CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Exploiting Multi-cores for Efficient Interchange of Large Messages in Distributed Systems

Dongfang Zhao[⋆], Kan Qiao[†], Zhou Zhou[‡], Tonglin Li[‡], Xiaobing Zhou[◇], Ioan Raicu[‡,§]

[⋆]*Pacific Northwest Natoinal Laboratory, Richland, WA*
[†]*Google Inc., Seattle, WA*
[‡]*Illinois Institute of Technology, Chicago, IL*
[◇]*Hortonworks Inc., San Francisco, CA*
[§]*Argonne National Laboratory, Chicago, IL*

SUMMARY

Conventional data serialization tools assume that objects to be coded are usually small in size so a single CPU core can encode it in a timely manner. In the era of Big Data, however, object gets increasingly complex and larger, which makes data serialization become a new performance bottleneck. This paper describes an approach to parallelize data serialization by leveraging multiple cores. Parallelizing data serialization introduces new questions such as how to split the (sub)objects, how to allocate the available cores, and how to minimize its overhead in practice. In this paper we design a framework for parallelly serializing large objects and analyze the design tradeoffs under different scenarios. To validate the proposed approach, we implemented Parallel Protocol Buffers (PPB)—the parallel version of Google's Protocol Buffers, a widely-used data serialization utility. Experimental results confirm the effectiveness of PPB: multiple cores employed in data serialization achieve highly scalable performance and incur negligible overhead. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

1. INTRODUCTION

Serialization is the de facto mechanism for data interchange in distributed systems. In essence, on the client side a data structure (or, an object) is encoded into another format (typically a string) that is transferred to and decoded on the server side. For example, serialization is widely used in remote procedure calls (RPC): the client marshals (i.e., serializes) the procedure parameters and sends the packed message to the server who unmarshals (i.e., deserializes) the message and calls its local procedure with the unpacked parameters; after the local procedure is finished, the server then conducts a similar but reversed process to return the value to the client. Although only running on a single CPU core, conventional techniques for data serialization are sufficient for most workloads because the objects are usually very small. In the era of Big Data, however, objects of large-scale distributed systems are becoming increasingly larger, which is challenging the viability of our conventional wisdom. As a case in point, at Google our systems experience many RPC messages in the size of hundreds of megabytes. Our serialization tool—Google's Protocol Buffers [27]—was originally not designed for messages of this size; Instead, Protocol Buffers assumes that objects are small enough to be efficiently encoded and decoded with a single CPU core in a serial manner. That is, a gap between the conventional assumption and the real-world situation is increasingly enlarged.

This work explores how to leverage modern computing systems' multi-cores to improve the serialization and deserialization speed of large objects. Rather than proposing new serialization

algorithms, we tackle the problem from a system's perspective. Specifically, we propose to leverage multiple CPU cores to split a large object into smaller sub-objects so to be serialized in parallel. While data parallelism is not a new idea in general, it has never been applied to data serialization and poses new problems. For instance, serializing multiple chunks of a large object incurs additional overhead such as metadata maintenance, thread and process synchronization, resource contention. In addition, the granularity (i.e., the number of sub-objects) is a machine-dependent choice: the optimal number of concurrent processes and threads might not align with the available CPU cores.

Table I. A partial list where Protocol Buffers [27] is deployed

Organization / Project	Description
Google	Internal projects, e.g. MapReduce [13], Google File System [15], Bigtable [12]
Twitter	For efficient and flexible data storage
Apache Camel [4]	Default data interchange format in enterprise integration
SWI-Prolog [31]	Recently added support in logical programming
The R Programming Language	The `RProtoBuf` Package
Protobuf-Embedded-C [30]	For resource constrained applications in embedded systems
FusionFS Filesystem [41]	Metadata of distributed file systems; particularly for large directories
ZHT Key-Value Store [22]	Default data interchange format in distributed key value stores

In order to overcome these challenges and better understand whether the proposed approach could improve the performance of data serialization of large objects, we provide detailed analysis on the system design, for example how to determine the sub-object's granularity for optimal performance and how to ensure that the performance gain is larger than the cost. To demonstrate the effectiveness of our proposed approach, we implemented a system prototype called parallel protocol buffers (PPB) by extending a widely-used open-source serialization utility (Google's Protocol Buffers [27]). We have evaluated PPB on a variety of test beds: a conventional Linux server, the Amazon EC2 cloud, and an IBM Blue Gene/P supercomputer. Experimental results confirm that the proposed approach could significantly accelerate the serialization process. In particular, PPB could accelerate the metadata interchange 3.6x faster for an open-source distributed file system (FusionFS [41]).

To summarize, this work makes the following contributions:

- *We identify a new performance bottleneck on data serialization in parallel and distributed systems, and propose a data parallelism approach to address it;*
- *We discuss the design tradeoff and quantitatively analyze the abstracted model under different scenarios;*
- *We implement a system prototype by extending Google's Protocol Buffers and evaluate it at a variety of test beds.*

In the following we give more background of this work in Section 2. Section 3 presents PPB's design trade-off and analysis. In Section 4 we detail the implementation of PPB. Section 5 reports the experimental results. In Section 6 we discuss open questions of PPB. Section 7 reviews related work. We finally conclude this paper in Section 8.

2. BACKGROUND AND MOTIVATION

Although we illustrate the parallelization of data serialization by extending Google's Protocol Buffers [27], the approach is generally applicable to other serialization tools as well. The reason we choose to extend Protocol Buffers is two-fold. First, Protocol Buffers is used as the serialization utility at Google internally, so it is straightforward to work with its intrinsic designs and features. Second, Protocol Buffers is widely used in both production systems and research projects (see Table I for examples); therefore we hope the community could largely benefit from an extension of this popular tool.

We give a motivating example on the serialization bottleneck at Google. Among many others, one of our servers' tasks is to take the incoming query requests from our enterprise clients. The end-to-end time for completing a RPC is up to a few minutes when the message size is large, and more than

DOI: 10.1002/cpe

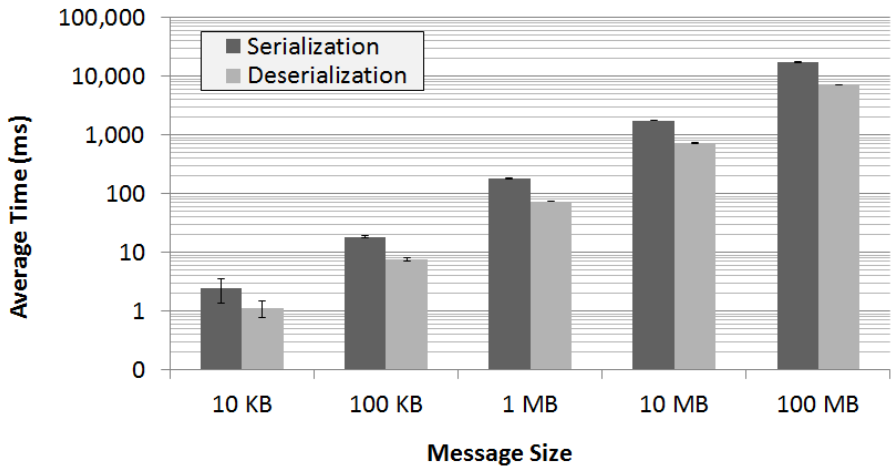


Figure 1. Serialization and deserialization with Protocol Buffers [27]

35% of time is spent on data serialization and deserialization. At the same time, the CPU utilization rate is low (i.e., several cores stay idle) even though RPCs require intensive computation to coding large objects. That is, on one hand, the overhead of computing the (de)serialization is significant; on the other hand, many computing resources are not being utilized. Consequently, the coding time is proportional to the size of the message. As shown in Fig. 1, the (de)serialization time increases at the same rate as the message size on one of our servers. For all message sizes tested in the figure, only one CPU core is busy (more than 98% CPU usage) with the coding job. We only report the performance of message size up to 100 MB in Fig. 1, as it is on par with the largest objects we observe at Google. Yet, this O(100 MB) is beyond the original design goal of Protocol Buffers, as large messages pose new challenges such as security concerns. In fact, the default maximal message size of Protocol Buffers is 64 MB. For messages between 64 MB and 512 MB, developers need to manually lift the upper limit (for example, the 100 MB case in Fig. 1). Messages larger than 512 MB are not supported in any release.

