DtCraft: A Distributed Execution Engine for Compute-intensive Applications

Tsung-Wei Huang*, Chun-Xun Lin†, and Martin D. F. Wong‡
*twh760812@gmail.com, †clin99@illinois.edu, ‡mdfwong@illinois.edu
Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA

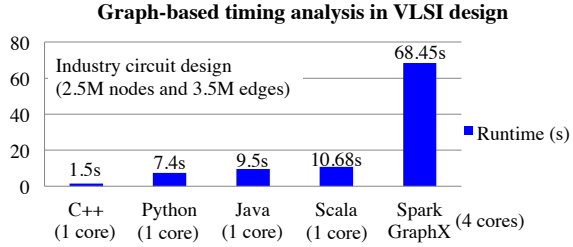
***Abstract*—Recent years have seen rapid growth in *data-driven* distributed systems such as Hadoop MapReduce, Spark, and Dryad. However, the counterparts for high-performance or *compute-intensive* applications including large-scale optimizations, modeling, and simulations are still nascent. In this paper, we introduce *DtCraft*, a modern C++17-based distributed execution engine that efficiently supports a new powerful programming model for building high-performance parallel applications. Users need no understanding of distributed computing and can focus on high-level developments, leaving difficult details such as concurrency controls, workload distribution, and fault tolerance handled by our system transparently. We have evaluated DtCraft on both micro-benchmarks and large-scale optimization problems, and shown promising performance on computer clusters. In a particular semicondictor design problem, we achieved $30\times$ speedup with 40 nodes and $15\times$ less development efforts over hand-crafted implementation.**

I. Introduction

Recently, cluster computing frameworks such as MapReduce, Spark, and Dryad have been widely used for big data processing [1], [2], [3]. The availability of allowing users without any experience of distributed systems to develop applications that access large cluster resources has demonstrated great success in many big data analytics. Existing platforms, however, mainly focus on big data processing. Research for high-performance or *compute-driven* counterparts such as large-scale optimizations and engineering simulations has failed to garner the same attention. As horizontal scaling has proven to be the most cost-efficient way to increase compute capacity, the need to efficiently leverage numerous computations is quickly becoming the next challenge [4], [5].

Compute-intensive applications have many different characteristics from big data. First, developers are obsessed about performance. Striving for high performance typically requires intensive CPU computations and efficient memory managements, while big data computing is more data-intensive and I/O-bound. Second, performance-critical data are more connected and structured than that of big data. Design files cannot be easily partitioned into independent pieces, making it difficult to fit into MapReduce paradigm [1]. Also, it is fair to claim most compute-driven data are *medium-size* as they must be kept in memory for performance purpose [4]. The benefit of MapReduce may not be fully utilized in this domain. Third, performance-optimized programs are normally hard-coded in C/C++, whereas the mainstream big data languages are Java, Scala, and Python. Rewriting these ad-hoc programs that have been robustly present in the tool chain for decades is not a practical solution.

To prove the concept, a recent research study has reported an experiment comparing the performance of running VLSI timing analysis under different languages and system frameworks [7]. As shown in Figure 1, the hand-crafted C/C++ program is much faster than many of mainstream big data languages such as Python, Java, and Scala. It can even outperform one of the best big data cluster computing frameworks, the distributed Spark/GraphX-based implementation, by $45\times$ faster. Many industry experts have realized that big data is not an easy fit to their domains, for example, semiconductor design optimizations and engineering simulations. Unfortunately, the ever-increasing design complexity will far exceed what many old ad-hoc methods have been able to accomplish. In addition to having researchers and practitioners acquire new domain knowledge, we must rethink the approaches of developing software to enable the proliferation of new algorithms combined with readily reusable toolboxes. To this end, the key challenge is to discover an elastic programming paradigm that lets developers place computations at customizable granularity wherever the data is – which is believed to deliver the next leap of engineering productivity and unleash new business model opportunities [4].



	Compute-intensive	Big data
Computation	CPU-bound	I/O-bound
Data traits	Structured, monolithic	Unstructured, sharded
Storage	NFS, GPFS, Ceph	HDFS, GFS
Programming	Ad-hoc, C/C++	MapReduce, Java, Scala
Example	EDA, optimization, simulation	Log mining, database, analytic

Fig. 1. An example of VLSI timing analysis and the comparison between compute-intensive applications and big data [6], [7].

One of the main challenges to achieve this goal is to define a suitable programming model that abstracts the data computation and process communication effectively. The success of big data analytics in allowing users without any experience of distributed computing to easily deploy jobs that access large cluster resources is a key inspiration to our system design [1], [2], [3]. We are also motivated by the fact that existing big data systems such as Hadoop and Spark are facing the bottleneck in support for compute-optimized codes and general dataflow programming [5]. For many compute-driven or resource-intensive problems, the most effective way to achieve scalable performance is to force developers to exploit the parallelism. Prior efforts have been made to either breaking data dependencies based on domain-specific knowledge of physical traits or discovering independent components across multiple application hierarchies [7]. Our primary focus is instead on the generality of a programming model and, more importantly, the simplicity and efficiency of building distributed applications on top of our system.

While this project was initially launched to address a question from our industry partners, "*How can we leverage the numerous computations of semiconductor designs to improve the engineering productivity?*", our design philosophy is a general system that is useful for compute-intensive applications such as graph algorithms and machine learning. As a consequence, we propose in this paper *DtCraft*, a general-purpose distributed execution engine for building high-performance parallel applications. DtCraft is built on Linux machines with modern C++17, enabling end users to utilize the robust C++ standard library along with our parallel framework. A DtCraft application is described in the form of a *stream graph*, in which vertices and edges are associated with each other to represent generic computations and real-time data streams. Given an application in this framework, the DtCraft runtime automatically takes care of all concurrency controls including partitioning, scheduling, and work distribution over the cluster. Users do not need to

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

worry about system details and can focus on high-level development toward appropriate granularity. We summarize three major contributions of DtCraft as follows:

- **New programming paradigm**. We introduce a powerful and flexible new programming model for building distributed applications from sequential stream graphs. Our programming model is very simple yet general enough to support generic dataflow including feedback loops, persistent jobs, and real-time streaming. Stream graph components are highly customizable with meta-programming. Data can exist in arbitrary forms, and computations are autonomously invoked wherever data is available. Compared to existing cluster computing systems, our framework is more elastic in gaining scalable performance.
- **Software-defined infrastructure**. Our system enables fine-grained resource controls by leveraging modern OS container technologies. Applications live inside secure and robust Linux containers as work units which aggregate the application code with runtime dependencies on different OS distributions. With a container layer of resource management, users can tailor their application runtime toward tremendous performance gain.
- **Unified framework**. We introduce the first integration of user-space dataflow programming with resource container. For this purpose, many network programming components are re-devised to fuse with our system architecture. The unified framework empowers users to utilize rich APIs of our system to build highly optimized applications.

