A General-purpose Parallel and Heterogeneous Task Programming System at Scale

Dr. Tsung-Wei Huang

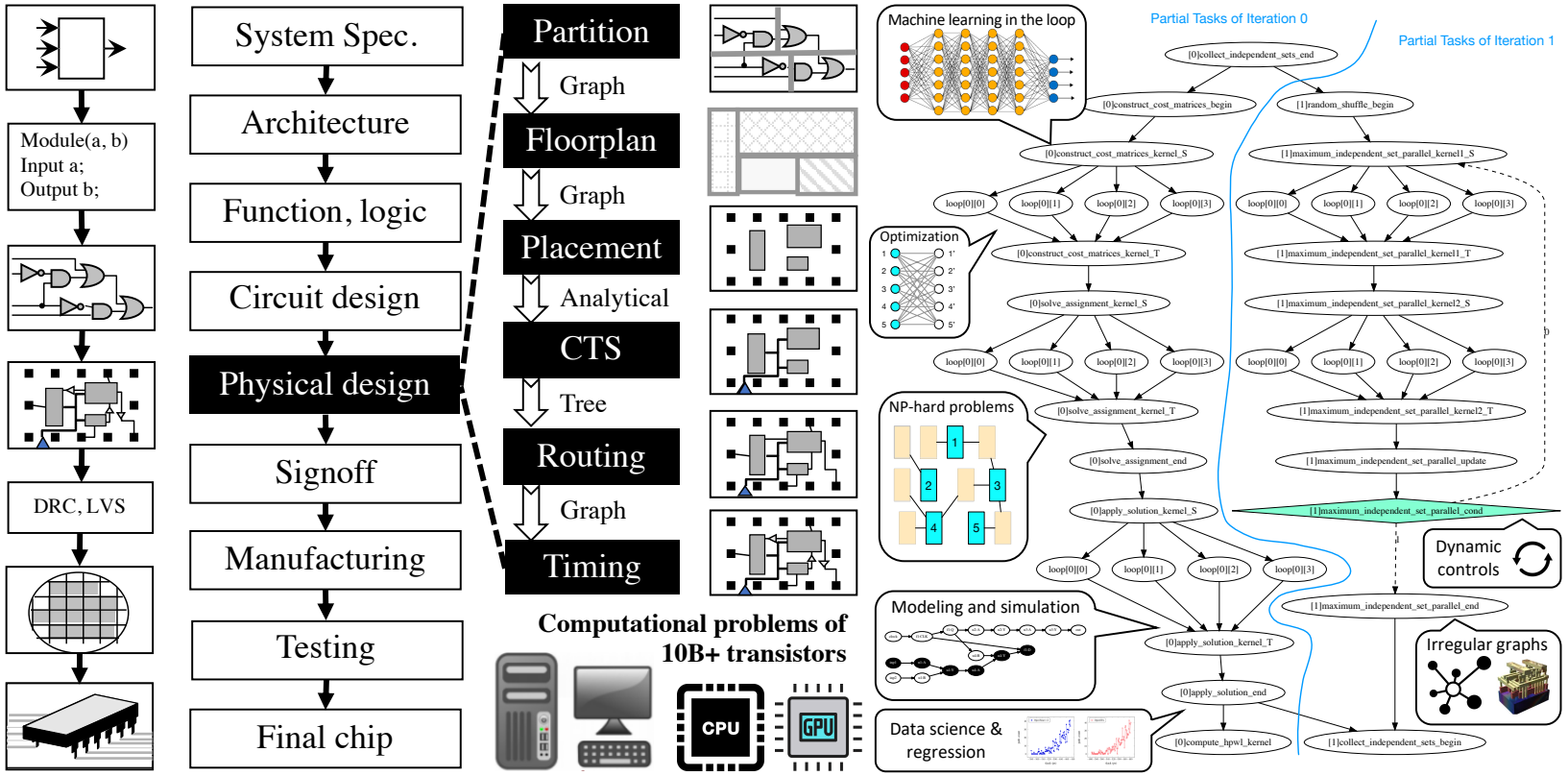
Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

How can we make it easier for scientific software developers to program large parallel and heterogeneous resources with high performance scalability and simultaneous high productivity?

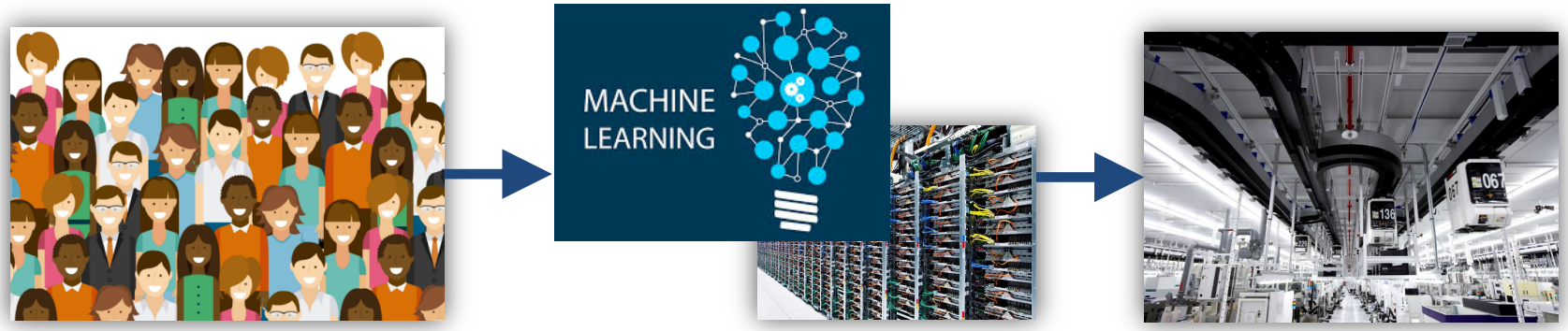
Parallelizing VLSI CAD Software

- **This is a seriously complicated process …**
  - Billions of tasks with dynamic control flows, cycles, irregularity, diverse computational patterns



IC Industry Seeks to Reduce Time and Effort

- **DARPA IDEA/POSH program (under ERI) 2018-2022**
  - *No human in the loop* 24-hour layout generator

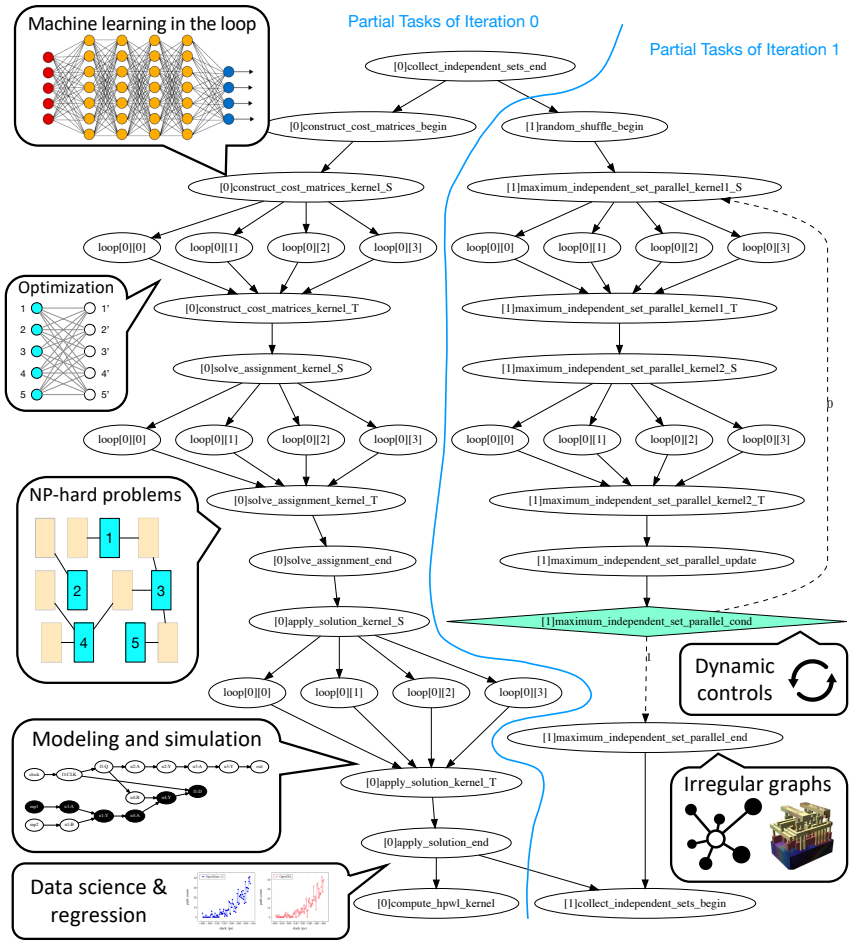


~~Expensive humans~~ **Machine Learning (ML) + Parallelism** No-touch foundry

- **Central theme: ML + Parallel Computing**
  1. ML must pervade CAD tools, both inside and outside
     - Remove expensive human decision making wherever possible
  2. CAD tools must evolve to new parallel targets
     - Free up time for design space exploration and ML optimization

This is Extremely Challenging for R&D

- How can we program a task graph like this?



Broad mix of algorithms (graph, analytics, simulation)

Domain-specific heuristics (branch bound, approximation)

Enormous parameter tuning (million lines of scripts)

Vast task/data dependencies (terabytes of design files)

Machine learning in the loop (numerous flow trajectories)

Today's CAD Software Landscape

- **Companies hire "heroic programmers"**
  - Handcraft everything to decide performance in detail
    - Solutions are heavily hard-coded
    - Augment existing codebase for incremental parallelism
    - Pthread, OpenMP, Intel TBB, Socket, Boost.Asio, MPI, CUDA
  - Explicitly manage scheduling and task distributions
    - Batch flow script to decide process mapping & partition
- **Why not use existing programming frameworks?**
  - DOE has enabled vast success of HPC software
    - Kokkos, SHAD, RAJA, exascale computing projects
  - Universities have released many open-source tools
    - Charm++, Legion, Spark, HPX, StarPU, PaRSEC

Three Big Limitations of Existing Tools

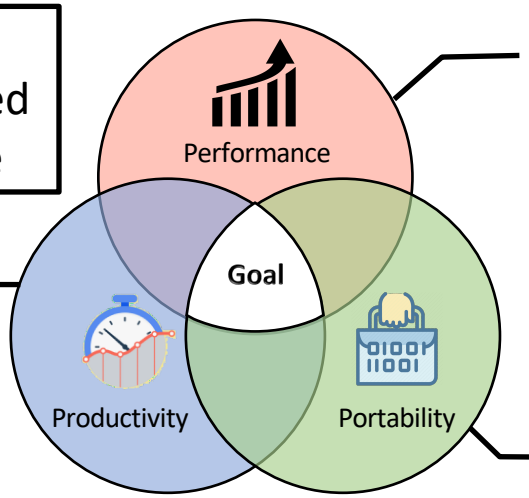
- **Lack of end-to-end parallelism**
  - Cause: ML enables complex workflows
  - Result: Composability is barely addressed in libraries
  - *Evidence: Simple ML-CAD pipeline ran 5—6x faster*
- **Lack of dynamic control flows and irregularity**
  - Cause: Task parallelism relies on DAG (acyclic graph)
  - Result: Non-deterministic workaround
  - *Evidence: Condition tasks saved 10 GB in VLSI placement*
- **Lack of automatic transition**
  - Cause: Programmers need significant rewrite of code
  - Result: Slow adoption by scientific software developers

