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

Tsung-Wei Huang
ECE Dept, UIUC, IL
twh760812@gmail.com

Chun-Xun Lin
ECE Dept, UIUC, IL
clin99@illinois.edu

Guannan Guo
ECE Dept, UIUC, IL
guannan4@gmail.com

Martin D. F. Wong
ECE Dept, UIUC, IL
mdfwong@illinois.edu

ABSTRACT

In this paper we present *DtCraft*, a distributed execution engine that enables a new powerful programming model to streamline cluster computing. Applications are described in a set of *data-parallel streams*, leaving difficult execution details and concurrency controls handled by our system kernel transparently. Compared with existing systems, DtCraft is unique in (1) an efficient stream-oriented programming paradigm using modern C++17, (2) an in-context resource controller and task executor based on Linux container technology, and (3) ease of development from prototyping machines to production cloud environments. These capabilities power industry applications and create new research directions in machine learning, stream processing, and distributed multimedia systems.

KEYWORDS

Distributed System, Stream Processing, Machine Learning

ACM Reference Format:
Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. 2018. A General-purpose Distributed Programming System using Data-parallel Streams. In *2018 ACM Multimedia Conference (MM '18), October 22–26, 2018, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3240508.3243654

1 INTRODUCTION

Recent years have seen tremendous success of cluster computing engines such as Hadoop MapReduce, DryadLINQ, and Apache Spark [1–3]. These frameworks offer high-level abstraction over system details and allow users without experience in distributed computing to quickly utilize the cluster resources to run big-data analytics. Despite promising advances, many industry experts and researchers have found these tools not an easy fit to their domains, in particular, real-time stream computing, distributed multimedia applications, and large-scale optimizations [4, 5]. A major reason is most existing frameworks target at *data-driven* applications. Data are divided into independent pieces followed by parallel MapReduce operations. Depending on applications, data might be highly connected and cannot be easily partitioned [6, 7]. Also, striving for higher cluster computing performance involves more complex resource managements and irregular compute patterns. The key challenge is thus to develop an elastic programming system–*which is believed to deliver the next leap of engineering productivity and unleash new business model opportunities* [4, 8].

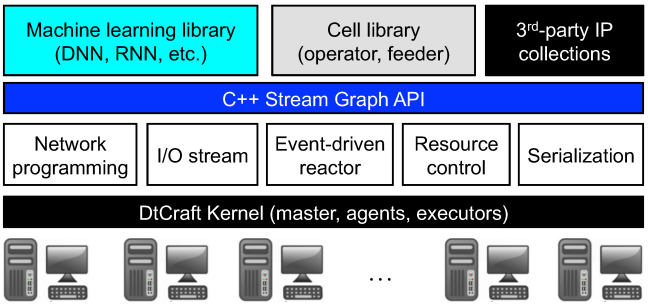

Figure 1: Software stack of the DtCraft system [9].

To this end, we have developed a system called DtCraft [9]. DtCraft is a general-purpose distributed programming system based on data-parallel streams [10]. Figure 1 gives an overview of DtCraft's software stack. The main theme is to make parallel and distributed programming easier to handle through our *stream graph* programming model. In order to ensure the best performance, we have redesigned many core system components from the ground up using modern C++17. These include network programming libraries, built-in serialization and deserialization interface, event-driven reactor, and so on. Users can make use of robust C++ standard library along with our parallel framework to deploy high-performance distributed applications on Linux clusters. In fact, we found most users are able to master DtCraft's application programming interface (API) required for most of the applications in a couple of days.

To enable an order of magnitude more users to build production applications using DtCraft, we developed several libraries on top to handle new types of workloads. The first library, *MLCraft*, implements a set of *estimators* with high-level API to greatly simplify machine learning programming in common use cases. Each estimator works seamlessly with the DtCraft core, allowing users to quickly scale out and create new machine learning algorithms without wrestling with low-level details. The second library, *Celllib*, provides a set of predefined stream graphs that can be readily added to an application graph. This largely simplifies the graph creation process and helps mitigate buggy implementations due to immature copy-and-paste mistakes. Users can also add new stream graphs to the library collections and expose them in other parts