3. DESIGN AND ANALYSIS

This section presents the design and analysis of the proposed parallelism for data (de)serialization. We will first provide a high-level overview of the system architecture, then discuss the design space on how to split and merge the objects, and finally analyze how to choose the number of cores as best practice.

3.1. System Architecture

The high-level overview of the system architecture is shown in Fig. 2. It shows the serialization and deserialization procedures from a sender to a receiver where six CPU cores work concurrently on the six sub-objects constituting the original large object.

It is worth noting, though, that Fig. 2 only shows one use case of parallel data serialization—the data is interchanged via the network between two nodes. The serialized messages can be used in other scenarios as well such as being persisted to the databases.

3.2. Manipulating Objects and Sub-Objects

It is not uncommon to deal with large objects in a distributed system. In scientific computing, for example as our prior work [40] showed, datasets were so large that they were usually compressed to be serialized onto the hard disk. As another example, in our previous work on implementing a

DOI: 10.1002/cpe

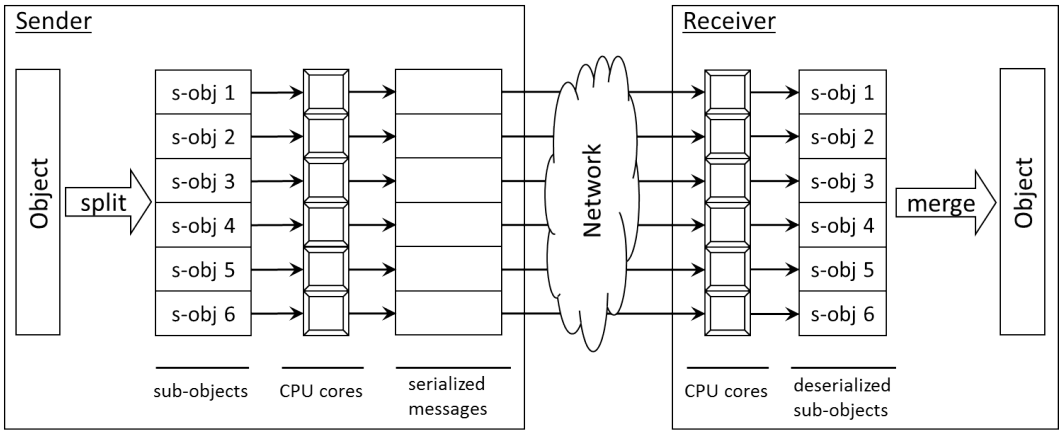


Figure 2. System architecture

distributed file system [39] we showed that a large directory comprised of thousands of small files could result in a huge metadata blob stored in a distributed key-value store [22]. In both cases, large objects should be better serialized in parallel to reduce the I/O time.

The first question we need to answer is how to split the given large object. In a homogeneous setting where all cores have the same computing capacity, it is straightforward to evenly split the original object. By doing this, the longest execution time for all sub-objects can be minimized. In practice, however, an equidistant splitting is not always possible because some variables might cross the boundary of the sub-objects. Then the question becomes: do we want to split the variable on the boundary?

We argue that a micro splitting at the variable granularity (in addition to the object level) is not worthy to trade for the strict evenness between sub-objects. The benefit of variable-level splitting is highly limited because the variable on the boundary is eventually a primitive data type such as integer, string, float, and so forth. That is, although a nested object could happen to reside on the boundary, we could still split this nested object and in the end it is either a variable or nothing right sitting on the boundary. A primitive variable is usually not larger than a memory page, thus splitting it into two processes brings limited benefit but incurs overhead and complexity. In the real world, here at Google the large messages we deal with comprise a huge number of repeated field (i.e., the data structure to store an array) of primitive data formats, rather than a few and large variables. This fact further justifies the choice not to apply the variable-level splitting.

Since splitting a variable is not a good choice in this context, the best we can do to approximate the evenness between sub-objects is to adjust the sub-objects' boundary to align with the variables'. In the following analysis, however, we assume that all sub-objects are of equal size; this greatly simplifies the analysis and is accurate enough to quantify the proposed approach.

Merging sub-objects at the receiver side is the reversed procedure from system's perspective. Algorithmically, the merging stage is significantly simpler than the splitting one. This is mainly because on the receiver side the number of sub-objects is fixed, as opposed to be decided on the sender side at runtime.

3.3. Optimal Number of Cores

This and the following sections quantitatively discuss some good practice in choosing the number of cores for the data parallelism in data (de)serialization. The environment parameters taken into consideration are listed in Table II. The end-to-end wall time of the original serialization and deserialization is denoted by T. When the workload is dispersed to multiple cores, overhead H is introduced by each core such as creating the thread or process, metadata update, and multicasting of messages. Lastly, the number of cores to be leveraged for the data parallelism is denoted by N.

DOI: 10.1002/cpe

Table II. Environment variables to parallelize data (de)serialization

Variable	Description
T	Single-core data interchange time
H	Per-core overhead of parallelization
N	Number of cores working in parallel

Since we assume each sub-object is of same size, the time for parallel (de)serialization is $\frac{T}{N}$. Similarly, the overhead by these N cores is $H \cdot N$. We therefore have the end-to-end time for the parallel version of data interchange as a function of N:

$$F(N) = \frac{T}{N} + HN$$

Intuitively, adding more cores reduces the (de)serialization time, but increases the overhead. We are interested in the choice of N to minimize $F(N)$. Suppose $\hat{N}$ is continuous, so if we take the first derivative of $\hat{N}$ ans solve the following equation,

$$\frac{d}{d\hat{N}}(F(\hat{N})) = H - \frac{T}{\hat{N}^2} = 0$$

we have

$$\hat{N} = \sqrt{\frac{T}{H}}$$

In addition, the second-order derivative of $\hat{N}$ is always positive:

$$\frac{d^2}{d\hat{N}^2}(F(\hat{N})) = \frac{T}{\hat{N}^3} > 0$$

Therefore, the optimal number of cores N_{opt} should be

$$\arg\min_N F(N) = \begin{cases} \left\lfloor \sqrt{\frac{T}{H}} \right\rfloor \text{ if } F\left(\left\lfloor \sqrt{\frac{T}{H}} \right\rfloor\right) < F\left(\left\lceil \sqrt{\frac{T}{H}} \right\rceil\right) \\ \left\lceil \sqrt{\frac{T}{H}} \right\rceil \text{ otherwise} \end{cases}$$

Accordingly, the minimal end-to-end time is

$$F_{min}(N) = min\left(F\left(\left\lfloor \sqrt{\frac{T}{H}} \right\rfloor\right),\ F\left(\left\lceil \sqrt{\frac{T}{H}} \right\rceil\right)\right)$$

If $\sqrt{\frac{T}{H}}$ is an integer, a simpler form of the above analysis is

$$N_{opt} = \arg\min_N F(N) = \sqrt{\frac{T}{H}} \tag{1}$$

and

$$F_{min}(N) = F\left(\sqrt{\frac{T}{H}}\right) = 2\sqrt{HT} \tag{2}$$

Note that in practice H could be significantly smaller than T, making N_{opt} a large number in Eq. 1.