**IPDPS19: T.-W. Huang, et al, "Cpp-Taskflow: Fast Task-Based Programming using Modern C++"*
**TCAD20: T.-W. Huang. et al, "DtCraft: High-performance Distributed Execution Engine at Scale"*

We Need a New Programming System

❑ Our project mantra:

We maximize productivity compared to handcrafted time



We maximize the performance compared to handcrafted solution

We maximize the portability to cover multiple hardware platforms

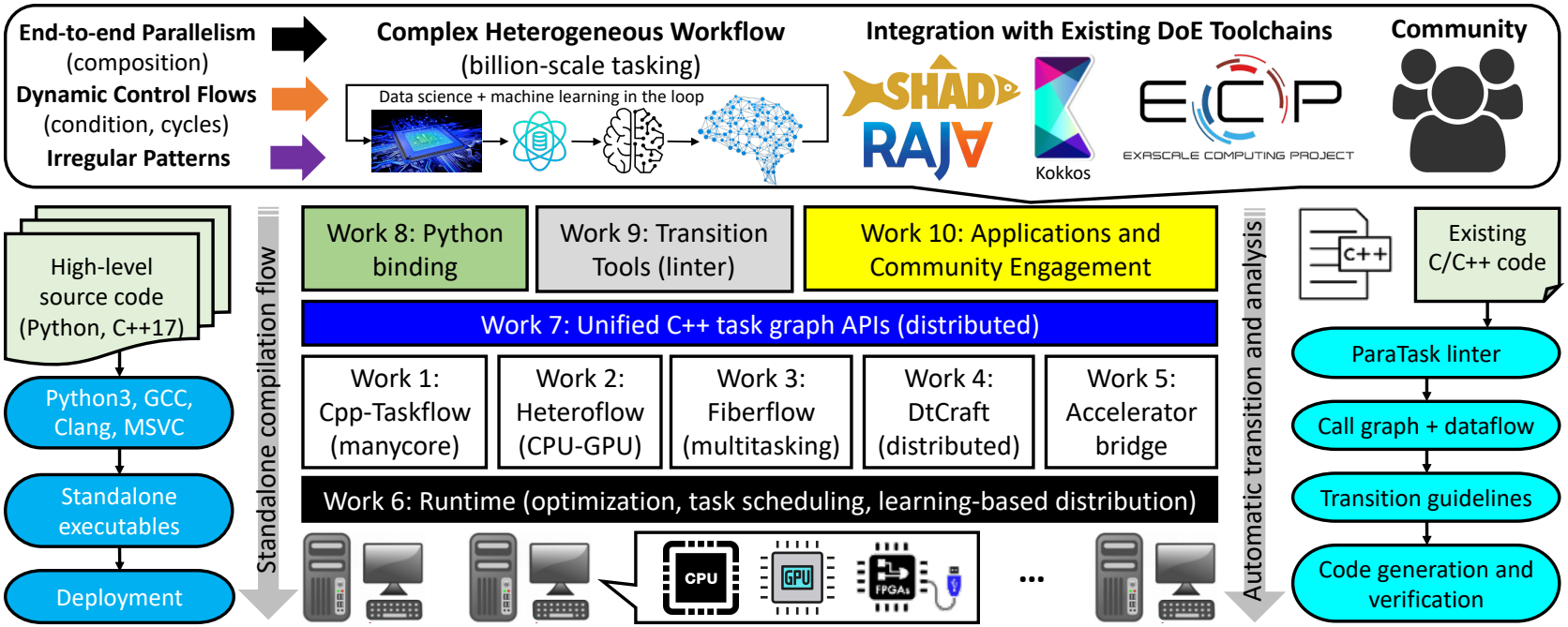
❑ We are not to replace existing tools but

1. Address their limitations on the task parallelism front
2. Develop compatible interface to reuse their facilities

Together, we can deliver complementary advantages to lay a foundation on which to innovate new scientific software and methodologies!

A General-purpose Task Programming System

- **Streamline parallel and heterogeneous programming**
  - Scalable to large parallel systems (CPUs, GPUs, FPGAs)



- **This has been an on-going project since my PhD**
  - Also a proposal to the DOE Early CAREER program

Selected Modules for the Rest of Talk

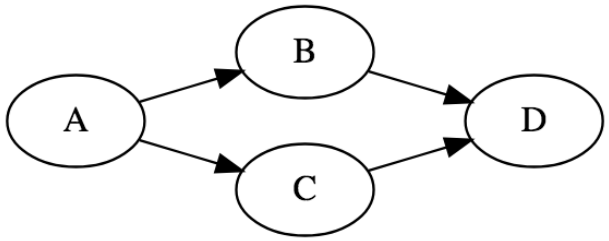
- **Vertical scalability**
  - Cpp-Taskflow: Parallel Task Programming in Modern C++
  - Result on ML-centric VLSI placement (>8M tasks)
  - Result on VLSI timing analysis (>1B tasks)
- **Horizontal scalability**
  - DtCraft: Distributed Programming and Execution Engine
  - Result on complex heterogeneous ML workflows
- **Technical details**
  - HeteroSteal: A Generalized Work-stealing Scheduler
  - Learning-based distributed scheduling
  - Result on improved system performance


WARNING
Code Ahead

“Hello World” in Cpp-Taskflow [IPDPS19]

```
#include <taskflow/taskflow.hpp>   // Cpp-Taskflow is header-only
int main(){
   tf::Taskflow taskflow;
   tf::Executor executor;
   auto [A, B, C, D] = taskflow.emplace(
      [] () { std::cout << "TaskA\n"; }
      [] () { std::cout << "TaskB\n"; },
      [] () { std::cout << "TaskC\n"; },
      [] () { std::cout << "TaskD\n"; }
   );
   A.precede(B);            // A runs before B
   A.precede(C);            // A runs before C
   B.precede(D);            // B runs before D
   C.precede(D);            // C runs before D
   executor.run(taskflow);  // create an executor to run the taskflow
   return 0;
}
```

*Only **17 lines** of code to get a parallel task execution!*



“Hello World” in OpenMP

```
#include <omp.h>  // OpenMP is a lang ext to describe parallelism using compiler directives
int main(){
   #omp parallel num_threads(std::thread::hardware_concurrency())
   {
      int A_B, A_C, B_D, C_D;
      #pragma omp task depend(out: A_B, A_C)
      {
         s t d : : c o u t << ”TaskA\n” ;
      }
      #pragma omp task depend(in: A_B; out: B_D)
      {
         s t d : : c o u t << ” TaskB\n” ;
      }
      #pragma omp task depend(in: A_C; out: C_D)
      {
         s t d : : c o u t << ” TaskC\n” ;
      }
      #pragma omp task depend(in: B_D, C_D)
      {
         s t d : : c o u t << ”TaskD\n” ;
      }
   }
   return 0;
}
```

Task dependency clauses

Task dependency clauses

Task dependency clauses

Task dependency clauses

OpenMP task clauses are static and explicit; Programmers are responsible for a proper order of writing tasks consistent with sequential execution

“Hello World” in Intel’s TBB Library

```
#include <tbb.h>  // Intel’s TBB is a general-purpose parallel programming library in C++
int main(){
    using namespace tbb;
    using namespace tbb:flow;
    int n = task_scheduler init::default_num_threads () ;
    task scheduler_init init(n);
    graph g;
    continue_node<continue_msg> A(g, [] (const continue msg &) {
        s t d : : c o u t << “TaskA” ;
    }) ;
    continue_node<continue_msg> B(g, [] (const continue msg &) {
        s t d : : c o u t << “TaskB” ;
    }) ;
    continue_node<continue_msg> C(g, [] (const continue msg &) {
        s t d : : c o u t << “TaskC” ;
    }) ;
    continue_node<continue_msg> C(g, [] (const continue msg &) {
        s t d : : c o u t << “TaskD” ;
    }) ;
    make_edge(A, B);
    make_edge(A, C);
    make_edge(B, D);
    make_edge(C, D);
    A.try_put(continue_msg());
    g.wait_for_all();
}
```

Use TBB’s FlowGraph for task parallelism

Declare a task as a continue_node

TBB has excellent performance in generic parallel computing. Its drawback is mostly in the ease-of-use standpoint (simplicity, expressivity, and programmability).

“Hello World” in Kokkos

```
struct A {
   template <class TeamMember> KOKKOS_INLINE_FUNCTION
   void operator()(TeamMember& member) {std::cout << "TaskA\n"; }
};
struct B {
   template <class TeamMember> KOKKOS_INLINE_FUNCTION
   void operator()(TeamMember& member) {std::cout << "TaskB\n"; }
};
struct C {
   template <class TeamMember> KOKKOS_INLINE_FUNCTION
   void operator()(TeamMember& member) {std::cout << "TaskC\n"; }
};
struct D {
   template <class TeamMember> KOKKOS_INLINE_FUNCTION
   void operator()(TeamMember& member) {std::cout << "TaskD\n"; }
};
auto scheduler = scheduler_type(/* ... */);
auto futA = Kokkos::host_spawn( Kokkos::TaskSingle(scheduler), A() );
auto futB = Kokkos::host_spawn( Kokkos::TaskSingle(scheduler, futA), B() );
auto futC = Kokkos::host_spawn( Kokkos::TaskSingle(scheduler, futA), C() );
auto futD = Kokkos::host_spawn(
  Kokkos::TaskSingle(scheduler, when_all(futB, futC)), D()
);
```

Fixed-layout task functor (no lambda interface …?)

