DtCraft: A General-purpose Distributed Programming System using Data-parallel Streams

Tsung-Wei Huang
Research Assistant Professor
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, IL, USA

Outline

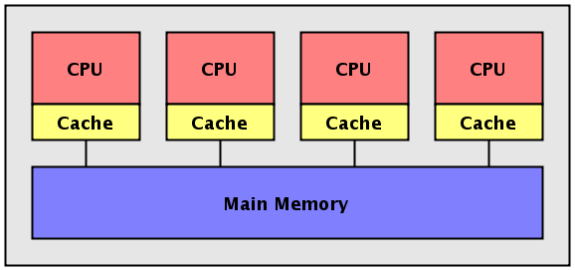
- **Express your parallelism in the right way**
  - A “hard-coded” distributed timing analysis framework
- **Boost your productivity in writing parallel code**
  - DtCraft system
- **Leverage your time to produce promising results**
  - Vanilla examples
  - Machine learning
  - Graph algorithms
  - Distributed timing

Outline

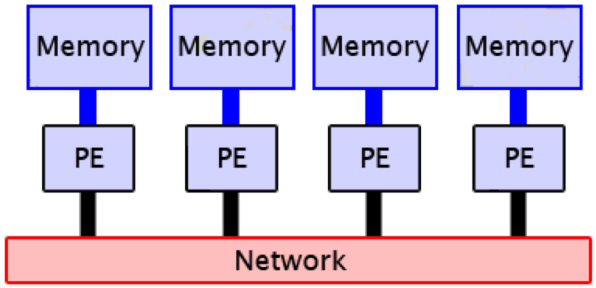
- **Express your parallelism in the right way**
  - A “hard-coded” distributed timing analysis framework
- **Boost your productivity in writing parallel code**
  - DtCraft system
- **Leverage your time to produce promising results**
  - Vanilla examples
  - Machine learning
  - Graph algorithms
  - Distributed timing

Motivation: Distributed Timing

- **Deal with the ever-increasing design complexity**
  - Billions of transistors result in very large timing graphs
  - Analyze the timing under different conditions
  - Vertical scaling is not cost efficient
- **Want to scale out our computations**
  - Leverage the power of computer clusters (cloud computing)



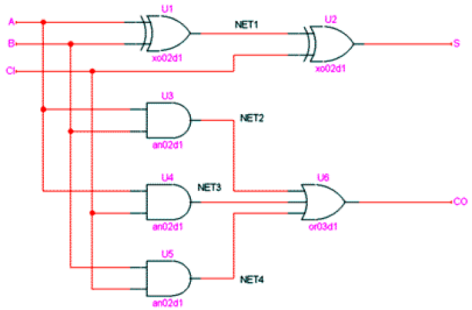
Single node (threaded)

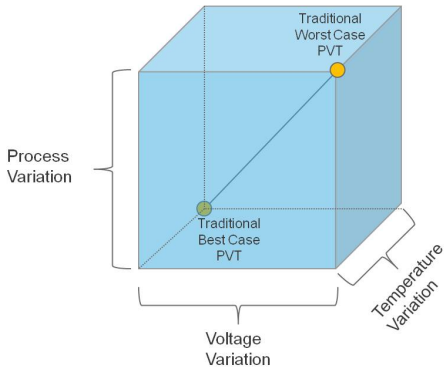


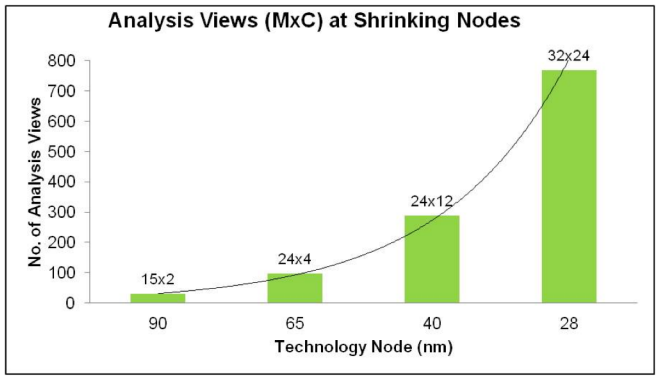
Datacenter (distributed)

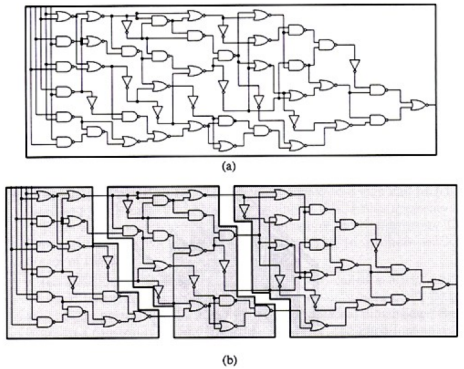
What Exactly is this Workload?

- **Input cannot be easily partitioned**
  - Circuit netlist, libraries are not partitionable
  - Must keep in memory for performance need
- **Incremental timing**
  - Iterative, irregular, graph-based operations
- **Multi-mode multi-corner (MMMC) timing analysis**
  - Timing runs across all combinations of modes and corners
  - Each timing view is logically independent of each other



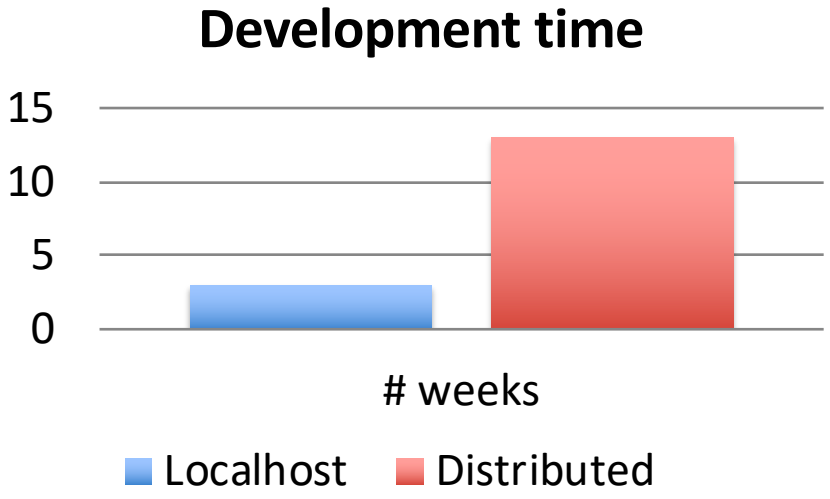






Good News and Bad News

- **We have many things to parallelize**
- **Developing a distributed program is very difficult**
  - Several weeks more than a single-machine counterpart
  - Network programming, subtly buggy code, etc
- **Scalability and transparency**
  - Intend to focus on high level rather than low level
  - Want better productivity
  - Want better flexibility
  - Want better performance



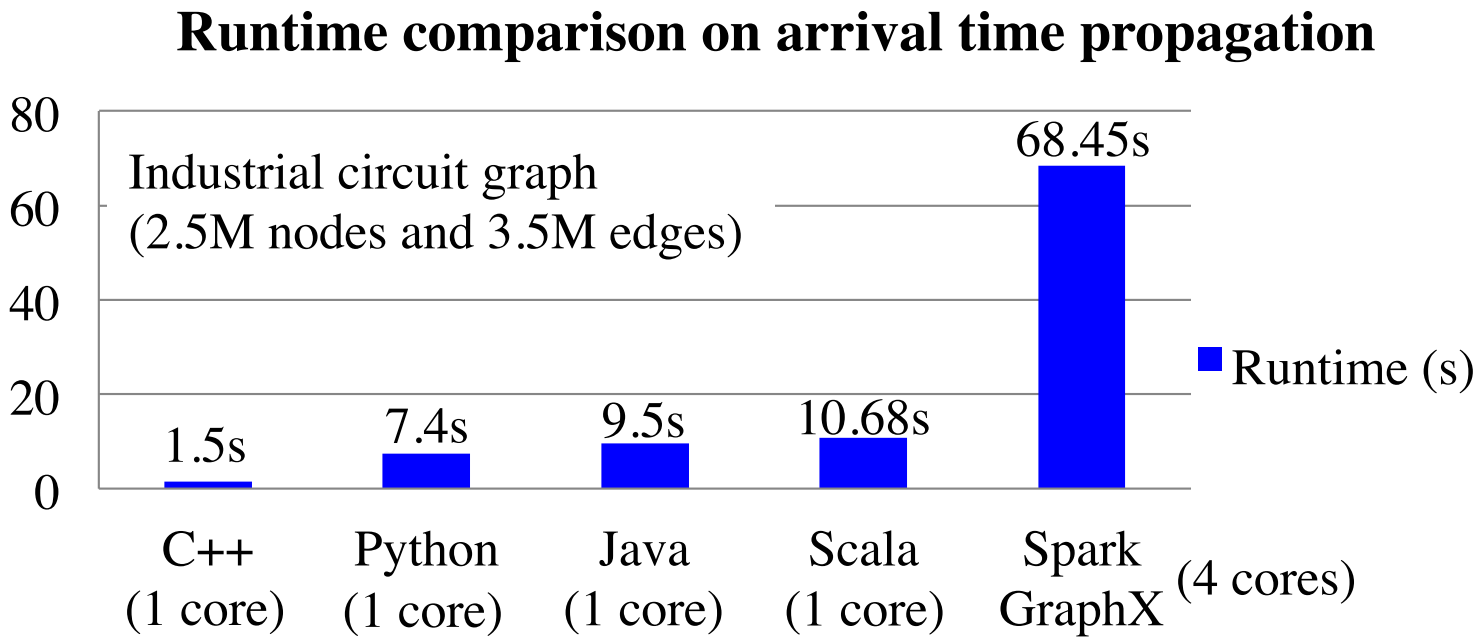
Distributed Systems in Big Data Community

- **Hadoop**
  - Distributed MapReduce platform on HDFS
- **Cassandra**
  - Scalable multi-master database
- **Chukwa**
  - A distributed data collection system
- **Zookeeper**
  - Coordination service for distributed application
- **Mesos**
  - A high-performance cluster manager
- **Spark**
  - A fast and general computing engine for big-data analytics

The Questions are

- ❑ Are these packages suitable for timing?
- ❑ What are the potential hurdles for me to use them?
- ❑ How much code rewrite do I need?
- ❑ What is the significance of adopting new languages?
- ❑ Will I lose performance?