DOI: 10.1002/cpe

3.4. Performance Gain Guarantee

We just show how to pick the optimal number of cores for the parallelism. In the real world, however, N_{opt} might not be viable; for example the available cores might be (much) fewer than N_{opt}. Therefore, a more realistic, or maybe more interesting, question is how to pick the number of cores to guarantee performance gain. That is, how to choose N to ensure

$$T > F(N) = \frac{T}{N} + HN$$

Again, this is not a trivial problem as the first term of $F(N)$ (i.e., $\frac{T}{N}$) decreases with a larger N while the second one (i.e., $H \cdot N$) increases with a larger N.

The above equation can be transformed to

$$H \cdot N^2 - T \cdot N + T < 0 \tag{3}$$

Again, let $\hat{N}$ denote the continuous N in the following analysis. If we solve this equation:

$$H \cdot \hat{N}^2 - T \cdot \hat{N} + T = 0$$

we have

$$\hat{N} = \frac{T \pm \sqrt{T^2 - 4 \cdot H \cdot T}}{2 \cdot H} \tag{4}$$

In order to make sure $\hat{N}$ has real values, we need to have this condition:

$$T^2 - 4 \cdot H \cdot T > 0$$

That is, we need to have

$$H < \frac{T}{4} \tag{5}$$

In other words, one precondition to apply multiple cores to parallelize data (de)serialization is to have the per-core overhead time smaller than a quarter of the processing time itself.

Note that the parabolic curve in Eq. 3 is open to the top, therefore if condition in Eq. 5 is satisfied, then $\hat{N}$ should be set to the value in between the two values in Eq. 4. That is, the number of cores should fall into the following range to guarantee that the parallelism is beneficial:

$$N \in \left[\left\lceil \frac{T - \sqrt{T^2 - 4HT}}{2H} \right\rceil, \ldots, \left\lfloor \frac{T + \sqrt{T^2 - 4HT}}{2H} \right\rfloor \right]$$

In practice, the above condition should be easy to satisfy. This is because H is usually a significantly smaller number than T for large messages. Therefore, $\sqrt{T^2 - 4HT}$ could be highly close to T, which consequently sets the lower bound (i.e., $\left\lceil \frac{T - \sqrt{T^2 - 4HT}}{2H} \right\rceil$) as a small integer. In other words, a small number of cores leveraged in the data parallelism have a high chance to outperform sequential (de)serialization on a single core.

4. IMPLEMENTATION

We implement the data parallelism for data serialization and data deserialization on top of Google's Protocol Buffers [27] (C++ version), which is called parallel protocol buffers (PPB). We will discuss some implementation details in the following aspects: user interface, multiprocessing, and multithreading.

DOI: 10.1002/cpe

4.1. User Interface

The long term goal of PPB is to provide the built-in support of data parallelism for large messages. Users will only need to specify the number of concurrent computing cores as an argument during the (de)serialization. Or even better, PPB will probe the current system status (for example, the number of idle cores) and automatically set the concurrency according to the theoretical rules discussed in Section 3.

At the time of writing, PPB is implemented as a middleware that is loosely coupled from Protocol Buffers' own release. That is, PPB is a wrapper of Protocol Buffers; the (de)serialization requests from applications are redirected to PPB who calls the Protocol Buffers in parallel. The main reason of such an implementation decision is for quick system prototyping and studying the performance improvement quantitatively.

Admittedly, one potential drawback of a middleware is missing some chances of optimizing the code as a whole. For example some performance penalty is expected compared to the build-in support of data parallelism. Nevertheless, as we will demonstrate in the evaluation (Section 5), a loosely coupled middleware can significantly improve the performance, making the proposed approach more promising when the built-in implementation is completed.

4.2. PPB with Multiprocessing

Our first attempt for parallelizing Protocol Buffers is using message passing interface (MPI). In essence, MPI is a well-developed interface to implement multiprocessing applications. That is, the application is split into multiple processes and executed in parallel.

The overhead of MPI implementation of Protocol Buffers mainly comes from MPI initialization (i.e., `MPI_Init, MPI_Comm_rank, MPI_Comm_size`) and MPI synchronization (i.e., `MPI_Barrier, MPI_Finalize`). Fortunately, modern CPUs mostly comprise O(10) cores; such level of concurrency does not cause significant overhead on the MPI part if the targeted message is as large as O(100 MB). There are other communication cost associated with MPI, such as `MPI_Send, MPI_Recv`; but these are dominated by the network cost if we are talking about big messages.

There is also other overhead coming from the operating system. For instance, every new process needs to be forked with a new independent memory stack; as another example, for any inter-process communication a socket needs to be created. This type of overhead exists by nature, and is hardly eliminated as long as we want to apply some forms of parallelism. Yet, we can try to reduce it; an intuitive option is to use multithreading over multiprocessing.

4.3. PPB with Multithreading

Comparing with multiprocessing, multithreading saves some resource on memory stacks as all threads share the same heap. Moreover, no sockets are required since they are all within the same process. This is, thus, particularly an advantage for those workloads that are CPU-bound.

Nevertheless, a parallel data serialization does not only involve CPU computation, but also frequent and concurrent memory accesses. This complicates the implementation choice for multithreading because of the possible contention on shared resources. In fact, concurrent write accesses are not supported in Protocol Buffers [27] for thread safety. Note that multiprocessing does not suffer from this because every process owns its complete memory map isolated from others.

To verify that multithreading does not help PPB, we use OpenMP [26] to parallelize the serialization process on a multi-core server (i.e., Fusion, whose detailed specification will be provided in Section 5) and report the results in Fig. 3. As we expect, multiple threads (in a single process) actually degenerate the overall performance.

Figure 3 clearly shows that adding more threads to the serialization procedure does not bring any performance gain. This confirms our previous conjecture that multiple threads in Protocol Buffer compete for the global (shared) metadata of the objects. Such contention results in a pseudo parallelism but really a serial execution of competing threads and only introduce overhead such thread manipulation.

DOI: 10.1002/cpe

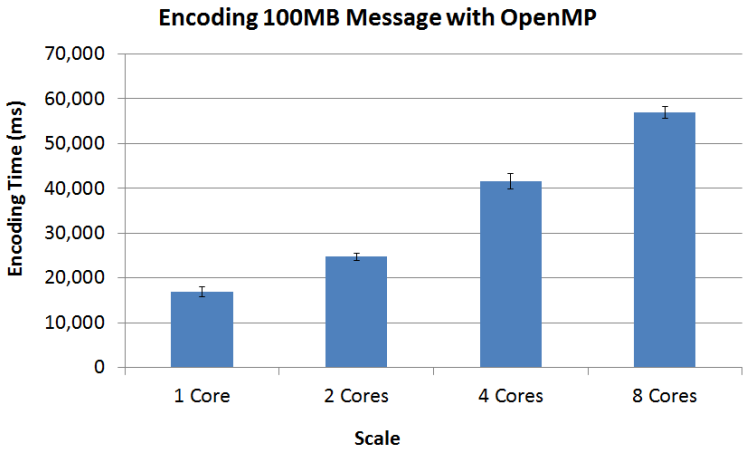


Figure 3. Parallel Serialization with OpenMP

Another critical disadvantage of multithreading is the extendibility to multiple nodes. In essence, a multithreading programming paradigm (for example, Pthread, OpenMP) targets only single-node environments. Therefore, if CPU cores are idle in a remote server, the local multithreading program is not able to leverage them. On the other hand, multiprocessing paradigm (for example, MPICH [24], OpenMPI [25]) does not have such a limit of working only on a single node.

