Lecture 05: Performance in Parallel Programming

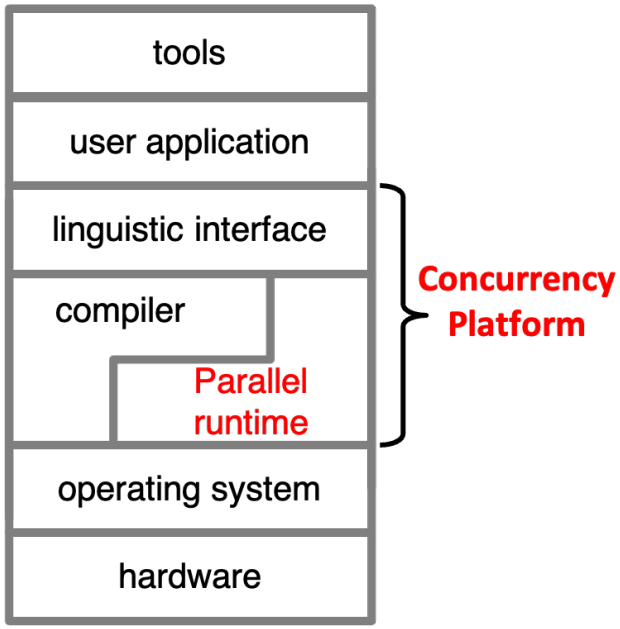
Vivek Kumar

Computer Science and Engineering

IIIT Delhi

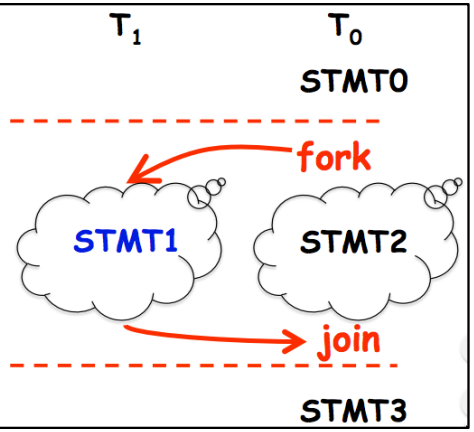
vivekk@iiitd.ac.in

Last Lecture




```
finish {
        async { Wash your clothes in washing machine
        async { Complete your PRMP project deadline
}
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!
```

```
// 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
```



Today's Lecture

- Parallel runtime system for task-scheduling
  - Work-stealing

Tasks-based parallel programming model and its underlying runtime system would be referred throughout in this course

