F	
7	F	
8	F	
9	F	
10	F	
11	F	
12	F	
13	G	
14	Time = 14	

Upper Bound on Execution Time of Greedy Schedules

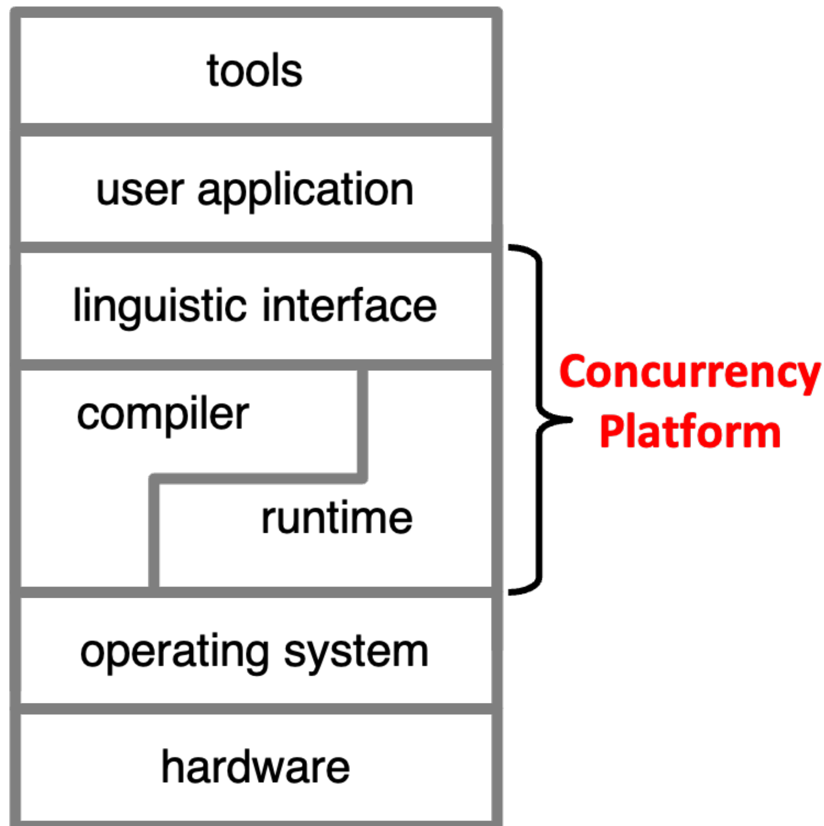
- Define a time step to be **complete** if ≥ P nodes are ready at that time, or **incomplete** otherwise
  - $\text{Total Steps}_{\text{Complete}} \leq \text{WORK}(G)/P$
    - If ≥ P nodes are ready, all of the P processors would be used
  - $\text{Total Steps}_{\text{Incomplete}} \leq \text{CPL}(G)$
    - If < P nodes are ready, then fewer than P tasks can run
    - Some nodes in CP can get executed during complete steps. Hence, incomplete steps may be less than CPL
  - $\text{Total Time Steps} = \#\text{Steps}_{\text{Complete}} + \#\text{Steps}_{\text{Incomplete}}$
    - As every time step can be either complete or incomplete type
  - $T_P \leq \text{WORK}(G)/P + \text{CPL}(G)$
    - Theorem [Graham'68, Brent'74]

Start Time	Proc1	Proc2
0	A	
1	B	C
2	D	E
3	F	
4	F	
5	F	
6	F	
7	F	
8	F	
9	F	
10	F	
11	F	
12	F	
13	G	
14	Time = 14	