Big-data Tool is Not an Easy Fit!



Method	Spark (RDD + GraphX Pregel)	Java (SP)	C++ (SP)
Runtime (s)	68.45	9.5	**1.50**

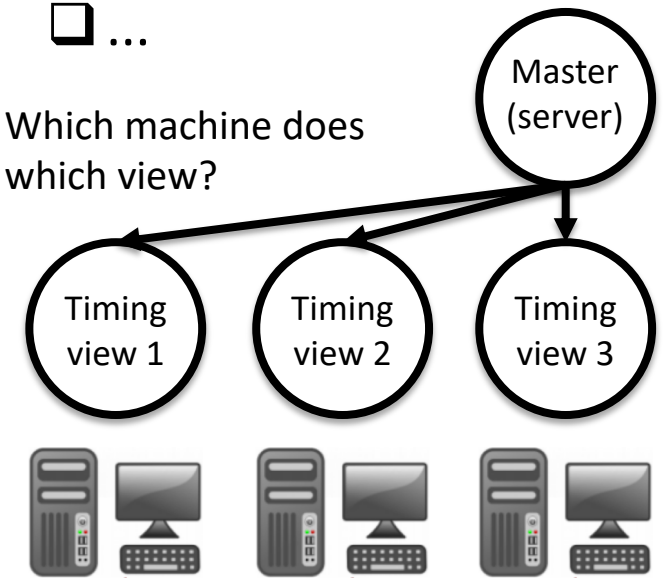
Overhead of MapReduce

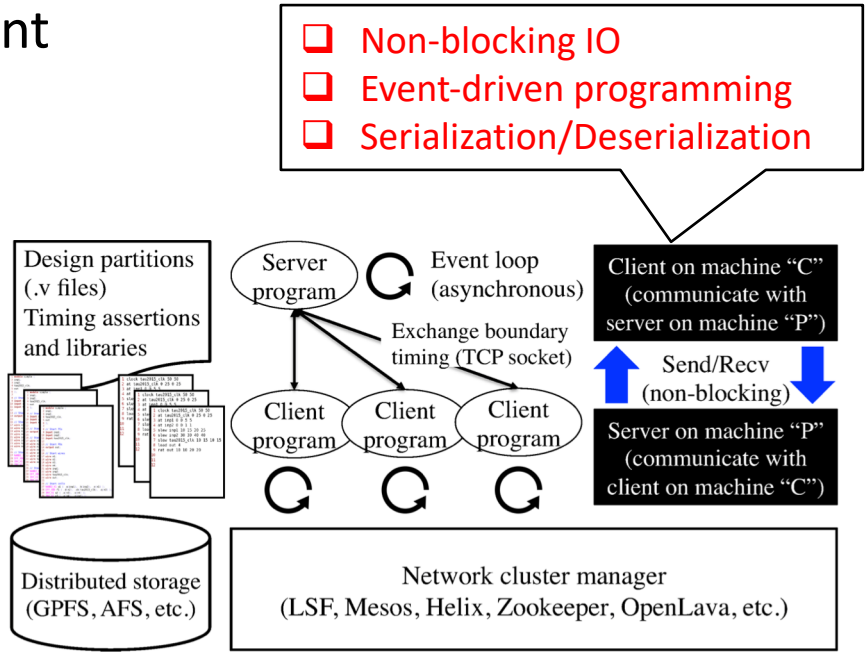
Language difference

A Hard-coded Distributed Timing Framework

- **Built from the scratch using raw Linux Socket**
  - Hard code using Linux sockets
  - Explicit data movement
  - Explicit job execution
  - Explicit parallelism management
  - …

Difficult scalability ☹
Large amount of code rewrites ☹





Huang et al., "A Distributed Timing Analysis Framework for Large Designs," IEEE/ACM DAC16

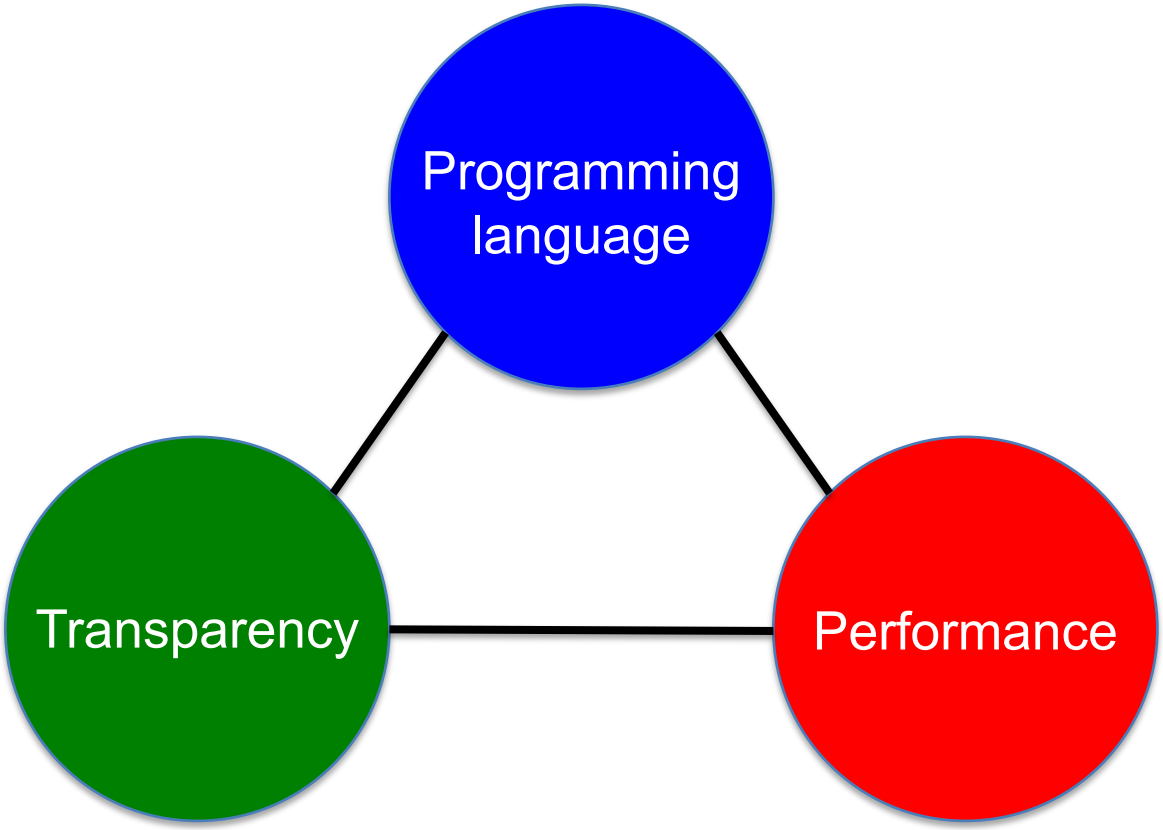
Observations

- **Big data doesn't fit well in my need**
  - IO-bound vs CPU-bound
  - Unstructured data vs structured data
  - JVM vs C/C++
- **Life shouldn't be hard-coded …**
  - Deal with low-level socket programming
  - Move data explicitly between compute nodes
  - Manage cluster resources on your own
  - Result in a large amount of development efforts
- **Want parallel programming *at scale* more *productive***

Outline

- **Express your parallelism in the right way**
  - A “hard-coded” distributed timing analysis framework
- **Boost your productivity in writing parallel code**
  - DtCraft system
- **Leverage your time to produce promising results**
  - Vanilla examples
  - Machine learning
  - Graph algorithms
  - Distributed timing

What does “Productivity” really mean?



Why is “Productivity” Important?