Define team handle

Task dependency is represented by instances of `Kokkos::BasicFuture`

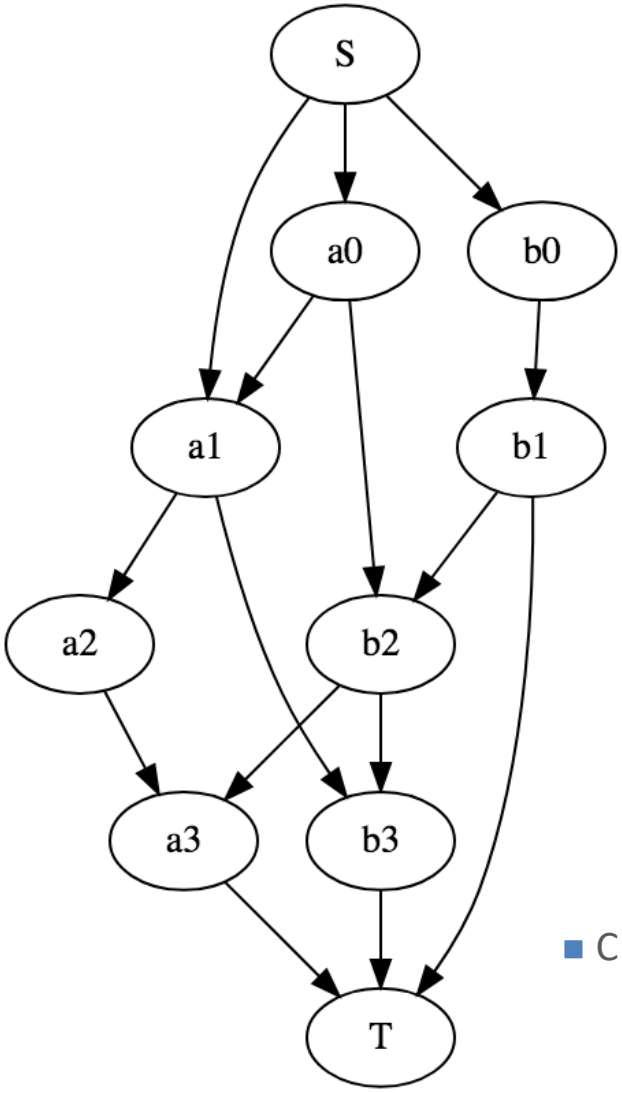
Aggregated dependencies

More scheduling code to follow …

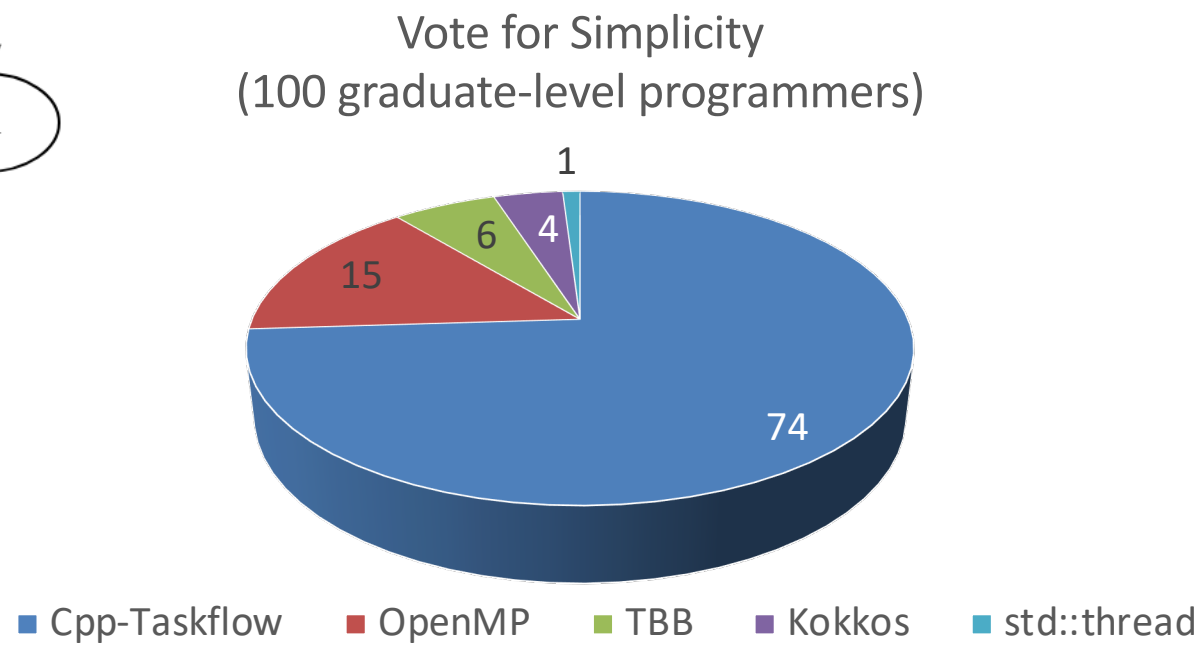
Kokkos is powerful in detailed data controls, but suffers from too many distinct notations with overly complex interface

Kokkos task parallelism: https://github.com/kokkos/kokkos/wiki/Task-Parallelism

Non-biased Opinion



How would you program this slightly more complicated task graph using Cpp-Taskflow, OpenMP, TBB, Kokkos and std::thread?

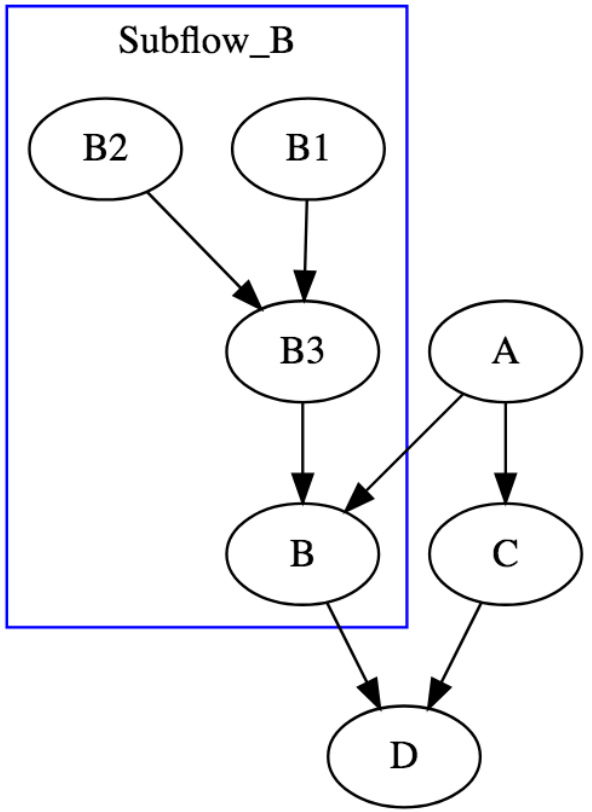


Dynamic Tasking (Subflow) in Cpp-Taskflow

```
// create three regular tasks
tf::Task A = tf.emplace([](){}).name("A");
tf::Task C = tf.emplace([](){}).name("C");
tf::Task D = tf.emplace([](){}).name("D");
```

```
// create a subflow graph (dynamic tasking)
tf::Task B = tf.emplace([] (tf::Subflow& subflow) {
  tf::Task B1 = subflow.emplace([](){}).name("B1");
  tf::Task B2 = subflow.emplace([](){}).name("B2");
  tf::Task B3 = subflow.emplace([](){}).name("B3");
  B1.precede(B3);
  B2.precede(B3);
}).name("B");
```

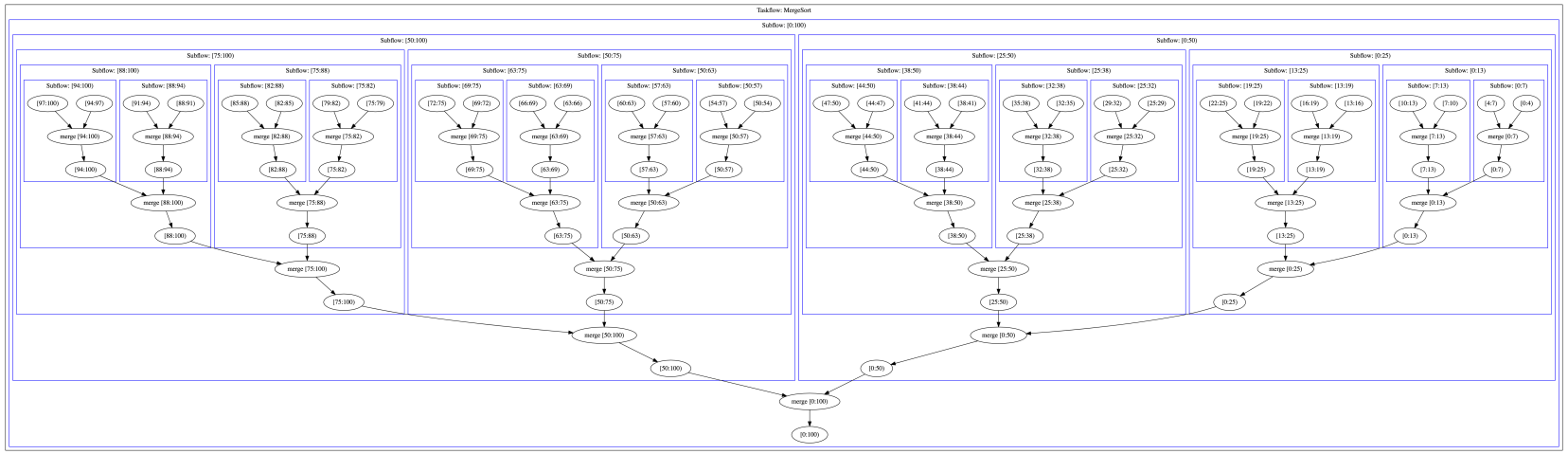
```
A.precede(B); // B runs after A
A.precede(C); // C runs after A
B.precede(D); // D runs after B
C.precede(D); // D runs after C
```



Cpp-Taskflow enables unified API for both static tasking and dynamic tasking using functional programming-styled semantic

Subflow can be Nested

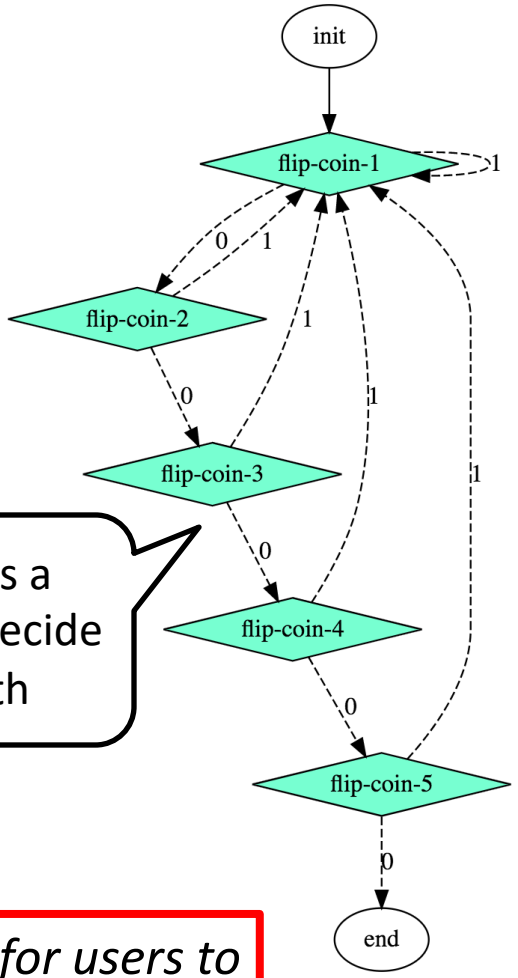
- Task graph for parallel merge sort algorithm