MM '18, October 22–26, 2018, Seoul, Republic of Korea

ACM ISBN 978-1-4503-5665-7/18/10…$15.00
https://doi.org/10.1145/3240508.3243654

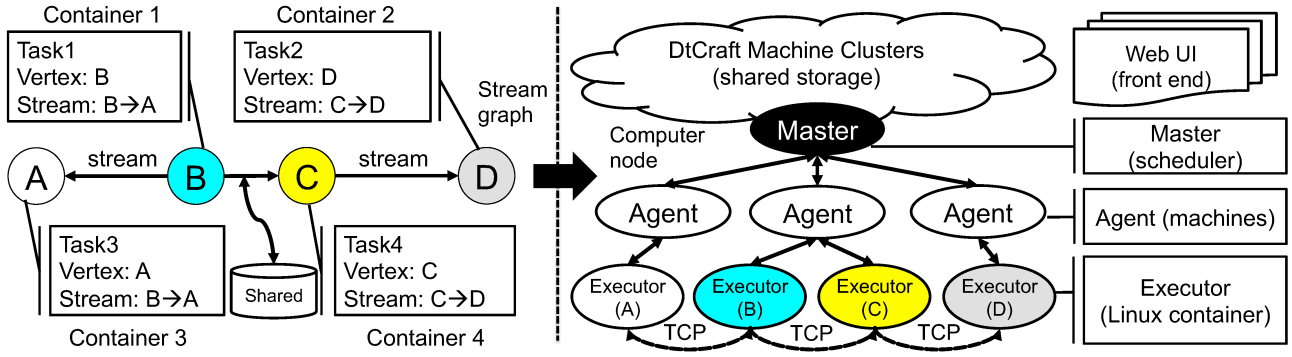

Figure 2: The system architecture of DtCraft. The kernel consists of a master daemon and one agent daemon per working machine. User describes an application in terms of a sequential stream graph and submits the executable to the master through our submission script. The kernel automatically deals with job scheduling, process communication, and work distribution that are known difficult to program correctly. Data is transferred through either TCP socket streams on inter-edges or shared memory on intra-edges, depending on the deployment by the scheduler. Application and workload are isolated in Linux containers.

of DtCraft including streaming and batch applications. These libraries can facilitate the developments of new multimedia, AI, and real-time streaming systems [11].

Our work and user experiences lead us to believe that each system has its own reason to exist. The judgement should be left for users. DtCraft is being actively maintained and has recently acquired multi-year supports from Defense Advanced Research Projects Agency (DARPA) to help advance the next generation distributed computing. This will allow us to provide long term support for each major release. For more details about DtCraft, please refer to our official website [9].

2 THE DTCRAFT SYSTEM

In this section we highlight the system architecture of DtCraft and discuss our stream graph programming model.

2.1 System Architecture

The overview of the DtCraft system architecture is shown in Figure 2. The system kernel contains a *master* daemon that manages *agent* daemons running on each cluster node. Each job is coordinated by an *executor* process that is either invoked upon job submission or launched on an agent node to run the tasks. A job or an application is described in a stream graph formulation. Users can specify resource requirements (e.g. CPU, memory, disk usage) and define computation callbacks for each vertex and edge, while the whole detailed concurrency controls and data transfers are automatically operated by the system kernel. A job is submitted to the cluster via a script that sets up the environment variables and the executable path with arguments passed to its `main` method. When a new job is submitted to the master, the scheduler partitions the graph into several *topologies* depending on current hardware resources and CPU loads. Each topology is then sent to the corresponding agent and is executed in an executor process forked by the agent. For those edges within the same topology, data is exchanged via efficient shared memory. On the other hand, connections across different topologies run through TCP sockets.

2.2 Stream Graph Programming Model

One of the key inventions of DtCraft is the stream graph programming model. The term "stream" is analogous to an assembly pipeline, where one end generates a series of products and the other end processes these products in a first-in-first-out (FIFO) manner. Listing 1 shows the gateway classes to create a stream graph.

```
class Vertex {
  any_type any;
  shared_ptr<OutputStream> ostream(key_type);
  shared_ptr<InputStream> istream(key_type);
};

class Stream {
  function<Signal(Vertex&, OutputStream&)> on_os;
  function<Signal(Vertex&, InputStream&)> on_is;
};

class Graph {
  VertexBuilder vertex();
  StreamBuilder stream(key_type, key_type);
  ContainerBuilder container();
};
```

Listing 1: Gateway classes to create a stream graph.

A stream graph consists of three major components, *vertex*, *stream*, and *container*. A vertex is a place to store computation results and a stream represents a directed channel to send or receive data. Each stream is associated with two computation callbacks, one on input side and one on output side. These computation callbacks occur whenever data is available on the underlying device which can be a file, a socket, or a pipe. Multiple streams are independent of each other and can run in parallel. A container represents one unique partition of a stream graph. Unlike existing cluster computing frameworks, DtCraft delegates the scheduler control to users. Gaining valuable hints from users instead of blind graph partitions can guide the scheduler toward the best job deployment.

3 APPLICATIONS AND EXAMPLES

In this section, we give several concrete examples to help readers better understand DtCraft.