Today's Lecture

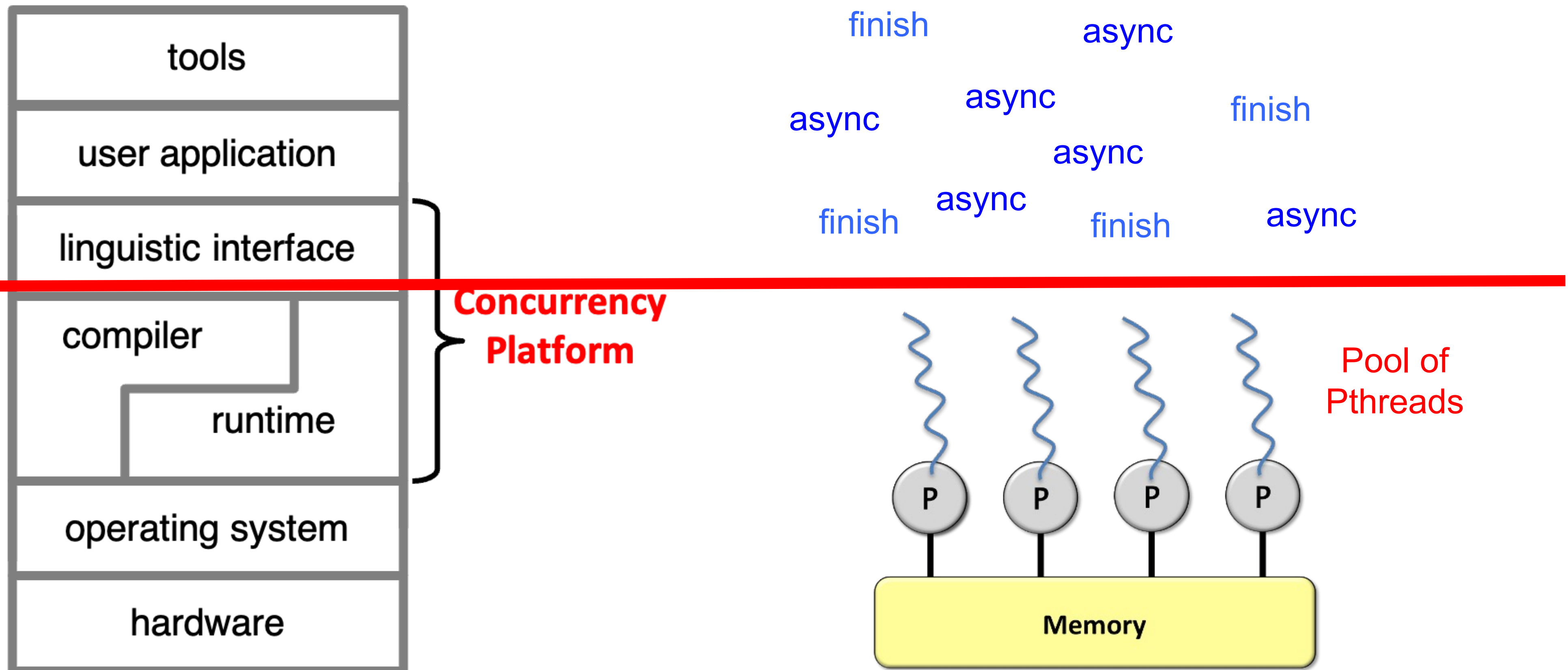
- Greedy scheduling of computation graph on fixed number of processors
  - Lower and upper bound on execution time
- **Thread pool**

Concurrency Platforms (Recap Lec04)



- A concurrency platform should provide:
  - an interface for specifying the ***logical parallelism*** of the computation;
  - a runtime layer to automate scheduling and synchronization; and
  - guarantees of performance and resource utilization competitive with hand-tuned code.