100 items, 7 subflow hierarchies

Conditional Tasking

```
auto A = taskflow.emplace([&](){ } );
auto B = taskflow.emplace([&](){ return rand()%2; } );
auto C = taskflow.emplace([&](){ return rand()%2; } );
auto D = taskflow.emplace([&](){ return rand()%2; } );
auto E = taskflow.emplace([&](){ return rand()%2; } );
auto F = taskflow.emplace([&](){ return rand()%2; } );
auto G = taskflow.emplace([&](){});

A.precede(B).name("init");
B.precede(C, B).name("flip-coin-1");
C.precede(D, B).name("flip-coin-2");
D.precede(E, B).name("flip-coin-3");
E.precede(F, B).name("flip-coin-4");
F.precede(G, B).name("flip-coin-5");
G.name("end");
```



Each task flips a binary coin to decide the next path

Cpp-Taskflow defines condition tasks for users to express dynamic control flows and cyclic flows

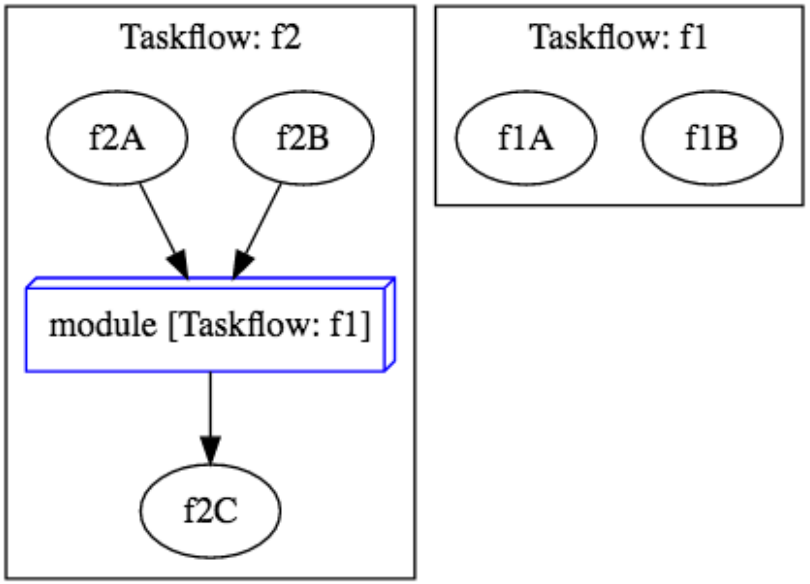
Existing Frameworks on Conditions?

- **Expand simple static loop across iterations**
  - Code size is linearly proportional to decision points
- **Nested loops?**
- **Non-deterministic conditions?**
- **Dynamic control flows and dynamic tasks?**
- **…**

In fact, existing frameworks on conditional tasking or dynamic control flows suffer from exponential growth of code complexity

Composable Tasking

```
tf::Taskflow f1, f2;

auto [f1A, f1B] = f1.emplace(
  []() { std::cout << "Task f1A\n"; },
  []() { std::cout << "Task f1B\n"; }
);
auto [f2A, f2B, f2C] = f2.emplace(
  []() { std::cout << "Task f2A\n"; },
  []() { std::cout << "Task f2B\n"; },
  []() { std::cout << "Task f2C\n"; }
);
```

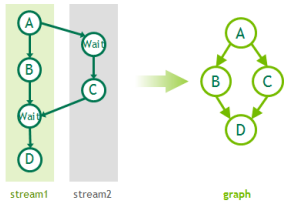



```
auto f1_module_task = f2.composed_of(f1);

f1_module_task.succeed(f2A, f2B)
              .precede(f2C);
```

Runtime sees the entire graph and performs whole-graph optimization for end-to-end parallelism

Concurrent CPU-GPU Tasking

```
const unsigned N = 1<<20;
std::vector<float> hx(N, 1.0f), hy(N, 2.0f);
float *dx{nullptr}, *dy{nullptr};
auto allocate_x = taskflow.emplace([&](){ cudaMalloc(&dx, 4*N);});
auto allocate_y = taskflow.emplace([&](){ cudaMalloc(&dy, 4*N);});
```

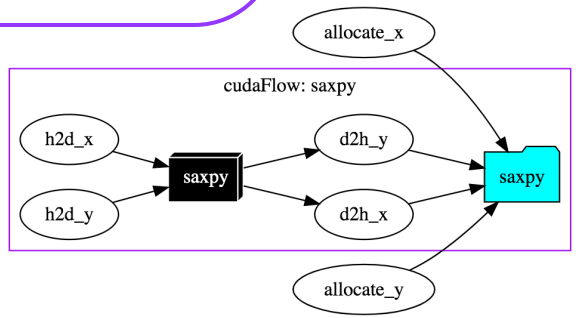


To Nvidia cudaGraph

```
auto cudaflow = taskflow.emplace([&](tf::cudaFlow& cf) {
   auto h2d_x = cf.copy(dx, hx.data(), N);  // CPU-GPU data transfer
   auto h2d_y = cf.copy(dy, hy.data(), N);
   auto d2h_x = cf.copy(hx.data(), dx, N);  // GPU-CPU data transfer
   auto d2h_y = cf.copy(hy.data(), dy, N);
   auto kernel = cf.kernel((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
   kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
});
```

```
cudaflow.succeed(allocate_x, allocate_y);
executor.run(taskflow).wait();
```

Users define GPU work in a graph rather than aggregated operations → single kernel launch to reduce overheads



Concurrent CPU-FPGA Tasking

- **Integrate with Princeton's OpenPiton FPGA emulator**
  - Prototype a "PitonFlow" of sequential operations

```
auto pitonflow = taskflow.emplace([&](tf::PitonFlow& pf) {
  auto launch = pf.bitstream("mybitstream", "localhost:200");
  auto writer = pf.write_led("OpenPiton");
  auto multiplier = pf.command("multiply");
  launch.precede(writer);
  writer.precede(multiplier);
});
```

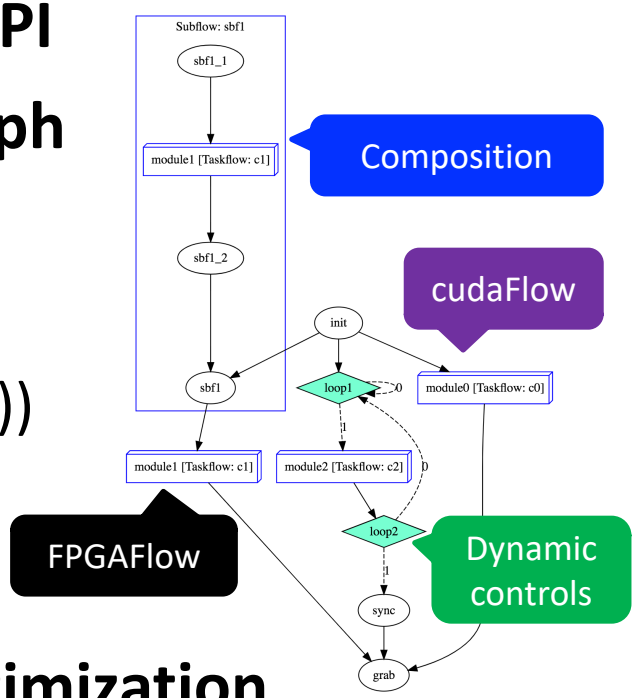
CPU-FPGA tasking

OpenPiton

Our functional programming-styled interface is extensible to various devices, provided a custom execution policy

Everything is Unified in Cpp-Taskflow

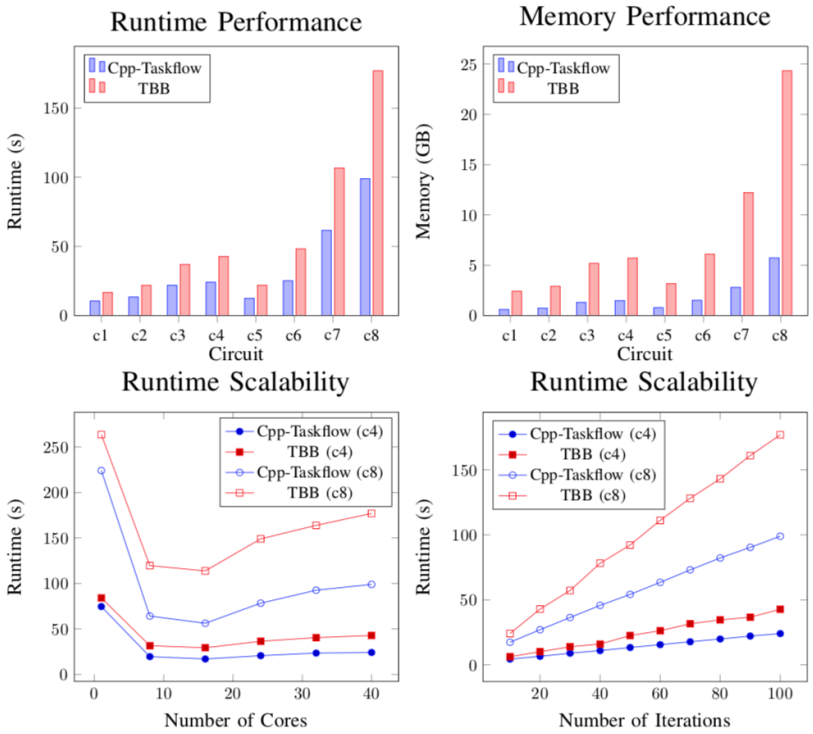
- Use the “emplace” method to create a task
- Use the “precede” method to add a task dependency
- No need to learn different sets of API
- You can create a really complex graph
  - Subflow(ConditionTask(cudaFlow))
  - ConditionTask(StaticTask(cudaFlow))
  - Composition(Subflow(ConditionTask))
  - Subflow(ConditionTask(FPGAFlow))
  - …
- Scheduler performs end-to-end optimization
  - Runtime, energy efficiency, and throughput



Reflect on the Monster Task Graph

❑ Now, it's no-brainer ☺

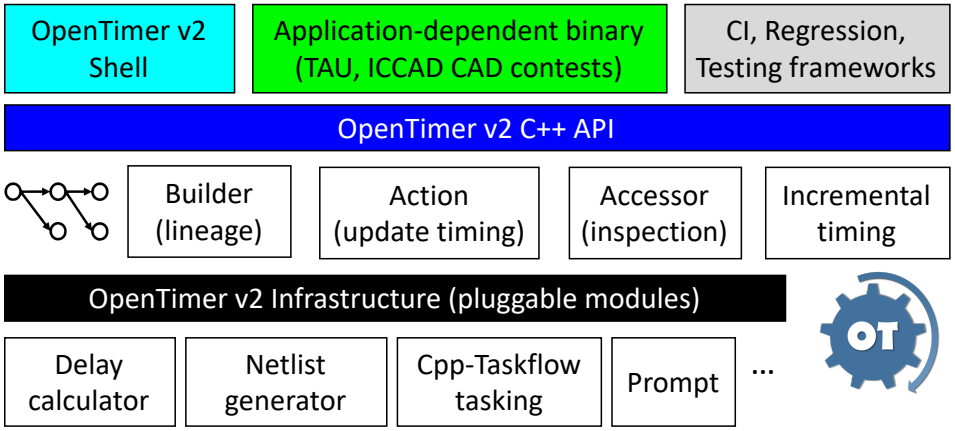
On a VLSI placement application:
47.81% faster than Intel TBB
4.24x fewer memory than Intel TBB

