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)
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 (MMMCM) 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

![Development time chart]

- # weeks
- Development time
- Localhost
- Distributed
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!

### Runtime comparison on arrival time propagation

<table>
<thead>
<tr>
<th>Method</th>
<th>Spark (RDD + GraphX Pregel)</th>
<th>Java (SP)</th>
<th>C++ (SP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (s)</td>
<td>68.45</td>
<td>9.5</td>
<td>1.50</td>
</tr>
</tbody>
</table>

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
  - ...

Which machine does which view?

Master (server)

Timing view 1
Timing view 2
Timing view 3

Difficult scalability 😞
Large amount of code rewrites 😞

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

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?

- Programming language
- Transparency
- Performance
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

```cpp
// 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

2016 average software engineer salary > 100K USD
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

High-level C++17-based Stream Graph API

| Network programming | I/O stream | Event-driven reactor | Resource control | Serialization |

DtCraft Programming Environment

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/](http://dtcraft.web.engr.illinois.edu/)
DtCraft github: [https://github.com/twhuang-uiuc/DtCraft](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

Generate data

Compute unit

Data stream

Stream graph

Generate data

Compute unit

buffer
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
```

Diagram:

```
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

Step 1: stream graph

- **A**
  - “hello from A”
  - Stop when no active streams

- **B**
  - “hello from B”
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 1: Stream graph

- A sends "hello from A"
- B sends "hello from B"
- Stop when no active streams

Step 2:

```
string msg;
```
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 1: stream graph

Step 2:

```
string msg;
```

“hello from A”

Step 3: A→B callback

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

“hello from B”

Step 3: A←B callback

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

A ← istream

B ← istream
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 1: stream graph

Step 2: string msg;

“hello from A”

Step 3: A\(\rightarrow\)B callback

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

“hello from B”

Step 4: A’s resource
1 CPU / 1 GB RAM

Step 3: A\(\leftarrow\)B callback

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

Step 4: B’s resource
1 CPU / 1 GB RAM
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 1:** stream graph

**Step 2:**

```cpp
string msg;

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

**Step 3:** A ➔ B callback

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

**Step 4:**

- A’s resource
  - 1 CPU / 1 GB RAM
- B’s resource
  - 1 CPU / 1 GB RAM

**Step 5:**

```
./submit –master=127.0.0.1 hello-world
```
Seeing is Believing

- 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

```cpp
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) {
    dtc::cout("hello world from A\'s");
    dtc::cout("Sent \'hello world from A\' to stream ", AB, '\n\');
});

B.on([&BA] (dtc::Vertex& v) {
    dtc::cout("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();```
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_bytes();
    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);
}

Branch your code to server and client for distributed computation!

```
simple.cpp → server.cpp + client.cpp
```
MapReduce Flow Example

Word counting

Find the frequency of each word in a data set

Step 1: stream graph

Step 2: String Data;

Step 2: Map WordCount;

Step 3: A_i \rightarrow M \text{ callback}

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

Step 3: A_i \rightarrow M \text{ callback}

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

Step 4: M’s resource

8 CPU / 8 GB RAM

Step 4: A_i’s resource

1 CPU / 1 GB RAM

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

Runtime comparison of machine learning applications

- 4-11x speedup on logistic regression (LR)
- 5-14x speedup on $k$-means (KM)

Distributed storage

V1 → V2 → ... → Vn

Points

Update weight

Stream graph

Number of machines (4 CPUs / 16GB each)
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

![Graph Algo Image](image)

**Runtime comparison of shortest path finding**

- 10-20x speedup by DtCraft

**Performance scalability (runtime vs graph size)**

- 10-17x speedup by DtCraft
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 (under 1 min)

---

On a cluster with 10 machines (4 CPUs/16 GB each)

- **Runtime (s)**
  - **Stream graph**
  - **MPI**

- **Power grid size (# of nodes) in million (M)**

---

Distributed Timing Analysis

- **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

```cpp
// 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
        }
    }
);
```

Existing framework

Google Protocol Buffer

- Hard-code your message format
- 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

DtCraft

- In-context streaming with < 30 lines

Existing framework

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 M2 = 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);

~$ ./submit --master=127.0.0.1 binary

Existing framework

Duplicate the code for each partition
Top.cpp M1.cpp M2.cpp

Duplicate the code for each partition

Only three lines for resource control in Linux container

Wrap up with submission scripts

Container 1 Container 2 Container 3
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

---

**Runtime scalability (timing analysis)**

Up to 30× speedup over baseline
15× fewer lines of codes than ad hoc

7 minutes

<table>
<thead>
<tr>
<th>Number of machines (4 CPUs / 16GB each)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8.2</td>
</tr>
<tr>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>30</td>
<td>19</td>
</tr>
<tr>
<td>40</td>
<td>30.1</td>
</tr>
</tbody>
</table>

- **DtCraft**
- **Ad hoc***

*: Hard-coded

* 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”.

![Development time of distributed timer](chart)

- **DtCraft**: Development time of 3 weeks
- **Hard-coded**: Development time of 15 weeks

Weeks
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/