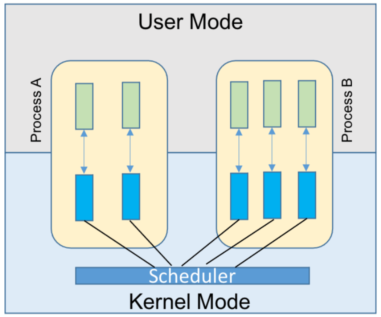
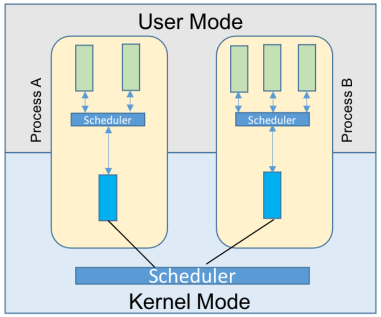
Lecture 05: Boost Fibers and Argobots

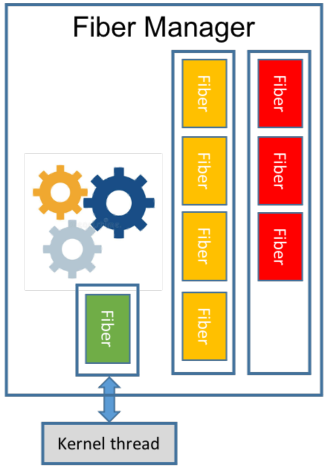
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

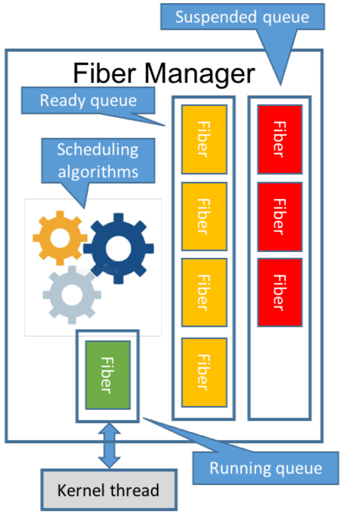
Last Lecture (Recap)






```
#include <boost/context/all.hpp>
ctx::continuation A(ctx::continuation cont) {
  cout<< "IN-A" << endl;
  cont = cont.resume();
  /* Do something */
  cout<< "OUT-A" << endl;
  return std::move(cont);
}
/* Methods B & C rewritten as A above */
int main() {
  ctx::continuation a = ctx::callcc(A);
  ctx::continuation b = ctx::callcc(B);
  ctx::continuation c = ctx::callcc(C);
  a.resume();
  b.resume();
  c.resume();
}
```





- ULT v/s KLT
- Boost context
  - Used for capturing execution state in current thread (stack, registers, etc.), and then jump to a different execution state on current thread (cooperative multitasking)
- Boost fibers
  - Emulates std::thread operations, but as a ULT instead of a KLT
    - Based on boost context

```
#include <boost/fiber/all.hpp>
#define millisleep(x) boost::this_fiber::sleep_for(std::chrono::milliseconds(a))
.....
boost::fibers::fiber f1 ([=]() { millisleep(500); }); // Fiber F1 launched
boost::fibers::fiber f2 ([=]() { millisleep(100); }); // Fiber F2 launched
f1.join(); // Wait for termination of F1
f2.join(); // Wait for termination of F2
```

Today’s Class

- Boost C++ libraries for concurrency
  - Fibers (Contd.)
  - Introduction to Coroutines
- Argobots library

Fiber Futures

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    std::future<uint64_t> f1 (std::async([=](){ return fib(n-1); }));
    std::future<uint64_t> f2 (std::async([=](){ return fib(n-2); }));
    //get will block until result is ready
    return f1.get() + f2.get();
  }
}
```

```
uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    boost::fibers::future<uint64_t> f1 (boost::fibers::async([=](){ return fib(n-1); }));
    boost::fibers::future<uint64_t> f2 (boost::fibers::async([=](){ return fib(n-2); }));
    //get will block until result is ready
    return f1.get() + f2.get();
  }
}
```

- Which of these programs would be faster?
- Which of these programs is a parallel program?

Yielding Fibers

