Data Processing at the Speed of 100 Gbps using Apache Crail

Patrick Stuedi
IBM Research
Apache Crail (incubating) is a high-performance distributed data store designed for fast sharing of ephemeral data in distributed data processing workloads.

Download Now
Apache Crail (Incubating) is a high-performance distributed data store designed for fast sharing of ephemeral data in distributed data processing workloads.
Ephemeral Data

Input data: HDFS, S3 → Broadcast → Map-reduce job → Shuffle → Reduce → Output data: HDFS, S3
Ephemeral Data

Map-reduce job

Input data

HDFS, S3

Broadcast

Map

Shuffle

Reduce

Output data

HDFS, S3
Ephemeral Data

Input data

Broadcast
Map
Shuffle
Reduce

Apache Crail

Output data

HDFS, S3

HDFS, S3
Ephemeral Data

Apache Crail

HDFS, S3

Input data

Broadcast
Map
Shuffle
Reduce

Intermediate data

HDFS, S3

HDFS, S3
Ephemeral Data

Input data

ML pre-processing (map-reduce job)

normalized images

ML training (Tensorflow job)

Apache Crail

HDFS, S3
Ephemeral Data

Apache Crail

Input data

ML pre-processing (map-reduce job)

normalized images

ML training (Tensorflow job)

HDFS, S3

normalized images

HDFS, S3

HDFS, S3
Ephemeral Data

Input data

ML pre-processing (map-reduce job)

normalized images

ML training (Tensorflow job)

Apache Crail

HDFS, S3
Why/when to use Crail
Why/when to use Crail

No Crail needed

HDD

100MB/s
10ms

10 Gb/s Ethernet

10 Gb/s
20us
Why/when to use Crail

- No Crail needed
- 100GB/s Ethernet
- 100 MB/s 10ms
- 10 Gb/s 20us

100x

Crail land

- 10 GB/s 10us
- 200 GB/s 1us

HDD

10 Gb/s Ethernet

100 Gb/s Infiniband

3D XPoint Memory

Memory

nvm EXPRESS

RDMA
Why/when to use Crail

No Crail needed

100x

Spark/Crail

Spark/Vanilla

Terasort

12.8 TB data

128 nodes
Performance Challenge

<table>
<thead>
<tr>
<th></th>
<th>1 Gbps</th>
<th>HDD</th>
<th>100 Gbps</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth</td>
<td>117 MB/s</td>
<td>140 MB/s</td>
<td>12.5 GB/s</td>
<td>3.1 GB/s</td>
</tr>
<tr>
<td>cycle/unit</td>
<td>38,400</td>
<td>10,957</td>
<td>360</td>
<td>495</td>
</tr>
</tbody>
</table>

Sorting Application
- Data Processing Framework
  - sockets
  - filesystem
  - TCP/IP
  - block layer
  - Ethernet
  - iSCSI
  - NIC
  - SSD

Sorter
Serializer
Netty
JVM
Performance Challenge

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>1 Gbps</th>
<th>HDD</th>
<th>100 Gbps</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth</td>
<td>117 MB/s</td>
<td>140 MB/s</td>
<td>12.5 GB/s</td>
<td>3.1 GB/s</td>
</tr>
<tr>
<td>cycle/unit</td>
<td>38,400</td>
<td>10,957</td>
<td>360</td>
<td>495</td>
</tr>
</tbody>
</table>

- Sorting Application
- Data Processing Framework
- sockets
- filesystem
- TCP/IP
- block layer
- Ethernet
- iSCSI
- NIC
- SSD

Process chunk
In reduce task

Fetch chunk
Over the network

HotNets’16

- 1 Gbps
- 10 Gbps
- 40 Gbps
Performance Challenge

<table>
<thead>
<tr>
<th></th>
<th>1 Gbps</th>
<th>HDD</th>
<th>100 Gbps</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth</td>
<td>117 MB/s</td>
<td>140 MB/s</td>
<td>12.5 GB/s</td>
<td>3.1 GB/s</td>
</tr>
<tr>
<td>cycle/unit</td>
<td>38,400</td>
<td>10,957</td>
<td><strong>360</strong></td>
<td><strong>495</strong></td>
</tr>
</tbody>
</table>

Sorting Application
- Data Processing Framework
  - sockets
  - filesystem
  - TCP/IP
  - Ethernet
  - iSCSI
- NIC
- SSD

Sorter
Serializer
Netty
JVM

HotNets’16
Performance Challenge

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>1 Gbps</th>
<th>HDD</th>
<th>100 Gbps</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle/unit</td>
<td>38,400</td>
<td>10,957</td>
<td>360</td>
<td>495</td>
</tr>
</tbody>
</table>

Software overhead are spread over the entire stack.