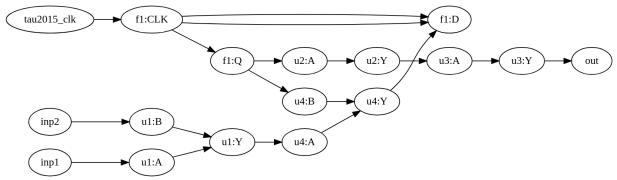


**TCAD20 (submitted): T.-W. Huang, et al, "Cpp-Taskflow: A General-purpose Parallel Task Programming System"*

Another Result: VLSI Timing Anslysis

- **OpenTimer v1: A VLSI Static Timing Analysis Tool**
  - v1 first released in 2015 (open-source under GPL)
  - Loop-based parallelisms using OpenMP 4.0
- **OpenTimer v2: A New Parallel Incremental Timer**
  - v2 first released in 2018 (open-source under MIT)
  - Task-based parallel decomposition using Cpp-Taskflow





V2 is completely rewritten using Cpp-Taskflow (real use case benchmark)

OpenTimer: https://github.com/OpenTimer/OpenTimer

Software Cost between v1 and v2

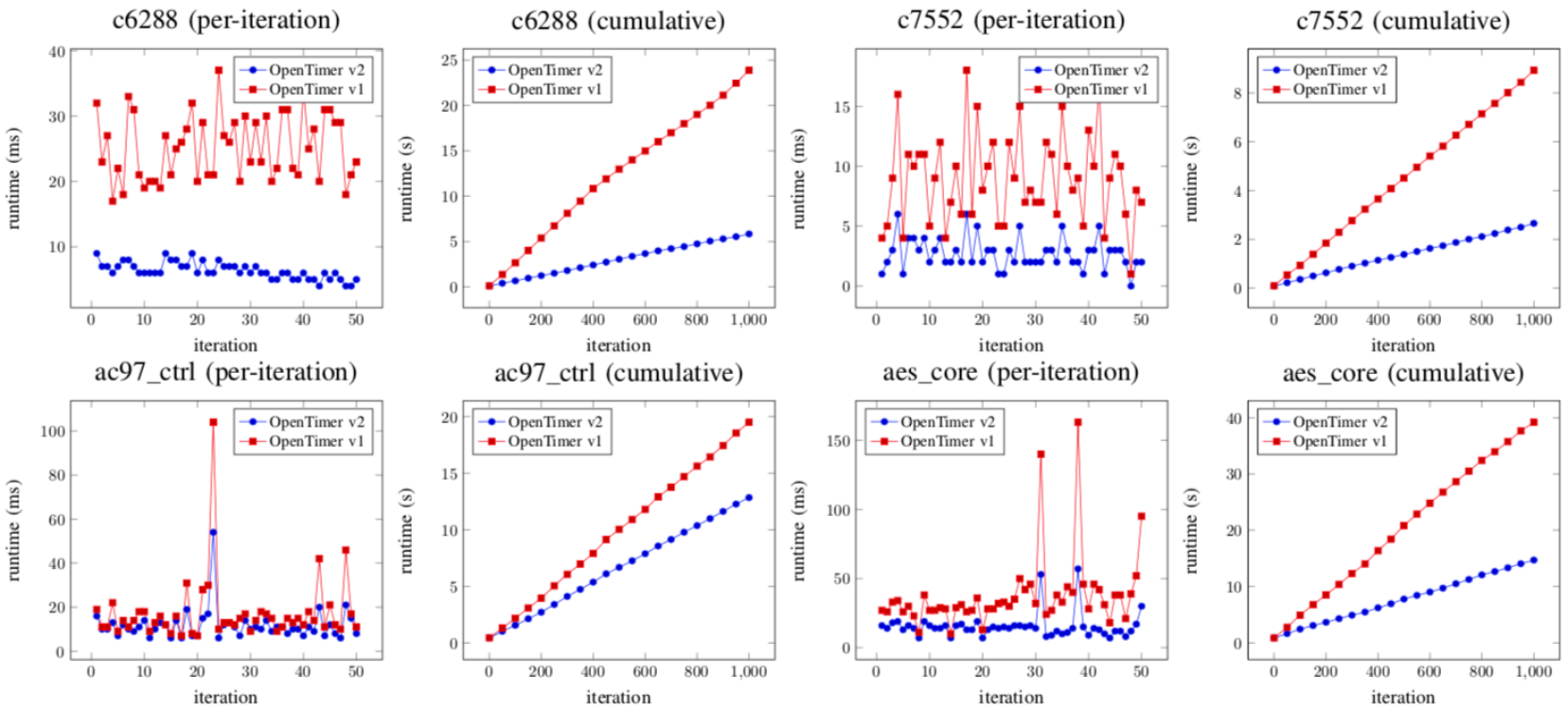
Tool	Task Model	LOC	MCC	Effort	Dev	Cost
v1	OpenMP 4.5	9,123	58	2.04	2.90	$275,287
v2	Cpp-Taskflow	4,482	20	0.97	1.83	$130,523

MCC: maximum cyclomatic complexity in a single function
Effort: development effort estimate, person-years (COCOMO model)
Dev: estimated average number of developers (efforts / schedule)
Cost: total estimated cost to develop (average salary = $56,286/year).

With Cpp-Taskflow, we saved 4K lines of parallel code, most from the sections to maintain dynamic data structures in support for OpenMP's loop-based task decomposition strategies. Reported by SLOCCount, the cost to develop is $275K with OpenMP and $130K with Cpp-Taskflow.

SLOCCount: https://dwheeler.com/sloccount/

Runtime Performance between v1 and v2



The new runtime is 1.4-3.8x faster. Task-based strategies enable more efficient parallel timing propagation; computations flow naturally with the structure of the timing graph, in no need of synchronization level-by-level.

Programming models matter

Different models give different implementations. The parallel code sections may run fast, yet the data structures to support a parallel decomposition strategy may overwhelm all its runtime benefits.

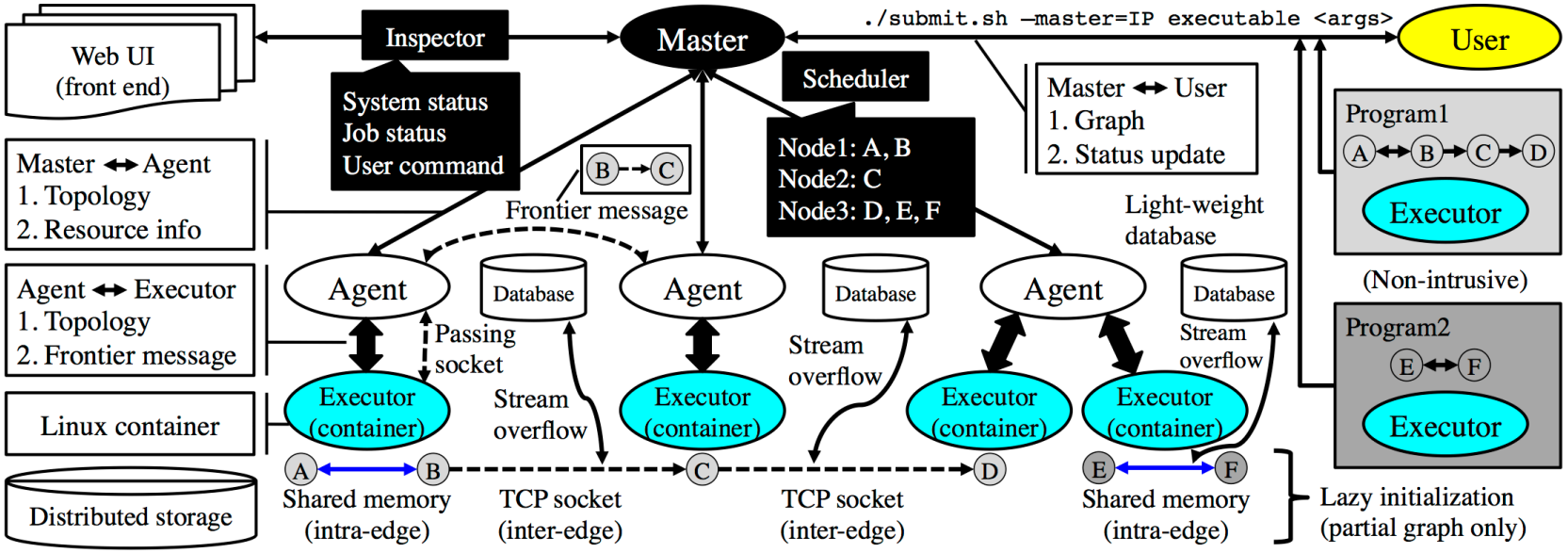
In OpenTimer v1, loop-based OpenMP code is very fast. But it's too costly to maintain the levellist data structure over iterations.

Selected Modules for the Rest of Talk

- **Vertical scalability**
  - Cpp-Taskflow: Parallel Task Programming in Modern C++
  - Result on ML-centric VLSI placement (>8M tasks)
  - Result on VLSI timing analysis (>1B tasks)
- **Horizontal scalability**
  - DtCraft: Distributed Programming and Execution Engine
  - Result on complex heterogeneous ML workflows
- **Technical details**
  - HeteroSteal: A Generalized Work-stealing Scheduler
  - Learning-based distributed scheduling
  - Result on improved system performance

DtCraft Programming System [TCAD19]

- **A new stream graph programming models**
  - Vertex program & data-parallel streams (computations)
  - No difficult distributed computing details
- **Everything is by default distributed**



DtCraft: https://github.com/tsung-wei.huang/DtCraft

A Hello-World Example

- **An iterative and incremental flow**
  - Two vertices + two streams

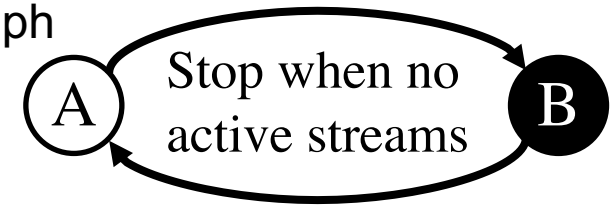
Step 4: A's resource
1 CPU / 1 GB RAM

Step 2:
string msg;

Step 3: A➔B callback

```
[=] (auto& B, auto& is) {
  Extract string from is;
  print string;
}
```

Step 1: stream graph



istream ➔ B

Step 3: A➔B callback

```
[=] (auto& A, auto& is) {
  Extract string from is;
  print string
}
```

Step 2:
string msg;