```
boost::fibers::fiber f1([=]() {
  cout << “A ”;
  boost::this_fiber::yield();
  cout << “B ”;
  boost::this_fiber::yield();
  cout << “C ”;
});

boost::fibers::fiber f2([=]() {
  cout << “D ”;
  boost::this_fiber::yield();
  cout << “E ”;
  boost::this_fiber::yield();
  cout << “F ”;
});

f1.join();
f2.join();
```

- yield() saves the context of currently running fiber, and places it inside the **ready** queue
  - Manager can schedule it again based on the scheduling algorithm
- What will be the output of this program?

Producer-Consumer using Fibers

```
boost::fibers::mutex mtx;
boost::fibers::condition_variable cnd;
std::string str;

boost::fibers::fiber f1([=]() {
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  if(str.size() == 0) {
    cnd.wait(lck);
  }
  cout << str << endl;
});

boost::fibers::fiber f2([=]() {
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  str = "Hello Fiber";
  cnd.notify_one();
});

f1.join();
f2.join();
```

- Fiber F1 moving into suspended queue, and then back into ready queue after a notify from F2
  - Single thread execution!

Fiber Pitfalls

```
std::mutex mtx;
std::condition_variable cnd;
std::string str;

boost::fibers::fiber f1([=]() {
  std::unique_lock<std::mutex> lck(mtx);
  if(str.size() == 0) {
    cnd.wait(lck);
  }
  cout << str << endl;
});

boost::fibers::fiber f2([=]() {
  std::unique_lock<std::mutex> lck(mtx);
  str = "Hello Fiber";
  cnd.notify_one();
});

f1.join();
f2.join();
```

- Can you spot the difference?
  - What effect it would cause, and why?

Work-Stealing Scheduling of Fibers (1/3)

```
// mutex and condition variable for shutdown
boost::fibers::mutex mtx;
boost::fibers::condition_variable cnd;
int pool_size;

int main() {
  // Step-1: Launch pool_size-1 number of workers calling "worker_routine"
  // Step-2: Set scheduling policy at fiber manager
  boost::fibers::use_scheduling_algorithm
        <boost::fibers::algo::work_stealing>(pool_size);
  // Step-3: Handshake with each of the spawned workers
  // Step-4: Launch compute kernel having fiber based concurrency
  int result = compute_kernel(/*some input*/);
  // Step-5: Shutdown the thread pool
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  cnd.notify_all();
}
```

- Main thread
  - Manages the creation and termination of thread pool
  - Create the top-level fiber with user computation

Work-Stealing Scheduling of Fibers (2/3)

```
// mutex and condition variable for shutdown
boost::fibers::mutex mtx;
boost::fibers::condition_variable cnd;
int pool_size;
void worker_routine(int id) {
  // Step-1: Set scheduling policy at fiber manager
  boost::fibers::use_scheduling_algorithm
        <boost::fibers::algo::work_stealing>(pool_size);
  // Step-2: Handshake with the master worker
  // Step-3: Suspend current fiber until master signals
  std::unique_lock<boost::fibers::mutex> lck(mtx);
  cnd.wait(lck);
}
```

- Worker routine
  - Each worker will have a pool of fibers instead of tasks
  - Fiber manager at each worker would use Boost’s inbuilt work-stealing algorithm for dynamic load balancing of fibers across workers

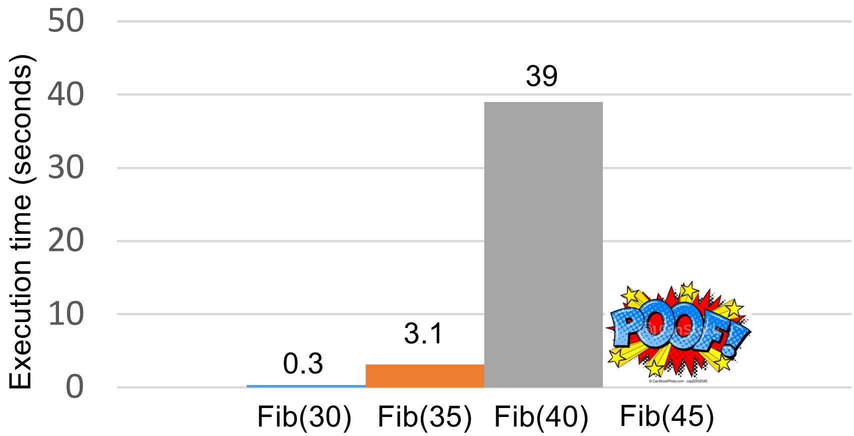
Work-Stealing Scheduling of Fibers (3/3)

