Accelerating Complex Data Transfer for Cluster Computing

Alexey Khrabrov
University of Toronto

Eyal de Lara
University of Toronto

Abstract

The ability to move data quickly between the nodes of a distributed system is important for the performance of cluster computing frameworks, such as Hadoop and Spark. We show that in a cluster with modern networking technology data serialization is the main bottleneck and source of overhead in the transfer of rich data in systems based on high-level programming languages such as Java. We propose a new data transfer mechanism that avoids serialization altogether by using a shared cluster-wide address space to store data. The design and a prototype implementation of this approach are described. We show that our mechanism is significantly faster than serialized data transfer, and propose a number of possible applications for it.

1 Introduction

Modern distributed computational systems are becoming more and more CPU-bound [12]. In contrast with the common belief that I/O (network and disk) operations are usually the main bottleneck, new network technologies, such as InfiniBand, allow bandwidths of up to 100 Gbit/s, and the advent of RDMA (remote direct memory access) has drastically shrunk the latency gap between remote and local memory accesses.

In this paper, we show that for distributed applications running modern clusters the main bottleneck for data transfer is data serialization, which is a CPU-bound operation. This is especially the case for complex data structures used by high-level programming languages.

Data serialization is the process of translating objects into an intermediate format such that the object can be reconstructed from it on another machine. Many of the reasons for serialization, such as the ability to transfer data between machines with different word size or endianness, are becoming irrelevant. We can assume that a modern cluster consists of nodes with the same hardware architecture. The only real reason for serialization is that pointer-based data structures become invalid when copied directly to another address space.

This paper introduces a novel fast data transfer mechanism that avoids data serialization altogether by using a shared cluster-wide address space to store objects that can be transferred between nodes. This approach preserves the validity of object pointers even as they get copied to another node. The objects are allocated in a special "compact" manner. They occupy contiguous regions of address space (similar to the serialized representation), which makes them amenable for fast network transfer, especially with RDMA.

The idea of a shared cluster-wide address space has been around for many years. An extreme version even uses the same address space for disk storage [13]. Distributed shared memory (DSM) systems reduce programming complexity by providing a consistent single-image view of the system. Effectively, these approaches trade performance for transparency. In contrast, we forego transparency in favour of performance. We require programmers to manage objects explicitly, and only leverage shared memory to increase performance by allowing direct copy of pointer-based data structures between nodes.

Many of the modern cluster computing systems (such as Hadoop [1] and Spark [2]) are based on Java Virtual Machine (JVM) languages such as Java and Scala. That is why our implementation targets the JVM, although the idea of our direct object transfer mechanism is not specific to any programming language or platform. The implementation involves modifying the JVM, the memory management subsystem in particular.

We have implemented this novel object transfer mechanism for the JVM and evaluated its performance. It can speedup object transfer up to 5× compared to optimized serialization using Kryo [4] and up to 10× compared to standard Java serialization, on a 5.5 Gbit/s network.

The rest of the paper is organized as follows. In Section 2, we discuss the motivation behind the paper

and present an analysis of Java serialization performance compared to potential performance of direct transfer. Section 3 describes design and implementation of the proposed mechanism. Section 4 contains performance evaluation of our implementation. In Section 5 we propose a few potential applications of this mechanism. Section 6 compares this paper to related work. Section 7 concludes the paper and describes future work directions.

2 Motivation

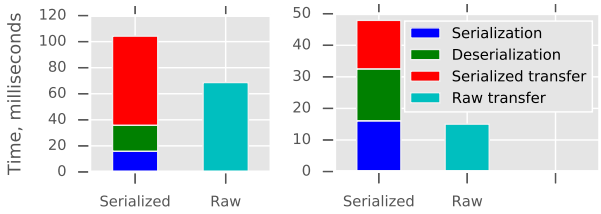
Modern cluster computing systems such as Spark and their workloads tend to be CPU-bound [12]. For instance, improving network I/O performance can only affect overall performance by less than 2% for typical workloads. Based on these observations, we conclude that optimizing CPU-bound operations such as data serialization is a reasonable way to improve performance of cluster computing systems.