We believe DtCraft stands out as a unique system considering the ensemble of software tradeoffs and architecture decisions we have made. With these features, DtCraft is suited for various applications both on systems that search for transparent concurrency to run compute-optimized codes, and on those that prefer distributed integration of existing developments with vast expanse of legacy codes in order to bridge the performance gap. We have evaluated DtCraft on micro-benchmarks including machine learning, graph algorithms, and large-scale semiconductor engineering problems. We have shown DtCraft outperforms one of the best cluster computing systems in big data community by more than an order of magnitude. Also, we have demonstrated DtCraft can be applied to wider domains that are known difficult to fit into existing big data ecosystems.

II. The DtCraft System

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. Edges between different topologies are communicated through TCP sockets. Stream overflow is resolved by per-process key-value store, and users are perceived with virtually infinite data sets without deadlock.

A. Stream Graph Programming Model

DtCraft is strongly tight to modern C++ features, in particular the concurrency libraries, lambda functions, and templates. We have struck a balance between the ease of the programmability at user level and the modularity of the underlying system that needs to be extensible with the advance of software technology. The main programming interface including gateway classes is sketched as follows:

```
class Vertex {
  function<void()> on;
  once_flag flag;
  Adjacency<DeviceWriter> writers;  // weak pointers
  Adjacency<DeviceReader> readers;  // weak pointers
};

class Stream {
  weak_ptr<DeviceWriter> writer;
  weak_ptr<DeviceReader> reader;
  function<Signal(Vertex&, DeviceWriter&)> on_os();
  function<Signal(Vertex&, DeviceReader&)> on_is();
};

class Graph {
  template <typename C> // vertex
  auto insert(C&&...);

  template <typename O, typename I> // stream
  auto insert(const auto&, O&&, const auto&, I&&)

  template <typename... U>
  auto containerize(U&&...);
};

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

Programmers formulate an application into a stream graph and define computation callbacks in the format of standard function object for each vertex and edge (stream). Vertices and edges are highly customizable subject to the inheritance from classes `Vertex` and `Stream` that interact with our back-end. The vertex callback is a constructor-like call-once barrier that is used to synchronize all adjacent edge streams at the beginning. Each edge is associated with two callbacks, one for output stream at the tail vertex and another one for input stream at the head vertex. Our stream interface follows the structure of standard C++ `iostream` library. We have developed specialized *stream buffer* classes in charge of performing reading and writing operations on stream objects. The stream buffer class hides from users a great deal of work such as non-blocking communication, stream overflow and synchronization, and error handling. Vertices and edges are explicitly connected together through the `Graph` and its method `insert`. Users can configure the resource requirements for different portions of the graph using our container method `containerize`. Finally, an executor class forms the graph along with application-specific parameters into a simple closure and dispatches it to the remote master for execution.

B. A Concurrent Ping-pong Example

To understand our programming interface, we describe a concrete example of a DtCraft application. The example we have chosen is a representative class in many software libraries – *concurrent ping-pong*, as it represents a fundamental building block of many iterative or incremental algorithms. The flow diagram of a concurrent ping-pong and its runtime on our system are illustrated in Figure 3. The ping-pong consists of two vertices, called "*Ball*", which asynchronously sends a random binary character to each other, and two edges that are used to capture the data streams. Iteration stops when the internal counter of a vertex reaches a given threshold.

```
auto Ball(Vertex& v, auto& k) {
  v.writers.at(k).lock->ostream((rand()%2));
  return Stream::DEFAULT;
};
auto PingPong(auto& v, auto& r, auto& k, auto& c) {
  int data;
  reader.istream(data);
  if((c+=data) >= 100) return Stream::REMOVE_THIS;
  return Ball(v, k)
}

