DtCraft: A Distributed Execution Engine for Compute-intensive Applications

Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, IL, USA

DtCraft
Parallel computing

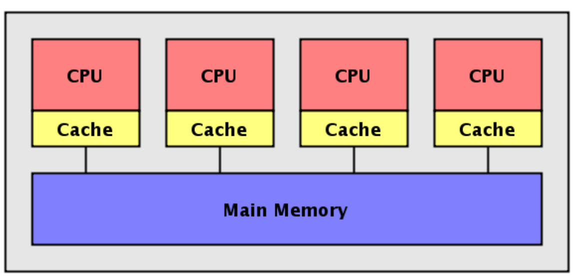
Boost your productivity in writing parallel code!

Agenda

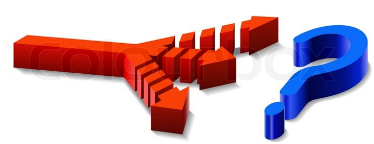
- ❑ **Express your parallelism in the right way**
- ❑ **Boost your productivity in writing parallel code**
- ❑ **Leverage your time to produce promising results**

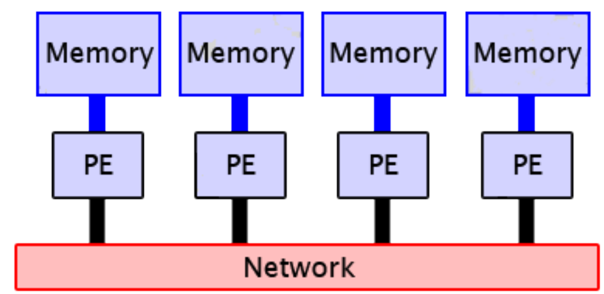
Motivation

- “*We need a new parallel/distributed timing analysis method to deal with the large design complexities,*” **IBM Timing Group, Fishkill, NY, 2015**
  - Explore a feasible framework
  - Prototype a distributed timer
  - Scale to billions of transistors



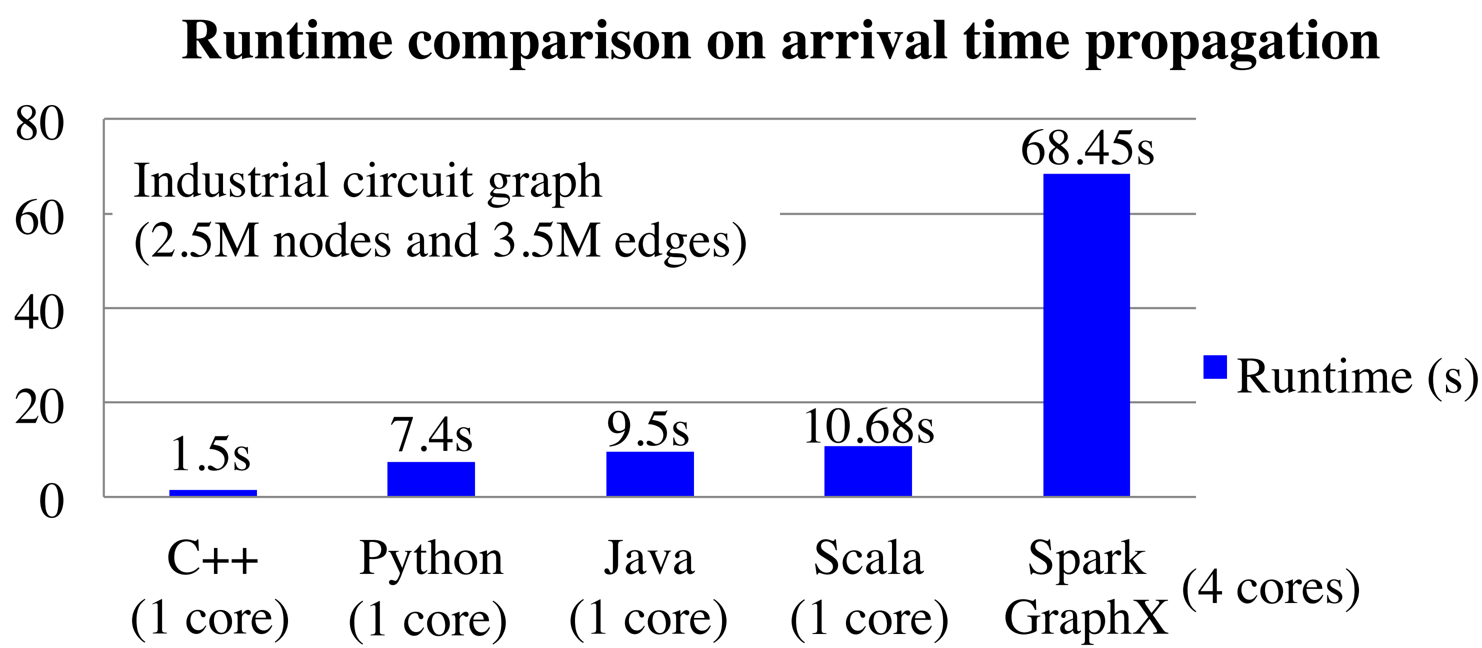
Multi-threaded timer





Distributed timer

Big data is NOT an easy fit in EDA!

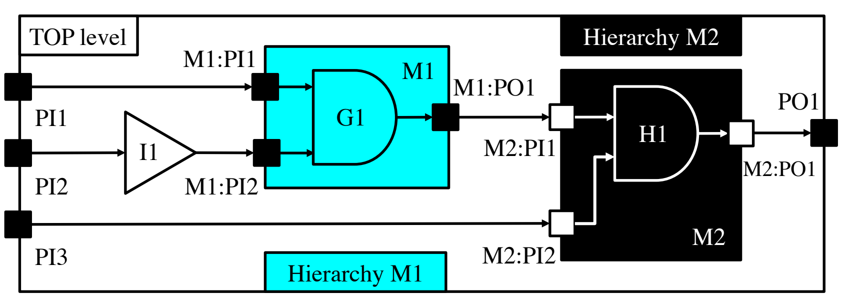


Method	Spark 1.4 (RDD + GraphX Pregel)	Java (SP)	C++ (SP)
Runtime (s)	68.45	9.5	**1.50**

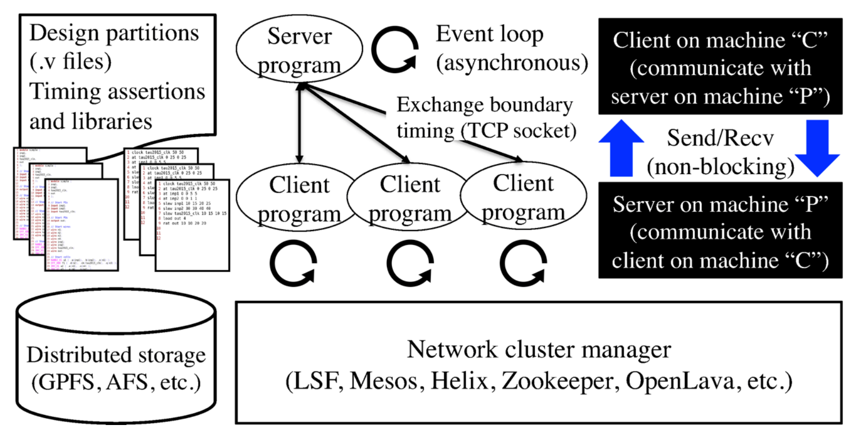
A "hard-coded" distributed timer

- ❑ **General design partitions**
  - ❑ Logical, physical, or hierarchical partitions
  - ❑ Design data are stored in a shared storage (e.g., NFS, GPFS)
- ❑ **Single-server multiple-client model**
  - ❑ Server is the centralized coordinator
  - ❑ Clients exchange boundary timing with server

❑ Non-blocking IO
❑ Event-driven programming
❑ Serialization/Deserialization



Three partitions, top-level, M1, and M2 (given by design teams)



Observations

- **Big-data is not an easy fit in EDA**
  - IO-bound *vs* CPU-bound
  - Unstructured *vs* Structured
  - JVM *vs* C/C++
- **Hard-coded method is error-prone and not scalable**
  - Expose to the low-level socket message passing
  - Move data between compute nodes' memories
  - Manage the cluster resource by yourselves
  - Difficult to maintain between software generations
  - Cause you a significant amount of coding efforts
- **Want parallel programming *at scale* far more *productive***
  - Better productivity means better performance for most people

What does "Productivity" mean to you?