- **In cloud era, machines are cheaper than coding**
  - Hardware is just a commodity
  - Coding takes people and time
- **Today’s parallel programming**
  - Many redundant steps
  - Many boilerplate code blocks
  - Many explicit thread managements

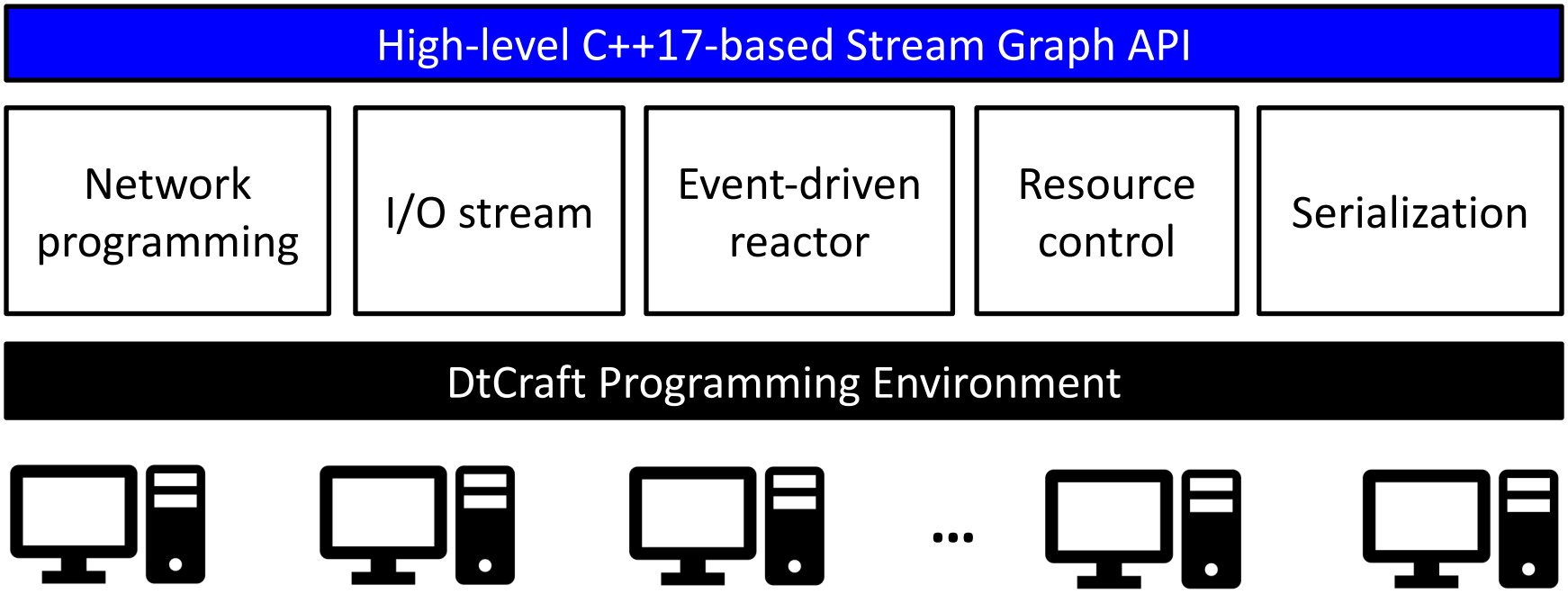
2016 average software engineer salary > 100K USD

```
// C++ thread example
std::thread t1([=](){ /* do something */ });
std::thread t2([=](){ /* do another thing */ });
t1.join();  // release thread 1 resource
t2.join();  // release thread 2 resource
```

- **We want computationally productive code**

A New Solution: DtCraft

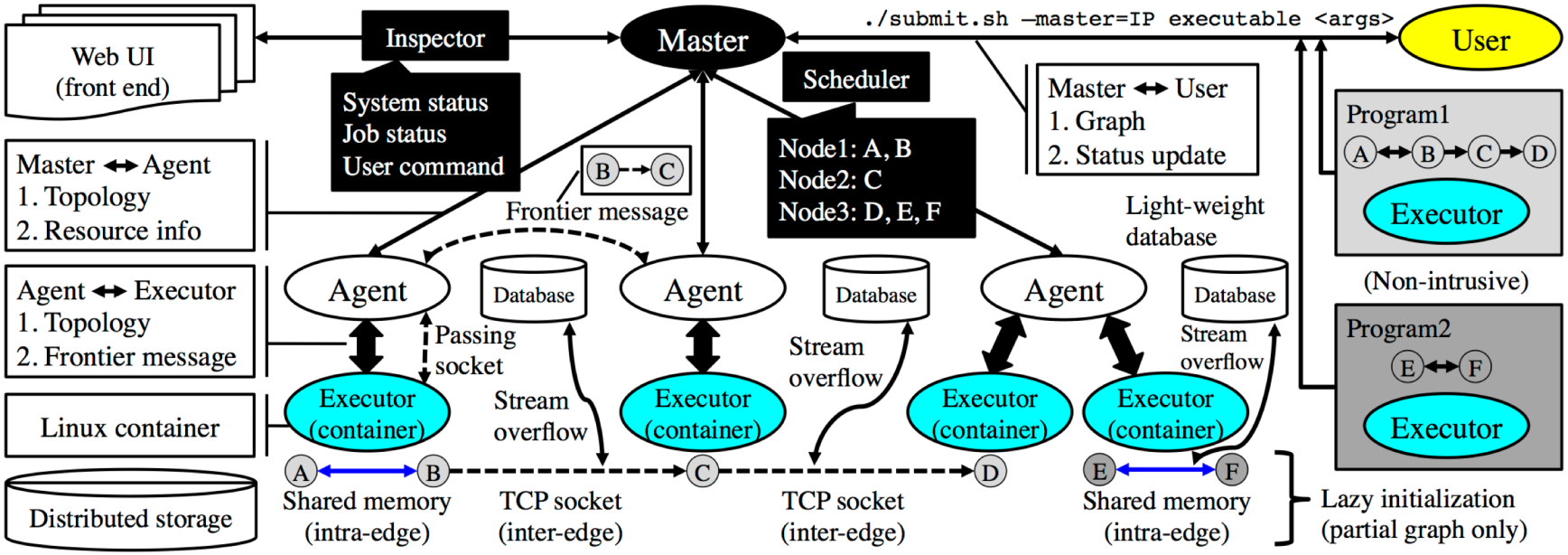
- **A unified engine to simplify cluster programming**
  - Completely built from the ground up using C++17
- **Save your time away from the pain of DevOps**



T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "DtCraft: A high-performance distributed execution engine at scale," IEEE TCAD, to appear, 2018

System Architecture

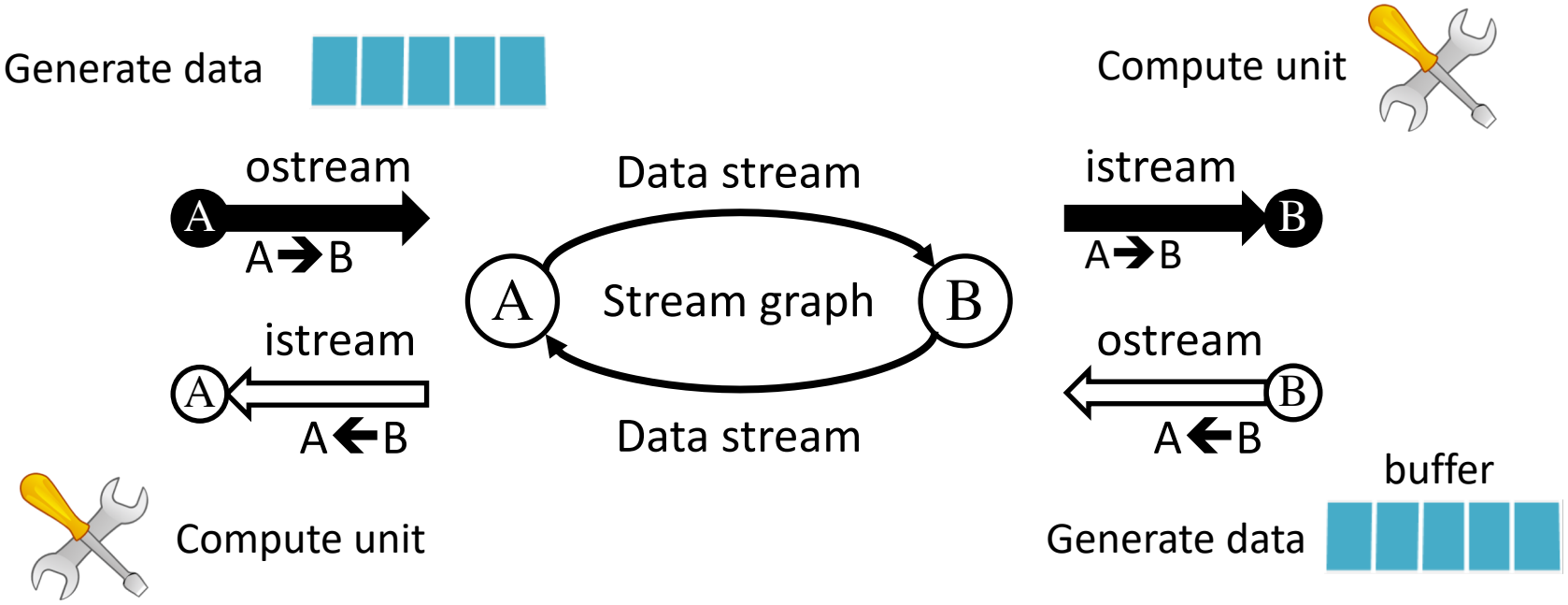
- **Express your parallelism in our *stream graph* model**
  - Generic dataflow at any granularity
- **Deliver transparent concurrency through the kernel**
  - Automatic workload distribution and message passing