Mapping the Linguistic Interface to the Parallel 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);
  }
}
```

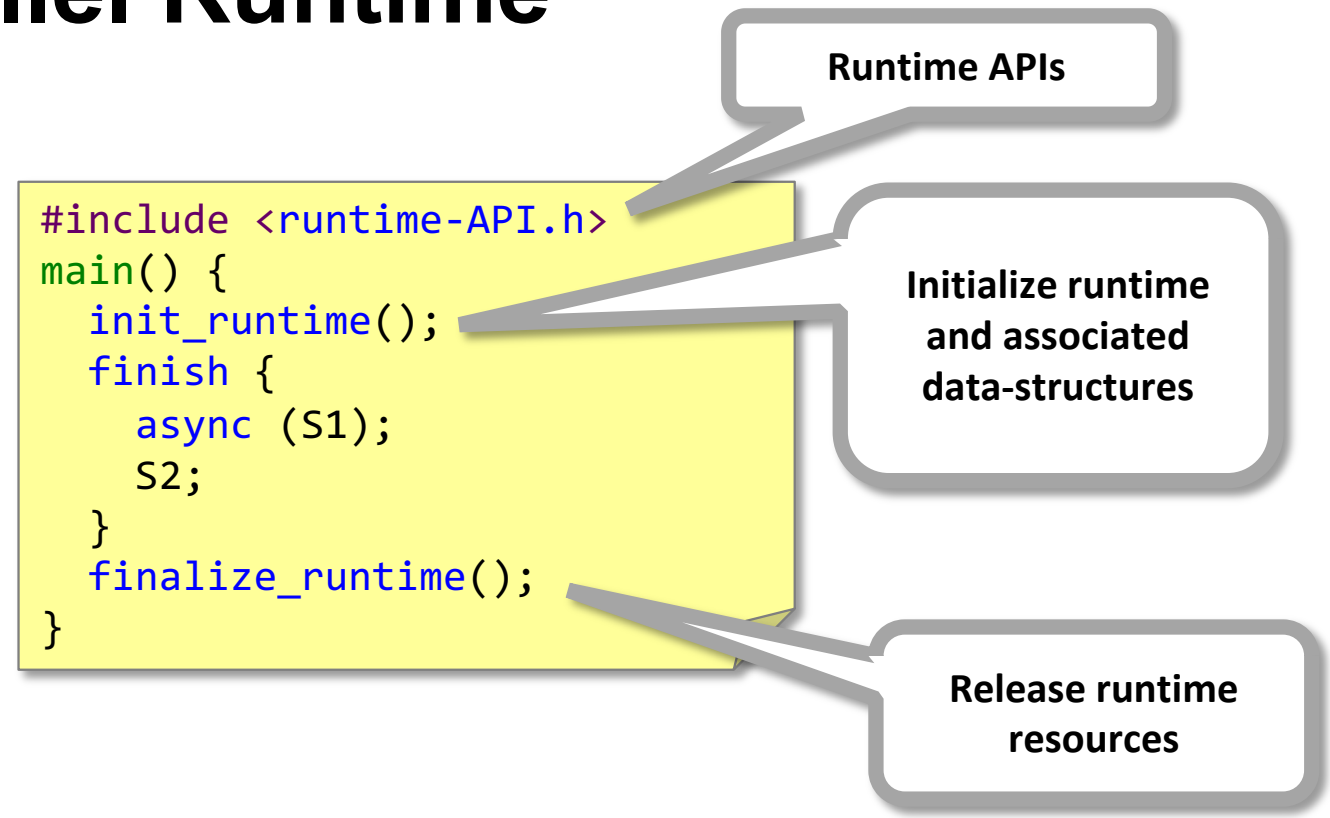
- High **productivity** due to **serial elision**
  - Removing all async and finish constructs results in a valid sequential program
  - Several existing frameworks support this programming model, although the name of the APIs for tasking would be different
- Uses an underlying high **performance parallel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