```
int compute_kernel(int arg) {
  // Step-1: Wrap callable target to asynchronously compute the return value
  boost::fibers::packaged_task<int()> task([=]() {
    // Launch computation that may also recursively spawn more fibers
    return value;
  });
  // Step-2: Get the future object associated with the above target
  boost::fibers::future<int> future = task.get_future();
  // Step-3: Spawn the fiber and detach it to enable work-stealing
  boost::fibers::fiber(std::move(task)).detach();
  // Step-4: Wait for the fiber to complete
  return future.get();
}
```

- Computation kernel
  - It can recursively create more fibers that would participate in thread pool based work-stealing
  - Fibers intended to participate in work-stealing must be detached

Details on fibers packaged_task and future: https://www.boost.org/doc/libs/1_80_0/libs/fiber/doc/html/fiber/synchronization/futures.html

Fiber Overheads / Limitations



- Language restriction
  - Fiber library requires C++11
  - Cannot be used in C-based HPC libraries/programs
- (Serious) Runtime overheads
  - Graph shown for calculating recursive Fibonacci that spawns detached fiber for every recursive call until threshold reached (n<10)
    - Total nested fibers created
      - Fib [30, 57K], Fib [35, 635K], Fib [40, 7049K]
  - Single worker used!
  - Platform details
    - AMD EPYC 7551 32-core processor
    - Ubuntu 18.04.3 LTS
    - GCC version 7.5.0
      - -O3 flag used