- **Programming language**
  - "I use Python/Matlab/Scala to prototype my project"
- **Transparency**
  - "I use Hadoop/Spark to express my parallel computations without understanding architecture-specific details"
- **Performance**
  - "I use C/C++/Fortran/MPI to ensure full control over resources to achieve the best CPU and memory performance"
- **DtCraft project**
  - "We let *less-experienced* users express their parallel computing workload without taking away the control over system details to achieve *high performance*, using our *groovy* API written in modern C++17"

Why is being "Productive" important?

- ❑ **Code costs are more than machine costs**
  - ❑ Hardware is a commodity resource
  - ❑ Coding takes people and time
- ❑ **I hate writing boilerplate code**
  - ❑ Redundant steps to write parallel code

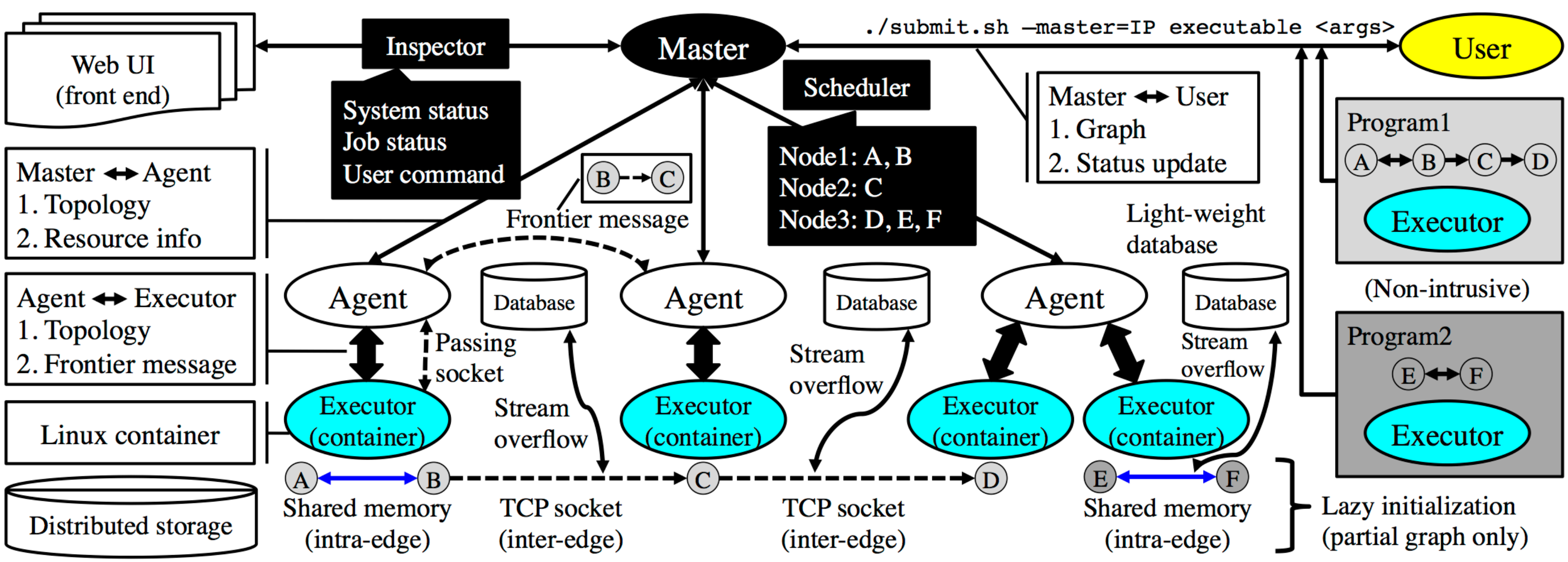
```
// C++ thread example
std::thread t1([=](){ /* do something */ });
std::thread t2([=](){ /* do another thing */ });
t1.join();  // release thread 1 resource
t2.join();  // release thread 2 resource
```

  - ❑ Code becomes massy when data dependencies exist
- ❑ **We want computationally productive code**
  - ❑ The cloud businesses reduce the hardware factor
  - ❑ Everything must be parallel moving forward

2016 average software engineer salary > 100K USD

DtCraft – A distributed execution engine

- ❑ Modernize yourself with C++17
- ❑ Express your workload in our groovy API
- ❑ Stay away from difficult concurrency controls
- ❑ Make the most use of cluster resources
- ❑ Gain security and reliability with Linux container



Stream graph programming model

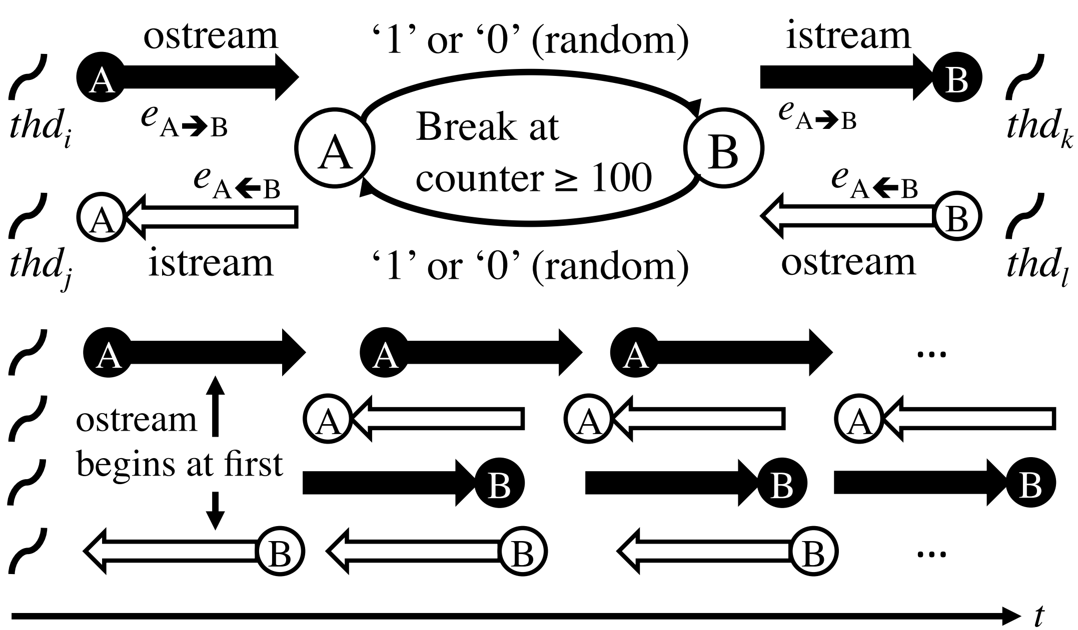
```
class Stream {
  weak_ptr<OutputStream> ostream();
  weak_ptr<InputStream>  istream();
  function<Signal(Vertex&, OutputStream&)> on_ostream;
  function<Signal(Vertex&, InputStream&)> on_istream;
};

class Vertex {
  shared_ptr<OutputStream> ostream(key_type) const;
  shared_ptr<InputStream> istream(key_type) const;
  function<void(Vertex&)> on;
  unordered_map<key_type, Stream*> streams;
};

class Graph {
  VertexDescriptor vertex();
  StreamDescriptor stream(key_type, key_type);
  ContainerDescriptor container();
};

class Executor : public Reactor {
  Executor(Graph&);
};
```

Only a single executable is required to enable distributed execution!