Thread Pool in Concurrency Platforms



In this course we are only going to consider the case where a thread pool has total number of threads (Pthreads) equal to total number of available cores

Mapping the Linguistic Interface to the Thread Pool Runtime

- Compiler based runtimes
  - User code translated to runtime code and then compiled using a native compiler (e.g., gcc)
  - Compiler maintenance is a costly affair and it is not so easy to use new features from mainstream languages
  - Using standard debugger (e.g., gdb) is not possible as the line number information inside the symbol table is w.r.t. the compiler generated code and not w.r.t. the user written code
  - However, compiler based approach provide several opportunities for code optimizations and doing smart things
- Library based runtimes — Our focus
  - Removes all the drawbacks of a compiler based approach

Tasks Based Parallel Programming Model

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

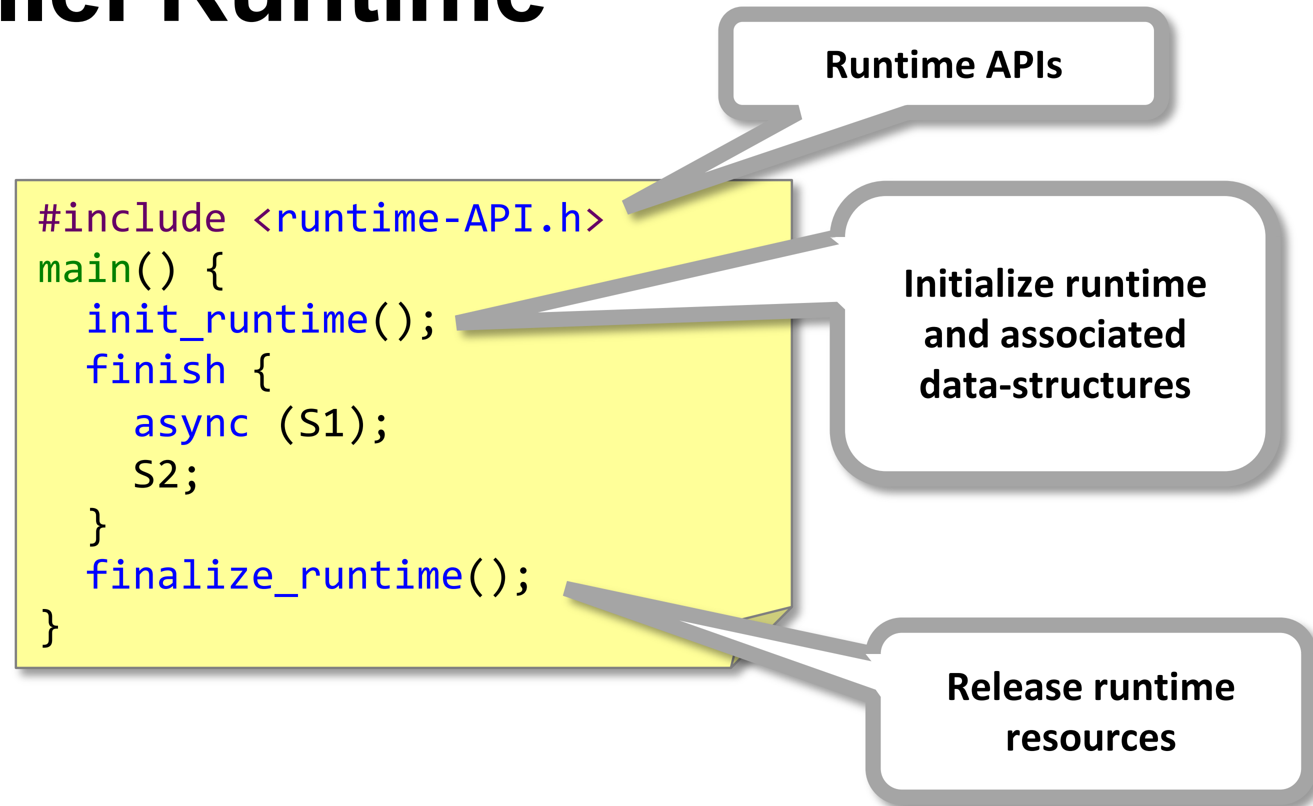
- High **productivity** due to **serial elision**
  - Removing all async and finish constructs results in a valid sequential program
  - Several existing frameworks support this programming model, although the name of the APIs for tasking would be different
- Uses an underlying high **performance parallel runtime** system for load balancing of dynamically created asynchronous tasks