    - Boost version 1_80_0

Coroutine v/s Fiber: Brief Overview

- Coroutines are like functions which allow suspending and resuming execution at certain locations
  - Preserves the local state of execution and allows re-entering the function more than once
  - Control is passed to the caller once a coroutine yields
- Coroutines do not resemble threads
  - Cannot synchronize across coroutines
  - Coroutine library provides no schedule
- Coroutine cannot outlive its invoker
  - Calling code instantiates a coroutine, passes control back and forth with it for some time, and then destroys it
    - Invoker can call it in any order

More info: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4024.pdf

Today's Class

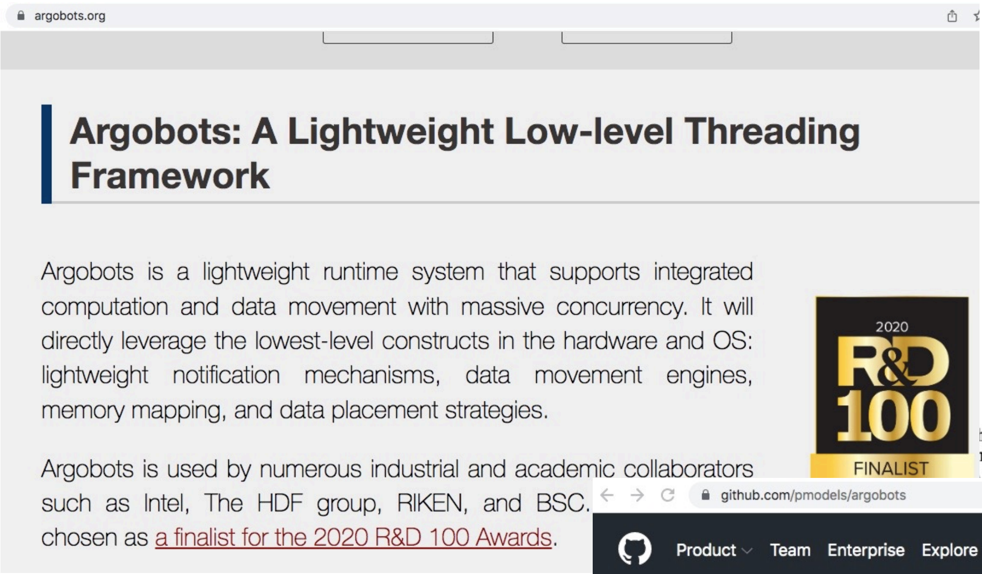
- Boost C++ libraries for concurrency
  - Fibers (Contd.)
  - Introduction to Coroutines
- Argobots library

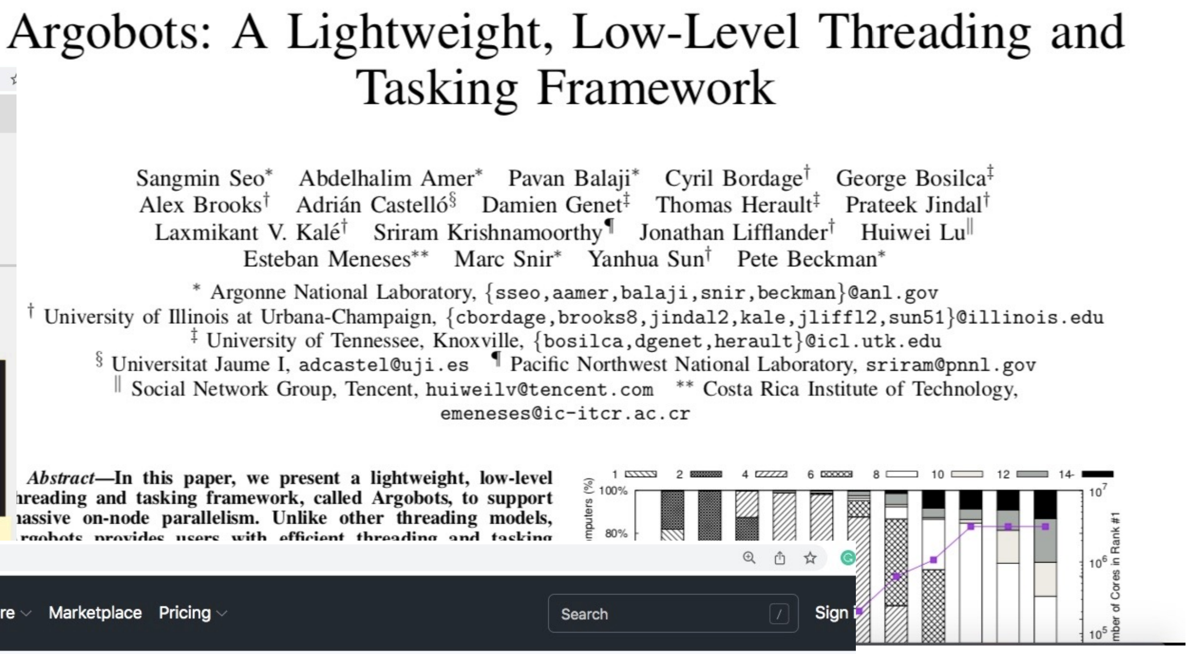
Alternative to Fibers: Options

- Qthreads
- MassiveThreads
- Maestro
- Nanos++
- StackThreads
- …..
- Argobots
  - Latest in the arsenal
  - Open sourced
  - High performance!
  - C-based implementation

To get a broad overview of all these alternatives, you can read the following paper: https://ieeexplore.ieee.org/document/7776544

Argobots



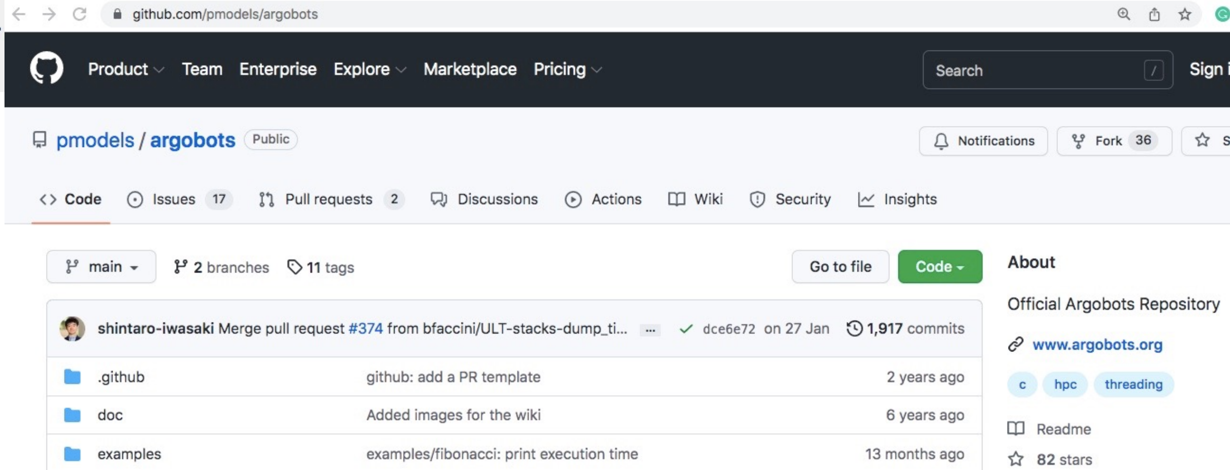


Argobots: A Lightweight, Low-Level Threading and Tasking Framework

Sangmin Seo* Abdelhalim Amer* Pavan Balaji* Cyril Bordage† George Bosilca‡