- **Graph**
  - Vertex and stream creation
  - Resource assignment
- **Vertex**
  - One-time callback
  - Access adjacent streams
- **Stream**
  - Level-triggered I/O callback
  - Close stream on return
- **Executor**
  - Submit your graph
  - Debug your graph
  - Execute your graph

A concurrent ping-pong example

- ❑ **A representative workload in parallel computing**
  - ❑ Message passing back and forth concurrently
  - ❑ A fundamental building block of incremental flow



Method	Parallelism
C++17 thread	Local/ Distributed
MPI	Distributed

Method to be compared with DtCraft

C++ thread on a local machine

❑ Standard C++ thread coding doesn't scale easily

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

// Boilerplate code to open file handles
int fds[2] = {-1, -1};
auto ret = socketpair(
  AF_UNIX,
  SOCK_NOBLOCK | SOCK_CLOEXEC,
  SOCK_STREAMfd,
  fds
 );

if(ret == -1) {
  throw system_error("Failed on socketpair");
}

// 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;
}
```

```
// Create a ping thread
thread t1 (
  [=] () {
    do {
      pingpong(fds[0], count_A);
    }while(count_A < 100);
  }
);

// Create a pong thread
std::thread t2 (
  [=] () {
    do {
      pingpong(fds[1], count_B);
    }while(count_B < 100);
  }
);

// Join the threads
t1.join();
t2.join();
```

Amount of code grows with thread count and problem size!

C++ thread on distributed machines

❑ Things become massy going distributed …

```
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;

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

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

server.cpp

client.cpp

```
// Create a ping thread
thread t1 (
  [=] () {
    do {
      pingpong(fd, count_A);
    }while(count_A < 100);
  }
);

// Create a pong thread
std::thread t2 (
  [=] () {
    do {
      pingpong(fds[1], count_B);
    }while(count_B < 100);
  }
);

t1.join();
t2.join();
```

Branch your code to server and client for distributed computation!

simple.cpp → server.cpp + client.cpp

(explicit and manual message passing)

Uh… you wonder how they look?

- ❑ `make_socket_server_fd` **and** `make_socket_client_fd`

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

  struct addrinfo hints;
  struct addrinfo* res {nullptr};
  struct addrinfo* ptr {nullptr};

  std::memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_protocol = IPPROTO_TCP;
  hints.ai_flags = AI_PASSIVE;          // let it fill

  int one {1};
  int ret;

  if((ret = ::getaddrinfo(nullptr, port.data(), &hints
    errc = make_posix_error_code(ret);
    return -1;
  }

  // Try to connect to the first one that is available
  for(ptr = res; ptr != nullptr; ptr = ptr->ai_next) {

    // Ignore undefined ip type.
    if(ptr->ai_family != AF_INET && ptr->ai_family != 
      goto try_next;
    }

    if((fd = ::socket(ptr->ai_family, ptr->ai_socktype
      errc = make_posix_error_code(errno);
      goto try_next;
    }

    ::setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &one, s

    if(::bind(fd, ptr->ai_addr, ptr->ai_addrlen) == -1
      errc = make_posix_error_code(errno);
      goto try_next;
    }

    if(::listen(fd, 128) == -1) {
      errc = make_posix_error_code(errno);
      goto try_next;
    }
    else {
      break;
    }

    try_next:

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

  }

  ::freeaddrinfo(res);

  // Assign the socket to the underlying event native handle.
  return fd;
}


int make_socket_client_fd(
  std::string_view H,
  std::string_view P,
  std::error_code& errc
) noexcept {

  errc.clear();

  struct addrinfo hints;
  struct addrinfo* res {nullptr};

  std::memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_protocol = IPPROTO_TCP;

  int ret;
  int fd {-1};
  int tries;

  if((ret = ::getaddrinfo(H.data(), P.data(), &hints, &res)) != 0
    errc = make_posix_error_code(ret);
    return -1;
  }

  // Try each internet entry.
  for(auto ptr = res; ptr != nullptr; ptr = ptr->ai_next) {

    // Ignore undefined ip type.
    if(ptr->ai_family != AF_INET && ptr->ai_family != AF_INET6) {
      goto try_next;
    }

    if((fd = ::socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_p
      errc = make_posix_error_code(errno);
      goto try_next;
    }

    make_fd_nonblocking(fd);
    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 conne
    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;
}
```

Actually more than the parallel code you need…

Massage Passing Interface (MPI)

- ❑ **Explicitly move EVERYTHING between compute nodes**

```c
#define  MAX_LEN 1 << 18    /* maximum vector length    */
#define  TRIALS  100        /* trials for each msg length   */
#define  PROC_0  0      /* processor 0          */
#define  B0_TYPE 176        /* message "types"      */
#define  B1_TYPE 177


int numprocs, p,        /* number of processors, proc index */
myid,           /* this processor's "rank"      */
length,         /* vector length            */
i, t;

double b0[MAX_LEN], b1[MAX_LEN];    /* vectors          */

double start_time, end_time;    /* "wallclock" times        */

MPI_Status stat;        /* MPI structure containing return  */
            /* codes for message passing operations */

MPI_Request send_handle, recv_handle;   /* For nonblocking msgs */

MPI_Init(&argc,&argv);          /* initialize MPI   */

MPI_Comm_size(MPI_COMM_WORLD, &numprocs);   /*how many processors?  */
MPI_Comm_rank(MPI_COMM_WORLD, &myid);   /*which one am I?   */

if (myid == PROC_0)
{
  srand48(0xFEEDFACE);

/* generate processor 0's vector */

for (i=0; i<MAX_LEN; ++i)
    b0[i] = (double) drand48();
}

MPI_Barrier(MPI_COMM_WORLD);
```

Hard-coded message passing

```c
for (length=1; length<=MAX_LEN; length*=2)
{
MPI_Barrier(MPI_COMM_WORLD);

if (myid == PROC_0)
    start_time = MPI_Wtime();

for (t=0; t<TRIALS; ++t)
{
#ifdef BLOCKING
    if (myid == PROC_0)
    {
        MPI_Send(b0, length, MPI_DOUBLE, 1,
                 B1_TYPE, MPI_COMM_WORLD);
        MPI_Recv(b1, length, MPI_DOUBLE, 1,
                 B0_TYPE, MPI_COMM_WORLD, &stat);
    }
    else
    {
        MPI_Recv(b1, length, MPI_DOUBLE, 0,
                 B1_TYPE, MPI_COMM_WORLD, &stat);
        MPI_Send(b0, length, MPI_DOUBLE, 0,
                 B0_TYPE, MPI_COMM_WORLD);
    }
#else
    MPI_Isend(b0, length, MPI_DOUBLE, (myid+1)%numprocs,
              B0_TYPE, MPI_COMM_WORLD, &send_handle);
    MPI_Irecv(b1, length, MPI_DOUBLE, (myid+1)%numprocs,
              B0_TYPE, MPI_COMM_WORLD, &recv_handle);

    MPI_Wait(&send_handle, &stat);
    MPI_Wait(&recv_handle, &stat);
#endif
}

MPI_Finalize();
```

DEADLOCK

It's user's fault to introduce deadlock

Concurrent ping-pong with DtCraft