Step 4: B's resource
1 CPU / 1 GB RAM

A ⬅ istream

Step 5: ./submit –master=127.0.0.1 hello-world

DtCraft Code of Hello World

```
dtc::Graph G;
auto A = G.vertex(); // create a vertex A
auto B = G.vertex(); // create a vertex B
auto AB = G.stream(A, B);  // create a data stream A->B
auto BA = G.stream(B, A);  // create a data stream B->A
G.container().add(A).cpu(1).memory(1_GB);
G.container().add(B).cpu(1).memory(1_GB);
A.on( [&AB] (dtc::Vertex& v) {
   (*v.ostream(AB))("hello world from A"s);
   dtc::cout("Sent 'hello world from A' to stream ", AB, "\n"); }
);
B.on( [&BA] (dtc::Vertex& v) {
   (*v.ostream(BA))("hello world from B"s);
   dtc::cout("Sent 'hello world from B' to stream ", BA, "\n"); }
);
AB.on( [] (dtc::Vertex& B, dtc::InputStream& is) {
   if(std::string b; is(b) != -1) {
      dtc::cout("Received: ", b, '\n');
      return dtc::Event::REMOVE;
   }
   return dtc::Event::DEFAULT;
});
BA.on( [] (dtc::Vertex& A, dtc::InputStream& is) {
   if(std::string a; is(a) != -1) {
     dtc::cout("Received: ", a, "\n");
     return dtc::Event::REMOVE;
   }
   return dtc::Event::DEFAULT; }
);
dtc::Executor(G).run();
```

Create two vertices A and B

Create two data parallel streams AB and BA

Assign vertex program (constructor)

Define computation callback on data streams

Create an executor to execute the stream graph

Only a couple lines of code → fully capable of distributed computing

Without DtCraft …

```cpp
auto count_A = 0;
auto count_B = 0;

// Send a random binary data to fd and add the
// received data to the counter.
auto pinpong(int fd, int& count) {
  auto data = random<bool>()
  auto w = write(fd, &data, sizeof(data));
  if(w == -1 && errno != EAGAIN) {
    throw system_error("Failed on write");
  }
  data = 0;
  auto r = read(fds, &data, sizeof(data));
  if(r == -1 && errno != EAGAIN) {
    throw system_error("Failed on read");
  }
  count += data;
}

int fd = -1;
std::error_code errc;
```

server.cpp

```cpp
if(getenv("MODE") == "SERVER") {
  fd = make_socket_server_fd("9999", errc);
}
else {
  fd = make_socket_client_fd("127.0.0.1", "9999", errc);
}
```

client.cpp

```cpp
if(fd == -1) {
  throw system_error("Failed to make socket");
}
```

Branch your code to server and client for distributed computation!

simple.cpp → server.cpp + client.cpp

```cpp
int make_socket_server_fd(
  std::string_view port,
  std::error_code errc
) {
  int fd {-1

  struct adc
  struct adc
  struct adc

  std::memse
  hints.ai_f
  hints.ai_s
  hints.ai_p
  hints.ai_f

  int one {1
  int ret;

  if((ret =
    errc = m
    return -
  }

  // Try to
  for(ptr =

    // Ignor
    if(ptr->
      goto t
    }

    if((fd =
      errc =
      goto t
    }

    ::setsoc

    if(::bir
      errc =
      goto t
    }

    if(::lis
      errc =
      goto t
    }
    else {
      break;
    }

  try_next
```

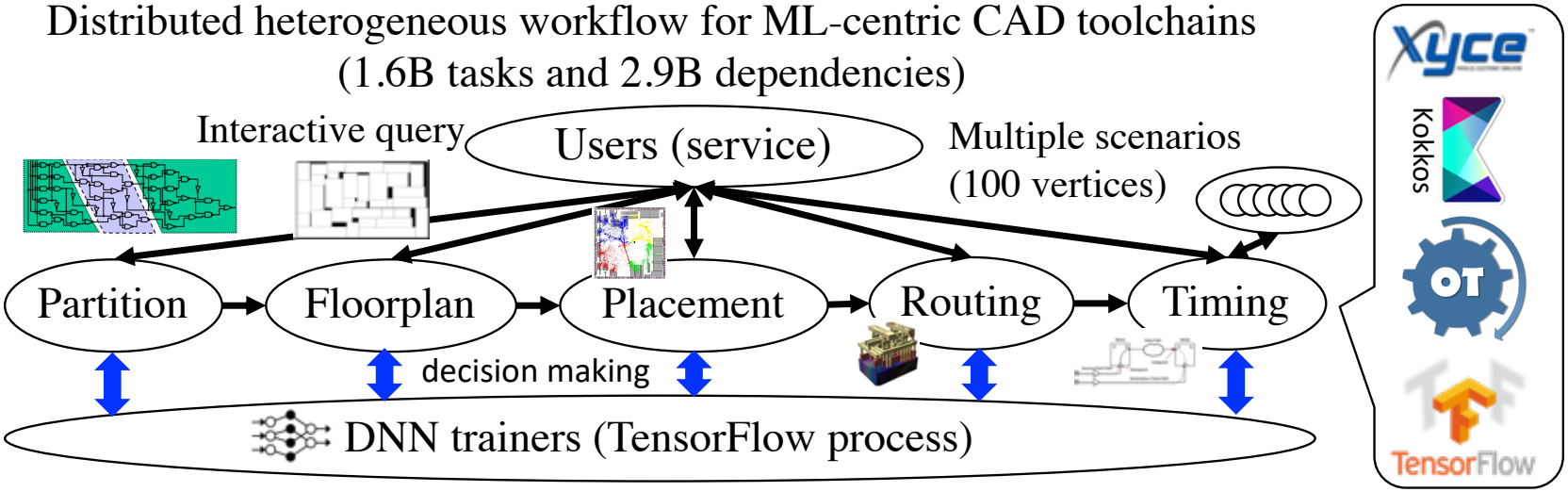
```cpp
    if(fd != -1) {
      ::close(fd);
      fd = -1;
    }
  }

  ::freeac

  // Assig
  return f
}

int make_s
  std::str
  std::str
  std::err
) noexcept

  errc.cle

  struct a
  struct a

  std::mem
  hints.ai
  hints.ai
  hints.ai

  int ret;
  int fd =
  int tri

  if((ret
    errc =
    return
  }

  // Try e
  for(auto

    // Ign
    if(pt
      goto
    }

    if((fd
      err
      goto
    }

    make_f
```

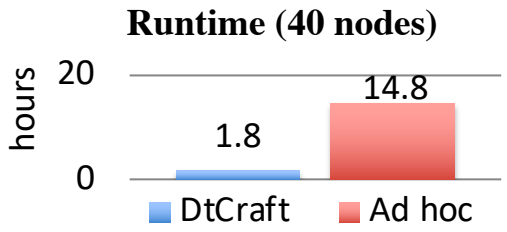
```cpp
    make_fd_close_on_exec(fd);

    tries = 3;

    issue_connect:
    ret = ::connect(fd, ptr->ai_addr, ptr->ai_addrlen);

    if(ret == -1) {
      if(errno == EINTR) {
        goto issue_connect;
      }
      else if(errno == EAGAIN && tries--) {
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
        goto issue_connect;
      }
      else if(errno != EINPROGRESS) {
        goto try_next;
      }
      errc = make_posix_error_code(errno);
    }

    // Poll the socket. Note that writable return doesn't mean it is connec
    if(select_on_write(fd, 5, errc) && !errc) {
      int optval = -1;
      socklen_t optlen = sizeof(optval);
      if(::getsockopt(fd, SOL_SOCKET, SO_ERROR, &optval, &optlen) < 0) {
        errc = make_posix_error_code(errno);
        goto try_next;
      }
      if(optval != 0) {
        errc = make_posix_error_code(optval);
        goto try_next;
      }
      break;
    }

    try_next:

    if(fd != -1) {
      ::close(fd);
      fd = -1;
    }
  }

  ::freeaddrinfo(res);

  return fd;
}
```

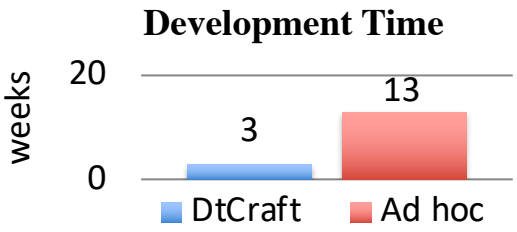
A lot of boilerplate code plus hundred lines of scripts to enable distributed flow…

Heterogeneous ML Workflows

- **Prototype a fully automated layout generator**
  - Ran on a 40-node Amazon cluster (4CPU/1GPU each)







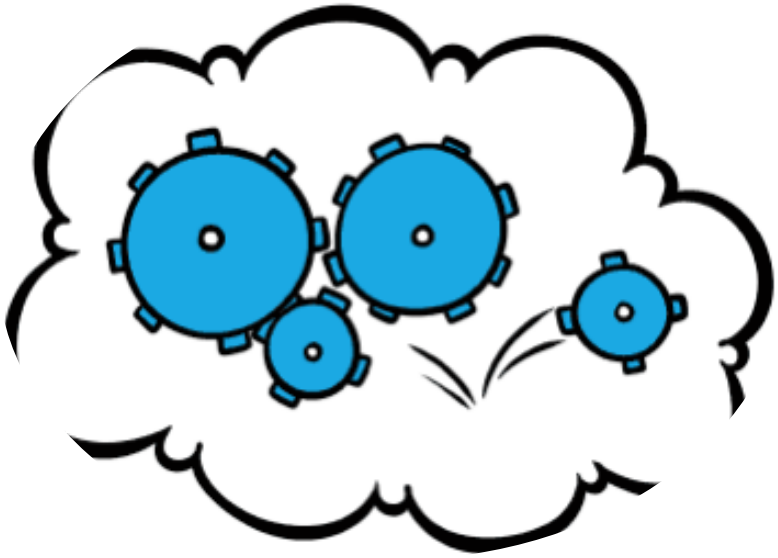
DtCraft Cluster computing + Cpp-Taskflow

New programming models enable simultaneous performance and productivity gain

End-to-end parallelism matters

Handcrafting a complex heterogeneous workflow using ad hoc scripts can result in result in suboptimal performance due to the lack of runtime optimization on the whole flow with available resources.

In our distributed workflow prototype, we found even simple pipeline optimization can boost >2x performance compared to the batch flow.