DtCraft website: http://dtcraft.web.engr.illinois.edu/
DtCraft github: https://github.com/twhuang-uiuc/DtCraft

Stream Graph Programming Model

- **A general representation of a dataflow**
  - Abstraction over computation and communication
- **Analogous to the assembly line model**
  - Vertex storage ➔ goods store
  - Stream processing unit ➔ independent workers



Outline

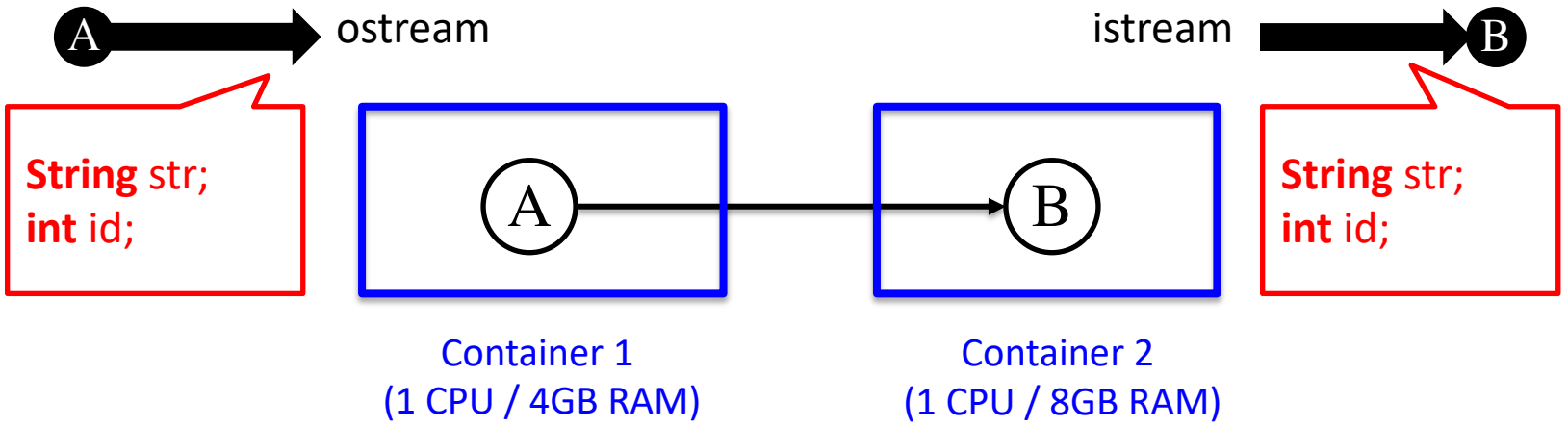
- **Express your parallelism in the right way**
  - A “hard-coded” distributed timing analysis framework
- **Boost your productivity in writing parallel code**
  - DtCraft system
- **Leverage your time to produce promising results**
  - **Vanilla examples**
  - **Machine learning**
  - **Graph algorithms**
  - **Distributed timing**

Write a DtCraft Application

- ❑ Step 1: Decide the stream graph of your application
- ❑ Step 2: Specify the data to stream between vertices
- ❑ Step 3: Define the stream computation callback
- ❑ Step 4: Attach resources on vertices (optional)
- ❑ Step 5: Submit

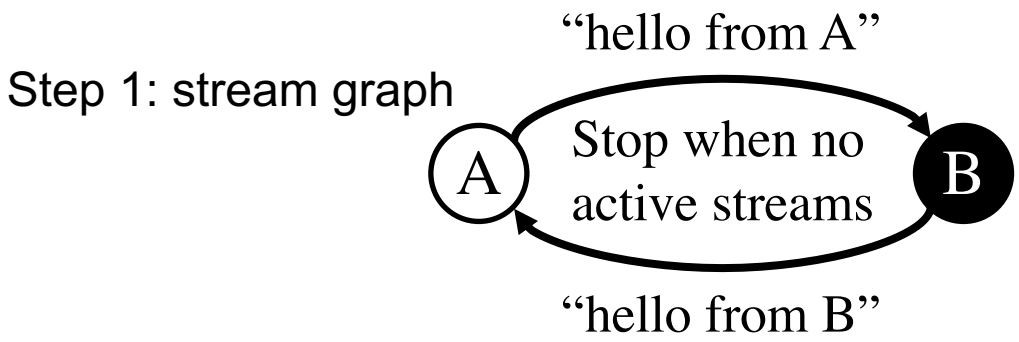
./submit –master=host hello-world

```
auto L = [=] (auto& vertex, auto& istream) {
  if(string data; istream(data) != -1) {
    // Your program control flow.
  } else { ... }
};
```



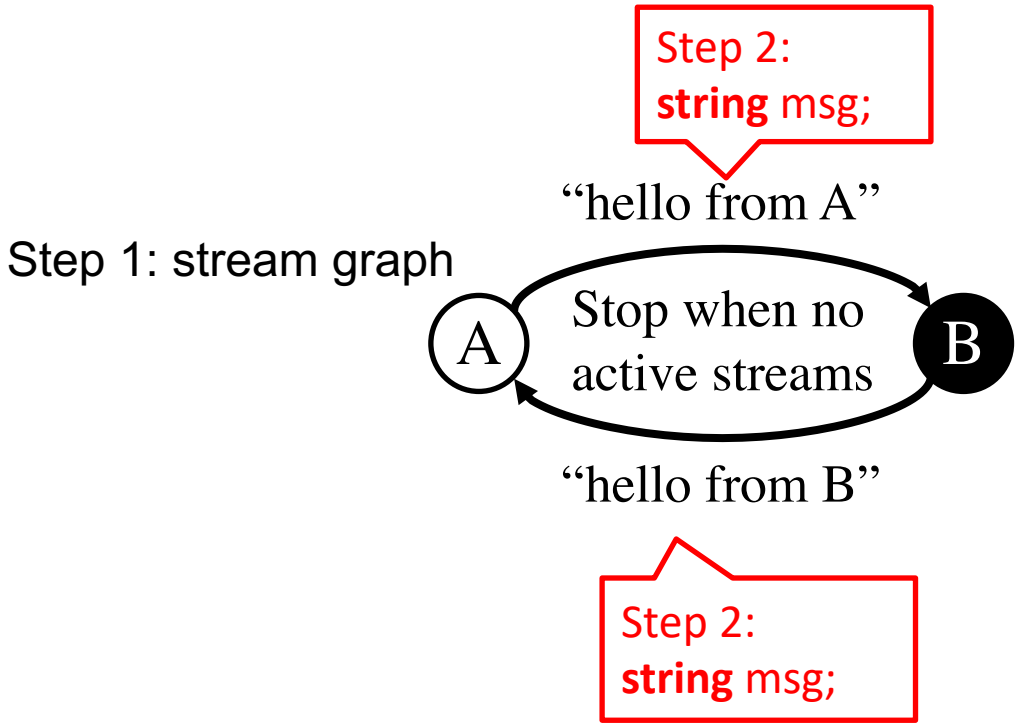
A Vanilla Example

- **A cycle of two vertices and two streams**
  - Each vertex sends a hello message to the other
  - Closes the underlying stream channel