```
auto Ball(Vertex& v, auto& k) {
  (*v.ostreams(k))((rand()%2));
  return Stream::DEFAULT;
};

auto PingPong(auto& v, auto& is, auto& k, auto& c) {
  int data;
  is(data);
  if((c+=data) >= 100) return Stream::REMOVE_THIS;
  else return Ball(v, k)
}

key_type AB, BA;
auto count_A {0}, count_B {0};

Graph G;
auto A = G.vertex().on([&](auto& v){ Ball(v, AB); });
auto B = G.vertex().on([&](auto& v){ Ball(v, BA); });

// Insert an iostream A->B
AB = G.stream(A, B).on(
  [&] (auto& B, auto& istream) {
    return PingPong(v, istream, BA, count_B);
  }
);

// Insert an iostream B->A
BA = G.stream(B, A).on(
  [&] (auto& A, auto& istream) {
    return PingPong(v, istream, AB, count_A);
  }
);

// Resource control using Linux container
G.containerize(A, "memory=1KB", "num_cpus=1");
G.containerize(B, "memory=1KB", "num_cpus=1");

Executor(G).dispatch();
```

- Fewer lines of code overall
- Less boilerplate code
- Single program
- No explicit data management
- Easy-to-use streaming interface
- Asynchronous by default
- Scalable to many threads
- Scalable to many machines
- In-context resource controls
- Scale out to heterogeneous devices
- Transparent concurrency controls
- Robust runtime via Linux container

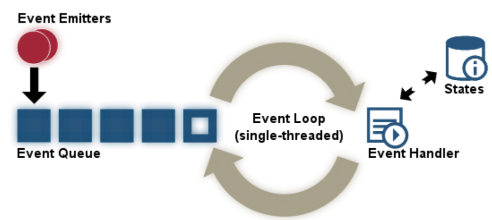
… and more

Be gentle to existing systems

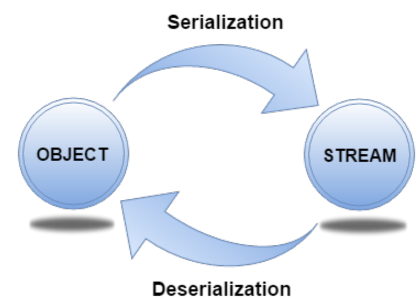
- ❑ **No one can claim their system general**
  - ❑ If yes, I understand it's for business purpose ☺
- ❑ **Big-data tools**
  - ✓ Good for data-driven and MapReduce workload
  - x Bad for CPU/memory-intensive applications
- ❑ **High-performance computing (HPC) language**
  - ✓ Enabled the vast majority of HPC results for 20 years
  - x Suffer from too many distinct notations for parallel programming
  - x Analogous to assembly language (bottom-up design)
- ❑ **DtCraft**
  - ✓ A higher-level alternative to higher-level technologies
  - ✓ Transparent concurrency without taking away low-level controls
  - x Currently best suitable for compute-intensive applications

System implementation of DtCraft

- ❑ **Kernel – *Master*, *Agent*, and *Executor***
  - ❑ Master: global scheduling, deployment, and front-end
  - ❑ Agent: local scheduling, containerization
  - ❑ Executor: task execution (local, distributed, submitted modes)
- ❑ **Event-driven programming environment**
  - ❑ Redesign the reactor library
  - ❑ Thread-safe, lock-free, non-blocking IO



- ❑ **Streaming interface**
  - ❑ Redesign the serialization/deserialization library
  - ❑ Thread-safe, strongly typed, memory efficient



- ❑ **Linux container**
  - ❑ A thin layer of fine-grained resource control
  - ❑ Secure, safe, and robust

A modern reactor library for event-driven programming

❑ The key component to our system kernel

```
// Smart pointer for effective concurrency control.
class Event : enable_shared_from_this <Event> {
  friend class Reactor;
  enum Type {TIMEOUT, PERIODIC, READ, WRITE};
  const function<Signal(Event&)> on;
};

class Reactor {

  // Executing event callback on a shared thread pool
  Threadpool threadpool;
  unordered_set<shared_ptr<Event>> eventset;

  template <typename T, typename... U>
  future<shared_ptr<T>> insert(U&&... u) {
    auto e = make_shared<T>(forward<U>(u)...);
    return promise(
      [&, e=move(e)](){
        _insert(e); // insert an event into reactor
        return e;
      }
    );
  }
};
```

- Written in C++17
- Thread-safe
- Lock-free
- Flattened event type
- Task-based parallelism
- Single producer (promise)
- Multiple consumers (future)
- Smart pointer
- Non-blocking IO controls
- Support multiple back-ends
- Shared thread pool
- Callback in a critical section

A memory-efficient serialization/deserialization library

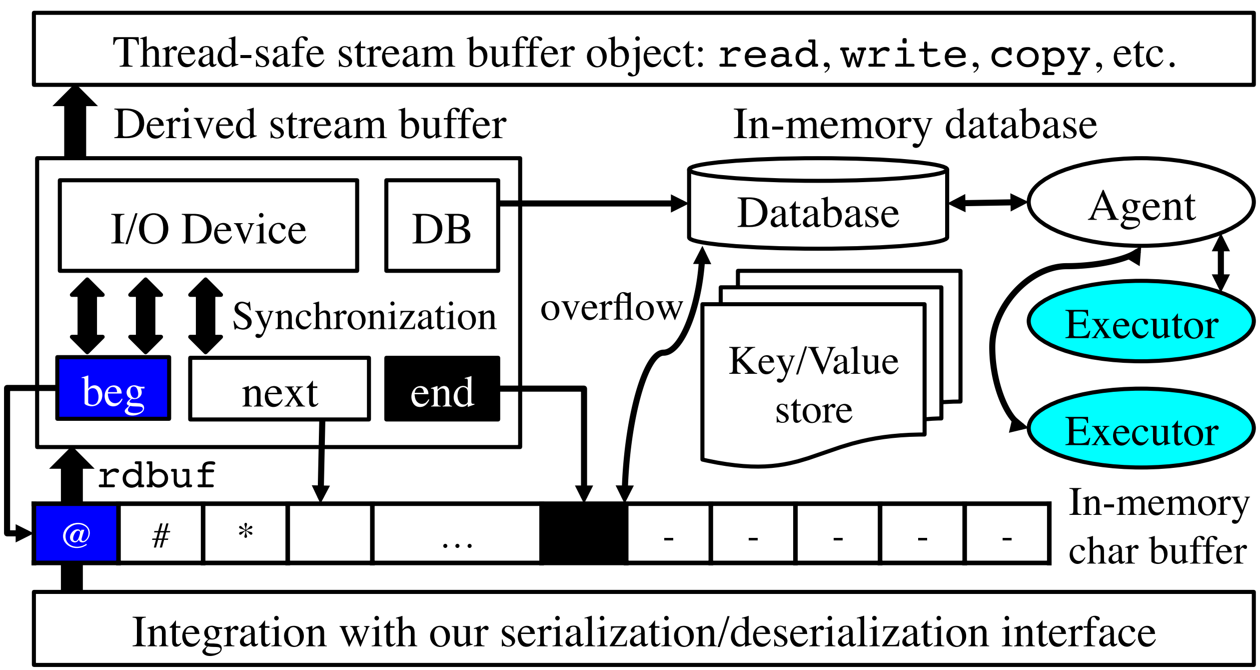
❑ The key component to our message passing

```
struct MyData {
  int raw;
  std::string str;
  std::vector<int> vec;