After considering all factors we believe multiprocessing is a more appropriate implementation choice than multithreading. In the following discussions, the PPB is a parallel version of Protocol Buffers [27] implemented by MPI. Nevertheless, it should be clear that multithreading is in general a more efficient approach to achieve parallelism compared to multiprocessing as long as the application is not competing for some resources. From a system's point of view, multiprocessing is more widely used in inter-node parallelism.

5. EVALUATION

5.1. Experiment Setup

The test beds where we evaluate PPB are the Fusion Linux server, the Amazon EC2 cloud platform, and an IBM Blue Gene/P supercomputer (i.e., Intrepid [18]). Fusion is a 48-core Linux server with 256 GB RAM and one 2 TB HDD and one 100 GB SSD. For Amazon EC2, the instance types where PPB is deployed are summarized in Table III. Intrepid has 40K nodes in total, each of which is equipped with quad-core 850 MHz PowerPC 450 processors and runs a light-weight Linux ZeptoOS [38] with 2 GB memory.

Table III. Representative Instance Types of Amazon EC2 Cloud Platform

Instance Name	EC2 Category	Cores	CPU Speed	Chip Type	Memory	Storage
t2.medium	General Purpose	2	2.5 GHz	Intel Xeon Family	4 GB	EBS only
m3.xlarge	General Purpose	4	2.5 GHz	Intel Xeon E5-2670v2	15 GB	80 GB
m3.2xlarge	General Purpose	8	2.5 GHz	Intel Xeon E5-2670v2	30 GB	160 GB
c3.8xlarge	Compute Optimized	32	2.8 GHz	Intel Xeon E5-2680v2	60 GB	640 GB
r3.8xlarge	Memory Optimized	32	2.5 GHz	Intel Xeon E5-2670v2	244 GB	640 GB
i2.8xlarge	Storage Optimized	32	2.5 GHz	Intel Xeon E5-2670v2	244 GB	6,400 GB

All results are obtained from the PPB middleware implemented with MPI. The MPI library we use is MPICH 3.0.4. The C++ compiler is g++ 4.4.1. The base Protocol Buffers version is 2.6.0.

DOI: 10.1002/cpe

All experiments are repeated at least five times until results become stable (within 5% margin of error). The reported numbers are the average of all runs. The standard derivations are also plotted along with the averages whenever available.

5.2. *PPB on the Conventional Server*

Fig. 4 shows the serialization time of parallel protocol buffers for different message sizes on 1 to 32 cores on the Fusion Linux server. For messages larger than 1 MB, the parallelism obviously shows its advantages with excellent scalability. For instance, while coding a 100 MB message takes about 20 seconds with a single core, the same workload only takes 1 second when 32 cores are used. For small messages such as 10 KB and 100 KB, adding more cores do not necessarily improve the performance because the overhead incurred by the parallelism outweighs the savings on the coding.

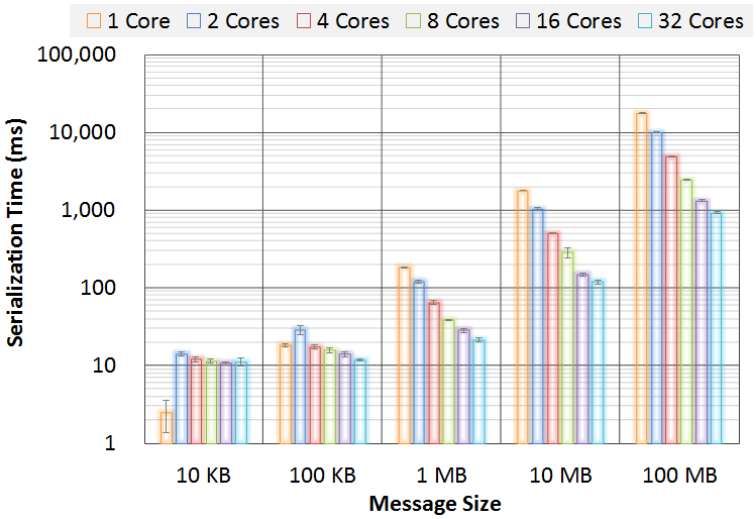


Figure 4. Serialization time of PPB on Fusion server

We show the speedup of the above experiment in Fig. 5. We observe that larger message sizes have closer performance to the theoretical upper bound—the absolutely linear scalability. This is because for larger messages the MPI overhead takes a smaller portion of the overall execution time.

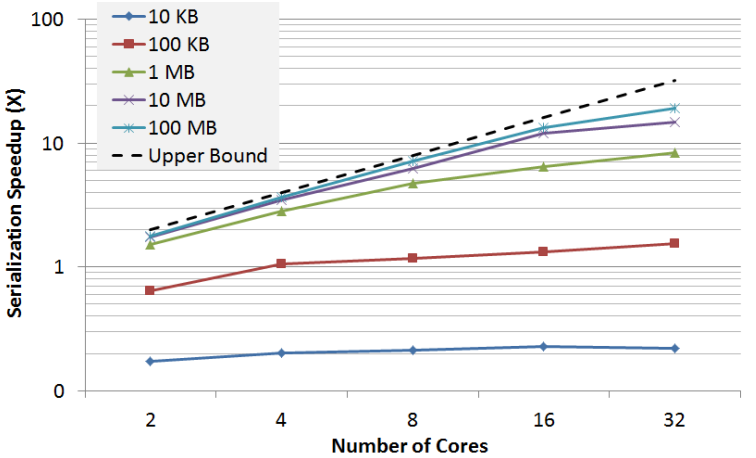


Figure 5. Serialization speedup of PPB on Fusion server

Fig. 6 shows the deserialization time of parallel protocol buffers for different message sizes on 1 to 32 cores. As opposed to the serialization case for small messages, we observe that PPB is

DOI: 10.1002/cpe

as effective as for large messages. The results suggest that the MPI overhead of deserialization is smaller than that of serialization, thus is almost negligible even for small messages.

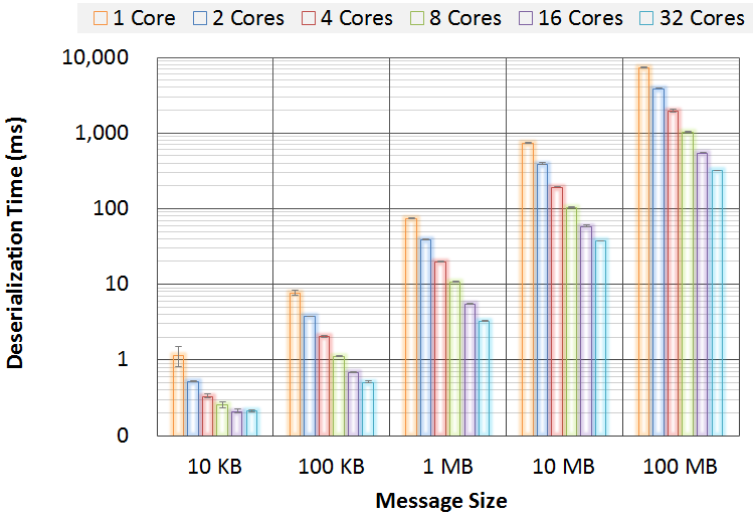


Figure 6. Deserialization time of PPB on Fusion server

We show the speedup of the parallel deserialization in Fig. 7. Similarly to the serialization case, a larger message has a better scalability. In this experiment, a 1 MB seems to be the cut-off size to saturate the 8+ cores as the speedup is almost identical to 1 MB, 10 MB, and 100 MB messages on 8, 16, and 32 cores.