3.1 A Vanilla Stream Graph

Listing 2 presents a simple yet representative stream graph example. The stream graph consists of two vertices, A and B, and two streams. Each vertex sends a greeting message to the other end and closes the underlying stream channel. Closing one end of a stream will subsequently force the other end to close. The program terminates when no active stream events exist. Finally we create two containers each of 1 GB memory and 1 CPU to distribute A and B to two machines.

```
Graph G;
auto A = G.vertex();
auto B = G.vertex();
auto lambda = [] (Vertex& v, InputStream& is) {
  if(string s; is(s) != -1) {
    cout << "Received: " << s << '\n';
    return Event::REMOVE;
  }
  return Event::DEFAULT;
};
auto AB = G.stream(A, B).on(lambda);
auto BA = G.stream(B, A).on(lambda);

A.on([&AB] (Vertex& v) {
  (*v.ostream(AB))("hello world from A"s);
});
B.on([&BA] (Vertex& v) {
  (*v.ostream(BA))("hello world from B"s);
});

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

Executor(G).run();
```

Listing 2: A distributed hello-world stream graph.

There are three important aspects. First, data streams are parallel. The program might print A's message before B or B's message before A, whichever arrives first. Second, even though the stream graph contains a cycle, developers should be aware of the fact that computation occurs asynchronously. There is no explicit synchronization or loop linearization to break the cycle. Third, distributing A and B to different nodes takes only two lines of code. The same code running on a local machine can easily scale out to multiple machines. These advantages allow DtCraft to create more general and faster dataflow graphs compared to [2, 3, 12].

3.2 Online Machine Learning

The second example is a online image classifier using our built-in deep neural network (DNN) library. Implementing a production distributed or online machine learning algorithm is a notoriously difficult task not because of the learning algorithm but the complex peripheral works such as streaming, model exposure, and data collection [8]. DtCraft's machine learning library aims to reduce this barrier. Figure 3 shows the stream graph to train a DNN classifier through a sequence of images from the MNIST database [13]. The stream feeder is a cell of a predefined stream graph that is readily useable to create image streams. The other vertex takes a DNN classifier and performs online training on each image stream.

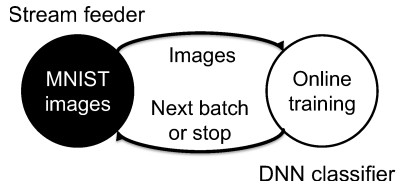

Figure 3: A online image classifier stream graph.

```
Graph G;
auto src = G.insert<cell::MnistStreamFeeder>(
  mnist_image_file, mnist_label_file
);
auto dnn = G.vertex();
auto d2s = src.in(dnn);

dnn.on([&] (Vertex& v) {
  auto& c = v.any.emplace<DnnClassifier>();
  printf("DNN classifier [784x30x10]\n");
  c.layer<FullyConnectedLayer>(784, 30, RELU);
  c.layer<FullyConnectedLayer>(30, 10);
  (*v.ostream(d2s))(10000);
});

G.stream(src.out(), dnn).on(
  [&] (Vertex& v, InputStream& is) mutable {
    auto& c = any_cast<DnnClassifier&>(v.any);
    Eigen::MatrixXf M;
    Eigen::VectorXi L;
    while(is(M, L) != -1) {
      auto cnt = ((L - c.infer(M)) == 0).count();
      auto acc = cnt / static_cast<float>(M.rows());
      printf("Accuracy: %f\n", acc);
      c.train(M, L, 1, 64, 0.01f, [] () {});
      (*v.ostream(d2s))(acc < 0.95f ? 10000 : -1);
    }
    return Event::DEFAULT;
  }
);

G.container().add(dnn);
G.container().add(src);

Executor(G).run();
```

Listing 3: Implementation of the online image classifier.

The implementation is shown in Listing 3. In a rough view, there are only a couple lines of code to implement a distributed online machine learning algorithm. This sequential code is capable of running distributively on two nodes, one for dnn and another for src. At each time step, the DNN classifier dnn requests 10000 images from the stream feeder src and performs training until the accuracy reaches 95%. It is observed that the feeder cell simplifies the graph creation process. By calling src.in(dnn), a stream d2s connecting dnn to src is automatically added to the graph. In short, the cell interface introduces a new way to develop "reusable" software at cluster scale. Users can create new cells to encapsulate certain applications, thereby allowing better reuse of design efforts.

3.3 External Program

DtCraft supports the execution of external programs. This is particularly useful when users want to create new streaming applications on top of existing or 3rd-party programs. When a vertex is specified as an external program, the kernel spawns a new program supplied by the given command. Connections will be passed to the

program through the environment variable DTC_BRIDGES, where key is the stream name and value is the associated file descriptor. Users can retrieve these handles and establish stream channels using their own libraries or our I/O stream API. As shown in Figure 4, the demo contains two vertices and one stream to emulate real-time image processing via external programs.