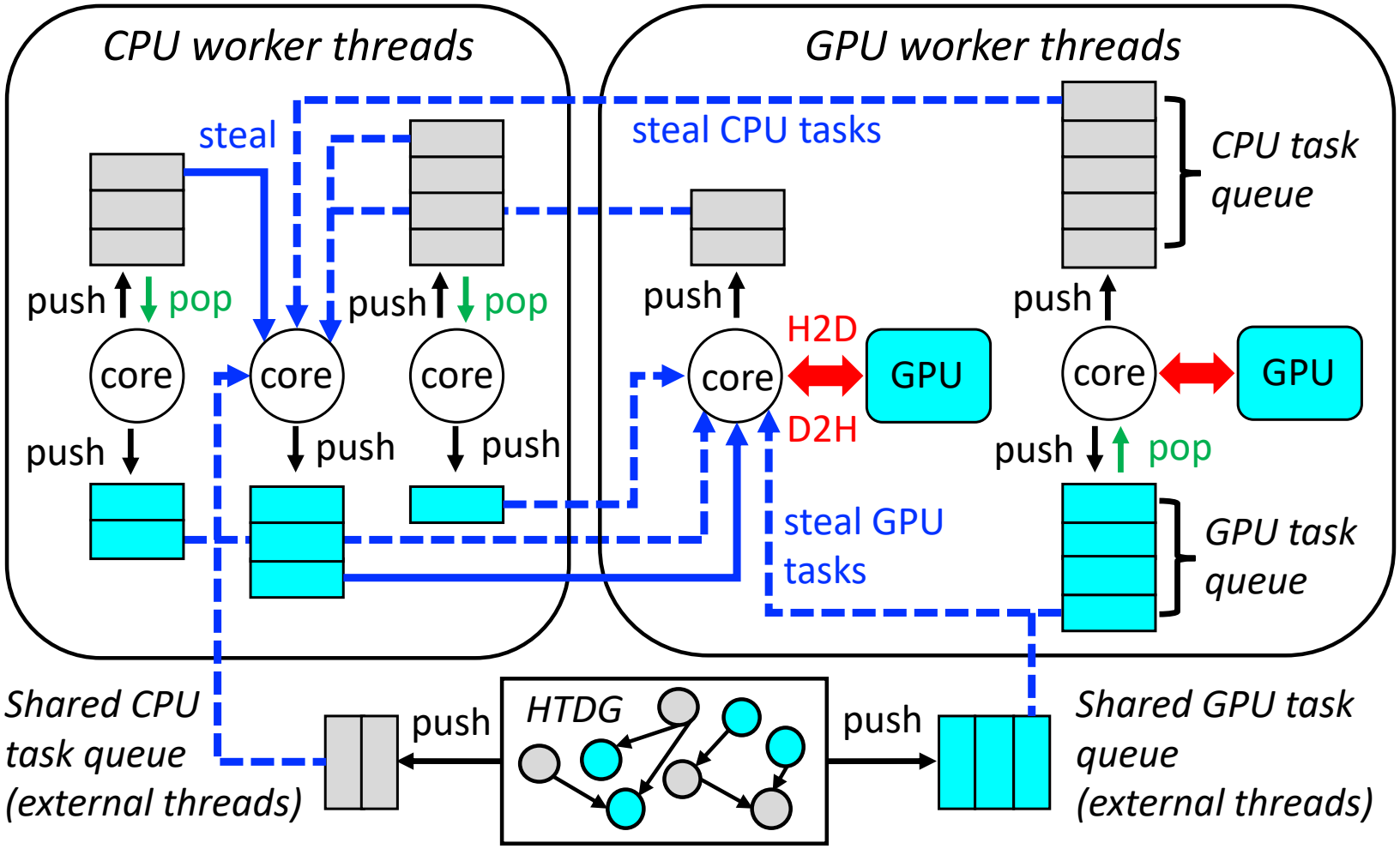


What about technical details?

Selected Modules for the Rest of Talk

- **Vertical scalability**
  - Cpp-Taskflow: Parallel Task Programming in Modern C++
  - Result on ML-centric VLSI placement (>8M tasks)
  - Result on VLSI timing analysis (>1B tasks)
- **Horizontal scalability**
  - DtCraft: Distributed Programming and Execution Engine
  - Result on complex heterogeneous ML workflows
- **Technical details**
  - HeteroSteal: A Generalized Work-stealing Scheduler
  - Learning-based distributed scheduling
  - Result on improved system performance

HeteroSteal: Generalized Work Stealing



Improved energy efficiency, throughput, and performance

Key Property and Components

- ***O(N)* sync cost on *N* heterogeneous domains**
  - Cost to decide when to put a worker to sleep or to work
- **Scheduler consists of two parts:**
  1. Task-level scheduling
     - Decide which task to enqueue at runtime
     - Support our unified tasking interface
     - Model flows via strong dependency and weak dependency
  2. Worker-level scheduling
     - Decide which worker to preempt and which worker to wake up
     - Adapt the number of workers to dynamically generated tasks
     - Control the wasteful steals within a bounded interval
- **Maximize the entire system performance**

Provably Good Scheduling Strategy

- **Balance workers with dynamically generated tasks**
- **We prove to avoid under-subscription**
  - Worker threads can't be lower than available tasks
  - Unless all workers are fully loaded
- **We prove to avoid over-subscription**
  - Worker threads can't exceed too much available tasks
  - Wasteful thieves are bounded

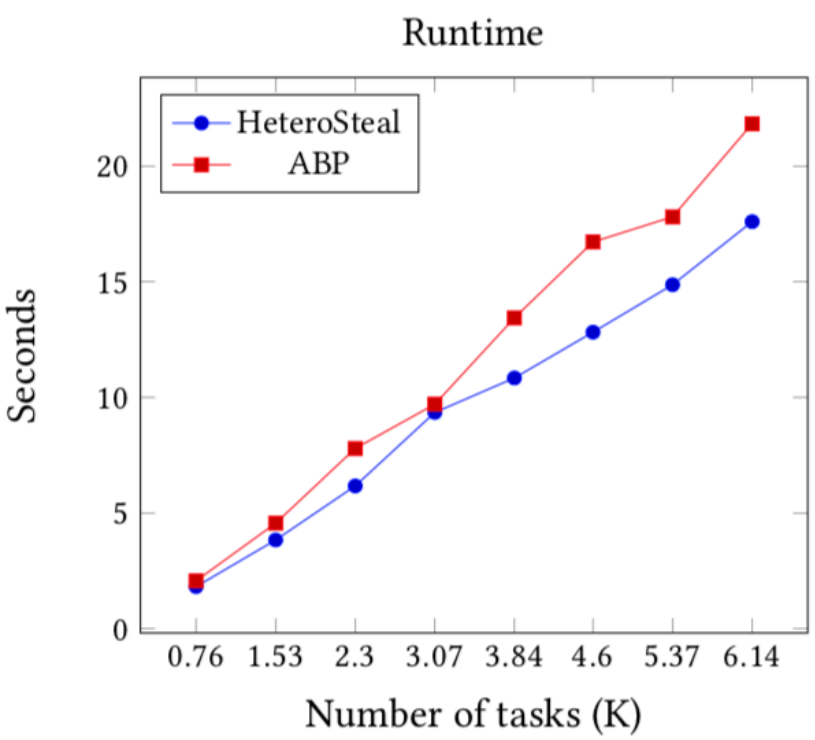
Lemma 1. *When a worker is active and at least one worker is inactive, one thief always exists.*

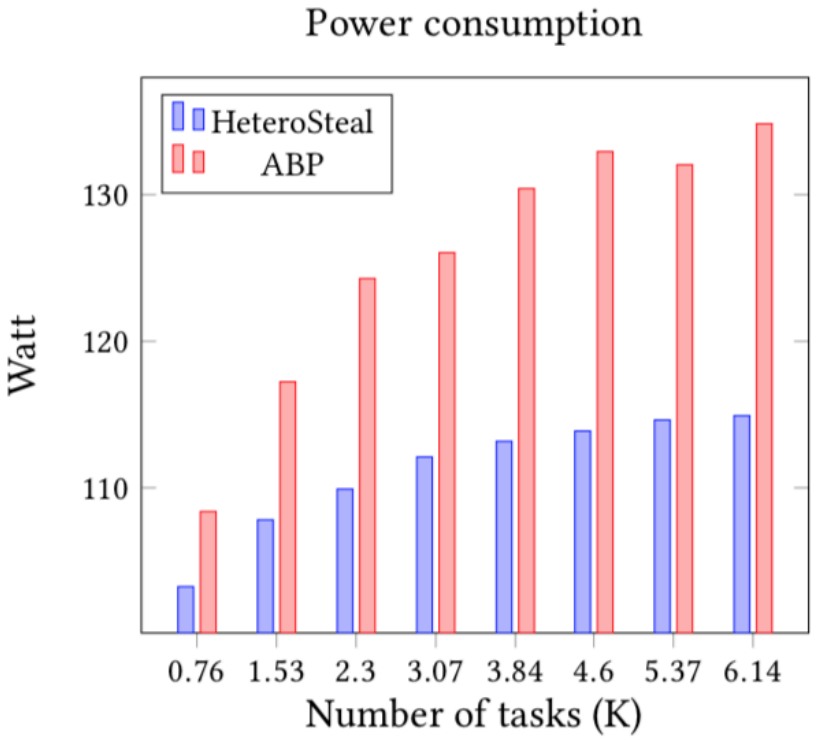
Lemma 2. *Given a group of thieves, only one thief in the group exists after* $O((STEAL\_BOUND + YIELD\_BOUND) * S + C)$ *time, where* S *is the time to perform a steal and* C *is a constant.*

We developed a two-phase synchronization to reach this goal

HeteroSteal vs ABP on a Mix BS-AES Graph

- **Comparison of runtime and power consumption**
  - 40 CPU cores and 4 Nvidia GeForce RTX 2080 GPUs