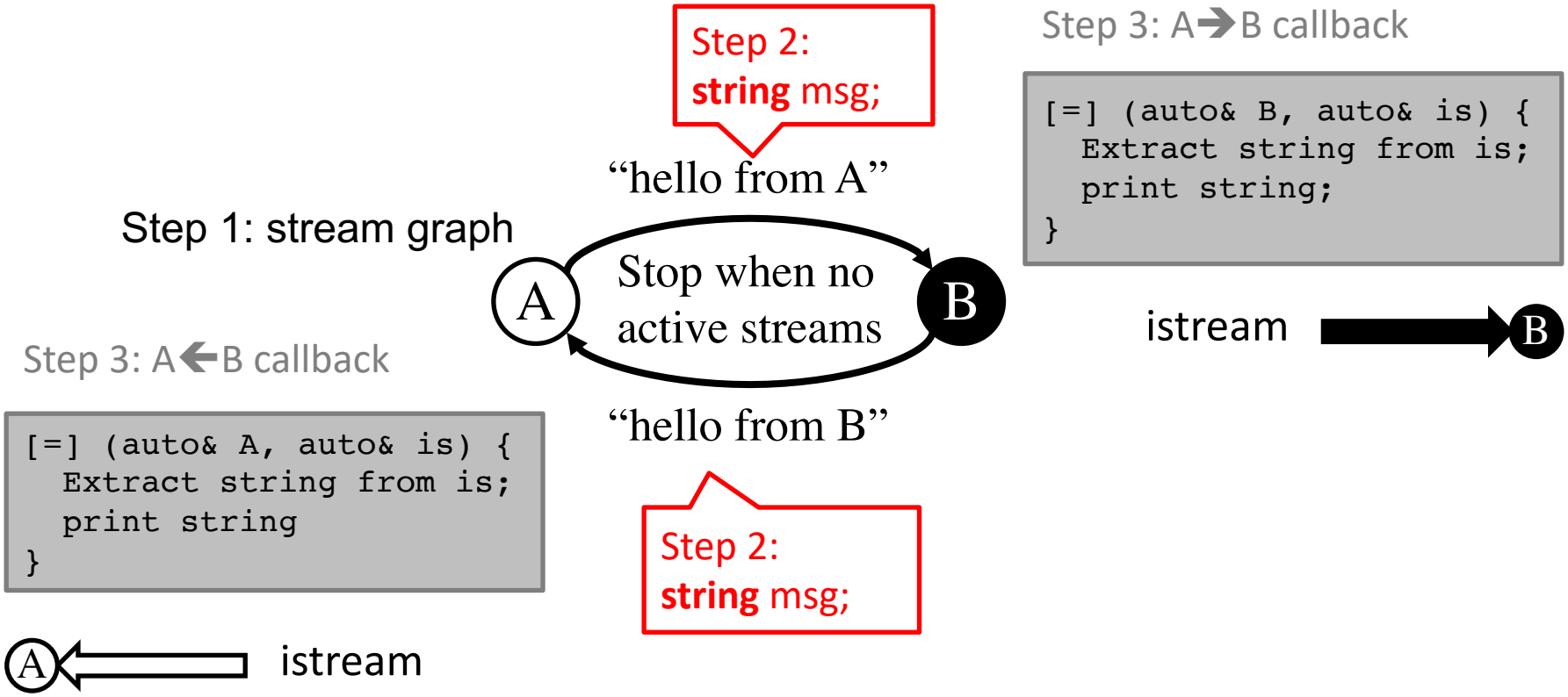
A Vanilla Example

- **A cycle of two vertices and two streams**
  - Each vertex sends a hello message to the other
  - Closes the underlying stream channel



A Vanilla Example

- **A cycle of two vertices and two streams**
  - Each vertex sends a hello message to the other
  - Closes the underlying stream channel



A Vanilla Example

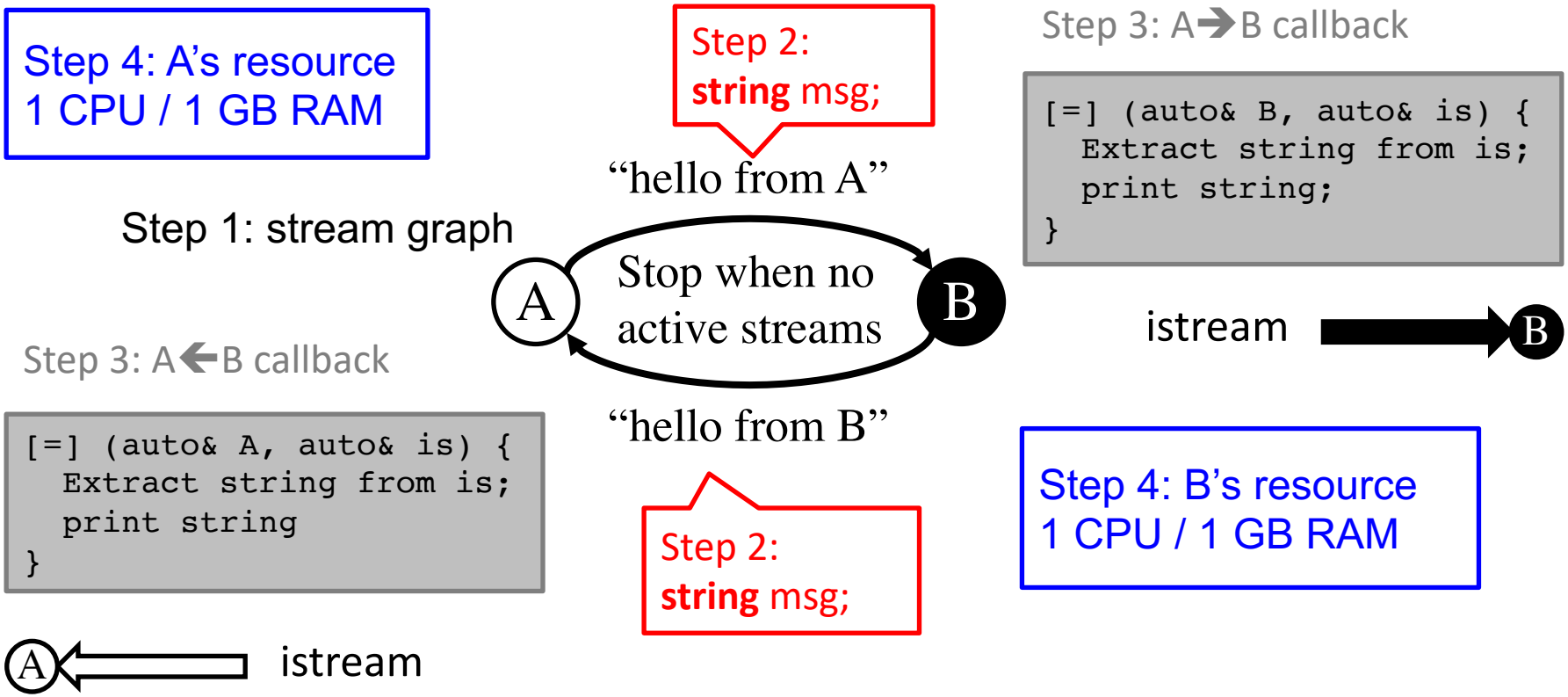
- **A cycle of two vertices and two streams**
  - Each vertex sends a hello message to the other
  - Closes the underlying stream channel

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;
}
```



A Vanilla Example

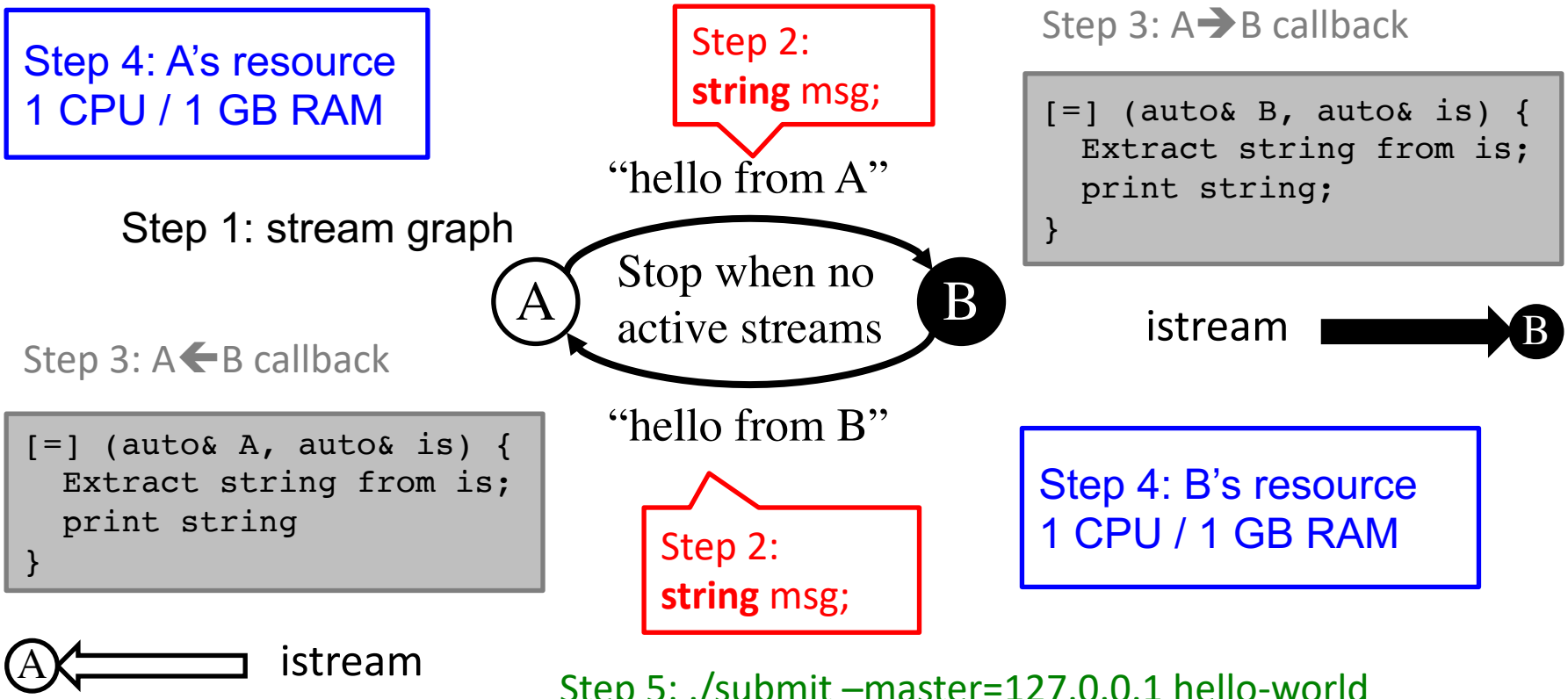
- **A cycle of two vertices and two streams**
  - Each vertex sends a hello message to the other
  - Closes the underlying stream channel

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;
}
```



Seeing is Believing

```
dtc::Graph G;

auto A = G.vertex();
auto B = G.vertex();

G.container().add(A).cpu(1).memory(1_GB);

auto AB = G.stream(A, B).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;
  }
);

auto BA = G.stream(B, A);

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");
});

G.container().add(B).cpu(1).memory(1_GB);

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");
});

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();
```

- **Only a couple lines of code**
- Single sequential program
- Distributed across computers
- No explicit data management
- Easy-to-use streaming interface
- Asynchronous by default
- Scalable to many threads
- Scalable to many machines
- In-context resource controls
- Transparent concurrency controls
- Robust runtime via Linux container

… and more

Distributed Hello-world 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!