	Java Fork/Join	Cilk	OpenMP	HClib[1]	TBB	C++11
Serial Elision	NO	Yes spawn-sync	Yes #pragma omp task #pragma omp taskwait	Yes async-finish	NO	Yes async-future
Performance	Limited	High	Limited	High	High	NO

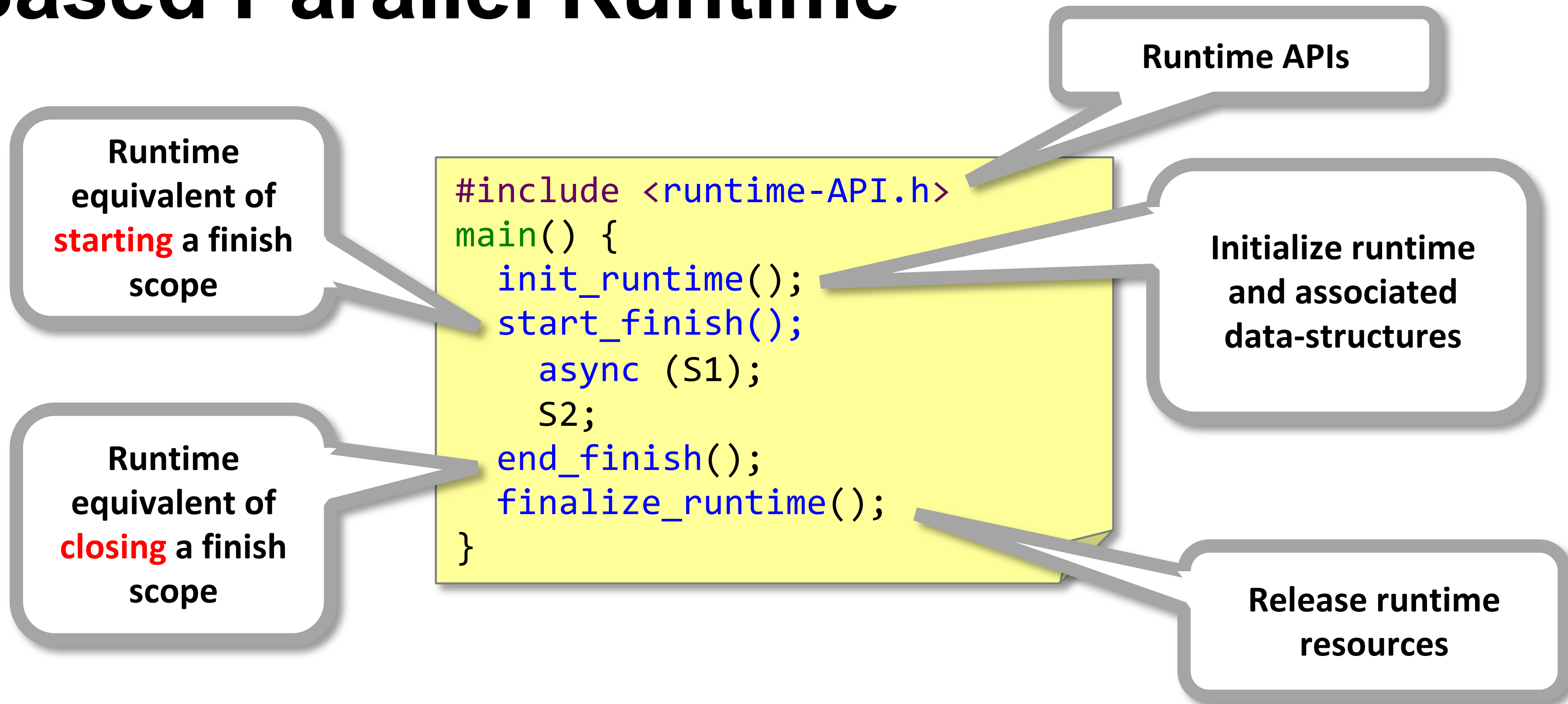
Popular options for simple tasks based parallel programming model