  // All you need is to include necessary members
  template <typename Archiver>
  auto archive(Archiver& ar) {
    return ar(raw, str, vec);  // we use variadic template
  }
};

class BinaryOutputArchiver {

    OutputStreamBuffer& osbuf;

    template <typename... U>
    constexpr streamsize operator()(U&&... u) {

        if constexpr (is_POD<U>) {
          osbuf.write(&u, sizeof(u));
        }
        else if constexpr (is_std_string<U>) {
          osbuf.write(u.data(), u.size());
        }
        else if constexpr (is_std_vector<U>) {
          // ...
        }
        // ...
        else
          return archive(forward<U>(u)...);
    }
};
```

- Written in C++17
- Heavy meta-programming
- Thread-safe
- Strongly-typed
- Convenient to use
- Integrated with our IO buffer
- Binary data format
- No extra parsing/unpacking
- No secondary representation
- Memory-efficient
- STL ready-to-use

Concurrent input/output stream buffer

- **In charge of reading/writing operations on devices**
  - Work directly with our serialization/deserialization interface
  - Zero copy in user space



- Written in C++17
- Thread-safe
- Recursive lock
- In-memory buffer
- Shared memory
- Network socket
- FIFO
- Domain socket

A Linux container-based resource control

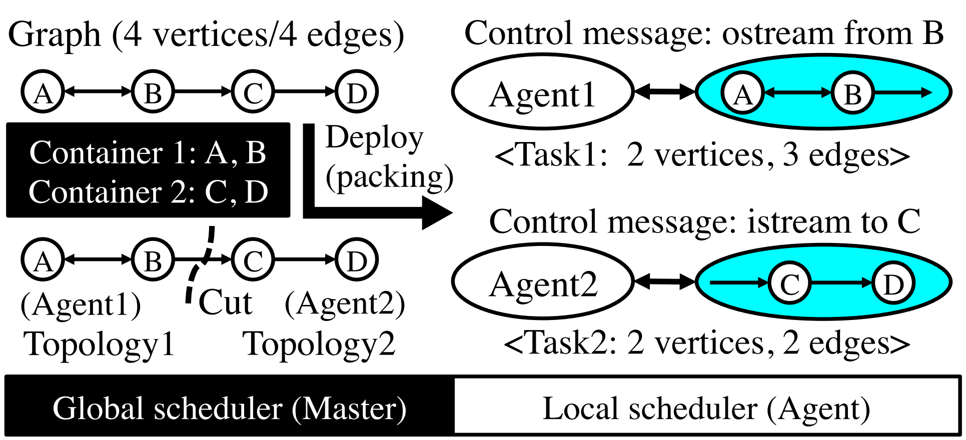
❑ Namespace isolation & resource control

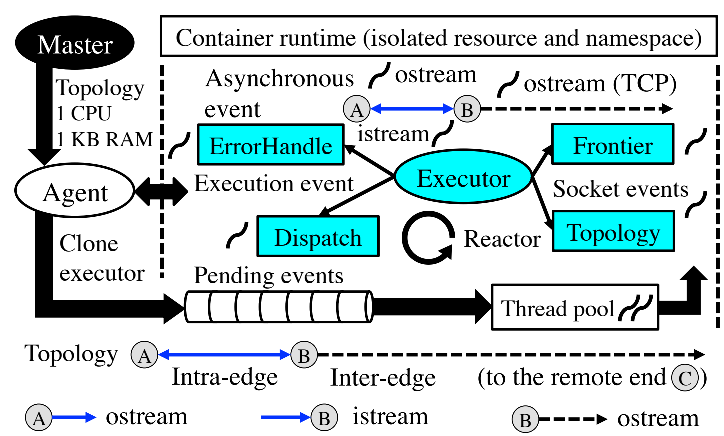
```
// Linux container
class ContainerDescriptor {

  friend class Graph;

  const key_type key;
  operator key_type() const;

  ContainerDescriptor& add(key_type);
  ContainerDescriptor& num_cpus(unsigned);
  ContainerDescriptor& memory_limit_in_bytes(uintmax_t);
  ContainerDescriptor& blkio(uintmax_t);
  // ...
};

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

// Create a container for vertex A with 1 CPU,
// 1 MB memory, and 1 GB block IO
G.container().add(A)
             .num_cpus(1)
             .memory_limit_in_bytes(1_MB)
             .blkio(1_GB);