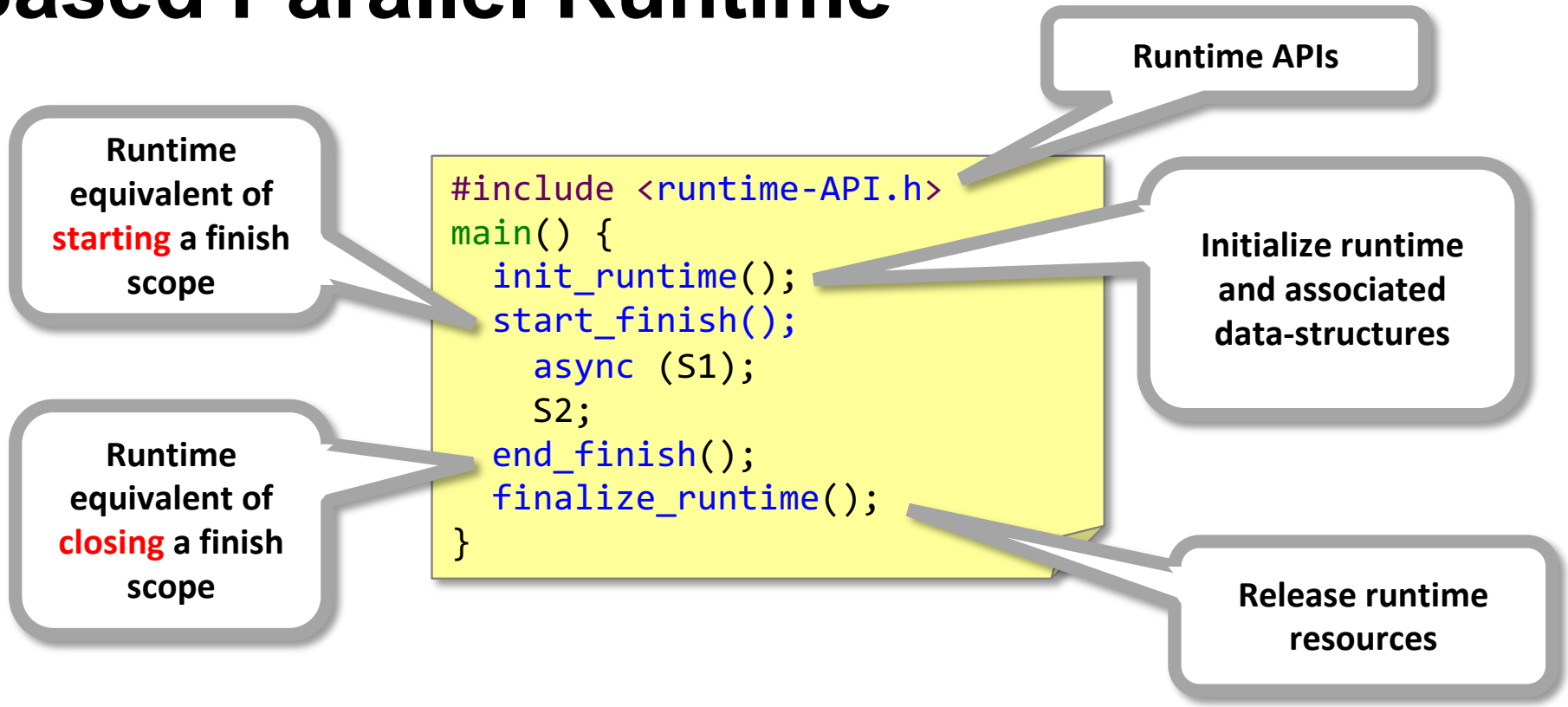
Popular options for simple tasks based parallel programming model

[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();
}
```

```
void finalize_runtime() {
   //all spinning workers
   //will exit worker_routine
   shutdown = true;
   int size = runtime_pool_size();
   // master waits for helpers to join
   for(int i=1; i<size; i++) {
     pthread_join(thread[i]);
   }
}
```