Graph G;
key_type AB, BA;
auto count_A {0}, count_B {0};
auto A = G.insert([&](auto& v){ Ball(v, AB); })
```

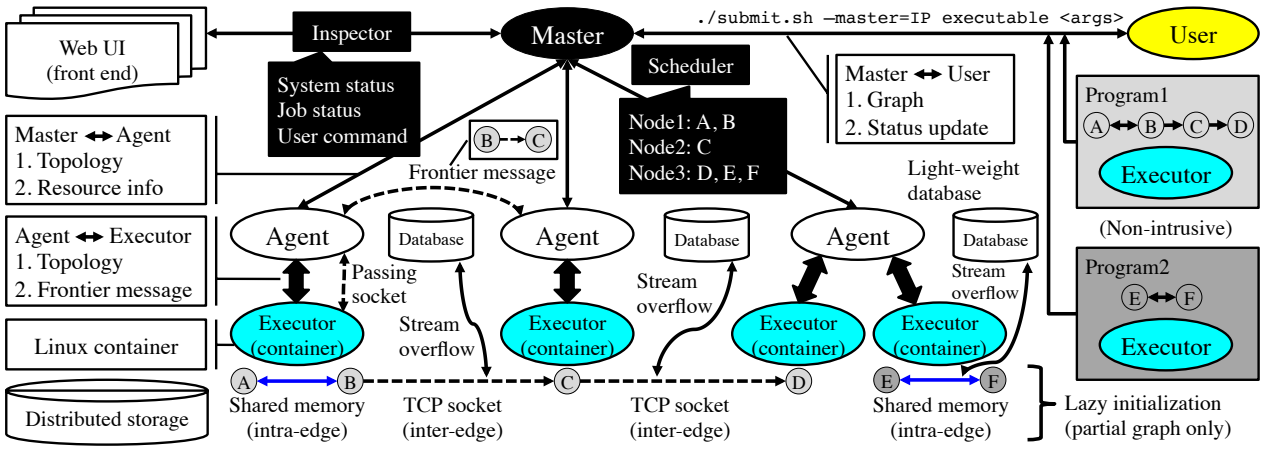


Fig. 2. The system architecture of DtCraft. The kernel consists of a master daemon and one agent daemon per working machine. User describes an application in terms of a sequential stream graph and submits the executable to the master through our submission script. The kernel automatically deals with concurrency controls including scheduling, process communication, and work distribution that are known difficult to program correctly. Data is transfered 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 secure and robust Linux containers.

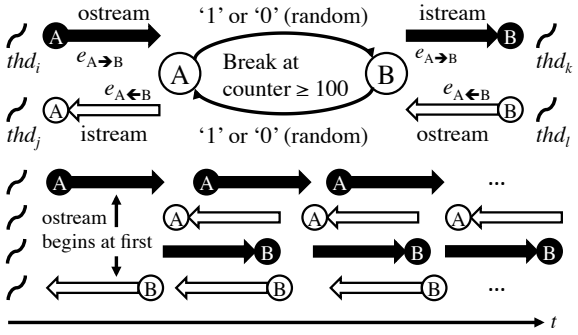


Fig. 3. Flow diagram of the concurrent ping-pong example. Computation callbacks on streams are simultaneously invoked by multiple threads.

```
auto B = G.insert([&](auto& v){ Ball(v, BA); })
AB = G.insert(
  A, [&](auto& v, auto& writer) {}, // ostream
  B, [&](auto& v, auto& reader) {   // istream
    return PingPong(v, reader, BA, count_B);
  }
);
BA = G.insert(
  B, [&](auto& v, auto& writer) {}, // ostream
  A, [&](auto& v, auto& reader) {   // istream
    return PingPong(v, reader, AB, count_A)
  }
);
G.containerize(A, "memory=1KB", "num_cpus=1");
G.containerize(B, "memory=1KB", "num_cpus=1");
Executor(G).dispatch();
```

As presented in the above code snippet, we define a function `Ball` that writes a binary data through the stream `k` on vertex `v`. We define another function `PingPong` to retrieve the data arriving in vertex `v` followed by `Ball` if the counter hasn't reached the threshold. We next define vertices and streams using the class method `insert` from the graph, as well as their callbacks based on `Ball` and `PingPong`. The vertex first reaching the threshold will close the underlying stream channels via a return of `Stream::REMOVE_THIS`. This is a handy feature of our system. Users do not need to invoke extra function call to signal our stream back-end. Closing one end of a stream will subsequently force the other end to be closed, which in turn updates the stream ownership on corresponding vertices. We configure each vertex with 1KB memory and 1 CPU. Finally, an executor instance is created to wrap the graph into a closure and dispatch it to the remote master for execution.

C. Advantages of the Proposed Model

DtCraft provides a programming interface similar to those found in C++ standard libraries. Users can learn how to develop a DtCraft application at a faster pace. The same code that executes distributively can be also deployed on a local machine for debugging purpose. No programming changes are necessary except the options passed to the submission script. Note that our framework needs only a *single entity* of executable from users. The system kernel is not intrusive to any user-defined entries, for instance, the arguments passed to the `main` method. We encourage users to describe stream graphs with C++ lambda and function objects. This functional programming style provides a very powerful abstraction that allows the runtime to bind callable objects and captures different runtime states.

Although conventional dataflow thinks applications as "computation vertices" and "dependency edges" [2], [8], [9], [10], our system model does not impose explicit boundary (e.g., DAG restriction). As shown in previous code snippets, vertices and edges are logically associated with each other and are combined to represent generic stream computations including feedback controls, state machines, and asynchronous streaming. Stream computations are by default long-lived and persist in memory until the end-of-file state is lifted. In other words, our programming interface enables straightforward *in-memory* computing, which is an important factor for iterative and incremental algorithms. This feature is different from existing data-driven cluster computing frameworks such as Dryad, Hadoop, and Spark that rely on either frequent disk access or expensive extra caching for data reuse [1], [2], [3]. In addition, our system model facilitates the design of real-time streaming engines. A powerful streaming engine has the potential to bridge the performance gap caused by application boundaries or design hierarchies. It is worth noting that many engineering applications and companies existed "*pre-cloud*", and the most techniques they applied were ad-hoc C/C++ [4]. To improve the engineering turnaround, our system can be explored as a distributed integration of existing developments with legacy codes.

Another powerful feature of our system over existing frameworks is *guided scheduling* using Linux containers. Users can specify *hard* or *soft* constraints configuring the set of Linux containers on which application pieces would

like to run. The scheduler can preferentially select the set of computers to launch application containers for better resource sharing and data locality. While transparent resource control is successful in many data-driven cluster computing systems, we have shown that compute-intensive applications has distinctive computation patterns and resource management models. With this feature, users can implement diverse approaches to various problems in the cluster at any granularity. In fact, we are convinced by our industry partners that the capability of explicit resource controls is extremely beneficial for domain experts to optimize the runtime of performance-critical routines. Our container interface also offers users secure and robust runtime, in which different application pieces are isolated in independent Linux instances. To our best knowledge, DtCraft is the first districuted execution engine that incorporates the Linux container into dataflow programming.

In summary, each system has its own merits in certain application domain, and it is impossible to provide thorough comparison with prior works due to the page limit. However, we believe DtCraft stands out as a unique system given the following attributes: (1) A compute-driven distributed system completely designed from modern C++17. (2) A new asynchronous stream-based programming model in support for general dataflow. (3) A container layer integrated with user-space programming to enable fine-grained resource controls and performance tunning. Developers are encouraged to investigate the structure of their applications and the properties of proprietary systems. Careful graph construction and refinement can improve the performance substantially.

III. System Implementation

DtCraft aims to provide a unified framework that works seamlessly with the C++ standard library. Like many distributed systems, network programming is an integral part of our system kernel. While our initial plan was to adopt third-party libraries, we have found considerable incompatibility with our system architecture (discussed in later sections). Fixing them would require extensive rewrites of library core components. Thus, we decided to re-design these network programming components from ground-up, in particular the event library and serialization interface that are fundamental to DtCraft. We shall also discuss how we achieve distributed execution of a given graph, including scheduling and transparent communication.

A. Event-driven Environment

DtCraft supports event-based programming style to gain benefits from asynchronous computations. Writing an event reactor has traditionally been the domain of experts and the language they obsessed about is C [11]. The biggest issue we found in widely-used event libraries is the inefficient support for object-oriented design and modern concurrency. Our goal is thus to incorporate the power of C++ libraries with low-level system controls such as non-blocking mechanism and I/O polling. Due to the space limit, we present only the key design principles of our event reactor as follows:

```
class Event : enable_shared_from_this <Event> {
  enum Type {TIMEOUT, PERIODIC, READ, WRITE};
  const function<Signal(Event&)> on;
};

class Reactor {
  Threadpool threadpool;
  unordered_set<shared_ptr<Event>> eventset;

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

Unlike existing libraries, our event is a *flattened* unit of operations including timeout and I/O. Events can be customized given the inheritance from class `Event`. The event callback is defined in a *function* object that can work closely with lambda and polymorphic function wrapper. Each event instance is created by the reactor and is only accessible through C++ *smart pointer* with shared ownership among those inside the callback scope. This gives us a number of benefits such as precise polymorphic memory managements and avoidance of ABA problems that are typically hard to achieve with raw pointers. We have implemented the reactor using *task-based* parallelism. A significant problem of existing libraries is the condition handling in multi-threaded environment. For example, a thread calling to insert or remove an event can get a nonsense return if the main thread is too busy to handle the request [11]. To enable proper concurrency controls, we have adopted C++ *future* and *promise* objects to separate the acts between the provider (reactor) and consumers (threads). Multiple threads can thus safely create or remove events in arbitrary orders. In fact, our unit test has shown 4–12× improvements in throughput and latencies over existing libraries [11].

B. Serialization and Deserialization

We have built a dedicated serialization and deserialization layer called *archiver* on top of our stream interface. The archiver has been intensively used in our system kernel communication. Users are strongly encouraged, though not necessary, to wrap their data with our archiver as it is highly optimized to our stream interface. Our archiver is similar to the modern *template-based* library `Cereal`, where data types can be reversibly transformed into different representations such as binary encodings, JSON, and XML [12]. However, the problem we discovered in `Cereal` is the lack of proper size controls during serialization and deserialization. This can easily cause exception or crash when non-blocking stream resources become *partially* unavailable. While extracting the size information in advance requires twofold processing, we have found such burden can be effectively leveraged using modern C++ template techniques. A code example of our binary archiver is given as follows:

```
class BinaryOutputArchiver {
  ostream& os;
  template <typename... U>
  constexpr streamsize operator()(U&&... u) {
    return archive(forward<U>(u)...);
  }
};
```

We developed our archiver based on extensive templates to enable a unified API. Many operations on stack-based objects and constant values are prescribed at compile time using constant expression and forwarding reference techniques. The archiver is a light-weight layer that performs serialization and deserialization of user-specified data members directly on the stream object passed to the callback. We also offer a packager interface that wraps data with a size tag for complete message processing. Both archiver and packager are defined as *callable* objects to facilitate dynamic scoping in our multi-threaded environment.

C. Input and Output Streams

One of the challenges in designing our system is choosing an abstraction for data processing. We have examined various options and concluded that developing a dedicated stream interface is necessary to provide users a simple but robust layer of I/O services. To facilitate the integration of safe and portable streaming execution, our stream interface follows the idea of C++ `istream` and `ostream`. Users are perceived with the API similar to those found in C++ standard library, while our *stream buffer* back-end implements the entire details such as device synchronization and low-level non-blocking data transfers.

Figure 4 illustrates the structure of a stream buffer object in our system kernel. A stream buffer object is a class similar to C++ `basic_streambuf` and consists of three components, *character sequence*, *device*, and *database pointer*. The character sequence is an in-memory linear buffer storing a particular window of the data stream. The device is an OS-level entity (e.g. TCP socket, shared memory) that derives reading and writing methods from an interface class with static polymorphism. Our stream buffer is *thread safe* and is directly integrated with our serialization and deserialization methods. To properly handle the buffer overflow, each stream buffer object is associated with a raw pointer to a database owned by the process. The database is initiated when a master, an agent, or an executor is created, and is shared among all stream buffer objects involved in that process. Unless the ultimate disk usage

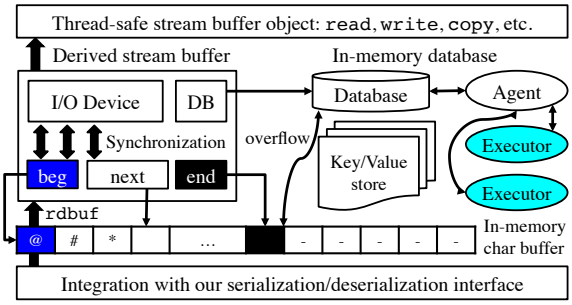


Fig. 4. DtCraft provides a dedicated stream buffer object in control of reading and writing operations on devices.

is full, users are virtually perceived with unbounded stream capacity in no worry about the deadlock.

D. Kernel: Master, Agent, and Executor

Master, agent, and executor are the three major components in the system kernel. There are many factors that have led to the design of our system kernel. Overall regard is the reliability and efficiency in response to different message types. We have defined a reliable and extensible message structure of type *variant* to manipulate a heterogeneous set of message types in a uniform manner. Each message type has data members to be serialized and deserialized by our archiver. The top-level class can inherit from a *visitor* base with dynamic polymorphism and derive dedicated handlers for certain message types.