Alex Brooks† Adrián Castelló§ Damien Genet‡ Thomas Herault‡ Prateek Jindal†
Laxmikant V. Kalé† Sriram Krishnamoorthy¶ Jonathan Lifflander† Huiwei Lu‖
Esteban Meneses** Marc Snir* Yanhua Sun† Pete Beckman*

* Argonne National Laboratory, {sseo,aamer,balaji,snir,beckman}@anl.gov
† University of Illinois at Urbana-Champaign, {cbordage,brooks8,jindal2,kale,jliffl2,sun51}@illinois.edu
‡ University of Tennessee, Knoxville, {bosilca,dgenet,herault}@icl.utk.edu
§ Universitat Jaume I, adcastel@uji.es ¶ Pacific Northwest National Laboratory, sriram@pnnl.gov
‖ Social Network Group, Tencent, huiweilv@tencent.com ** Costa Rica Institute of Technology, emeneses@ic-itcr.ac.cr

Abstract—In this paper, we present a lightweight, low-level hreading and tasking framework, called Argobots, to support nassive on-node parallelism. Unlike other threading models,



Argobots

- Supports M x N threading model similar to boost::fibers
  - C language based implementation built on top of pthreads
  - Designed to be used as underlying threading and tasking library for high-level runtimes of languages
- Uses Boost fcontext for context switches across lightweight user-level threads
- Supports flexible scheduling techniques
  - Although, such a policy could be implemented by creating a customized scheduling policy for boost::fibers [1]

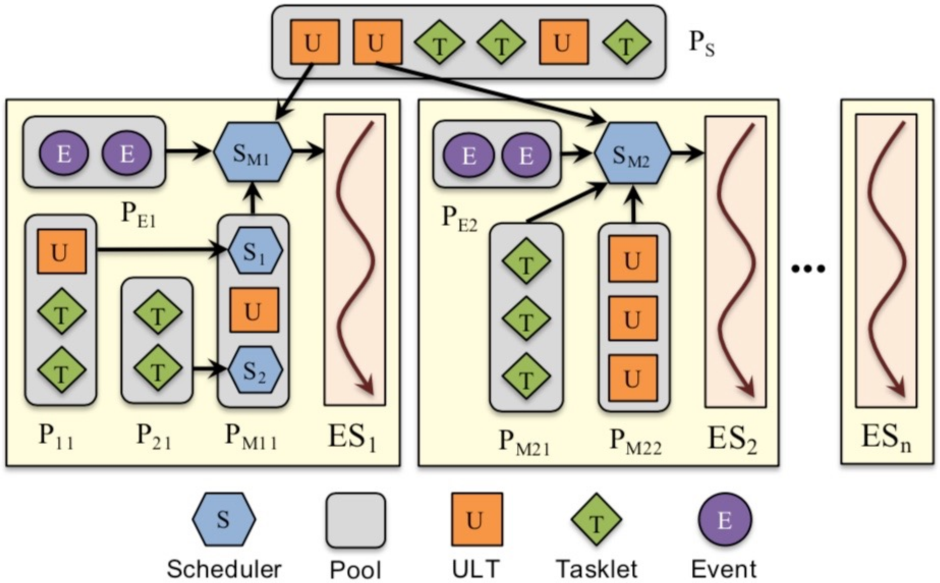
[1] https://www.boost.org/doc/libs/1_80_0/libs/fiber/doc/html/fiber/custom.html

Argobots: Programming Model