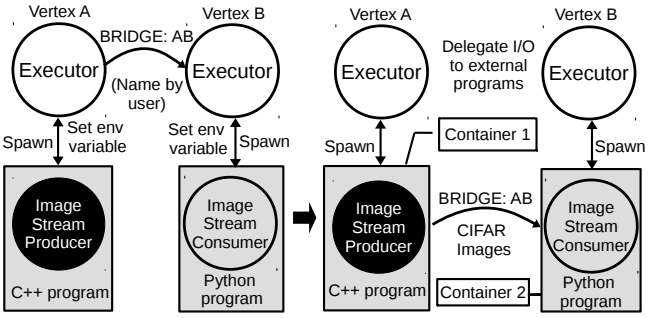

Figure 4: Image stream processing via external programs.

```
Graph G;
auto A = G.vertex().program("cifar-10_stream");
auto B = G.vertex().program("processing.py");
G.stream(A, B).tag("AB");
G.container().add(A);
G.container().add(B);
dtc::Executor(G).run();
```

Listing 4: The top-level graph implementation.

```
auto fd = get_bridge_fd("AB");
auto cifar = CIFARHandle("cifar_data.bin");
Reactor reactor;
reactor.insert<PeriodicEvent>(1s, true, [&] (auto&) {
  cifar.write_one_image(fd, batch_size);
  if(cifar.empty()) {
    return Event::REMOVE;
  }
  return Event::DEFAULT;
});
reactor.dispatch();
```

Listing 5: CIFAR-10 image stream generator.

```
fd = get_bridge_fd("AB")
while true:
  data = read_one_image(fd, size_per_image);
  # process image from data
  # ...
```

Listing 6: Python-based image stream consumer.

There are three programs, a top-level stream graph description in Listing 4, an image stream generator based on CIFAR database [14] in Listing 5, and a python-based image stream consumer in Listing 6. The stream generator employs DtCraft's event reactor to create a sequence of image streams for every one second. The image stream goes to a python program of a pre-define kernel for image processing. Please be mindful these two external programs are for demonstration purpose only. One can replace them with other applications. The top-level graph description is the key contribution of DtCraft. It is extremely simple to distribute a set of external programs and let them talk to each other. Currently, we are leveraging the existing container technologies to enhance this functionality. For example, with Docker containers [15] we can bring up different software to a DtCraft cloud. These containers can communicate with one another through our streamming interface, allowing users to create new software on top of exsiting ones at cluster scale.

4 AVAILABILITY

The source of DtCraft is published on GitHub under MIT license [16]. Project details, step-by-step tutorials, and cookbook are available on the project homepage [9].

5 ACKNOWLEDGMENT

The works is supported by DARPA under award FA8650-18-2-7843.

6 CONCLUSION

This paper presents a new distributed system DtCraft to streamline the programming for computer clusters. DtCraft introduces a powerful stream-oriented programming model and can power large-scale industry applications in machine learning, multimedia, and cloud computing. Since the first public release in December 2017, DtCraft has been used in multiple research projects at the University of Illinois at Urbana-Champaign. At the same time, we are collaborating with our industry partners to obtain state-of-the-art results. We also work with startups to prototype new workloads and business models. Future work will focus on both application- and system-level improvements on DtCraft.

REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.
[2] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *USNIX OSDI*, pages 1–14, 2008.
[3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USNIX NSDI*, 2012.
[4] L. Stok. The next 25 years in EDA: A cloudy future? *IEEE Design Test*, 31(2):40–46, April 2014.
[5] Mengfan Tang, Siripen Pongpaichet, and Ramesh Jain. Research challenges in developing multimedia systems for managing emergency situations. In *ACM MM*, pages 938–947, 2016.
[6] Tsung-Wei Huang and Martin D. F. Wong. Opentimer: A high-performance timing analysis tool. In *ACM/IEEE ICCAD*, pages 895–902, 2015.
[7] Tsung-Wei Huang, Martin D. F. Wong, Debjit Sinha, Kerim Kalafala, and Natesan Venkateswaran. A distributed timing analysis framework for large designs. In *ACM/IEEE DAC*, pages 116:1–116:6, 2016.
[8] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *NIPS*, pages 2503–2511, 2015.
[9] DtCraft project. http://dtcraft.web.engr.illinois.edu/.
[10] Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong. DtCraft: A High-performance Distributed Execution Engine at Scale. *IEEE TCAD*, 2018.
[11] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin D. F. Wong. Mt-Detector: A High-performance Marine Traffic Detector at Stream Scale. In *ACM DEBS*, 2018.
[12] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems, Wiley*, page 13, 2012.
[13] MNIST database. http://yann.lecun.com/exdb/mnist/.
[14] CIFAR image database. https://www.cs.toronto.edu/~kriz/cifar.html.
[15] Docker. https://www.docker.com/.
[16] DtCraft source. https://github.com/twhuang-uiuc/DtCraft.