[1] http://habanero-rice.github.io/hclib/

Mapping the Linguistic Interface to Library Based Parallel Runtime



Mapping the Linguistic Interface to Library Based Parallel Runtime



Mapping the Linguistic Interface to Library Based Parallel Runtime

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
volatile boolean shutdown = false;
void init_runtime() {
  int size = runtime_pool_size();
  for(int i=1; i<size; i++) {
    pthread_create(worker_routine);
  }
}
```

```
void worker_routine() {
  while( !shutdown ) {
     find_and_execute_task();
  }
}
```

Mapping the Linguistic Interface to Library Based Parallel Runtime

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
volatile int finish_counter = 0;
void start_finish() {
  finish_counter = 0; //reset
}
```

Note: in case of nested finish (e.g., Fibonacci), we need a better way to manage finish scopes. Recall, in Fibonacci every fib(n) call created a new finish, which ultimately creates a tree of finishes

Mapping the Linguistic Interface to Library Based Parallel Runtime

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
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;
}
```

Note: Runtime stores pointer to the tasks passed in the async. To ensure valid pointer during task execution, we heap allocate the task and store pointer to the task on heap.

Note: there are better ways to increment finish counter rather than doing it inside locks

Mapping the Linguistic Interface to Library Based Parallel Runtime

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

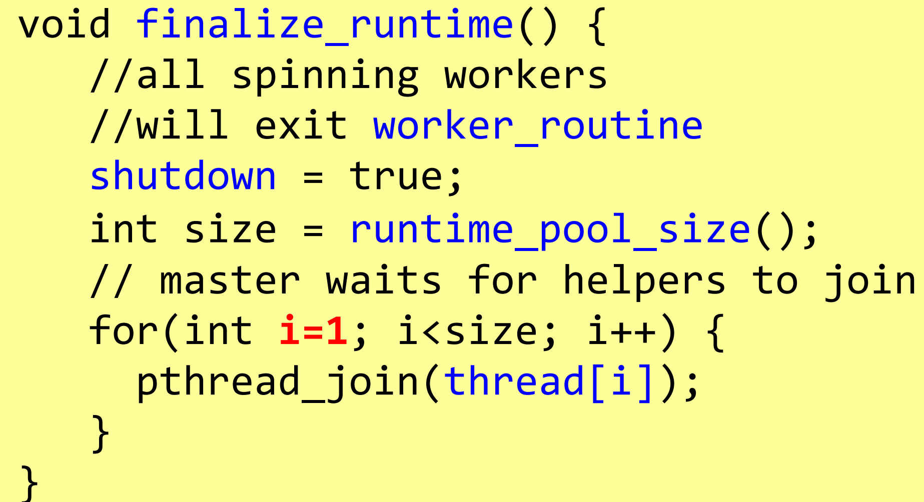
```
void end_finish() {
  while(finish_counter != 0) {
    find_and_execute_task();
  }
}
```

```
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();
   }
}
```

Note: there are better ways to decrement finish counter rather than doing it inside locks

Mapping the Linguistic Interface to Library Based Parallel Runtime

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```



How to Store Tasks in Runtime ?

- **push**_task_to_runtime()
- **pop**_task_from_runtime()

Data-structures for storing tasks in a thread pool based runtime plays a very important role in determining the scalability and performance of the runtime

Reading Materials

- https://doi.org/10.1007/s11227-018-2238-4
- https://gee.cs.oswego.edu/dl/papers/fj.pdf

Next Lecture

- Work-sharing and work-stealing runtimes