To efficiently react to each message, we have adopted the event-based programming style. Master, agent, and executor are persistent objects derived from our reactor with specialized events binding to each. While it is expectedly difficult to write non-sequential codes, we have found a number of benefits of adopting event-driven interface, for instance, asynchronous computations, natural task flow controls, and concurrency. We have defined several master events in charge of graph scheduling and status report. For agent, most events are designated as a proxy to monitor current machine status and fork an executor to launch tasks. Executor events are responsible for the communication with the master and agents as well as the encapsulation of asynchronous vertex and edge events. Multiple events are executed efficiently on a shared thread pool in our reactor.

1) Communication Channels: The communication channels between different components in DtCraft are listed in Table I. By default, DtCraft supports three types of communication channels, *TCP socket* for network communication between remote hosts, *domain socket* for process communication on a local machine, and *shared memory* for in-process data exchange. For each of these three channels, we have implemented a unique device class that effectively supports non-blocking I/O and error handling. Individual device classes are pluggable to our stream buffer object and can be extended to incorporate device-specific attributes for further I/O optimizations.

TABLE I
COMMUNICATION CHANNELS IN DTCRAFT.

Target	Protocol	Channel	Latency
Master–User	TCP socket	Network	High
Master–Agent	TCP socket	Network	High
Agent–Executor	Domain socket	Local processes	Medium
Intra-edge	Shared memory	Within a process	Low
Inter-edge	TCP socket	Network	High

Since master and agents are coordinated with each other in distributed environment, the communication channels run through reliable TCP socket streams. We enable two types of communication channels for graphs, shared memory and TCP socket. As we shall see in the next section, the scheduler might partition a given graph into multiple topologies running on different agent nodes. Edges crossing the partition boundary are communicated through TCP sockets, while data within a topology is exchanged through shared memory with extremely low latency cost. To prevent our system kernel from being bottlenecked by data transfers, master and agents are only responsible for control decisions. All data is sent between vertices managed by the executor. Nevertheless, achieving point-to-point communication is non-trivial for inter-edges. The main reason is that the graph structure is offline unknown and our system has to be general to different communication patterns deployed by the scheduler. We have managed to solve this by means of file descriptor passing through environment variables. The agent exports a list of open file descriptors to an environment variable which will be in turn inherited by the corresponding executor under fork.

2) Application Container: DtCraft leverages existing OS container technologies to enable isolation of application resources from one another. Because these technologies are platform-dependent, we implemented a *pluggable isolation module* to support multiple isolation mechanisms. An isolation module containerizes a process based on user-specified attributes. By default, we apply the Linux *control groups* (cgroups) kernel feature to impose per-resource limits (CPU, memory, block I/O, and network) on user applications. With cgroups, we are able to consolidate many workloads on a single node while guaranteeing the quota assigned to each application. In order to achieve secure and robust runtime, our system runs applications in isolated *namespaces*. We currently support IPC, network, mount, PID, UTS, and user namespaces. By essentially separating processes into independent namespaces, user applications are ensured to be invisible from others and will be unable to make connections outside of the namespaces. External connections such as inter-edge streaming are managed by agents through device descriptor passing techniques. Our container implementation also supports process snapshots, which is beneficial for checkpointing and live migration.

3) Graph Scheduling: Scheduler is an asynchronous master event that is invoked when a new graph arrives. Given a user-submitted graph, the goal of the scheduler is to find a deployment of each vertex and each edge considering the machine loads and resource constraints. A graph might be partitioned into a set of *topologies* that can be accommodated by the present resources. A topology is the basic unit of a task (container) that is launched by an executor process on an agent node. A topology is not a graph because it may contain dangling edges along the partition boundary. Once the scheduler has decided the deployment, each topology is marshaled along with graph parameters including the UUID, resource requirements, and input arguments to form a closure that can be sent to the corresponding agent for execution. An example of the scheduling process is shown in Figure 5. At present, two schedulers persist in our system, a *global scheduler* invoked by the master and a *local scheduler* managed by the agent. Given user-configured containers, the global scheduler performs resource-aware partition based on the assumption that the graph must be completely deployed at one time. The global scheduling problem is formulated into a bin packing optimization where we additionally take into account the number of edge cuts to reduce the latency. An application is rejected by the global scheduler if its mandatory resources (must acquire in order to run) exceed the maximum capability of machines. As a graph can be partitioned into different topologies, the goal of the local scheduler is to synchronize all inter-edge connections of a topology and dispatch it to an executor. The local scheduler is also responsible for various container setups including resource update, namespace isolation, and fault recovery.

Although our scheduler does not force users to explicitly containerize applications (resort to our default heuristics), empowering users fine-grained controls over resources can guide the scheduler toward tremendous performance gain. Due to the space limitation, we are unable to discuss the entire details of our schedulers. We believe developing a scheduler for distributed dataflow under multiple resource constraints deserves independent research effort. As a result, DtCraft delegates the scheduler implementation to a pluggable module that can be customized by organizations for their purposes.

4) Topology Execution: When the agent accepts a new topology, a special asynchronous event, *topology manager*, is created to take over the task. The topology manager spawns (fork-exec) a new executor process based on the parameters extracted from the topology, and coordinates with the executor until the task is finished. Because our kernel requires only a single entity of executable, the executor is notified by which execution mode to run via

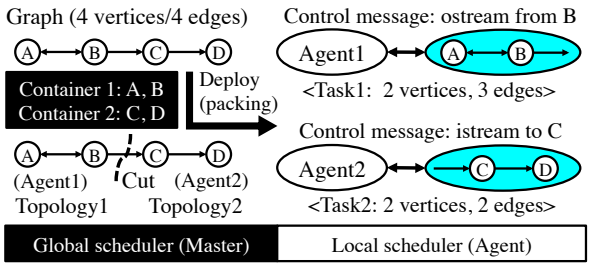


Fig. 5. An application is partitioned into a set of topologies by the global scheduler, which are in turn sent to remote agents (local scheduler) for execution.

environment variables. In our case, the topology manager exports a variable to "*distributed*", as opposed to aforementioned "*submit*" where the executor submits the graph to the master. Once the process controls are finished, the topology manager delivers the topology to the executor. A set of executor events is subsequently triggered to launch asynchronous vertex and edge events.