Before moving on to experimental data, let us briefly discuss some general aspects of data serialization performance. Serialization is almost always sequential, so it does not scale with increasing number of CPUs. Serialization of complex data structures such as binary trees is also not cache-friendly. Serialization of even the simplest objects requires an extra in-memory copy of the whole object on both the source and the destination nodes: to network buffers on the source, and from network buffers on the destination. In contrast, when data is transferred directly (without serialization) and via RDMA, there are no extra copies, and CPU is not involved.

Java serialization performance measurements were performed using OpenJDK 8 as the Java class library and HotSpot (in server mode) as the JVM. We used Kryo [4] - a popular (e.g. used by Apache Spark) custom fast serialization library. Experiments were run on two identical machines (the sender and the receiver) with the following hardware configuration: 8-core Intel Xeon L5420 CPU @2.5Hz with 16 GB of RAM, running Linux 3.16.7 kernel, connected with 1 Gbit/s and 10 Gbit/s (the actual bandwidth was 5.5 Gbit/s) Ethernet.

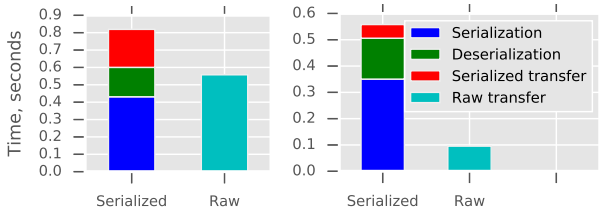
We used both simple (array of integers) and complex (a TreeMap with integer keys and arrays of strings as values) data. Figures 1 and 2 represent the time breakdown of serialized transfer compared to the time taken to transfer raw object data. Results are averaged over 50 measurements; the standard deviation is within 10%.

Serialization and deserialization take a significant portion of transfer time - up to 60% for simple data and up to 90% for complex. Raw data takes more time to send over the network since the serialized representation of objects is smaller. However, total transfer time is significantly smaller for raw data. Based on these results, we expect



(a) 1 Gbit/s (b) 5.5 Gbit/s

Figure 1: Transfer time breakdown for simple data



(a) 1 Gbit/s (b) 5.5 Gbit/s

Figure 2: Transfer time breakdown for complex data

that our proposed mechanism will be able to significantly speedup object transfer.

3 System Design and Implementation

We assume that the object transfer mechanism does not have to be completely transparent to the programmer. Objects shared among the nodes should be managed explicitly by either the application or the framework.

Shared objects are assumed to be immutable in the following sense. After an object is created on one node and copied to another node, modifications of the original object are not propagated. This requirement is not very restrictive in practice since common cluster computing frameworks operate with immutable objects.

The system should be implemented without any changes to the programming language. An example of such change is adding a new keyword to identify shared objects (as in PGAS languages [6]). Changing the programming language requires modifications of the compiler and the class file format at great engineering cost.

3.1 Programming model

Our data transfer mechanism introduces the notion of the global heap - a virtual address space region that is used to store shared objects. The global head is mapped to the same predefined range of virtual addresses in all the JVM processes that take part in the computation. The global heap architecture is illustrated in Figure 3.

We will use the term “global heap object” to denote the

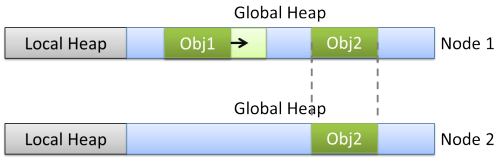


Figure 3: Global heap architecture

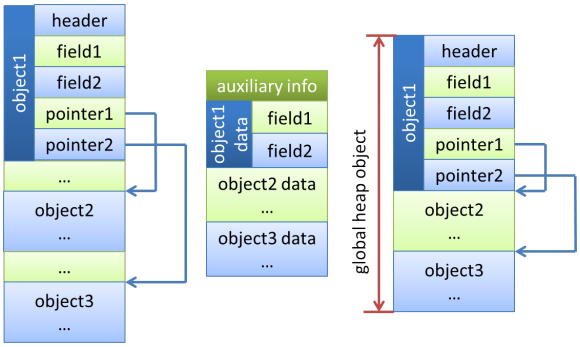


Figure 4: Object formats: raw, serialized, compact

whole complex object or collection that can be shared between nodes. The contents of global heap objects (Java objects linked by pointers) are allocated in a compact way such that the whole global heap object occupies a contiguous memory range (illustrated in Figure 4).