// Create a container for vertex B with 2 CPU,
// 2 MB memory, and 2 GB block IO
G.container().add(B)
             .num_cpus(2)
             .memory_limit_in_bytes(2_MB)
             .blkio(2_GB);
```

- Safe and robust runtime
- Minimize intruder’s effect
- Network isolation
- UTS isolation
- IPC isolation
- PID isolation
- User/Group isolation
- Cgroup isolation
- Mount point isolation
- In-context resource controls
- Give scheduler hints
- Maximize cluster performance

Graph deployment and workload distribution

- ❑ **Global scheduler – master**
  - ❑ Manage users' graph submissions
  - ❑ Partition graph through bin-packing optimization
- ❑ **Local schedulers – agents**
  - ❑ Fork-exec an executor for each topology
  - ❑ Containerize the executor under resource constraints

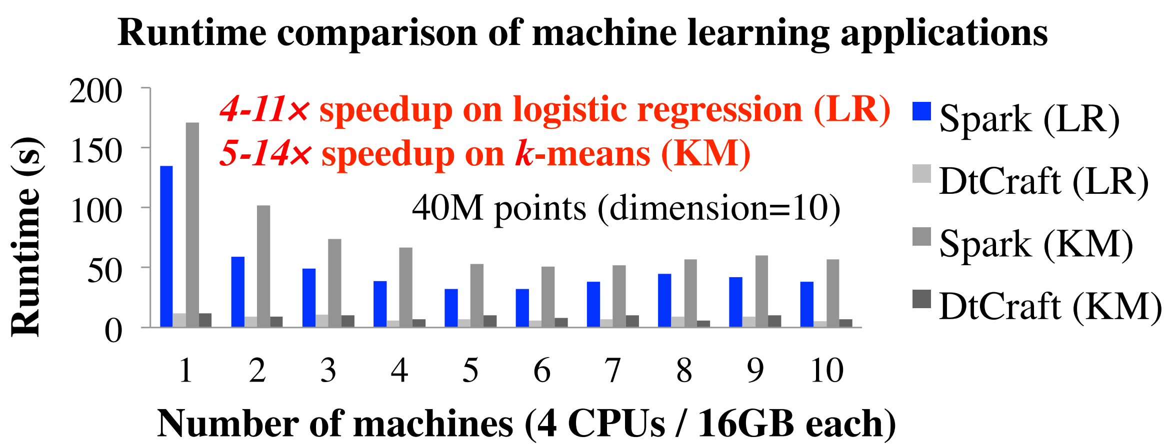


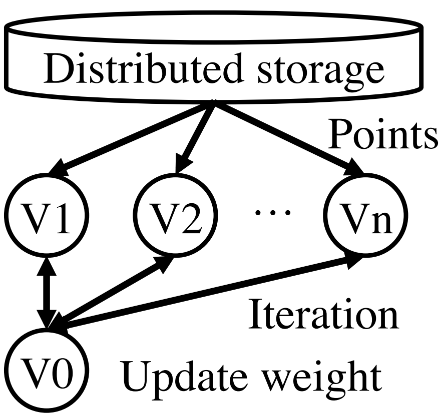


Intra-stream and inter-stream talk through shared memory and TCP socket, respectively

Experiments on machine learning

- **Logistic regression and *k*-means algorithms**
  - Mimic the MapReduce-based flow with ten iterations
- **Compared with Spark 2.0 MLib**
  - More than an order of magnitude faster
  - No extra overhead on the first iteration to cache data
  - Explicit resource controls outperform blind RDD partitions

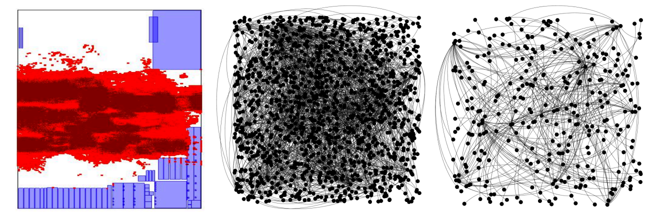




Stream graph

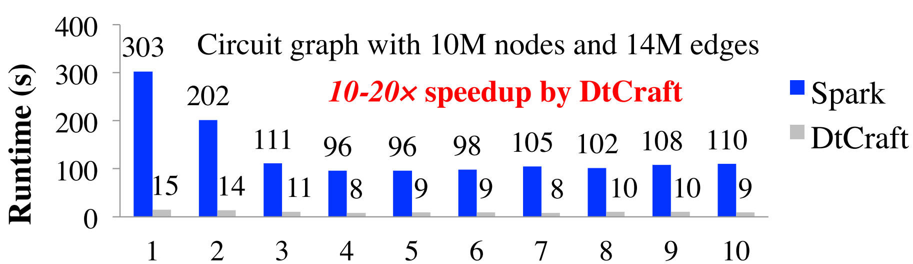
Experiments on graph algorithms

- ❑ **Shortest path algorithm**
  - ❑ Circuit graph with 10M nodes and 14M edges
  - ❑ Higher connectivity than many of big data graphs
  - ❑ Mimic the Pregel-based algorithm (Bellman-Ford style)
- ❑ **Compared with Spark 2.0 GraphX**
  - ❑ Less synchronization overhead
  - ❑ An order of magnitude faster
  - ❑ Scale up as the graph size increases



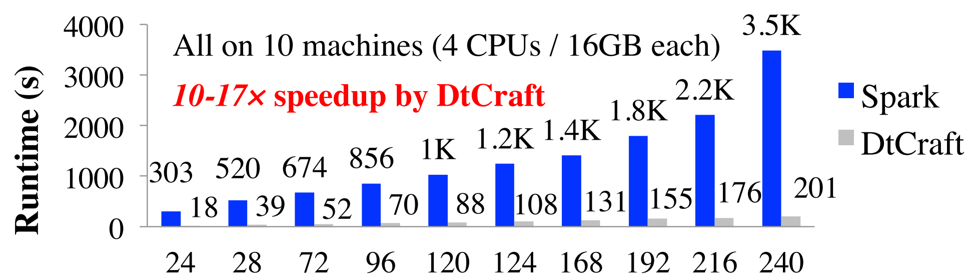
(a) Circuit (1.01mm²) (b) Graph (3M gates) (c) A signal path

Runtime comparison of shortest path finding



Number of machines (4 CPUs / 8GB each)

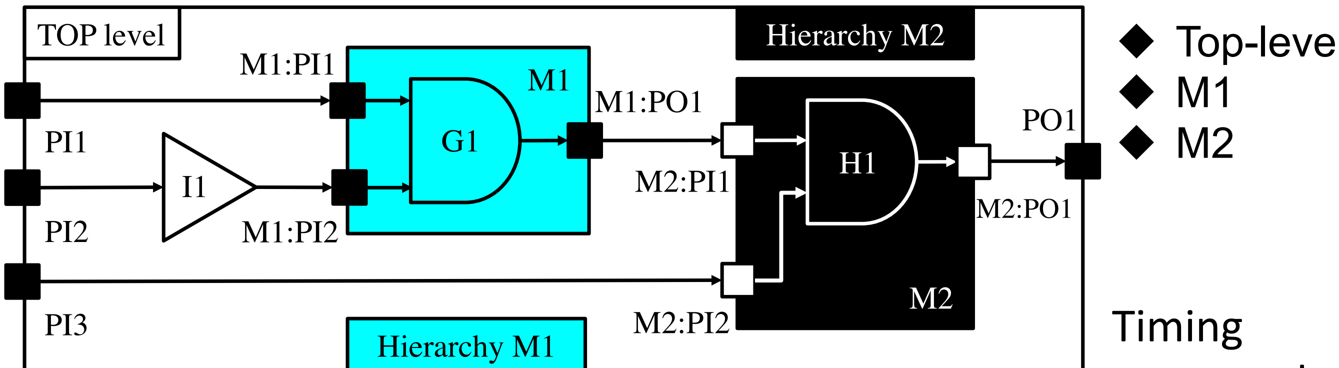
Performance scalability (runtime vs graph size)

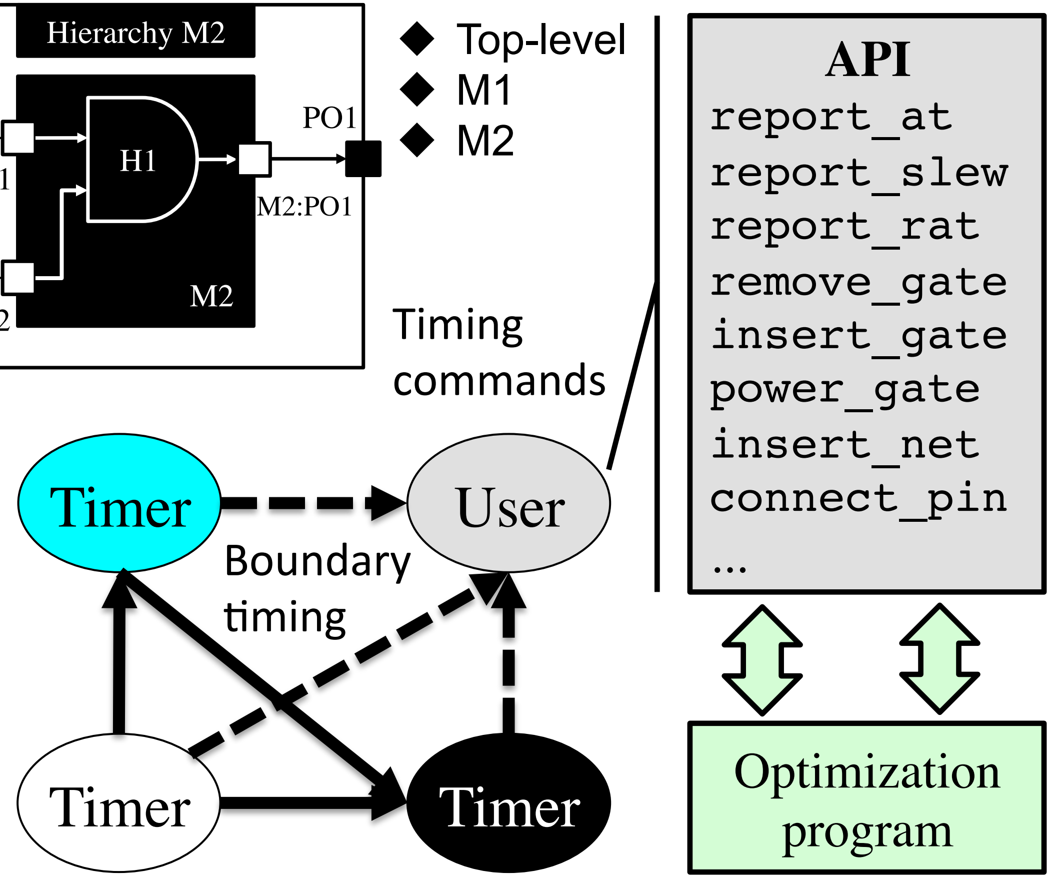


Graph size (# edges + # nodes) in million (M)

Distributed timing analysis using DtCraft

- ❑ **Two-level hierarchical design (three partitions)**





- ❑ 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

Google Protocol Buffer (open-source compiler)

C++/Java/Python source code generator