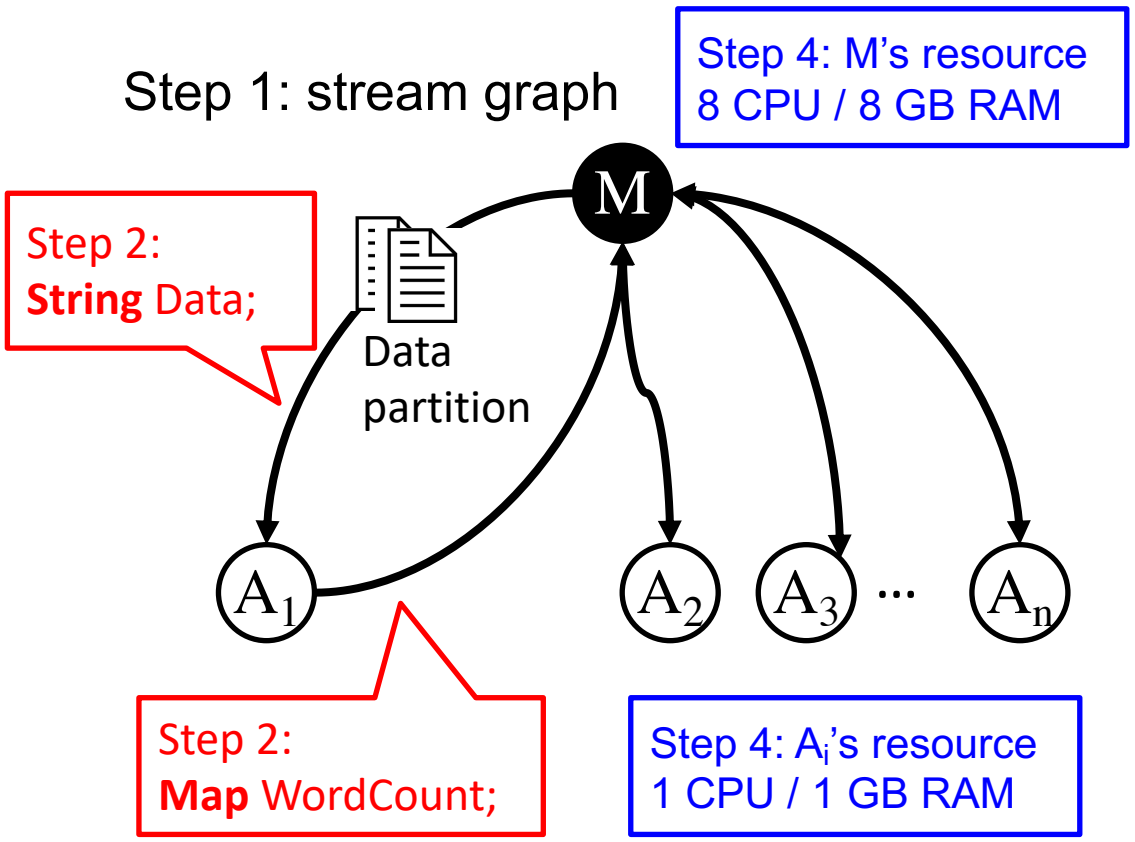
simple.cpp → server.cpp + client.cpp

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

  struct addr
  struct addr
  struct addr

  std::memset
  hints.ai_fa
  hints.ai_so
  hints.ai_pr
  hints.ai_fl

  int one {1}
  int ret;

  if((ret = :
    errc = ma
    return -1
  }

  // Try to c
  for(ptr = r

    // Ignore
    if(ptr->a
      goto tr
    }

    if((fd =
      errc =
      goto tr
    }

    ::setsock

    if(::bind
      errc =
      goto tr
    }

    if(::list
      errc =
      goto tr
    }
    else {
      break;
    }

    try_next:

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

  ::freea

  // Assi
  return
}

int make_
  std::si
  std::st
  std::er
) noexce

  errc.cl

  struct
  struct

  std::me
  hints.a
  hints.a
  hints.a

  int ret
  int fd
  int tri

  if((ret
    errc
    retur
  }

  // Try
  for(aut

    // Ig
    if(pt
      got
    }

    if((f
      er
      got
    }

    make_

        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 for this trivial distributed program…

MapReduce Flow Example

- **Word counting**
  - Find the frequency of each word in a data set



Step 5: ./submit –master=127.0.0.1 word-count

Step 3: A_i ← M callback

```
[=] (auto& Ai, auto& is) {
  Extract data from is;
  Count words in data;
  Store results in a map;
  Send the map to M;
}
```

A ⟵ istream

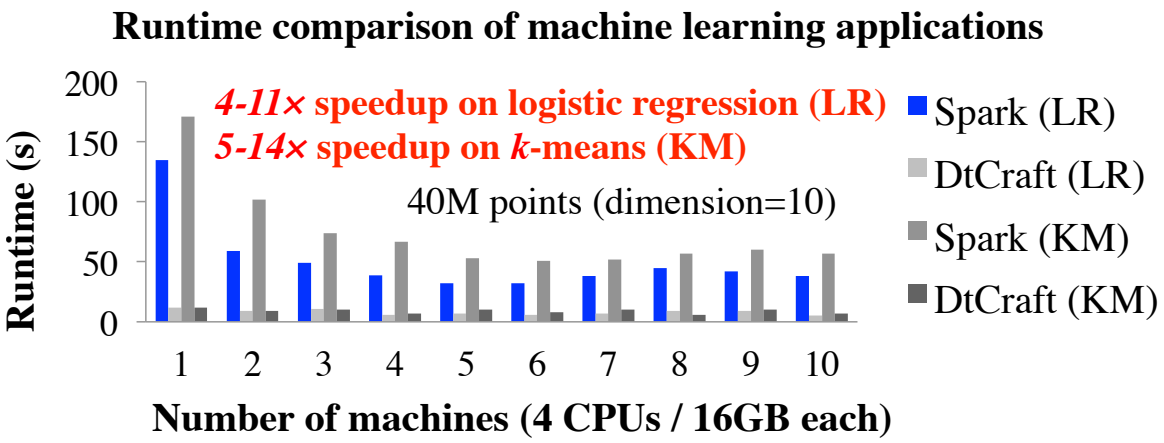
Step 3: A_i → M callback

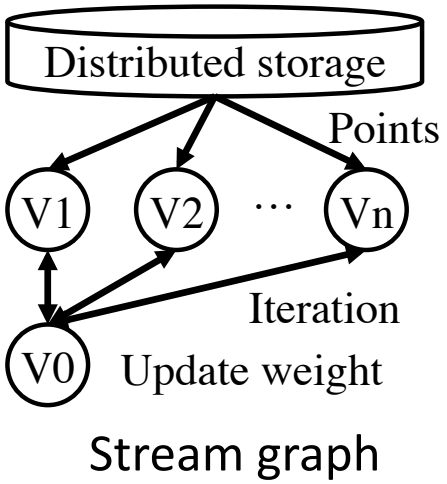
```
[=] (auto& M, auto& is) {
  Extract map from is;
  Reduce sum on map;
  if all done: print map;
  else: wait;
}
```

istream ⟶ M

Micro-benchmark: Machine Learning

- **Logistic regression and *k*-means learning**
  - MapReduce workload with 10 iterations
- **Compared with Spark 2.0 MLib**
  - Reduced dataflow size by 10x
  - Achieved up to 15x speed-up and 2-5x less memory
  - Required zero overhead to cache data for reuse