Each global heap object is located at a unique address, so when it is copied between nodes, no conflicts are possible. When a global heap object is created, an address range is reserved in the global heap. This range is released when all the copies of the object are destroyed. Object destruction can be performed either explicitly or implicitly (in a deferred fashion) using Java finalizers. We plan to explore the possibility of automatic garbage collection in the global heap in our future work.

In order to use our mechanism, the application has to perform some operations in a special way. In particular, it needs to let the JVM know which objects must be allocated on the global heap. Object creation is divided into three stages (described in Figure 5): (1) reserving space in the global heap; (2) populating the object with data; (3) making it available to other nodes (commit). The object becomes immutable after the commit operation.

Programming interface on the Java level is provided via special native methods implemented inside the JVM itself. They can access the JVM internal state such as memory allocator metadata. This allows us to avoid significant modifications of the bytecode interpreter.

Global classes (objects of which can be stored in the global heap) need to be registered before usage. Note that *all* the classes, instances of which are to be shared, must be registered, including classes of entries of various collections, which are usually declared private. We rely on reflection to automate registration of such auxiliary classes. Developers can also use Java annotations to declare their own classes as global instead of making registration calls in runtime.

```
// Source side:
// Reserve space in the global heap
GObject obj = new GObject(...);
// Populate the object with data
obj.data = ...;
// Make the object available to other nodes
obj.commit("key");
...
obj.release();// no longer in use
-----------------------------------------------
// Destination side:
// Obtain a copy of the object
GObject obj = GHeap.get("key");
...
obj.release();// no longer in use
```

Figure 5: Global heap object creation and transfer

3.2 Distributed global heap coordination and accessing remote objects

The global heap is designed in a way that minimizes the amount of distributed coordination and synchronization. Our design is inspired by multithreaded memory allocators such as Hoard [8]. The global heap is partitioned among the nodes: at every point in time, each node has a set of exclusive address space ranges that are used to allocate global heap objects created on this node. As a result, most of the time, allocation is performed entirely locally by each node. Furthermore, each node's exclusive heap is partitioned in the same fashion between the threads running on the node.

Global heap ranges for exclusive allocation are requested from the centralized coordinator service dynamically on demand and released when they are no longer in use. Distributed coordination is limited to reserving and releasing these (large) heap ranges. Dynamic partitioning scheme allows our mechanism to be used in dynamic cluster topologies and elastic cloud deployments.

Shared object are located using the object address and the node network address. This location information can be obtained either using a separate communication mechanism or the location service (key-object mapping) provided by the system. In the latter case, our system can be viewed as a distributed key-value store. Currently, our implementation only supports TCP transport; each node runs a TCP server servicing requests for object data. We plan to implement RDMA support in the near future.

3.3 JVM modifications

We have implemented our fast object transfer mechanism for JamVM [3] - a lightweight open source Java virtual machine. It is significantly simpler than HotSpot [5] (the reference implementation of the JVM), that is why we chose it for the initial prototype. We are currently working on the HotSpot-based implementation.

Each Java object's binary representation has a pointer to its class object containing the class metadata. These class pointers are the only type of "external" (pointing outside of the global heap) pointers that cannot be avoided. We store unique class IDs in object headers instead of pointers. The JVM maintains the class pointer table; class IDs act as indices into this table. IDs are assigned to classes on registration, so the order of registration must be the same on all the nodes.

For each global heap object created on the current node, the JVM maintains its own dedicated (small) memory heap for the address range reserved for this object. The memory allocator determines which range to allocate from using the context information (object currently being populated) stored in the Java thread. When the JVM allocates memory for objects, it dispatches the allocation to the corresponding heap range.

Since there is no need to free individual small objects or perform garbage collection within these ranges, memory is allocated using simple "bump pointer" method. The ranges are discarded as a whole when objects are released. This simplicity leads to our memory allocation mechanism being generally faster than ordinary JVM memory allocation and garbage collection.

Since we have changed the semantics of some of low-level JVM operations (namely dereferencing the class pointer), we had to modify a small part of the bytecode interpreter. Besides that, we had to modify the garbage collector (GC) in order to disable it for the global heap. JamVM uses a conservative GC - when discovering "live" objects, it treats all object fields that look like pointers into the Java heap as object references. Pointers into the global heap are simply ignored.