HotNets’16
Crail Overview

Multiple interfaces

Multiple storage backends (pluggable, open interface)

Data Processing Framework (e.g., Spark, TensorFlow, λ Compute)

Apache Crail (Incubating)

Fast Network, e.g., 100 Gbps RoCE

DRAM, NVMe, PCM, GPU

100 Gbps
10 μsec
Crail Overview

- Multiple interfaces
- Multiple storage backends (pluggable, open interface)
- Primary high-performance storage backends

Data Processing Framework (e.g., Spark, TensorFlow, λ Compute)

- FS
- Streaming
- KV
- HDFS

100 Gbps
10 μsec

Apache Crail (Incubating)

- TCP
- RDMA
- NVMeF
- SPDK

Fast Network, e.g., 100 Gbps RoCE

- DRAM
- NVMe
- PCM
- GPU
Crail Architecture & API

- MultiFile
  - Table Node
    - KeyValue Node
      - FileNode
        - Directory Node

- Application interface

- optimized for shuffle data
- key-value semantics
- append-only file
Crail Architecture & API

Java:

```java
CrailStore crail = CrailStore.newInstance();
Future<Node> fut = crail.create("/a.dat", CrailType.File);
//... do work
CrailFile file = fut.get().asFile();
CrailOutputStream stream = file.getDirectOutputStream();
ByteBuffer buffer = crail.allocateBuffer();
Future<CrailResult> ret = stream.write(buf);
//... do work
ret.get();
```

C++:

```cpp
CrailStore crail;
auto fut = crail.Create<CrailFile>("/tmp.dat");
//... do work
CrailFile file = fut.get();
CrailOutputStream stream = file.getOutputStream();
shared_ptr<ByteBuf> buf = make_shared<ByteBuf>(len);
Future<int> ret = stream.Write(buf);
//... do work
ret.get();
```
Crail Architecture & API

Java:

```java
crailStore crail = CrailStore.newInstance();
Future<
Node> fut = crail.create("/a.dat", CrailType.File);
//...do work
CrailFile file = fut.get().asFile();
CrailOutputStream stream = file.getDirectOutputStream();
ByteBuffer buf = crail.allocateBuffer();
Future<
CrailResult> ret = stream.write(buf);
//...do work
ret.get();
```

C++:

```cpp
crailStore crail;
auto fut = crailCreate<
CrailFile>("/tmp.dat");
//...do work
CrailFile file = fut.get();
CrailOutputStream stream = file.getOutputStream();
shared_ptr<
ByteBuffer> buf = make_shared<
ByteBuffer>(len);
Future<
int> ret = stream.write(buf);
//...do work
ret.get();
```
Crail Architecture & API

Java:
```java
CrailStore crail = CrailStore.newInstance();
Future<Node> fut = crail.create("/a.dat", CrailType.File);
//...do work
CrailFile file = fut.get().asFile();
CrailOutputStream stream = file.getDirectOutputStream();
ByteBuffer buffer = crail.allocateBuffer();
Future<CrailResult> ret = stream.write(buf);
//...do work
ret.get();
```

C++:
```cpp
CrailStore crail;
auto fut = crail.CreateCrailFile("/tmp.dat");
//...do work
CrailFile file = fut.get();
CrailOutputStream stream = file.getOutputStream();
shared_ptr<ByteBuffer> buf = make_shared<ByteBuffer>(len);
Future<int> ret = stream.write(buf);
//...do work
ret.get();
```
Where does the performance come from?
User-Level I/O: Metadata

1. auto fut = crail.Create<CrailFile>("/tmp.dat");
   //..do work
2. CrailFile file = fut.get();
User-Level I/O: Metadata

1. `auto fut = crail.Create<CrailFile>("/tmp.dat"); //...do work`
2. `CrailFile file = fut.get();`

No threads
No context switches
User-Level I/O: Data

1. Future<int> ret = stream.Read(buf);
   /// do work
   ret.get();

2. 

Diagram:
- NVMf server
  - Staging buffers
  - Linux
  - DMA

- DRAM server
  - Storage blocks
  - Linux
  - DMA
  - RDMA read

- Application
  - Data Buffer
  - send queue, recv queue, completion queue
  - DMA

- Crail client
  - NVMf "read"
User-Level I/O: Data

1. `Future<int> ret = stream.Read(buf);`  // do work
   `ret.get();`

2. Zero-copy, transfer only data that is requested.

Diagram:
- **NVMe server**
  - Staging buffers
  - DMA
- **DRAM server**
  - Storage blocks
  - DMA
- **Application**
  - Data Buffer
  - DMA
  - send queue, recv queue, completion queue
- **Client**
  - RDMA read
  - NVMf "read"
Crail Deployment Modes

- compute/storage co-located
- storage disaggregation
- flash storage disaggregation
YCSB KeyValue Workload

Crail offers Get latencies of ~12us and 30us for DRAM and NVM for 100 byte KV pairs
Crail offers Get latencies of ~30us and 40us for DRAM and NVM for 1000 byte KV pairs

Crail offers Get latencies of ~12us and 30us for DRAM and NVM for 100 byte KV pairs
Crail offers Get latencies of ~30us and 40us for DRAM and NVM for 1000 byte KV pairs
Spark GroupBy (80M keys, 4K)

Spark shuffling via Crail on a single core is 2x faster than vanilla Spark on 8 cores per executor (8 executors)
Crail enables disaggregation of temporary data at no cost
Using flash only increases the sorting time by around 48%
Conclusions

• Apache Crail: Fast distributed “tmp”
  - User-level I/O
  - Storage disaggregation
  - Memory/flash convergence

• Applications
  - Intra-job scratch space (shuffle, broadcast, etc.)
  - Multi-job pipelines

• Coming soon
  - Native Crail (C++)
  - Tensorflow-Crail