```
#define N 100

void example(){
  printf(“Hello\n”);
}

int main(){
  initialization();

  for(int i=0; i<N; i++)
          ULT_creation_to(example, dest);

  for(int i=0; i<N; i++) {
          ULT_join();
          UTL_free();
  }

  finalize()
}
```

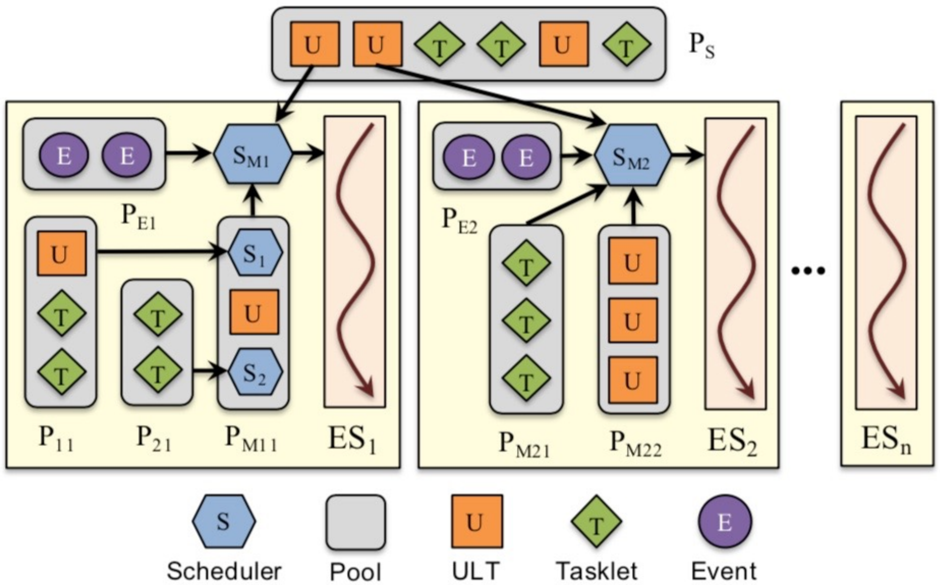
1. Environment Initialization
2. ULT/Tasklet creation
3. ULT/Tasklet join
4. ULT/Tasklet free
5. Environment Finalization

Argobots: Execution Model



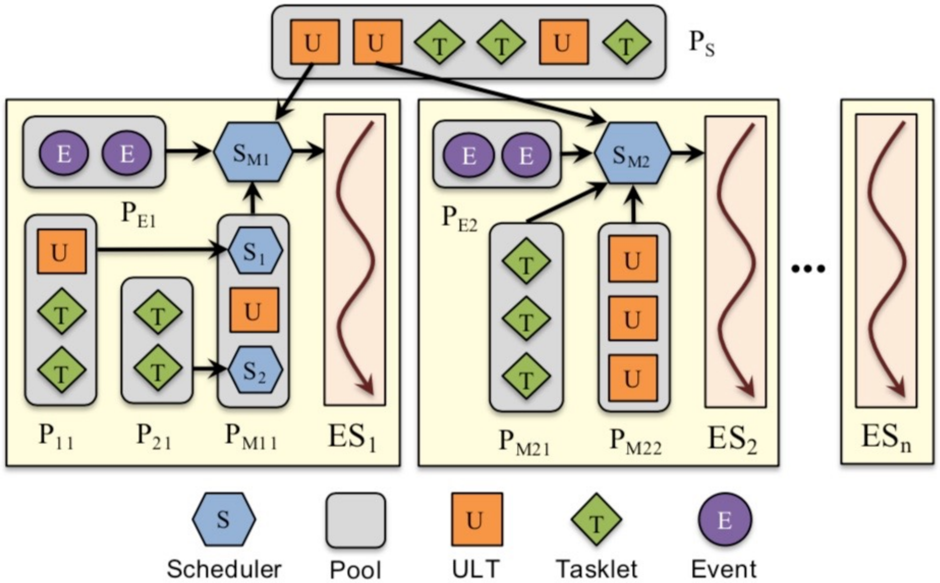
- Two levels of parallelism
  - Execution Stream (ES)
    - Each mapped to a single pthread
  - Work units
    - User Level Threads (ULTs)
    - Tasklets
- ULTs
  - Similar to boost fibers
  - Have their own stack
  - Can cooperatively yield to ES or another ULT
- Tasklets
  - Similar to HClib's async or OpenMP task
  - Borrows the stack of host ES
  - Cannot explicitly yield but run to completion before returning control to the host ES
    - No concurrent execution of work units in a single ES
    - Migratable unless stored inside a private pool

Argobots: Scheduler



- Each ES can have its own scheduler (S_M in the figure)
- A scheduler is associated with one or more pools of “ready” work units (P_{M11} in S_{M1} and P_{M21}, P_{M22} in S_{M2})
- Pools can be private or shared with other ES (P_s is shared pool)
- Event pool (E) stores lightweight notification events (e.g., message from remote node)
- Supports stackable scheduling framework with pluggable strategies
  - E.g., allowing switching between a scheduler with low priority work units and another scheduler with high priority work units
  - S_1 and S_2 in P_{M11} are stacked schedulers, which will be executed by the main scheduler S_{M1}

Argobots: Primitive Operations for Work Units

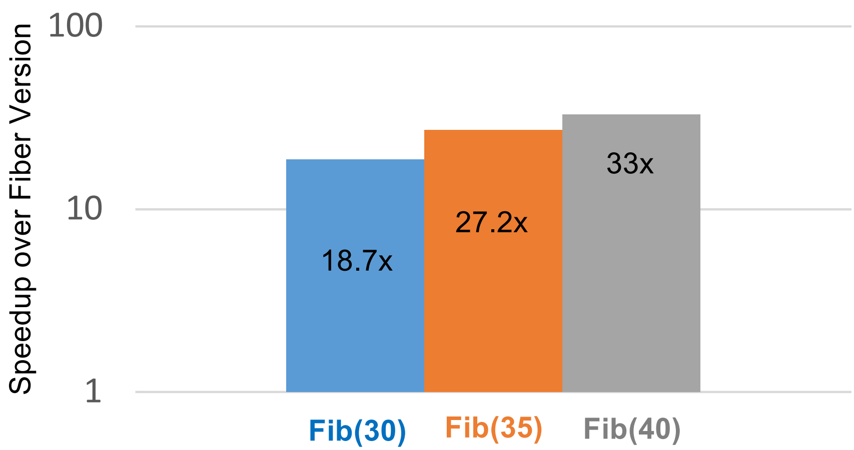


- Creation, join, and migration
  - Supported for both ULT and tasklets