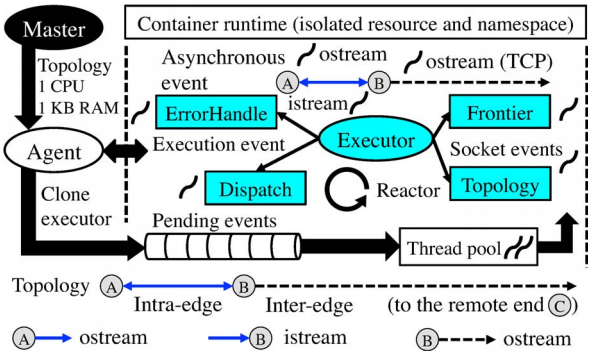


Fig. 6. A snapshot of the executor runtime in distributed mode.

A snapshot of the executor runtime upon receiving a topology is shown in Figure 6. Roughly speaking, the executor performs two tasks. First, the executor initializes the graph from the given topology which contains a key set to describe the graph fragment. Since every executor resides in the same executable, an intuitive method is to initialize the whole graph as a parent reference to the topology. However, this can be cost-inefficient especially when vertices and edges have expensive constructors. To achieve a generally effective solution, we have applied lazy lambda technique to suspend the initialization (see the code below). The suspended lambda captures all required parameters to construct a vertex or an edge, and is lazily invoked by the executor runtime. By referring to a topology passed from the "future", only necessary vertices and edges will be constructed.

```
template <typename V, typename... U>
VertexDescriptor Graph::insert_vertex(U&&... u) {
  auto key = generate_key();  // deterministic key
  tasks.emplace_back(  // lazy initialization
    [u..., key](Topology* t) {
      // local mode and distributed mode
      if(t == nullptr || t->has_key(key)) {
        auto v = make_shared<V>(u...);
        pm.set_value(move(v));
      }
      else t->insert(key);  // submit mode
    }
  );
  return key;
}
```

The second task is to initiate a set of events for vertex and edge callbacks. We have implemented an I/O event for each device based on our stream buffer object. Because each vertex callback is invoked only once, it can be absorbed into any adjacent edge events coordinated by modern C++ threading `once_flag` and `call_once`. Given an initialized graph, the executor iterates over every edge and creates shared memory I/O events and TCP socket I/O events for intra-edges and inter-edges, respectively. Notice that the device descriptor for inter-edges are fetched from the environment variables inherited from the agent.

IV. Fault Tolerance Policy

Our system architecture facilitates the design of fault tolerance on two fronts. First, master maintains a centralized mapping between active applications and agents. Every single error, which could be either heartbeat timeout on the executor or unexpected I/O behaviors on the communication channels, can be properly propagated. In case of a failure, the scheduler performs a linear search to terminate and re-deploy the application. Second, our container implementation can be easily extended to support periodic checkpointing. Executors are freezed to a stable state and are thawed after the checkpointing. The solution might not be perfect, but adding this functionality is already an advantage over our system framework, where all data transfers are exposed to our stream buffer interface and can be dumped without lost.

V. Experimental Results

We have implemented DtCraft in C++17 on a Linux machine with GCC 7. Given the huge amount of existing cluster computing frameworks, we are unable to conduct comprehensive comparison subject to the space limit. Instead, we compare with one of the best cluster computing engines, Apache Spark 2.0 [3], that has been extensively studied by other research works as baseline. To further investigate the benefit of DtCraft, we compared with an application hand-crafted with domain-specific optimizations [7]. The performance of DtCraft is evaluated on three sets of experiments. The first two experiments took classic algorithms from machine learning and graph applications and compared the performance of DtCraft with Spark. We have analyzed the runtime performance over different numbers of machines on an academic cluster [13]. The third experiment applied DtCraft to solve a large-scale semiconductor design problem. Our goal is to explore DtCraft as a distributed solution to mitigate the end-to-end engineering efforts along the design flow. The evaluation has been undertaken on a large cluster in Amazon EC2 cloud [14]. Overall, we have shown the performance and scalability of DtCraft on both standalone applications and cross-domain applications that have been coupled together in a distributed manner.

A. Machine Learning

We implemented two iterative machine learning algorithms, logistic regression and k-means clustering, and compared our performance with Spark. One key difference between the two applications is the amount of computation they performed per byte of data. The iteration time of k-means is dominated by computations, whereas logistic regression is less compute-intensive [3]. The source codes we used to run on Spark are cloned from the official repository of Spark. For the sake of fairness, the DtCraft counterparts are implemented based on the algorithms of these Spark codes.

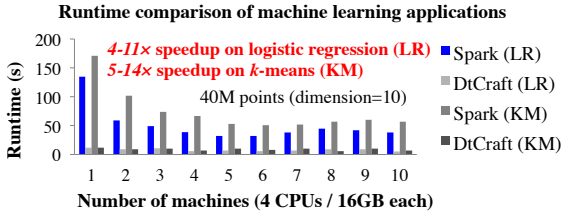


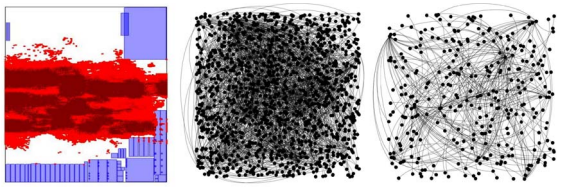
Fig. 7. Runtimes of DtCraft versus Spark on logistic regression and k-means.

Figure 7 shows the runtime performance of DtCraft versus Spark. Unless otherwise noted, the value pair enclosed by the parenthesis (CPUs/GB) denotes the number of cores and the memory size per machine in our cluster. We ran

both logistic regression and k-means for 10 iterations on 40M sample points. It can be observed that DtCraft outperformed Spark by 4–11× and 5–14× faster on logistic regression and k-means, respectively. Although Spark can mitigate the long runtime by increasing the cluster size, the performance gap to DtCraft is still remarkable (up to 8× on 10 machines). In terms of communication cost, we have found hundreds of Spark RDD partitions shuffling over the network. In order to avoid disk I/O overhead, Spark imposed a significant burden on the first iteration to cache data for reusing RDDs in the subsequent iterations. In contrast, our system architecture enables straightforward *in-memory* computing, incurring no extra overhead of caching data on any iterations. Also, our scheduler can effectively leverage the machine overloads along with network overhead for higher performance gain.