```
// 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;
}
```

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

Message wrapper

Derived packet struct
header_t header
void* buffer

Many extra stuff ☹

Extra.pb.h
Extra.pb.cpp
…
Source.cpp

Out-of-context streaming takes > 300 lines

Deploy the distributed timer in one line

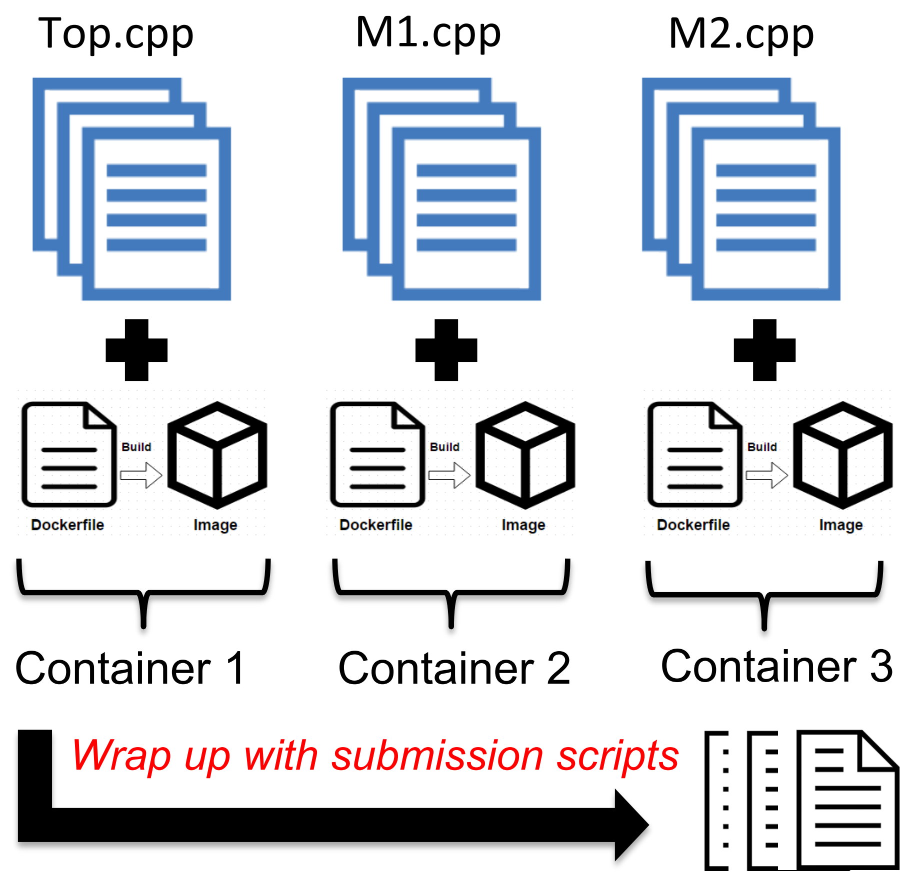
DtCraft ⟷ Existing framework

```
// 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

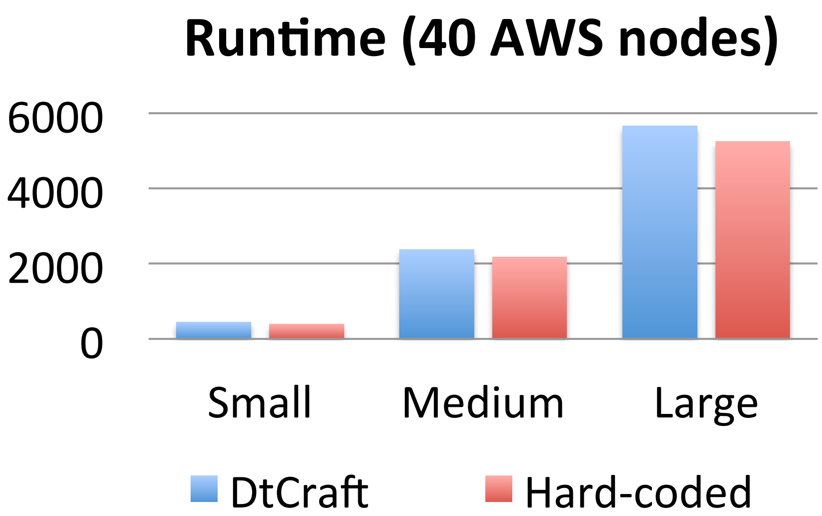
Duplicate the code for each partition

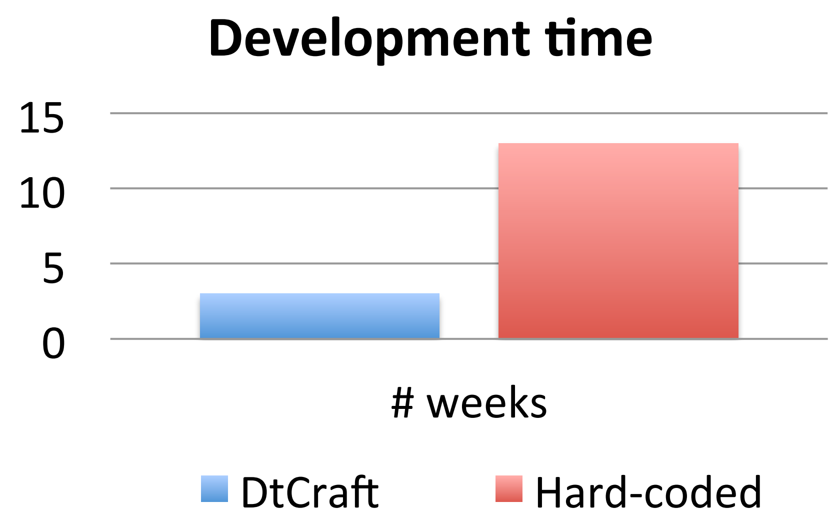


Comparison with the hard-coded method

- ❑ **×17 fewer lines of code**
  - ❑ 33% from message passing
  - ❑ 67% from boilerplate code
- ❑ **7-11% performance loss**
  - ❑ Transparent concurrency
  - ❑ API cost