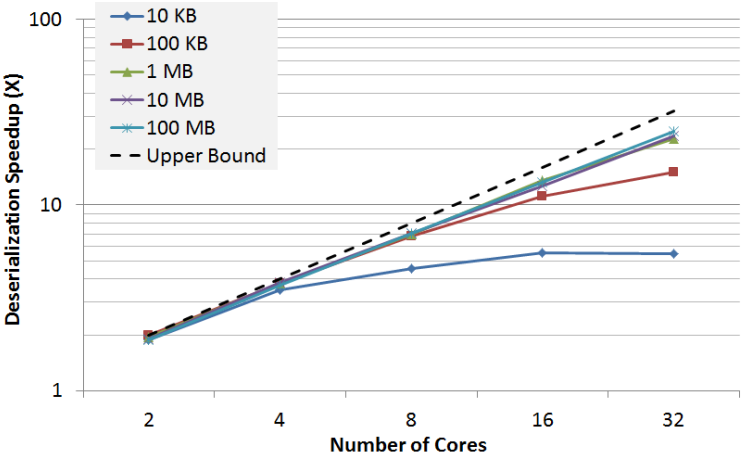


Figure 7. Deserialization speedup of PPB on Fusion server

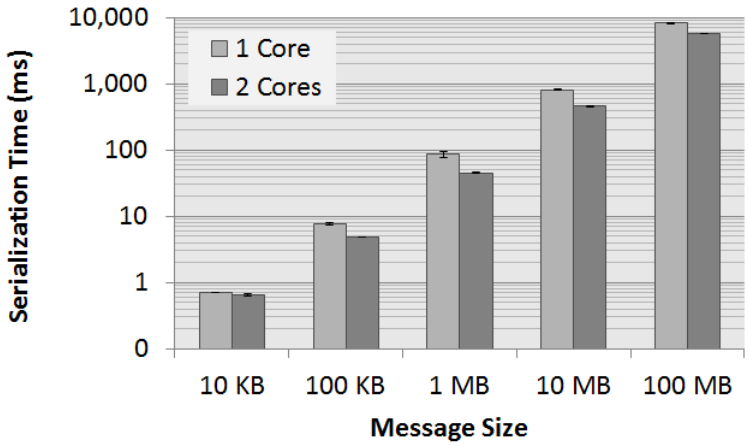
5.3. PPB on the Cloud

In order to understand the viability of parallel serialization in a more general setup particularly for cloud computing, we deploy PPB on a variety of Amazon EC2 instances. In the remainder of this section, we will use cores and vCPUs interchangeably because the latter is to indicate the number of computing units in Amazon EC2. We will not report the speedup as in Section 5.2 due to limited space.

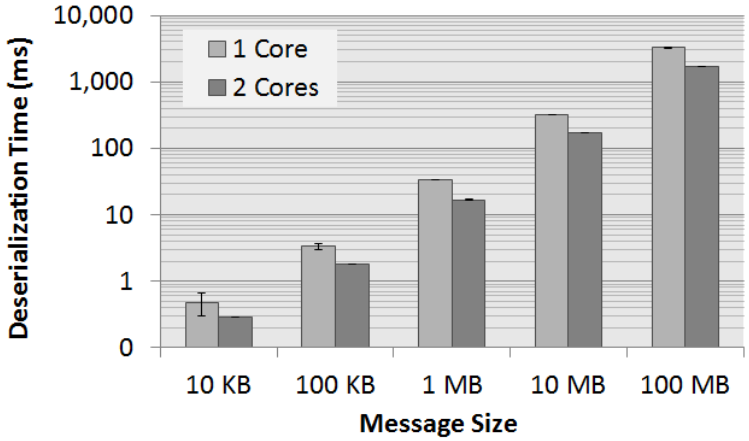
First of all, we report the effect of parallel serialization on a relatively less powerful instance—t2.medium. Fig. 8(a) and Fig. 8(b) show the serialization and deserialization time for different sizes of messages, respectively. Recall that a t2.medium instance has 2 vCPUs (i.e., cores), therefore we

DOI: 10.1002/cpe

carried out the experiments on both a serial process (1 Core) and two concurrent processes (2 Cores) scenarios.



(a) Serialization



(b) Deserialization

Figure 8. PPB on the EC2 t2.medium instance

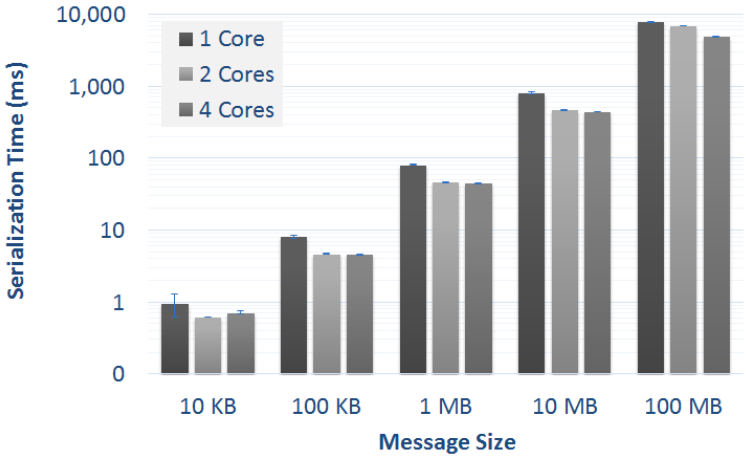
From Fig. 8(a) we learn that a small message such as 10 KB might benefit little from the data parallelism during serialization. Nevertheless, when the message size increases we observe significant saving when serializing the message (i.e., 100 KB to 100 MB). Therefore, when serializing small messages on small EC2 instances, a parallel version could bring limited benefit to the performance.

For deserialization in Fig. 8(b), however, we observe that both small and large messages could benefit the data parallelism. This result suggests that we apply the parallel version of data deserialization regardless the message size for small EC2 instances.

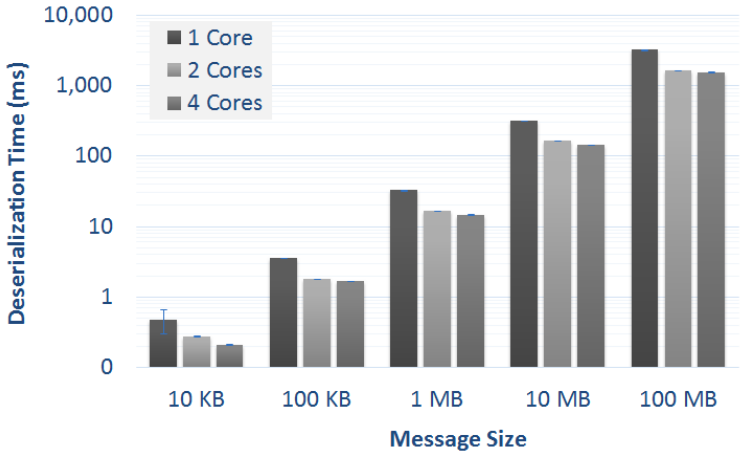
The evaluation results for a 4-core instance (m3.xlarge) are reported in Fig. 9(a) and Fig. 9(b). The execution time is at the same level of t2.medium for 1-core and 2-core.

Fig. 9(a) shows that for serializing small message (i.e., 10 KB) more cores do not necessarily improve the performance. This is because the overhead for splitting the original message and the multiplied disk write overhead outweigh the gain from the parallelism. Moreover, we observe that even for medium to large messages, 4-core improvement over 2-core is not as significant as that of 2-core over 1-core. We believe it can be best explained as that in m3.xlarge virtual machine 4 vCPUs impose significant switching overhead that offsets the data parallelism.