B. Graph Algorithm

We next examine the effectiveness of DtCraft by running a graph algorithm. Graph problems are challenging in concurrent programming due to the iterative, incremental, and irregular computing patterns. We considered the classic shortest path problem on a circuit graph with 10M nodes and 14M edges released by [7]. We implemented the Pregel-style shortest path finding algorithm in DtCraft, and compared it with Spark-based Pregel variation downloaded from the official GraphX repository [10].



(a) Circuit (1.01mm^2) (b) Graph (3M gates) (c) A signal path

Fig. 8. Visualization of our graph benchmark.

Figure 9 shows the runtime comparison across different machine counts. In general, DtCraft reached the goal by 10–20× faster than Spark. Our program can finish all tests within a minute regardless of the machine usage. We have observed intensive network traffic among Spark RDD partitions whereas in our system most data transfers were effectively scheduled to shared memory. To further examine the runtime scalability, we duplicated the circuit graph and created random links to form larger graphs, and compared the runtimes of both systems on different graph sizes. As shown in Figure 10, the runtime curve of DtCraft is far scalable against Spark. The highest speedup is observed at the graph of size 240M, in which DtCraft is 17× faster than Spark. To summarize this micro-benchmarking, we believe the performance gap between Spark and DtCraft is due to the system architecture and language features we have chosen. While we compromise with users on explicit dataflow description, the performance gain in exchange can scale up to more than an order of magnitude over one of the best cluster computing systems.

C. Electronic Design Automation (EDA)

The recent semiconductor industry is driving the need of massively-parallel integration to leverage the technology scaling [4]. We applied DtCraft to solve a large-scale EDA optimization problem, *physical design*, a pivotal stage that encompasses several steps from circuit partition to timing closure (see Figure 11). Each step has domain-specific solutions and engages with others through different internal databases. We used open-source tools and our internal developments for each step of the physical design [15], [6]. Individual tools have been developed based on C++ with default I/O on files, which can fit into DtCraft without significant rewrites of codes. Altering the I/O channels is unsurprisingly straightforward because our stream interface is compatible with C++ file streams. We applied DtCraft to handle a typical physical design cycle under multiple timing scenarios. As shown in Figure 12, our implementation ran through each physical design step and coupled them together in a distributed manner. Generating the timing report is the most time-consuming step. We captured each independent timing scenario by one vertex and connected it to a synchronization barrier to derive the final result. Users can interactively access the system via a service vertex.

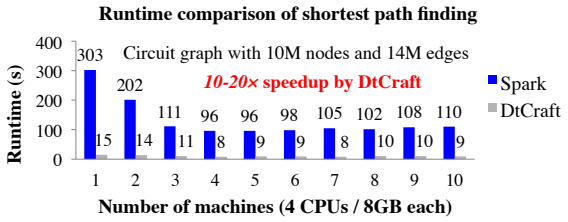


Fig. 9. Runtimes of DtCraft versus Spark on finding a shortest path in our circuit graph benchmark.

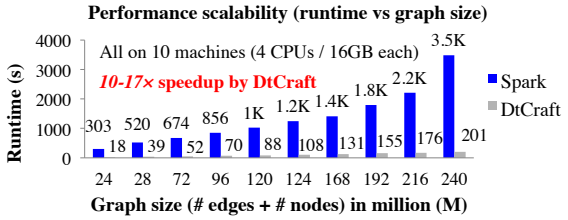


Fig. 10. Runtime scalability of DtCraft versus Spark on different graph sizes.

We derived a benchmark with two billion transistors from ICCAD15 and TAU15 contests [15]. The DtCraft-based solution is evaluated on 40 Amazon EC2 m4.xlarge machines [14]. The baseline we considered is a batch run over all steps on a single machine that mimicked the normal design flow. The overall performance is shown in Figure 13. The first benefit of our solution is the saving of disk I/O (65 GB vs 11 GB). Most data is exchanged on the fly including those that would otherwise come with redundant auxiliaries through disk (50 GB parasitics in the timing step). Another benefit we have observed is the asynchrony of DtCraft. Computations are placed wherever stream fragments are available rather than blocking for the entire object to be present. These advantages have translated to effective engineering turnaround – *13 hours saving over the baseline*. From designers' perspective, this value convinces not only a faster path to the design closure but also the chance for breaking cumbersome design hierarchies, which has the potential to tremendously improve the overall solution quality [7], [4].

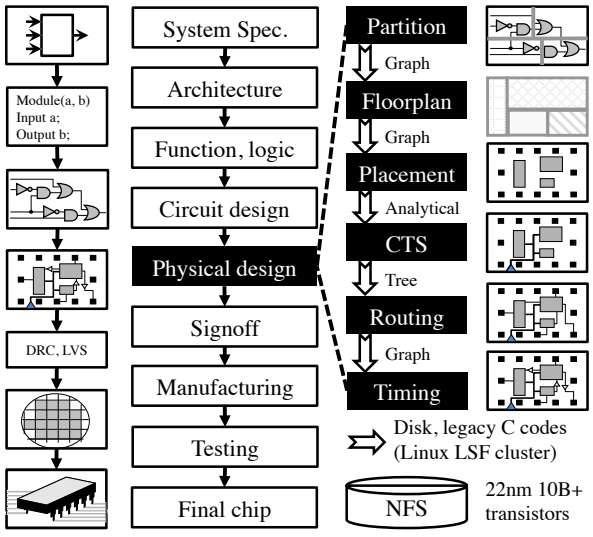


Fig. 11. Electronic design automation of VLSI circuits and optimization flow of the physical design stage.

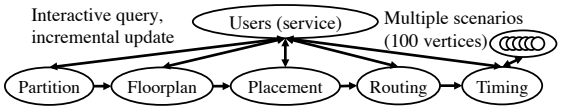


Fig. 12. Stream graph (106 vertices and 214 edges) of our DtCraft-based solution for the physical design flow.