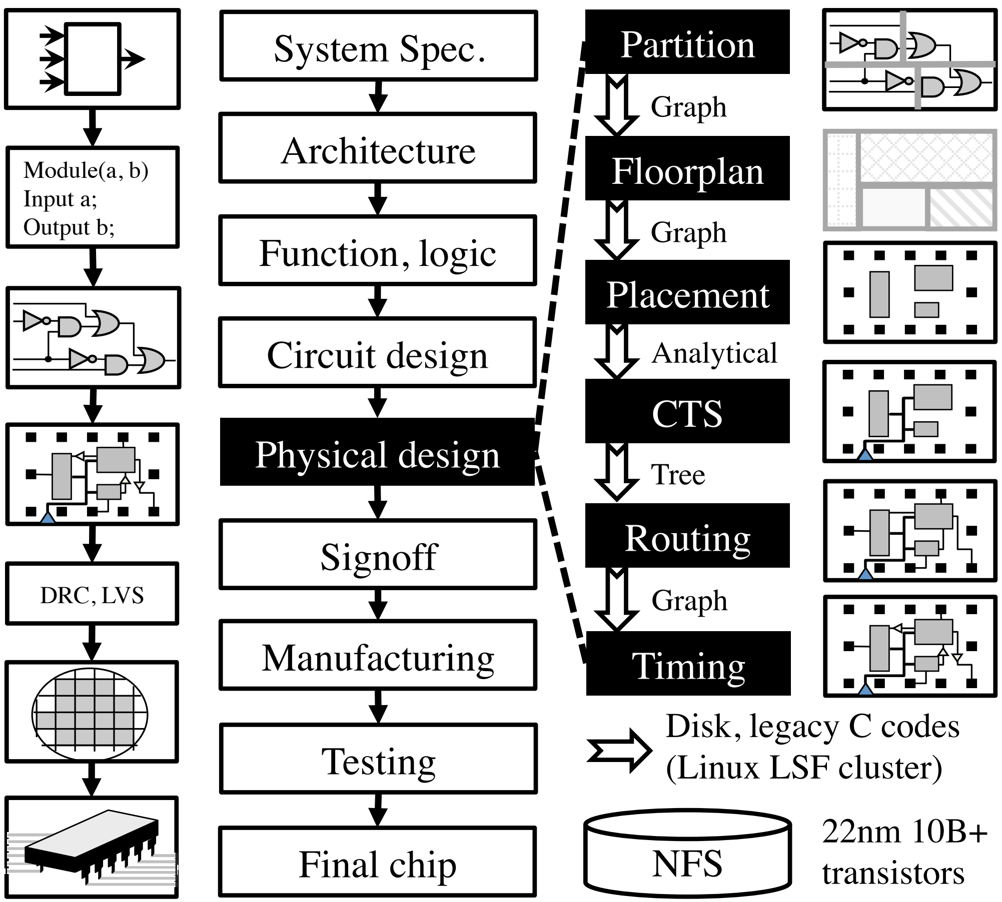
*"With DtCraft, it took me only three weeks, precisely, the **SPARE time** out of my summer internship at Citadel, to build a distributed timer that otherwise took my **whole summer internship** with IBM".*



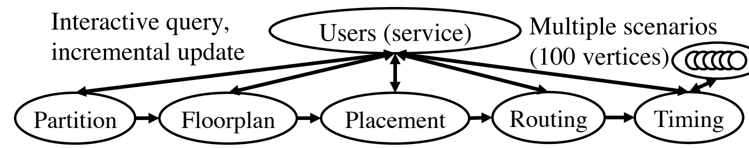


Experiments on EDA tool Integration

- Electronic design automation (EDA)



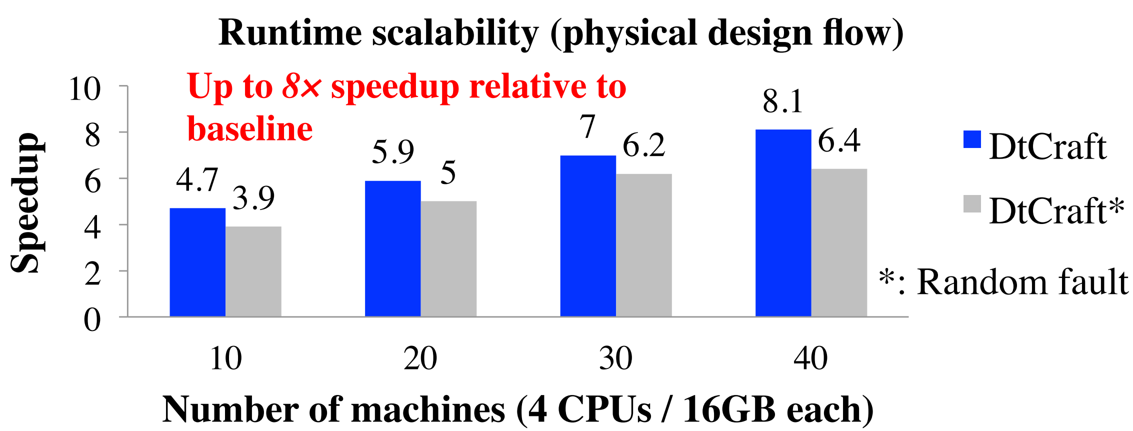
Stream graph for physical design

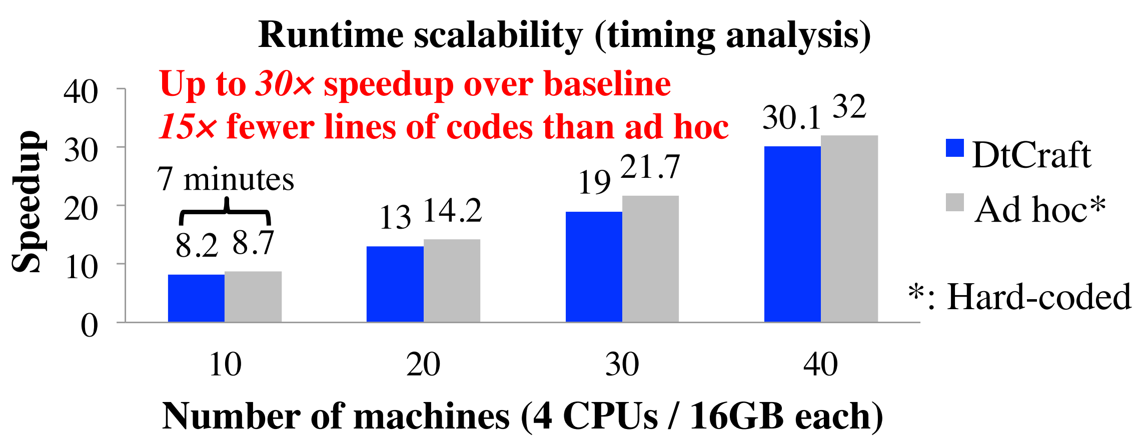


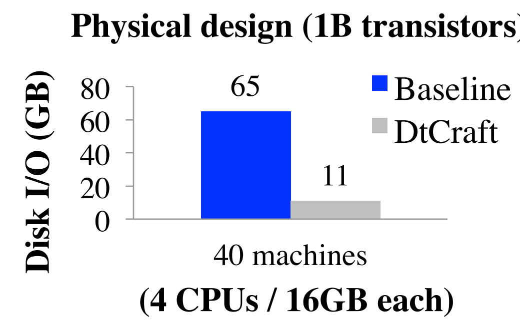
- Goal
  - New EDA methodology
  - distributed integration
  - Reduce tool-to-tool overhead
- Open-source tools
  - OpenTimer, placer, etc.

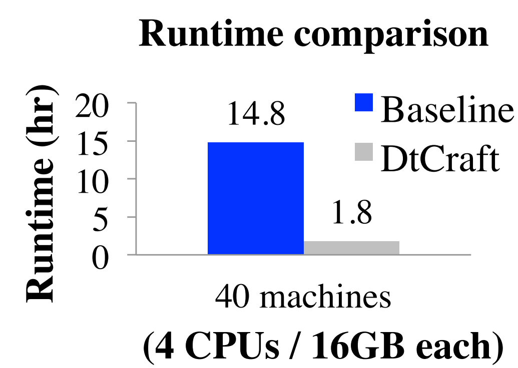
Experiments on EDA tool integration (cont'd)

- ❑ Physical design and timing analysis









Less disk IO translates to faster runtime

Conclusion

- **DtCraft: A distributed execution engine**
  - Creation of new parallel/distributed algorithms
  - Tool-to-tool integration at cloud scale
- **Tentative first release on 12/1**
  - Github repository
- **Acknowledgment**
  - UIUC CAD group

Thank you!

Tsung-Wei Huang
twh760812@gmail.com
(512) 815-9195

DtCraft
Parallel computing

Boost your productivity in writing parallel code!