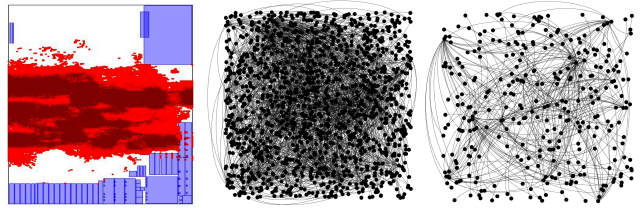


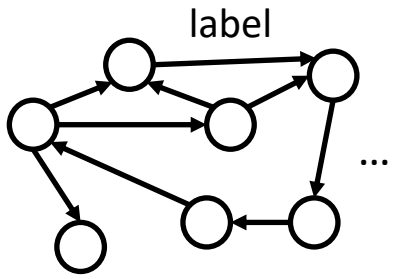


Stream graph

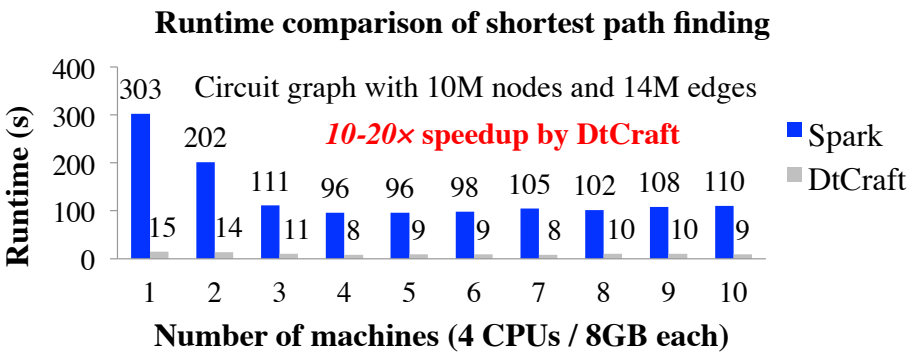
Micro-benchmark: Graph Algorithms

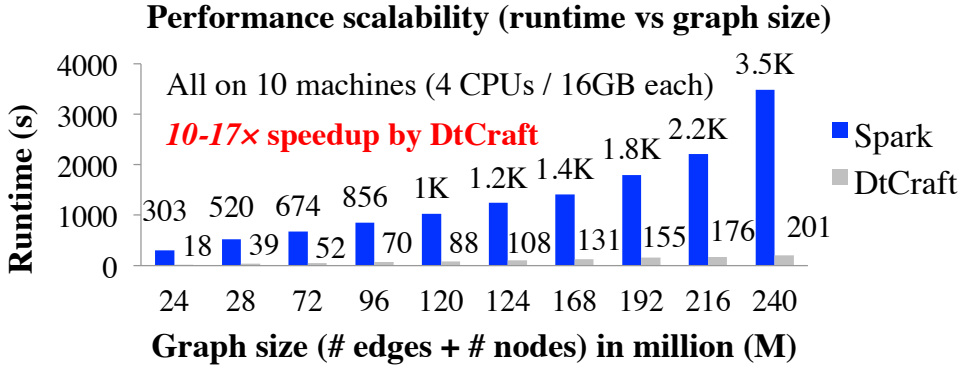
- **Shortest path finding**
  - Million-scale circuit graphs
  - Bellman-Ford algorithms
- **Compared with Spark GraphX (Pregel)**
  - Less synchronization overhead
  - Less communication cost
  - Achieved 10-20x speed-up and scalability


(a) Circuit (1.01mm²) (b) Graph (3M gates) (c) A signal path



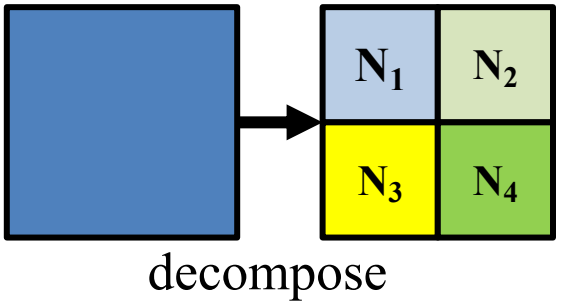
Stream graph

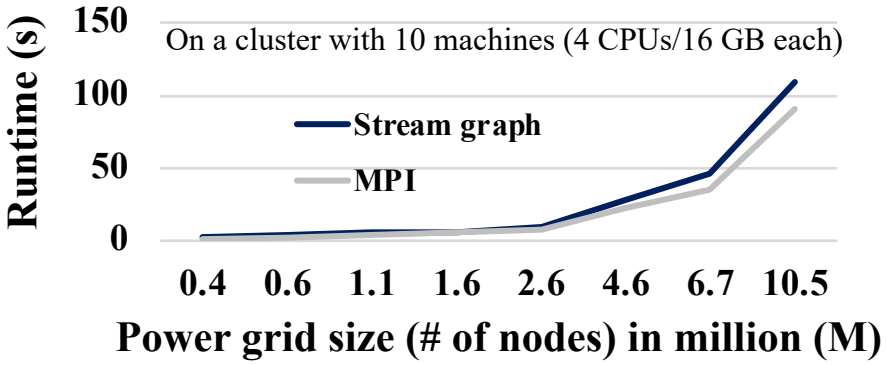


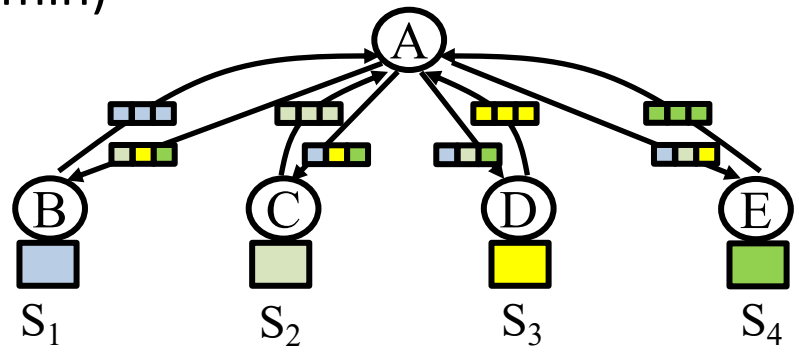


Distributed Power-grid Analysis

- **Power-grid analysis algorithm**
  - Solve a linear system *GV=I*
  - Domain decomposition method
- **Compared with MPI-based method**
  - Flexible partition scheme without CPU count constraint
  - Reduced coding efforts by 10x
  - Similar performance martin (< 1 min)





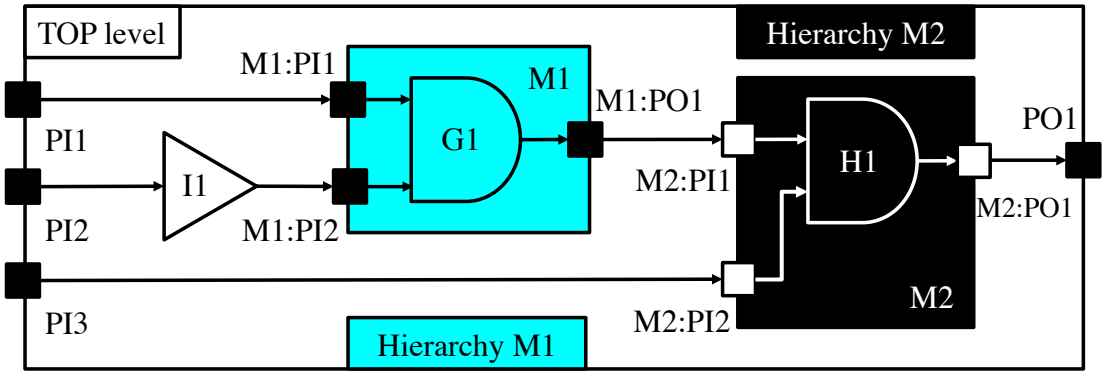


Stream graph

C.-X Lin, T.-W. Huang, and M. Wong., "A Distributed Power-grid Analysis Framework," ACM GLSVLSI

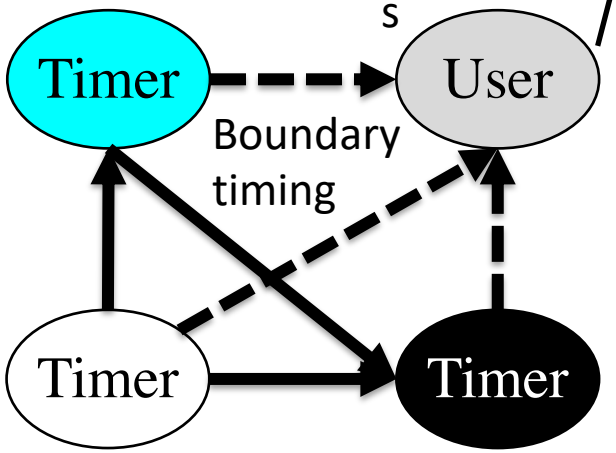
Distributed Timing Analysis

- Two-level hierarchical design (three partitions)



- Top-level
- M1
- M2

Timing commands



API

```
report_at
report_slew
report_rat
remove_gate
insert_gate
power_gate
insert_net
connect_pin
...
```

Optimization program

- Three timer vertices
- One user vertex
- Four Linux containers
- Six input/output streams

Each container has one OpenTimer operating on one design hierarchy

Exchange Timing Data – Delay, Slew, etc.