How to Store Tasks in Runtime ?

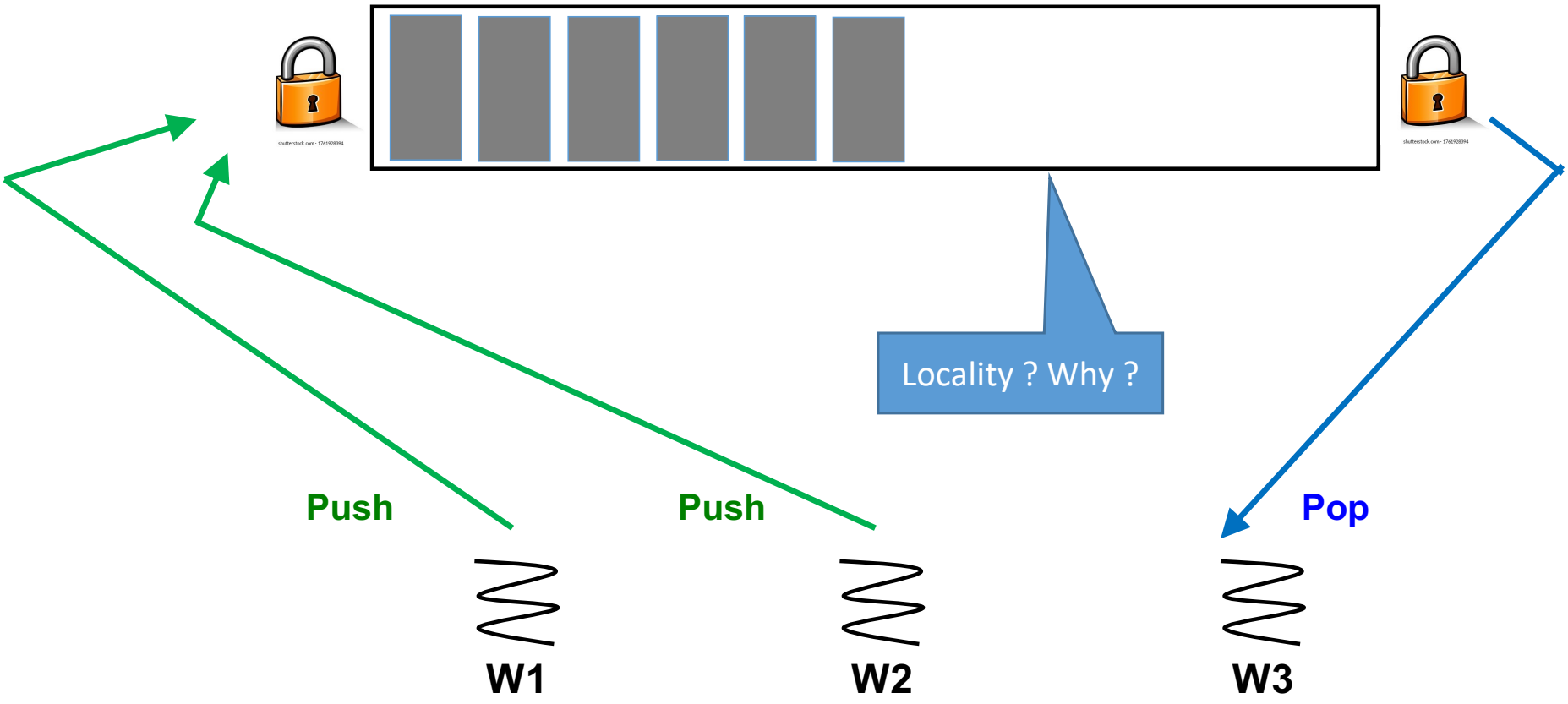
- **push**_task_to_runtime()
- **pop**_task_from_runtime()