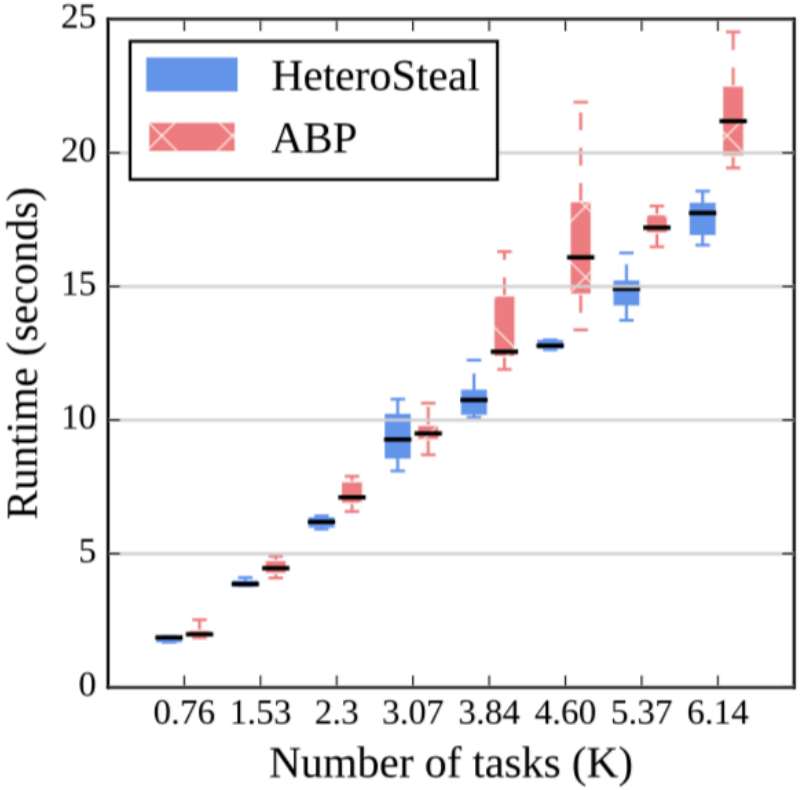


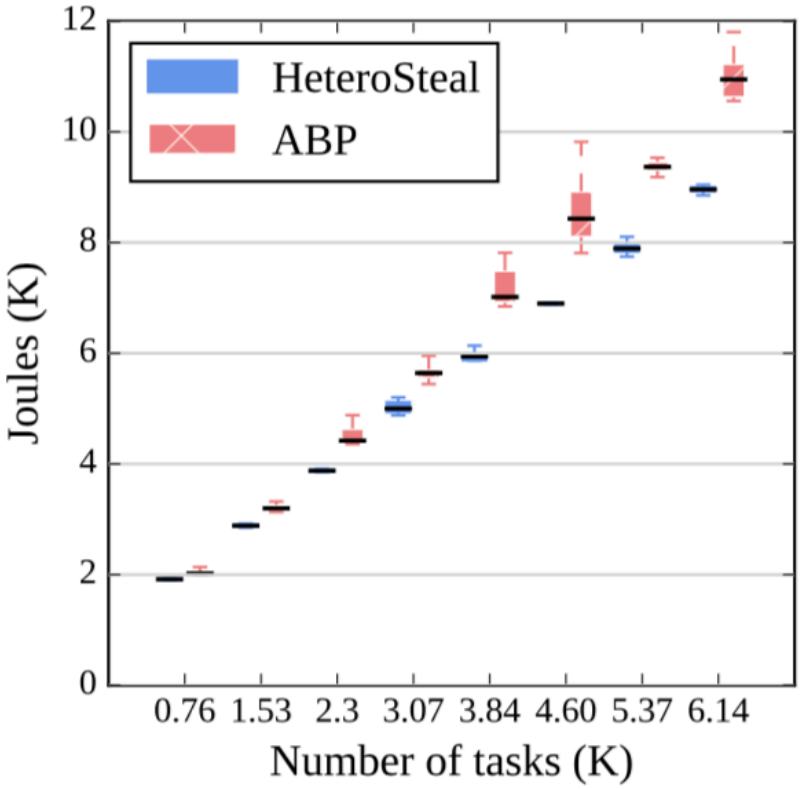


N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread Scheduling for Multiprogrammed Multiprocessors," ACM SPAA, pp. 119—129, 1998

HeteroSteal vs ABP on a Mix BS-AES Graph

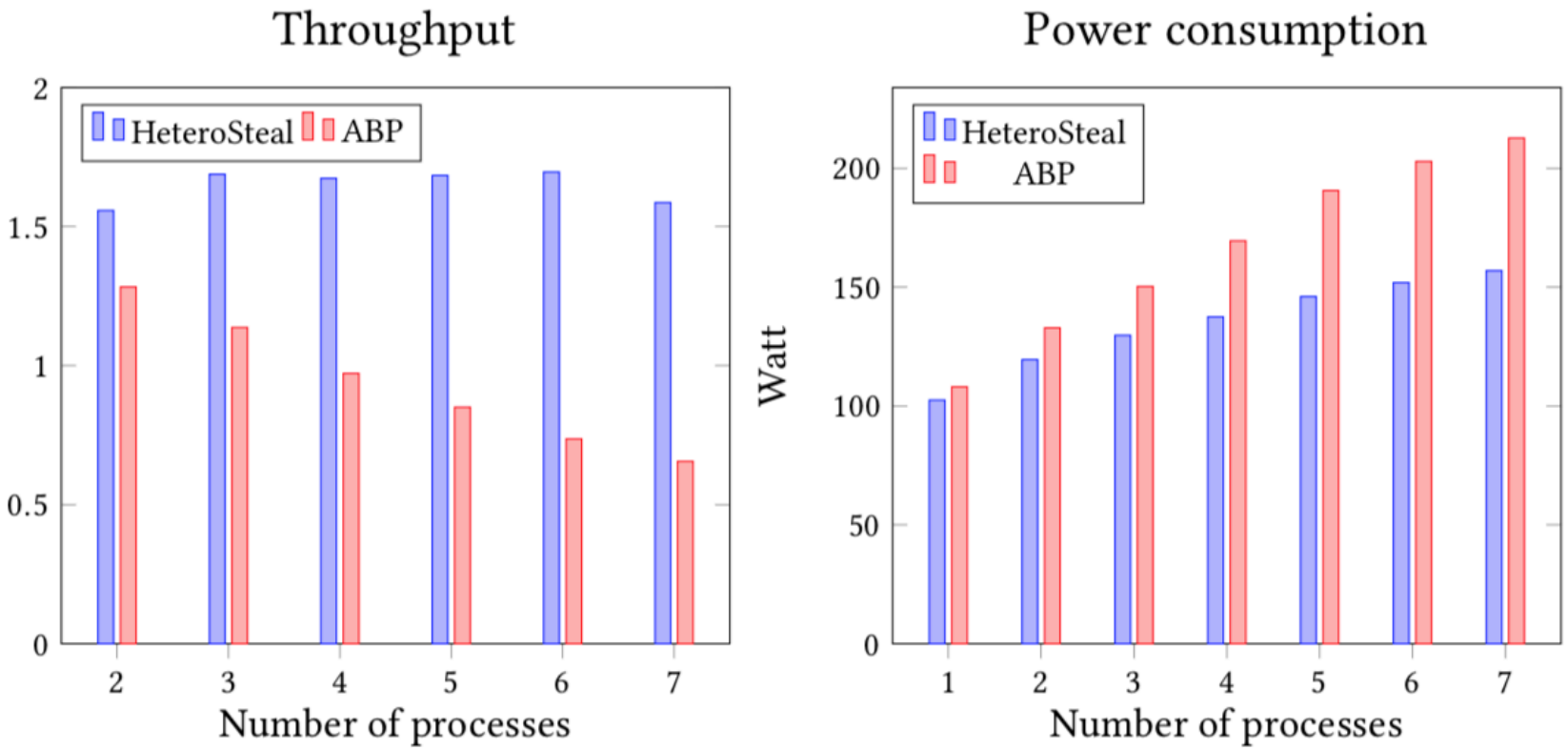
- **Runtime and energy distribution over multiple runs**
  - 40 CPU cores and 4 Nvidia GeForce RTX 2080 GPUs





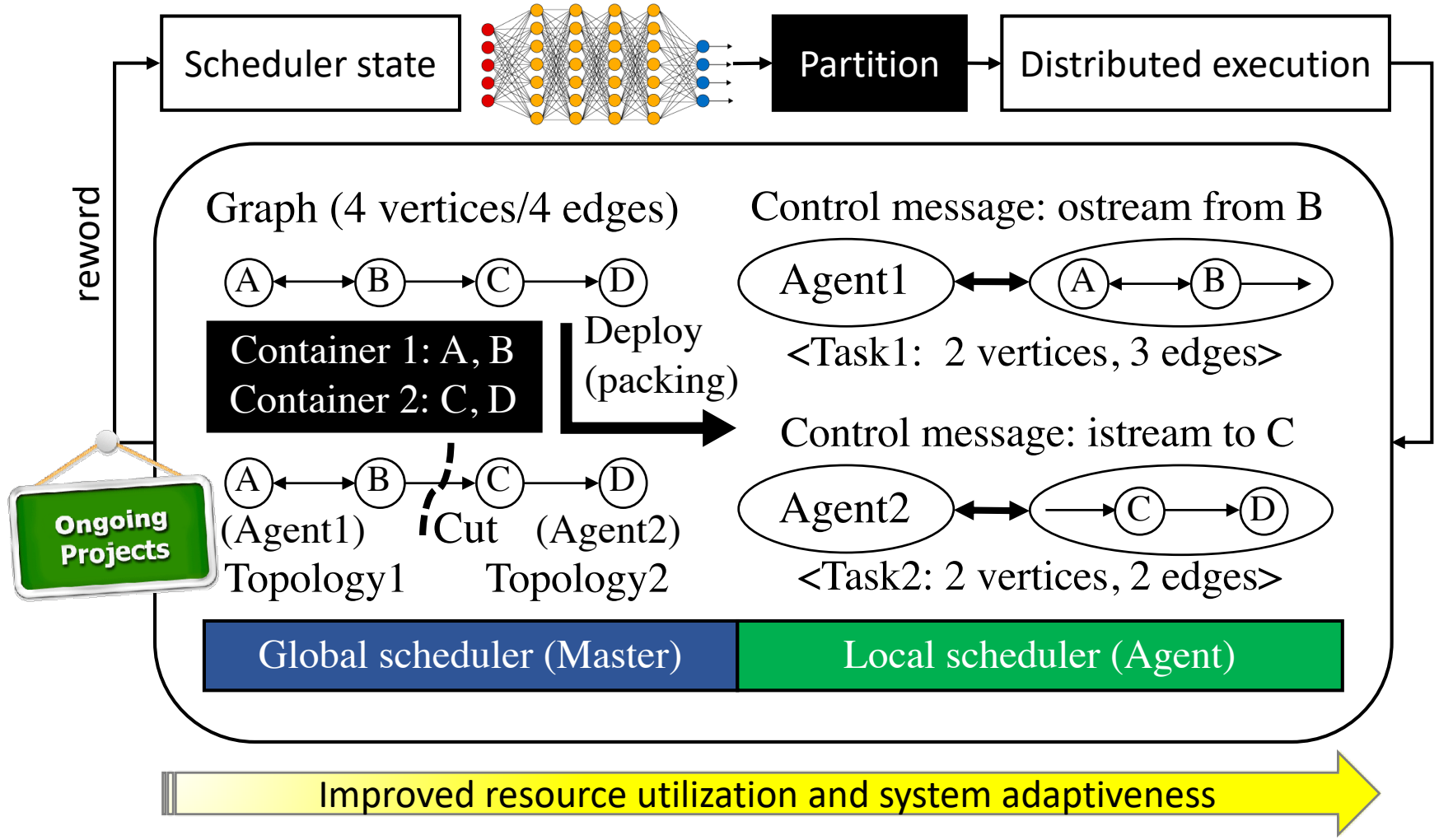
HeteroSteal vs ABP on a Mix BS-AES Graph

- **Co-run system throughput and power consumption**
  - 40 CPU cores and 4 Nvidia GeForce RTX 2080 GPUs



Learning-based Distributed Scheduling

- **Autonomously learn to optimize service objectives**



Who to sleep or work matters

Most existing work focus on "task-level" scheduling but ignore the impact of "worker-level" management on system performance (runtime, power, co-run throughput).

In VLSI timing experiment, our adaptive work-stealing scheduler achieved faster runtime using less CPU resources than Intel TBB and BWS (EuroSys15). Our result delivered higher energy efficiency and system throughput.

Conclusion

- **A general-purpose parallel task programming system**
  - Simple, efficient, and transparent tasking models
  - CPU-, GPU-, and FPGA-collaborative computing
  - Real case use in ML, VLSI (billion-scale tasking)
- **On-going and future work**
  - Improving the system in all aspects
  - Developing transition tools/linters
- **WE ARE OPEN TO COLLABORATION!!!**
  - https://github.com/cpp-taskflow/cpp-taskflow
  - https://github.com/tsung-wei-huang/DtCraft
  - https://tsung-wei-huang.github.io/

Community Engagement – Thank You All!