DOI: 10.1002/cpe



(a) Serialization



(b) Deserialization

Figure 9. PPB on EC2 m3.xlarge instance

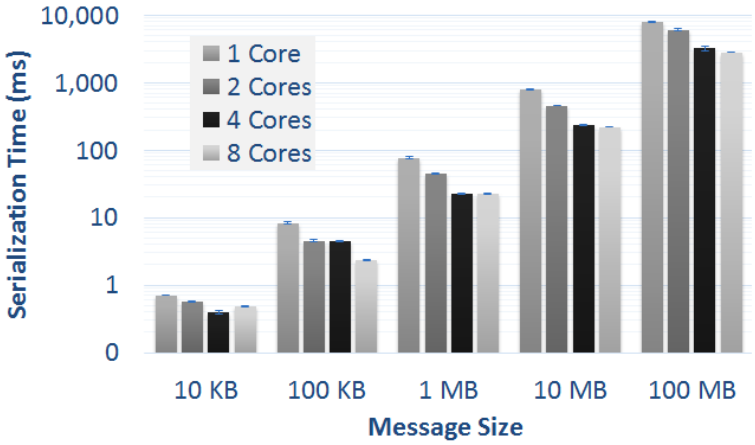
Fig. 9(b) shows the deserialization performance on m3.xlarge, which has a similar trend as serialization. That is, 4 cores do not bring significant improvement as 2 cores. The only exception is 10 KB where 4 cores almost scale linearly over 2 cores. Yet this is not our major goal as the original serial deserialization is fast enough (i.e., within 1 ms).

For the largest instance in the general purpose category—m3.2xlarge, we re-run the same workload and report the results in Fig. 10(a) and Fig. 10(b). We observe that in most cases 8 cores do not help improve much the performance. In some scenarios 8 cores even degrade the performance, such as serializing 10 KB and deserializing 100 KB. Therefore, although adding cores exploits more data parallelism, over-decomposing the data could cause performance degradation due to the potentially huge divide-conquer overhead.

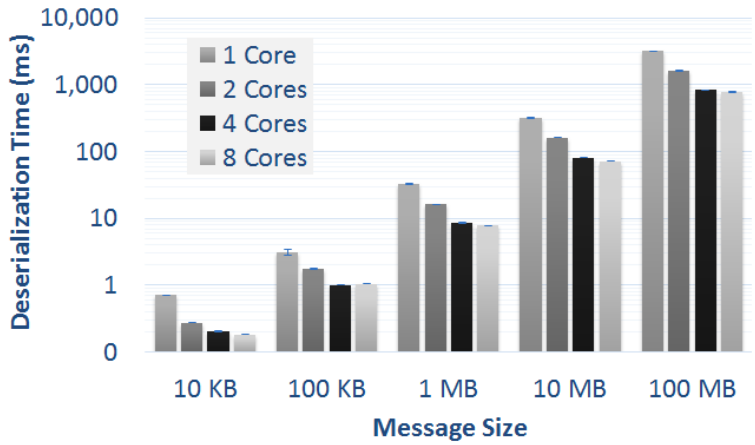
Lastly, we report the results of the 32-core instance types from the 3 optimized categories (i.e., compute-optimized, memory-optimized, and storage-optimized) in Fig. 11. Due to limited space, only the largest instances are considered and compared; that is, 16-cores and smaller cases are not shown here. Again, a similar trend is observed: for small messages the parallelism is not obvious for improved performance; it is the large message where PPB significantly outperforms Protocol Buffers.

While we provided a quantitative analysis on the optimality (see Equation 1, Section 3) about the number of cores concurrently processing the serialization, in practice the rule is much simpler. Our

DOI: 10.1002/cpe



(a) Serialization



(b) Deserialization

Figure 10. PPB on EC2 m3.2xlarge instance

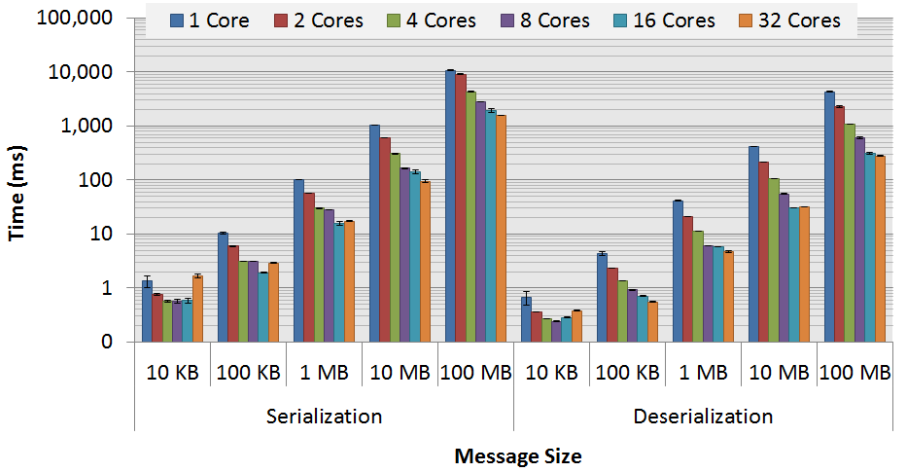
experiments showed that as long as the package size is non-trivially large (in terms of megabytes or more), current mainstream systems' configurations (i.e., up to 32 cores) are able to leverage the data parallelism and thus enhance the I/O performance. As a case in point, in Figure 11(c) we observed the with a single core the deserialization time of a 100 MB message took 4,743 ms while 2 cores reduce the time to 2,487 ms. If we calculate the overhead and plug the values into Equation 1, the optimal number of cores is 67. That said, the parallel approach introduced in this paper is much applicable in current mainstream system configurations.

5.4. PPB on the Supercomputer

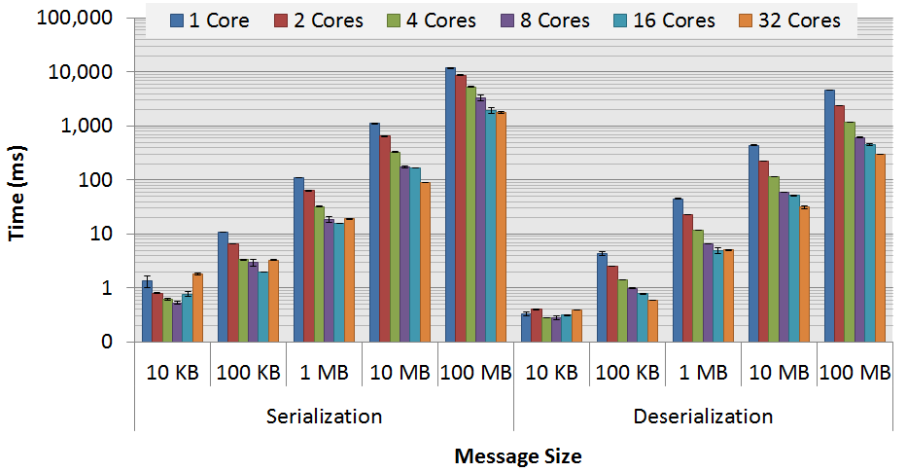
This section demonstrates one PPB use case where it accelerates the metadata performance in a distributed file system designed for world's top supercomputers. The goal of this experiment is two-fold as follows. First, it illustrates that PPB is beneficial to real-world applications in addition to the micro-benchmarks we show in Section 5.2 and Section 5.3. Second, it showcases the end-to-end performance improvement in a large-scale distributed system—Intrepid [18], one of world's fastest supercomputers when launched.