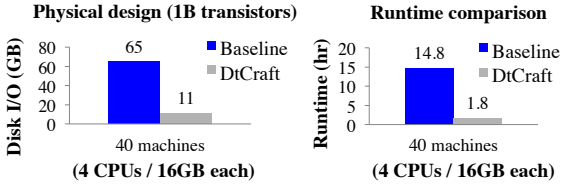


Fig. 13. Performance of DtCraft versus baseline in completing the physical design flow.

We next demonstrate the speedup relative to the baseline on different cluster sizes. In addition, we included the experiment in presence of a failure to demonstrate the fault tolerance of DtCraft. One machine is killed at a random time step, resulting in partial re-execution of the stream graph. As shown in Figure 14, the speedup of DtCraft scales up as the cluster size increases. The highest speedup is achieved at 40 machines (160 cores and 640 GB memory in total), where DtCraft is $8.1\times$ and $6.4\times$ faster than the baseline. On the other hand, we have observed approximately 10–20% runtime overhead on fault recovery. We did not see pronounced difference from our checkpoint-based fault recovery mechanism. This should be in general true for most EDA applications since existing optimization algorithms are designed for "*medium-size data*" (million gates per partition) to run in main memory [7], [4]. In terms of runtime breakdown, computation takes the majority while about 15% is occupied by system transparency.

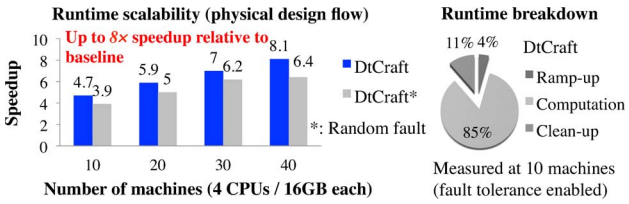


Fig. 14. Runtime scalability in terms of speedup relative to the baseline on different cluster sizes.

Since timing analysis exhibits the most parallelism, we investigate into the performance gain by using DtCraft. To discover the system capability, we compare with the distributed timing analysis algorithm (ad-hoc approach) proposed by [7]. To further demonstrate the programmability of DtCraft, we compared the code complexity in terms of the number of lines of codes between our implementation and the ad-hoc approach. The overall comparison is shown in Figure 15. Because of the problem nature, the runtime scalability is even remarkable as the compute power scales out. It is expected the ad-hoc approach is faster than our DtCraft-based solution. Nevertheless, the ad-hoc approach embedded many hard codes and supports neither transparent concurrency nor fault tolerance, which is difficult for scalable and robust maintenance. In terms of programmability, our programming interface can significantly reduce the amount of the codes by $15\times$. The corresponding engineering efforts can be far beyond this number. Although this comparison might not be fair, it indeed reflected the potential engineering productivity that can be improved by DtCraft.

To conclude this experiment, we have introduced a platform innovation to solve a large-scale semiconductor optimization problem with low integration cost. To our best knowledge, this is the first work in the literature that achieves a distributed EDA flow integration. In addition, DtCraft also opens new

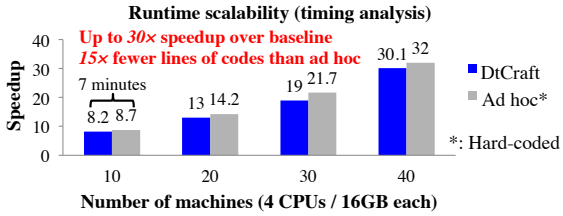


Fig. 15. Performance comparison on distributed timing analysis between DtCraft-based approach and the ad-hoc algorithm by [7].

opportunities for improving commercial tools, for example, distributed EDA algorithms and tool-to-tool integration. While this experiment demonstrates merely a successful prototype, we believe DtCraft can be extended to consider more general and complex design flows.

VI. Conclusion

We have presented DtCraft, a distributed execution engine for high-performance parallel applications. DtCraft is developed based on modern C++17 on Linux machines. Developers can fully utilize rich features of C++ standard libraries along with our parallel framework to build highly-optimized applications. Experiments on classic machine learning and graph applications have shown DtCraft outperforms the state-of-the-art cluster computing system by more than an order of magnitude. We have also successfully applied DtCraft to solve large-scale semiconductor optimization problems that are known difficult to fit into existing big data ecosystems. For many similar industry applications, DtCraft can be employed to explore integration and optimization issues, thereby offering new revenue opportunities for existing company assets. We plan to open the source of DtCraft as a vehicle for system research.

Acknowledgment

This work is partially supported by the National Science Foundation under Grant CCF-1421563 and CCF-171883. The authors thank the IBM Timing Analysis Group and the EDA group in UIUC for their helpful discussion to inspire this work.

References

[1] "Apache hadoop," http://hadoop.apache.org/.
[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *ACM EuroSys*, 2007, pp. 59–72.
[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*, vol. 31, no. 2, pp. 40–46, April 2014.
[5] "The future of big data," https://www2.eecs.berkeley.edu/patterson2016/.
[6] T.-W. Huang and M. D. F. Wong, "Opentimer: A high-performance timing analysis tool," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.
[7] T.-W. Huang, M. D. F. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A distributed timing analysis framework for large designs," in *ACM/IEEE DAC*, 2016, pp. 116:1–116:6.
[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," *Parallel Comput.*, vol. 38, no. 1-2, pp. 37–51, Jan. 2012.
[9] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Caf - the c++ actor framework for scalable and resource-efficient applications," in *ACM AGERE!*, 2014, pp. 15–28.
[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *ACM SIGMOD*, 2010, pp. 135–146.
[11] "Libevent," http://libevent.ogr/.
[12] "Cereal," http://uscilab.github.io/cereal/index.html.
[13] "Illinois campus cluster program," https://campuscluster.illinois.edu/.
[14] "Amazon ec2," https://aws.amazon.com/ec2/.
[15] "Tau contest," https://sites.google.com/site/taucontest2016/resources.