  - Tasklets can migrate only if they haven't started execution
- Yield (only for ULTs)
  - When a ULT yields control, the control goes to the scheduler that was in charge of scheduling in the ES at the point of yield time
  - ULTs must cooperatively yield control in order to enable progress of other work units
- Yield_to
  - When a ULT calls yield to, it yields control to a specific ULT instead of the scheduler
    - Eliminates the overhead of context switching to the scheduler and scheduling another ULT
- Synchronization (only for UTLs)
  - Mutex, condition variable, future, and barrier are supported (also supported by boost fibers)
  - ULT calling a blocking Argobots operation is context switched

Programming with Argobots

- Demo
  - https://github.com/pmodels/argobots/blob/main/examples/hello_world/hello_world.c
    https://github.com/pmodels/argobots/blob/main/examples/fibonacci/fibonacci.c

Argobots v/s Fibers



- Graph shown for calculating recursive Fibonacci that spawns task for every recursive call until threshold reached (n<10)
  - In the Boost version, every task is a detached fiber
  - In the Argobots version, every task is an ULT
  - Both version uses work-stealing scheduler (although its of no effect as single worker)
  - Fib(45) execution
    - Crashes with Fibers
    - Took 13.1 seconds with Argobots
- Single worker used in each case
- Platform details
  - AMD EPYC 7551 32-core processor
  - Ubuntu 18.04.3 LTS
  - GCC version 7.5.0
    - -O3 flag used
  - Boost version 1_80_0
  - Argobots commit id dce6e72

Reading Materials

- Argobots
  - http://pavanbalaji.github.io/pubs/2018/tpds/tpds18.argobots.pdf
  - https://www.argobots.org/
- Fibers
  - https://www.boost.org/doc/libs/1_80_0/libs/fiber/doc/html/index.html

Next Lecture (L #06)

- Managing Overheads from Blocking Tasks & Deques