Accelerating Complex Data Transfer for Cluster Computing

Alexey Khrabrov, Eyal de Lara
University of Toronto

HotCloud 2016
Motivation

• Data processing is now CPU-bound

• Software layers can’t leverage fast datacenter networks
  – network responsible for as low as 2% of overall performance

• Data [de]serialization is one of the bottlenecks
  – up to 26% of total CPU time
  – prevents from fully leveraging RDMA
Serialized data transfer

Source Node

Destination Node

Serialization

Transfer

Deserialization

Object1

Object2

Object3

Field1

Field2

Pointer1

Pointer2

Header

Auxiliary Info

Data

Node 1

Node 2
Transfer time breakdown: complex data

TreeMap; size: 64 MB raw, 24 MB serialized; 10 Gbit/s

80% overhead
(for 100 Gbit/s – 97%)
Transfer time breakdown: simple data

double[]; size: 80 MB; 10 Gbit/s

65% overhead
Eliminating data [de]serialization

• Reason: pointer-based data structures become invalid when copied directly to another address space
  – other reasons (e.g. different endianness) are irrelevant: assume that all nodes have the same architecture
• General idea: shared cluster-wide virtual address space
• Compact allocation of objects to be copied together
  – continuous regions copied in a single operation – RDMA-friendly
Compact object format and Direct transfer

Source Node

Global Heap Object

Destination Node

Transfer
Cluster-wide shared address space

• Virtual address space is huge -> can be shared
  – 128 TB ($2^{47}$), potentially $2^{63}$ bytes
• Limited version of DSM (distributed shared memory)

• DSM original goal: trade off performance for transparency / ease of programming
• We use DSM to *improve* performance (but increase programming complexity)
Assumptions

• Immutable shared objects
  – modifications of the original are not propagated
  – not very restrictive: e.g. immutable RDDs in Spark

• No need to be completely transparent to programmer
  – explicit management of global objects
  – possible to hide most of the details inside the framework
GObject obj = new GObject(...);

obj.data = new MyFancyClass(...);

//...

obj.commit("key");

//...

obj.release();

GObject obj = GHeap.get("key");

MyFancyClass data = obj.data;

//...

obj.release();
Global heap architecture

• Huge *virtual* address space region; the same on all nodes
• Partitioning: nodes allocate objects in own exclusive regions
  – minimal amount of coordination required
• Mapping to physical memory on demand
• Objects identified by keys mapped to <node, vaddr>
• 3-stage object creation: (1) reserve space; (2) populate with data; (3) commit (make available to other nodes)
• Explicit release of objects
JVM-based implementation

• Prototype based on JamVM
  – HotSpot ("standard" JVM) – in progress
• Most of functionality implemented in native methods
• Still need some JVM modifications
  – memory allocator / garbage collector
  – object header format
  – bytecode interpreter / JIT compiler
• Details: in the paper
Evaluation

• Microbenchmark (performance of the mechanism alone)
• Transfer objects between 2 identical nodes
• Direct copy vs. serialized
  – both standard Java serialization and Kryo
• HotSpot for serialized measurements, JamVM for direct copy
• TCP transport, 10 Gbit/s; expect better results with RDMA

• Overhead of JVM modifications: within 1%
Evaluation: complex data (TreeMap)

- Serialized
- Kryo serialized
- Direct copy

Transfer time, seconds vs. Raw data size, MB

10x improvement

5.5x improvement
Evaluation: simple data (double[])
Evaluation: small simple objects

Transfer time, milliseconds

Raw data size, bytes

- Red: Serialized
- Blue: Kryo serialized
- Green: Direct copy
Proposed applications

• Data processing frameworks: Spark, Hadoop, etc.
  – optimize shuffle stages (data exchange between all nodes)
  – possible scheduling improvements; data migration is now cheaper

• Distributed in-memory storage
  – store complex data efficiently
  – reduce latency of set/get operations

• Fast IPC and RPC
  – zero-copy within one machine (using shared memory)
Current and future work directions

• Applications and macrobenchmarks
• RDMA
• Reliability / fault tolerance
• Storage considerations (spills to disk)
• Multiple address spaces for extremely large datasets
• Global heap space management, other implementation details...
Conclusion

• Data [de]serialization is a bottleneck; doesn’t let us fully leverage fast network
• Designed a data transfer mechanism to avoid serialization
  – main idea: shared cluster-wide virtual address space
• Use DSM to improve performance, trading off increased programming complexity
• Evaluation shows significant (up to 10x) speedup of data transfer
• Will explore applications that can benefit from this mechanism
Questions?