4 Evaluation

To evaluate the performance of our object transfer mechanism we designed the following benchmark. We compare time taken to transfer objects between two nodes with serialization (using both Kryo and standard Java serialization) and directly using our mechanism. The experiments were conducted on the hardware described in Section 2, with a 5.5 Gbit/s network. For direct copy measurements we used a modified version of JamVM 2.0 and GNU Classpath 0.99 as Java class library (due to compatibility issues with OpenJDK). We used the same types of data as in experiments in Section 2.

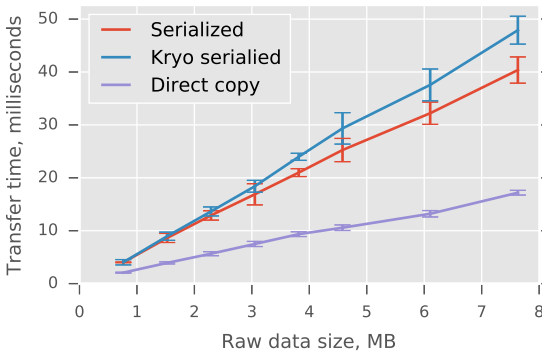


Figure 6: Serialization vs. direct copy for simple data

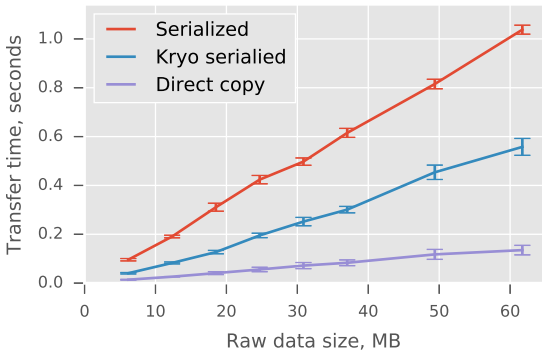


Figure 7: Serialization vs. direct copy for complex data

The results are presented in Figures 6, 7 and 8. All measurements are averaged over 50 runs; error bars represent the standard deviation. As we can see from the graphs, direct object transfer achieves significant speedups: up to 5× compared to Kryo and up to 10× compared to standard serialization for complex data, and up to 2.5× for simple data. We expect even more significant speedup with higher network bandwidth (e.g. 40 Gbit/s) and using RDMA as the transport.

Figure 8 represents the results for small objects. For

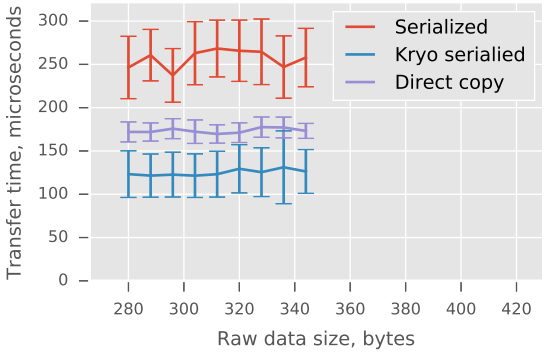


Figure 8: Serialization vs. direct copy for small objects

data sizes of a few hundred bytes (in case of simple objects), our mechanism performs worse than Kryo, but still better than standard serialization. Our implementation can be further optimized to minimize the fixed (independent of data size) overhead.

In order to estimate the overhead of JVM modifications, we used a test program that populates a data structure (TreeMap with objects as keys and values), serializes and deserializes it in memory. We chose this test because it involves a lot of memory allocations (during deserialization) and accesses to class metadata (to determine if each object is serializable); these are the two operations that we modified in the JVM. We measured total execution time for unmodified JamVM and for our modified version. The results of this experiment showed that the overhead is negligible - within 1% for all the runs.

5 Applications

As we already mentioned, one example of a system that could benefit from our fast data transfer mechanism is the Apache Spark [15] [2] cluster computing framework. One of the main bottlenecks in typical Spark workloads are shuffle operations that require data transfer between all the nodes. Optimizing data transfer with our proposed mechanism can significantly speed up shuffle operations.

Another example is distributed in-memory key-value stores. Our mechanism would significantly reduce the latency of `get` and `set` operations in the case when values are complex objects (instead of small primitives such as numbers and strings).

Our approach is not limited to data transfer between different machines. It can also be used to improve performance of local IPC (inter-process communication) between processes running on the same machine. Here we can go further than only getting rid of serialization: objects to be transferred can be stored in shared memory, which will provide a very fast zero-copy IPC mechanism for complex data transfer.