DtCraft

```
// Timing data (early/late rise/fall)
struct Timing {

  string pin;
  array<float, 4> value;

  template <typename T>
  auto archive(T& ar) {
    return ar(pin, value);
  }
};

// Timing path
struct Path {

  float slack;
  vector<string> pins;

  template <typename T>
  auto archive(T& ar) {
    return ar(slack, pins);
  }
};

// Exchange timing through DtCraft stream
stream.on(
  [](Vertex& v, InputStream& is) {
    if(Timing timing; is(timing) != -1) {
      // Call OpenTimer to run incremental timing
    }
  }
);
```

In-context streaming with < 30 lines

Existing framework*

Hard-code your message format

```
// OpenTimer.proto
package OpenTimer;

// Message format for timing
message Timing {
  required string pin = 0;
  required float er = 1;
  required float ef = 2;
  required float lr = 3;
  required float lf = 4;
}

// Message format for path
message Path {
  required float slack = 0;
  repeated string pins = 1;
}
```

Many extra stuff ☹

Extra.pb.h
Extra.pb.cpp
…
Source.cpp

Google Protocol Buffer (open-source compiler)

C++/Java/Python source code generator

.cpp/.h class methods
ParseFromArray(void*, size_t)
SerializeToArray(void*, size_t)

Message wrapper

Derived packet struct
header_t header
void* buffer

Out-of-context streaming takes > 300 lines

* *Huang et al., "A Distributed Timing Analysis Framework for Large Designs," IEEE/ACM DAC16*

Deploy the Distributed Timer in One Line

DtCraft

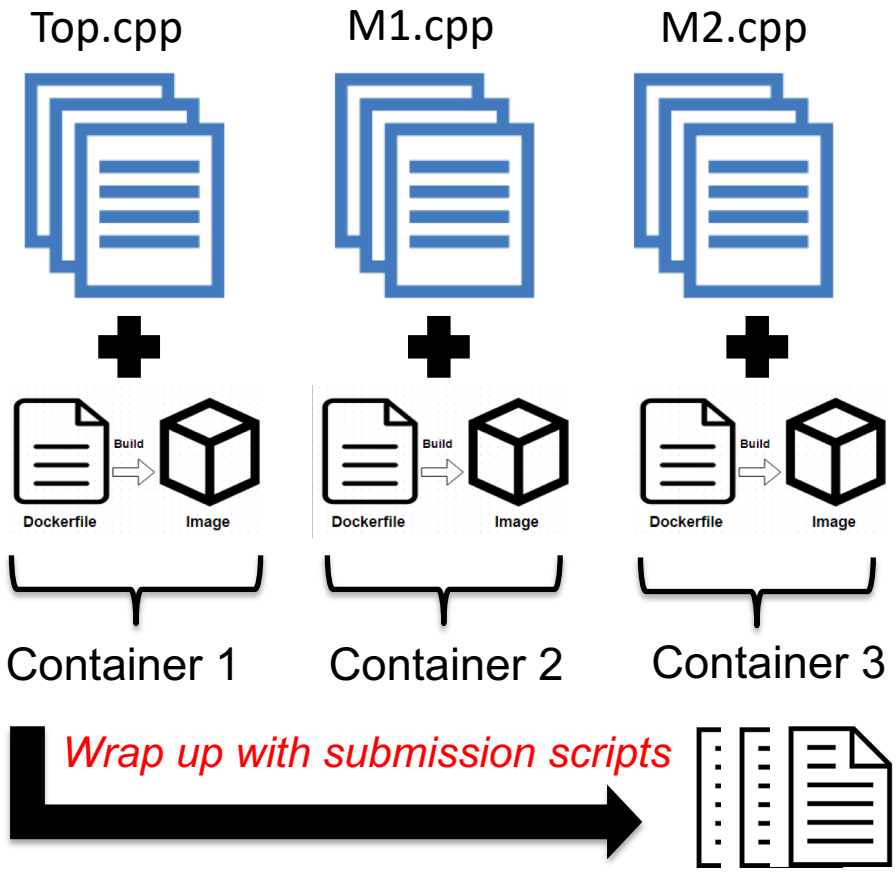
```
// Create a timer vertex for Top
auto Top = G.Vertex().on(
  [=] () {
    OpenTimer timer ("Top.v");
  }
);

// Create a timer vertex for Macro 1
auto M1 = G.Vertex().on(
  [=] () {
    OpenTimer timer ("M1.v");
  }
);

// Create a timer vertex for Macro 2
auto M1 = G.Vertex().on(
  [=] () {
    OpenTimer timer ("M2.v");
  }
);

// Create streams ...
...

// Distribute timers to machines.
G.container().add(Top).num_cpus(4).memory_(4_GB);
G.container().add(M1).num_cpus(1).memory(8_GB);
G.container().add(M2).num_cpus(2).memory(6_GB);
```

Only three lines for resource control in Linux container

~$ ./submit –master=127.0.0.1 binary

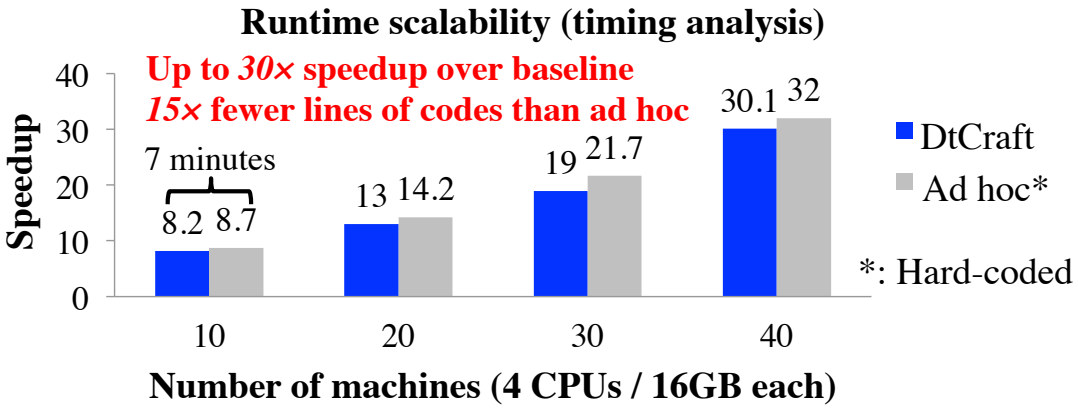
Existing framework

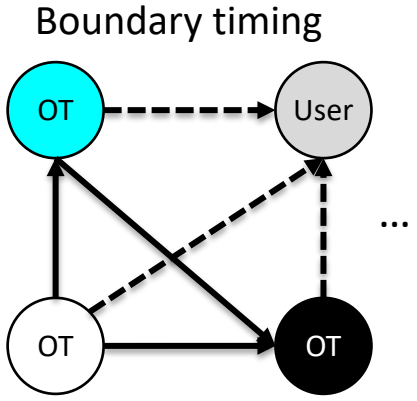
Duplicate the code for each partition



Comparison with Hard-coded Method

- **Static timing analysis under multiple scenarios**
  - One OpenTimer (OT) per scenario
  - Ran on 40 Amazon EC2 nodes
- **Compared with hard-coded implementation***
  - Saved 15x lines of code
  - Only 7-11% performance loss



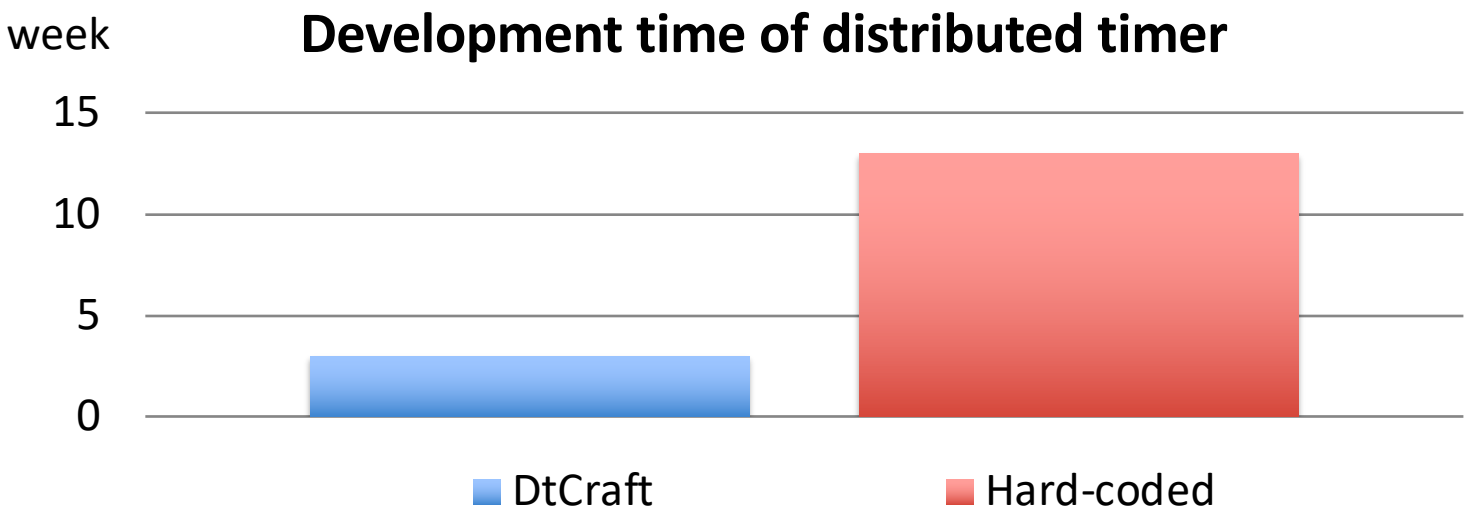


Stream graph
(250 IBM design partitions)

* *Huang et al., "A Distributed Timing Analysis Framework for Large Designs," IEEE/ACM DAC16*

The Productivity Gain is TREMENDOUS

*"With DtCraft, it took me only three weeks, precisely, the **SPARE time** out of my 2017 summer internship at Citadel HPC group, to build a distributed timer that took my **whole 2015 summer** with IBM EDA group".*



Conclusion

- **Express your parallelism in the right way**
  - A “hard-coded” distributed timing analysis framework
- **Boost your productivity in writing parallel code**
  - DtCraft system
- **Leverage your time to produce promising results**
  - Vanilla examples
  - Machine learning
  - Graph algorithms
  - Distributed timing

Thank you!

Dr. Tsung-Wei Huang

twh760812@gmail.com

Github: https://github.com/twhuang-uiuc

Website: http://web.engr.illinois.edu/~thuang19/