Specifically, we demonstrate how PPB could accelerate the processing speed for large directories in the FusionFS [41] file system in the following discussion. A large directory comprises many

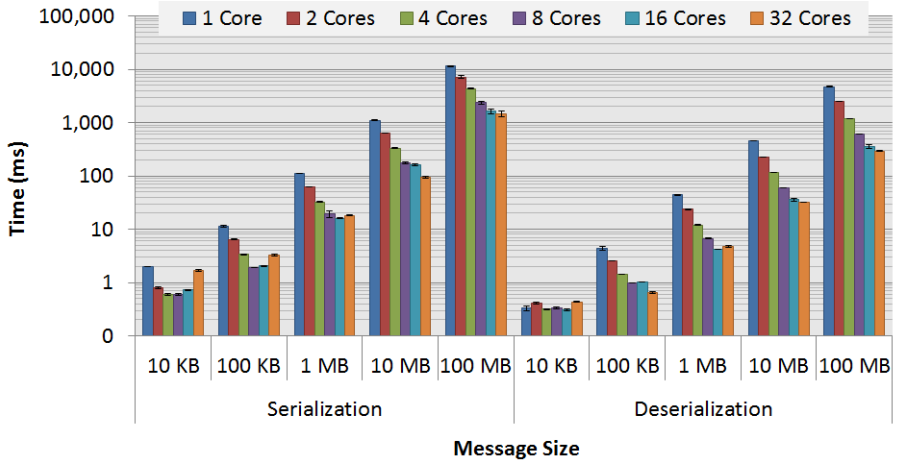
DOI: 10.1002/cpe



(a) c3.8xlarge



(b) r3.8xlarge



(c) i2.8xlarge

Figure 11. Large instances of optimized categories

DOI: 10.1002/cpe

file entries and poses an unprecedented challenge for metadata management [28] in extreme-scale systems. Before discussing PPB performance at this scale, we provide a brief overview of FusionFS.

The Fusion distributed file system (FusionFS [41]) was initially designed to address the I/O bottleneck in the conventional high-performance computing (HPC) systems. The state-of-the-art architecture of HPC systems have all their data stored in the remote storage nodes (for example, GPFS [29]). Therefore, every single I/O has to be transferred via the interconnect between compute and storage nodes. FusionFS, in order to ameliorate the performance bottleneck of the shared network, breaks the accepted practice of compute-storage separation and manipulates all its metadata and data stored right on the compute nodes. As a result, many I/O requests that used to be transferred over the network are now completed by local system calls. Our evaluations on both a real implementation and a simulation model confirm the superiority of FusionFS over the conventional parallel file system; the peak I/O throughput on 16K nodes of an IBM Blue Gene/P supercomputer (i.e., Intrepid [18]) reached 2.5 TB/s surpassing Titan [32] (1.4 TB/s), one of the fastest production storage system in the world ([33]).

One big challenge in FusionFS is how to support large directories. In theory, all metadata are distributed and balanced on all nodes because internally FusionFS maintains a distributed hash table (DHT) to manage them. Yet, one issue with DHT is that each directory information is stored as a single key-value pair. That is, the pathname is the key while all its entries are in the value of the key. Therefore, if the directory comprises many entries, the value becomes extremely large. Because the metadata is all distributed, these large key-value pairs need to be transferred between multiple nodes. This is exactly where data (de)serialization plays in: whenever the large directory is touched, it needs to be encoded into a serialized format for network transfer and then decoded back for the file system manipulation.

As a case in point, one of workloads we evaluate for FusionFS metadata is to let each of 1,024 nodes create 10,000 empty files in the same shared directory. That is, we expect a single directory to hold roughly 10 million files. The average length of each file name is 10, meaning that each file entry takes 10 bytes in the metadata. This roughly results in a total 100 MB message to be serialized, transferred, and deserialized.

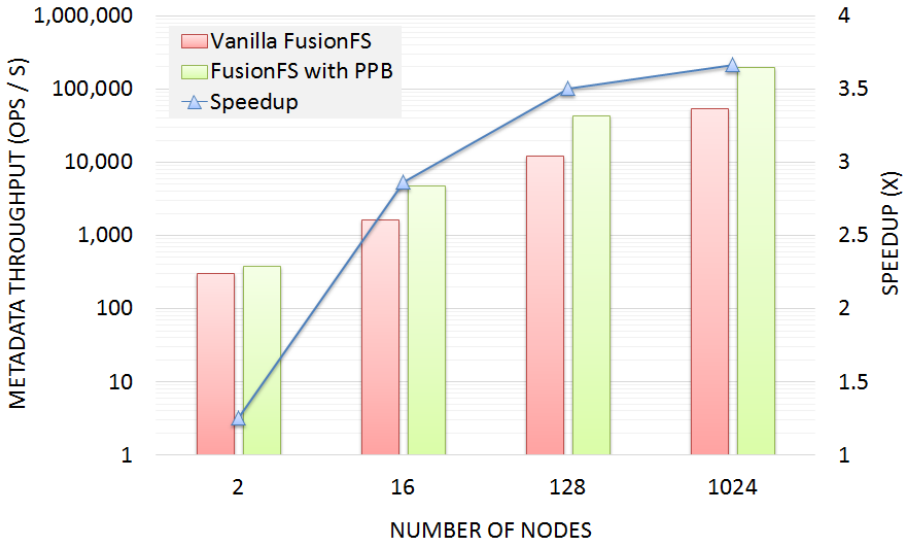


Figure 12. FusionFS metadata throughput on Intrepid [18]

Fig. 12 compares the metadata performance with and without PPB for the workload mentioned above on Intrepid [18]. We simulated FusionFS's performance with PPB enabled on up to 1,024 nodes; The simulation was run on the FusionSim simulator [39] that proved to be highly accurate. The red bar measures the performance with single-core Protocol Buffers. Based on our analysis in Section 3 and system configuration, the green bar predicts how the overall performance could

DOI: 10.1002/cpe

be improved by four CPU cores of each Intrepid compute node. For small scale such as 2 nodes, multi-cores do not bring much improvement. Nevertheless, for large scales the gap is significant; in particular, a 3.6X speedup is expected on 1,024 nodes.

6. DISCUSSION AND LIMITATION

6.1. User Interface

PPB is currently implemented as an MPI wrapper up on Google's Protocol Buffers. Therefore, users would need to make some modifications to application's source code. The change, however, is slight, if not minimal. We name the API names similar to those of Protocol Buffers [27]. For example, the original method `ParseFromCodedStream()`, which is to deserialize the given serialized stream, is now changed to `MPIParseFromCodedStream()` in PPB.

Our long-term goal is to have the parallelism option built in the future release of Protocol Buffers itself. There would be new challenges brought by this design though. One thing worth mentioning is that all the classes and methods provided by Protocol Buffers (the C++ version) are automatically compiled from a "proto"— a user-defined data structure following a C-like syntax. In other words, the parallelism code cannot be directly integrated to the original API; instead, the code change will need to be generated by Protocol Buffers' compiler. We will work closely with Google's Protocol Buffers team to figure out the new design for the parallelism support in the future release.

6.2. Dynamic Object Splitting

As mentioned in Section 3, current PPB design assumes the serialization occurs in a homogeneous machine. Thus it makes sense to split the large object into sub-objects of the same size. It is a fair question, however, to ask what if the system is heterogeneous.

As a case in point, more and more servers are equipped with high-end GPUs, which comprises a massive number (for example, O(100)) of computing cores. If we want to offload some sub-objects from CPU to GPU, the current design of PPB will need to change. The new design needs to consider the fact that the time for a GPU core to serialize a sub-object of the same size might take quite different time than a CPU core. A straightforward solution would be to split the large object into different sizes of sub-objects. The size would be proportional to the computing capacity (i.e., in GFlops) of CPU and GPU cores. We will also need to consider other factors; for example moving objects between CPU memory and GPU memory also introduces new overhead, which hopefully could be amortized by the performance gain from the massive GPU parallelism.

7. RELATED WORK

7.1. Data Interchange

Many serialization frameworks are developed to support transporting data over distributed systems. XML [14] represents a set of rules to encoding documents or text-based files. Another format, namely JSON [19], is treated as a lightweight alternative to XML in web services and mobile devices as well. While XML and JSON are the most widely used data serialization format for text-based files, binary format is also gaining its popularity. A binary version of JSON is available called BSON [9]. Two other famous binary data serialization frameworks are Google's Protocol Buffers [27] and Apache Thrift [7]. Both frameworks are designed to support lightweight and fast data serialization and deserialization, which could substantially improve the data communication in distributed systems. The key difference between Thrift and Protocol Buffers is that the former has the built-in support for RPC.