6 Related Work

There are many existing implementations of distributed Java virtual machines that target large-scale computational clusters. Some of them are also based on the idea of a shared address space [7]. However, the existing system is based on a full-featured DSM and requires hardware support in order to have good performance. Our approach is more suited for a specific problem of fast migration of complex data structures, and does not require a sophisticated underlying DSM system.

PGAS (partitioned global address space) languages [6] are based on the programming model with a global address space divided into regions shared between processes and regions local to one process. Separation into local heaps for each node and one global heap is very similar to PGAS. However, there are significant differences: (1) our mechanism doesn't involve programming language modifications; (2) PGAS model has more strict consistency properties, leading to worse performance.

As RDMA becomes a more popular and mature technology, researchers are adopting existing distributed systems to benefit from RDMA and are designing new ones. Notable examples include in-memory key-value stores [9] [10] and efforts to support RDMA in cluster computing frameworks [11]. Design of these systems is mostly focused on a different (from traditional TCP sockets) network communication paradigm. Our approach is not specific to RDMA, although would benefit the most from RDMA-capable hardware.

Project Tungsten [14] is an effort by Apache Spark developers to improve memory management performance in Spark. The main idea is re-implementing standard Java collections using "unsafe" interfaces provided by the JVM. These new collections are allocated in native memory and are not managed by the JVM. They benefit from decreased memory footprint and improved cache locality. Our approach is similar in a way that it also involves lower-level JVM-related optimizations. However, our mechanism allows using existing Java collections. The two approaches can be combined to achieve even better performance improvements.

7 Conclusion

In this paper we describe a novel approach for fast transfer of complex pointer-rich data structures between the nodes of a distributed system. The approach is based on avoiding serialization of data (which is the main bottleneck given modern high-speed networks) by providing a global cluster-wide virtual address space to store shared objects. This mechanism can significantly speedup transferring large datasets, although it adds programming complexity (due to explicit management of shared objects) in order to achieve performance benefits.

This fast data transfer mechanism can be applied to improve performance of various distributed applications such as cluster computing frameworks and distributed in-memory key-value stores. Another possible application is a fast IPC mechanism for transferring complex objects between different processes running on the same machine. We are planning to explore these and other applications in our future work.

One of the important aspects that we leave out of scope of this paper is fault tolerance, which we currently assume to be the responsibility of the application. Fault tolerance is among the main directions of our future work.

References

[1] Apache Hadoop. https://hadoop.apache.org/.

[2] Apache Spark. http://spark.apache.org/.

[3] JamVM. http://jamvm.sourceforge.net/.

[4] Kryo - Java serialization and cloning: fast, efficient, automatic. https://github.com/EsotericSoftware/kryo.

[5] OpenJDK and HotSpot JVM. http://openjdk.java.net/.

[6] Partitioned global address space. http://www.pgas.org/.

[7] ANDERSSON, J., WEBER, S., CECCHET, E., JENSEN, C., AND CAHILL, V. Kaffemik - a distributed jvm featuring a single address space architecture. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1* (Berkeley, CA, USA, 2001), JVM'01, USENIX Association, pp. 9–9.

[8] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), ASPLOS IX, ACM, pp. 117–128.

[9] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 401–414.

[10] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing* (Washington, DC, USA, 2011), ICPP '11, IEEE Computer Society, pp. 743–752.

[11] LU, X., RAHMAN, M. W. U., ISLAM, N., SHANKAR, D., AND PANDA, D. K. Accelerating spark with rdma for big data processing: Early experiences. In *Proceedings of the 2014 IEEE 22Nd Annual Symposium on High-Performance Interconnects* (Washington, DC, USA, 2014), HOTI '14, IEEE Computer Society, pp. 9–16.

[12] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 293–307.

[13] SKOUSEN, A., AND MILLER, D. Using a single address space operating system for distributed computing and high performance. In *Proceedings of the IEEE International Performance Computing and Communications Conference, IPCCC 1999, Phoenix/Scottsdale, Arizona, USA, 10-12 February 1999* (1999), pp. 8–14.

[14] XIN, R., AND ROSEN, J. Project Tungsten: Bringing Spark closer to bare metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[15] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.