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

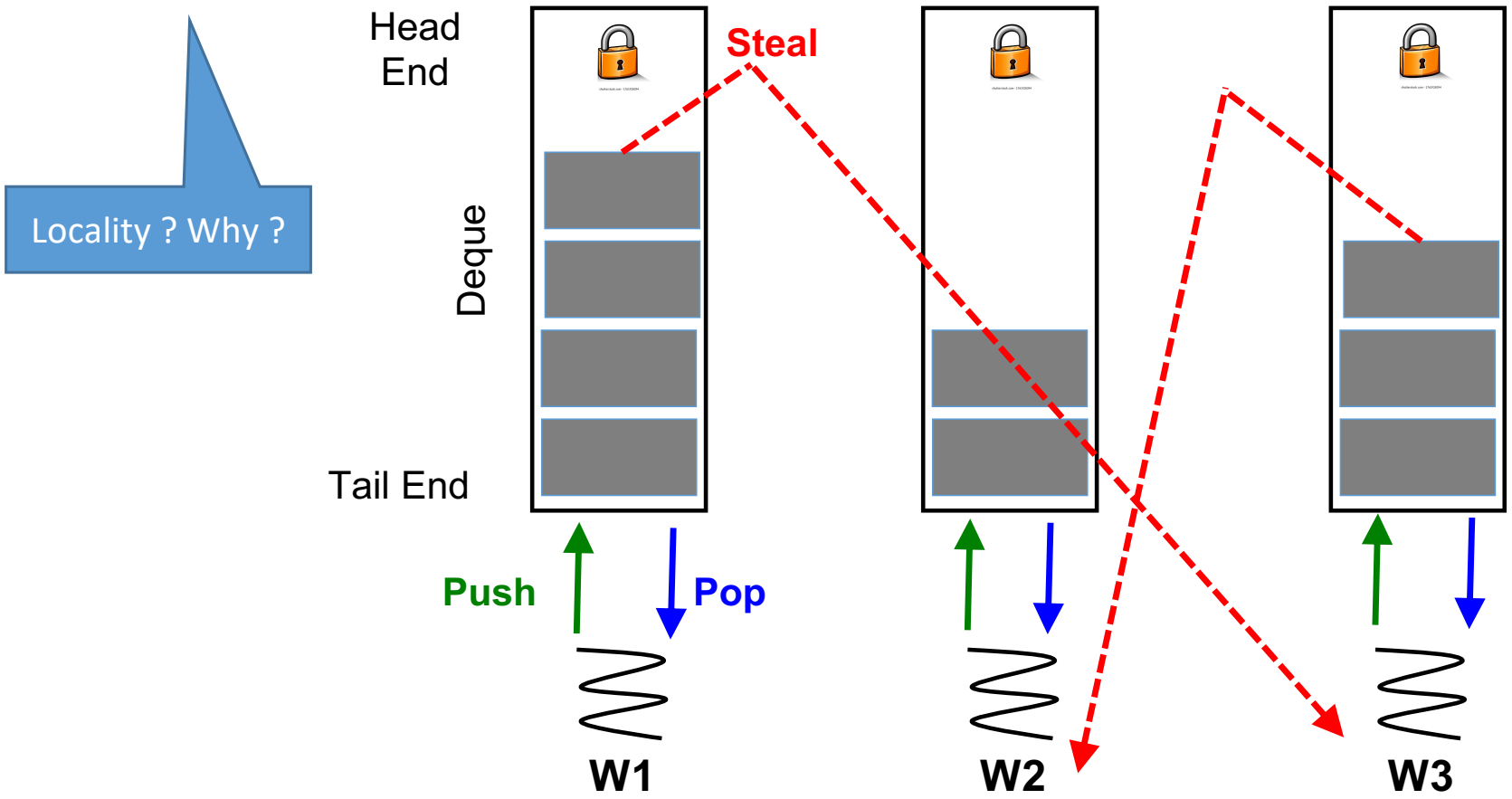
Parallel Runtime for Task Scheduling

- There are several different implementations of parallel runtimes, but at the core almost all of them uses either a **work-sharing** a **work-stealing** runtime underneath
- Tasks based parallel runtime systems primarily use work-stealing runtime only

Work-Sharing Runtime System



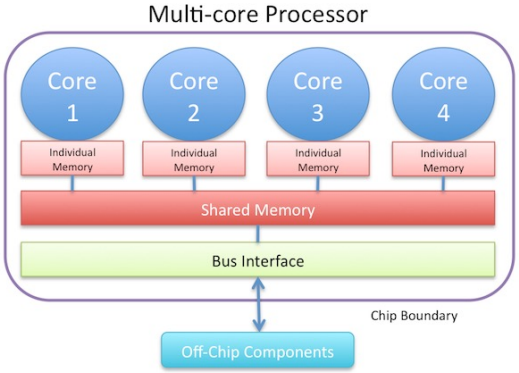
Work-Stealing Runtime System

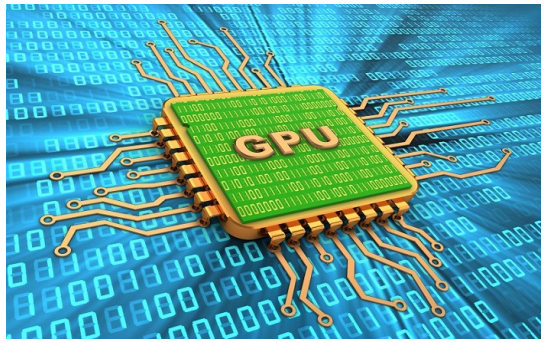


Work-Sharing v/s Work-Stealing

- 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 (only feasible by using atomic operations)
    - Idle worker steals the tasks from busy workers
  - Distributed task pools
  - Better scalability