Many other serialization utilities are available at the present. Avro [3] is used by Hadoop for serialization. Internally, it uses JSON [19] to represent data types and protocols and improves the performance of the Java-based framework. Etch [5] supports more flexible data models(for example,

DOI: 10.1002/cpe

trees), but it is slower and generates larger files. BERT [8] supports data format compatible with Erlang's binary serialization format. Message Pack [23] allows both binary data and non UTF-8 encoded strings. Hessian [16] is a binary web service protocol that is 2X faster than the Java serialization with significantly smaller compressed data size. ICE [17] is a middleware platform that supports object-oriented RPC and data exchange. CBOR [10] is designed to support extremely small message size.

None of the aforementioned systems, however, support data parallelism. Thus they suffer the low efficiency problem when multiple CPU cores are available particularly when the data is large in size. PPB, on the other hand, takes advantage of the idle cores and leverage them for parallelizing the compute-intensive process of data serialization.

7.2. Parallel Data Processing

Many frameworks are recently developed for parallel data processing. MapReduce [13] is a programming paradigm and framework that allows users to process terabytes of data over massive-scale architecture in a matter of seconds. Apache Hadoop [6] is one of the most popular open-source implementations of MapReduce framework. Apache Spark [37] is an execution engine which supports more types of workload than Hadoop and MapReduce.

Numerous efforts have been devoted to utilizing or improving data parallelism in cluster and cloud computing environment. Lee et al. [21] presented how to reduce data migration cost and improve I/O performance by incorporating parallel data compression on the client side. Klasky et al. [20] proposed a parallel data-streaming approach with multi-threads to migrate terabytes of scientific data across distributed supercomputer centers. Some work [1, 11, 35, 36] proposed data-parallel architectures and systems for large-scale distributed computing. In [2, 34], authors exploited the data parallelism in a program by dynamically executing sets of serialization codes concurrently.

Unfortunately, little study exists on data parallelism for data serialization, mainly because large messages are usually not the dominating cost by convention. This work for the first time identifies that large message is a challenging problem from our observations on real-world applications at Google. We hope our PPB experience could provide the community insights for designing the next-generation data serialization tools.

8. CONCLUSION

The fact that a significant overhead is incurred when serializing and deserializing large messages in modern parallel and distributed systems calls for a revisit to our conventional wisdom. This work pinpoints the root cause of the performance bottleneck on data serialization and deserialization—serial processing unaware of idle CPU cores, and proposes parallel serialization and deserialization to address the issue. The key idea is to split the data and leverage multiple computing cores to parallelize coding procedures. While data parallelism is not a new idea in general, it has never been applied to data serialization and deserialization, thus posed several new challenges such as divide-conquer strategies and the data-splitting granularity. We explore the design space according to the proposed approach and analyze its abstraction in depth. The system prototype is implemented and deployed on a variety of test beds from a single-node Linux server, to the Amazon EC2 cloud and an IBM Blue Gene/P supercomputer. Experiments confirm the effectiveness of the proposed approach to overcome the new performance bottleneck brought by interchanging large messages in modern distributed systems at scale.

In future we plan to work on the following two main directions of PPB. First, we will deploy PPB on more production systems to test its applicability under various workloads. Second, we will extend the data-parallelism approach to other serialization tools such as Thrift [7].

DOI: 10.1002/cpe

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under awards OCI-1054974 (CAREER). This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This work is also partially supported through the Amazon research grant.

REFERENCES

[1] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. Law, and M. Papka. Large-scale data visualization using parallel data streaming. *Computer Graphics and Applications, IEEE*, 21(4), Jul 2001.

[2] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, 2009.

[3] Apache Avro. http://avro.apache.org/, Accessed December 13, 2014.

[4] Apache Camel. http://camel.apache.org/, Accessed December 7, 2014.

[5] Apache Etch. https://etch.apache.org/, Accessed December 13, 2014.

[6] Apache Hadoop. http://hadoop.apache.org/, Accessed September 5, 2014.

[7] Apache Thrift. https://thrift.apache.org/, Accessed December 8, 2014.

[8] BERT. http://bert-rpc.org/, Accessed December 13, 2014.

[9] Binary JSON. http://bsonspec.org/, Accessed December 13, 2014.

[10] CBOR. http://cbor.io/, Accessed December 13, 2014.

[11] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of USENIX Symposium on Opearting Systems Design & Implementation*, 2004.

[14] Extensible Markup Language (XML). http://www.w3.org/xml/, Accessed December 13, 2014.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, 2003.

[16] Hessian. http://hessian.caucho.com/, Accessed December 13, 2014.

[17] ICE. http://doc.zeroc.com/display/ice34/home, Accessed December 13, 2014.

[18] Intrepid. https://www.alcf.anl.gov/user-guides/intrepid-challenger-surveyor, Accessed September 5, 2014.

[19] JSON. http://www.json.org/, Accessed December 8, 2014.

DOI: 10.1002/cpe

[20] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid-based parallel data streaming implemented for the gyrokinetic toroidal code. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, 2003.

[21] J. Lee, M. Winslett, X. Ma, and S. Yu. Enhancing data migration performance via parallel data compression. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, 2002.

[22] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, 2013.

[23] Message Pack. http://msgpack.org/, Accessed December 13, 2014.

[24] MPICH. http://www.mpich.org/, Accessed December 10, 2014.

[25] Open MPI. http://www.open-mpi.org/, Accessed December 10, 2014.

[26] OpenMP. http://openmp.org/wp/, Accessed December 9, 2014.

[27] Protocol Buffers. http://code.google.com/p/protobuf/, Accessed September 5, 2014.

[28] K. Ren, Q. Zheng, S. Patil, and G. Gibson. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *IEEE/ACM International Conference on Supercomputing*, 2014.

[29] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[30] W. Schwitzer and V. Popa. Using protocol buffers for resource-constrained distributed embedded systems. Technical report, Technische Universitaet Muenchen, 2011.

[31] SWI-Prolog. http://www.swi-prolog.org/, Accessed December 7, 2014.

[32] Titan. https://www.olcf.ornl.gov/titan/, Accessed December 10, 2014.

[33] Top500. http://www.top500.org/list/2014/06/, Published June 2014; Accessed September 5, 2014.

[34] M. Voss and R. Eigenmann. Reducing parallel overheads through dynamic serialization. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, IPPS '99/SPDP '99, 1999.

[35] D. Warneke and O. Kao. Nephele: Efficient parallel data processing in the cloud. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, 2009.

[36] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[38] ZeptoOS. http://www.mcs.anl.gov/zeptoos, Accessed September 5, 2014.

[39] D. Zhao, N. Liu, D. Kimpe, R. Ross, X.-H. Sun, and I. Raicu. Towards exploring data-intensive scientific applications at extreme scales through systems and simulations. *IEEE Transactions on Parallel and Distributed Systems*, (10.1109/TPDS.2015.2456896):1–14, 2015.

DOI: 10.1002/cpe

[40] D. Zhao, J. Yin, K. Qiao, and I. Raicu. Virtual chunks: On supporting random accesses to scientific data in compressible storage systems. In *Proceedings of IEEE International Conference on Big Data*, pages 231–240, 2014.

[41] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. FusionFS: Toward supporting data-intensive scientific applications on extreme-scale distributed systems. In *Proceedings of IEEE International Conference on Big Data*, pages 61–70, 2014.

DOI: 10.1002/cpe