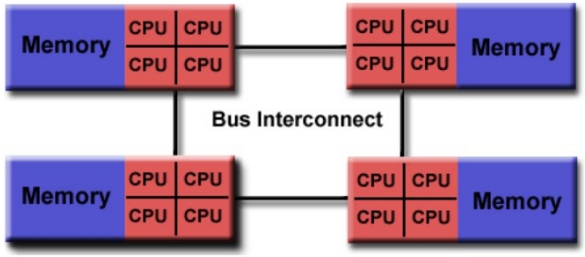
Supported on Wide Range of Architectures

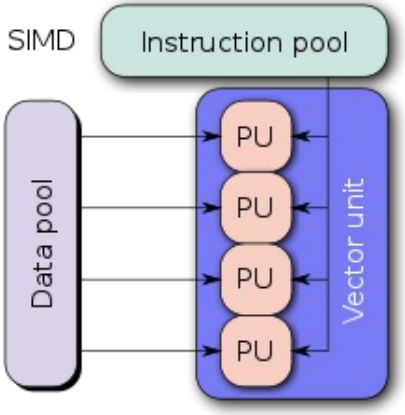




Multiprocessor System-on-Chip

Shared Memory (NUMA)





Supercomputers

Supported/Used by Several Companies/Projects

facebook/**folly**

An open-source C++ library developed and used at Facebook.

679 Contributors · 328 Issues · 23k Stars · 5k Forks

PYTORCH

Product Team Enterprise Explore Marketplace Pricing

pytorch / pytorch Public

Code Issues 5k+ Pull requests 838 Actions Projects 25 Wiki Security

master pytorch / caffe2 / utils / threadpool / ThreadPool.h

Karol Kosik [XROS][ML] System specific adjustments for UTs to work. (#65245)

11 contributors

67 lines (53 sloc) 1.94 KB

```
#ifndef CAFFE2_UTILS_THREADPOOL_H_
#define CAFFE2_UTILS_THREADPOOL_H_

#include "ThreadPoolCommon.h"

#include <atomic>
#include <functional>
#include <memory>
#include <mutex>
#include <vector>

#include "caffe2/core/common.h"

//
// A work-stealing threadpool loosely based off of pthreadpool
//

namespace caffe2 {
```

Caffe

Futures

A Non-actor re-implementation of Scala Futures.

```
import com.twitter.conversions.DurationOps._
import com.twitter.util.{Await, Future, Promise}

val f = new Promise[Int]
val g = f.map { result => result + 1 }
f.setValue(1)
Await.result(g, 1.second) // => this blocks for the futures result (and eventually returns 2)

// Another option:
g.onSuccess { result =>
  println(result) // => prints "2"
}

// Using for expressions:
val xFuture = Future(1)
val yFuture = Future(2)

for {
  x <- xFuture
  y <- yFuture
} {
  println(x + y) // => prints "3"
}
```

Twitter

Future interrupts

Method `raise` on `Future` (`def raise(cause: Throwable)`) raises the interrupt described by `cause` to the producer of this `Future`. Interrupt handlers are installed on a `Promise` using `setInterruptHandler`, which takes a partial function:

```
val p = new Promise[T]
p.setInterruptHandler {
  case exc: MyException =>
    // deal with interrupt..
}
```

Interrupts differ in semantics from cancellation in important ways: there can only be one interrupt handler per promise, and interrupts are only delivered if the promise is not yet complete.

Object Pool

The pool order is FIFO.

Reading Materials

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

Next Lecture (#06)

- Performance in parallel programming (contd.)
- Project deadline-1 will be announced tonight with a deadline of ten days