TÉCNICO LISBOA

Accelerating Machine Learning with GASPI-based Parameter Servers

Rafael Pestana de Andrade

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Pedro Faria Mendonça Barreto
Prof. Rodrigo Seromenho Miragaia Rodrigues

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. João Pedro Faria Mendonça Barreto
Member of the Committee: Prof. Paolo Romano

November 2021

Acknowledgments

Firstly, I want to thank professors Rodrigo Rodrigues, João Barreto and José Monteiro, for all the corrections, sugestions and guidance provided during the thesis, as well as their availability in the final weeks of the thesis report and dissertation delivery. I am also grateful for the opportunity I had to learn from them as a student.

I am also grateful to Tom Vander Aa and Roman Iakymchuk; to Tom for providing access to the cluster at critical phase in record time, and for his great patience in answering my questions; and to Roman for opening doors and providing helpful insights.

Finally, I have to thank my family for being patient with me in this last academic task; and also to a multitude of friends which offered their support and companionship, particularly my classmates.

Non nobis, Domine, sed nomini Tuo da gloriam

Abstract

When using distributed Machine Learning, one problem that arises is how to communicate efficiently the contributions made by the various computing nodes; one of the most common approaches is the Parameter Server architecture. Recent work has improved this approach and achieved state-of-the-art speed records in Machine Learning tasks, but focused on synchronous and consistent settings, in which the computing nodes are executing a given task in lockstep; this imposes a synchronization overhead which may penalize large distributed systems, namely those that can resort to asynchronous execution and relaxed consistency models. We present Lapser, a communication library that provides one-sided communication with configurable consistency and an easy-to-use programming interface. Lapser was implemented over GASPI, taking advantage of its powerful abstraction and capabilities. The evaluation results shows that Lapser has a competitive performance when compared with custom-made application specific communication routines.

Keywords

Parameter Server, Stale Synchronous Parallel, One-Sided Communication, Machine Learning

Resumo

No ramo de Aprendizagem Automática distribuída, um dos problemas de fundo é como comunicar as contribuições feitas pelos vários nós de computação de forma eficiente; uma das abordagens mais comuns é a arquitectura de Servidor de Parâmetros. Avanços recentes melhoraram esta abordagem e conseguiram recordes de velocidade de estado da arte em várias tarefas de Aprendizagem Automática, mas esses avanços estavam focados em cenários síncronos e consistentes, nos quais os nós de computação estão a executar uma dada tarefa ao mesmo ritmo; isto impõe um custo adicional de sincronização que pode penalizar grandes sistemas distribuídos, nomeadamente aqueles que podem recorrer a execução assíncrona e modelos de consistência relaxada. Com esta tese apresentamos Lapser, uma biblioteca de comunicação que oferece comunicação unilateral com consistência configurável e uma interface de programação fácil de usar. Lapser foi implementado sobre GASPI, aproveitando as suas poderosas abstrações e capacidades. Os resultados da avaliação mostram que Lapser tem uma performance competitiva quando comparado com rotinas de comunicação customizadas à aplicação.

Palavras Chave

Servidor de Parâmetros, Atraso Síncrono Paralelo, Comunicação Unilateral, Aprendizagem Automática

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

ML Machine Learning

RMSE Root Mean Squared Error

MF Matrix Factorization

BPMF Bayesian Probabilistic Matrix Factorization

PS Parameter Server

API Application Programming Interface

BSP Bulk Synchronous Parallel

A-BSP Arbitrary-BSP

SSP Stale Synchronous Parallel

HPC High-Performance Computing

RDMA Remote Direct Memory Access

GASPI Global Address Space Programming Interface

NIC Network Interface Controller

1

Introduction

In the last few years, the field of Machine Learning (ML) has observed an ever-growing need for computing power due to the increasing amounts of training data and model complexity [2]; as a consequence, the field has increasingly looked at using distributed computing, which presents new challenges on how to run ML algorithms in the most efficient way.

A key strategy to increase the efficiency of distributed computing is to overlap computation and communication whenever possible to avoid waiting times. In ML's case, many of its algorithms are *iterative convergent* algorithms, which means that they repeat the same calculations multiple times, varying some internal state, in order to reach a desirable one. When adapting these algorithms to run in a distributed fashion, there is the question of how to share the updates to this shared state among nodes in the most efficient way, as no progress can be achieved while the computing nodes wait for the propagation of the next state; in other words, sharing the intermediate state is a synchronization point, preventing a beneficial overlap between computation and communication. As such, this is a prime target for optimization, which is the focus of a large body of ongoing research.

At the architecture level, one can choose between two main approaches, collectives and Parameter Server (PS) [2]. In the first approach, the updates are directly communicated between computing nodes, which then apply over their local copy of the state, using an operation such as an allreduce; the latter approach relies on the existence of nodes (the ”parameter server”) which receive the contributions from the computing nodes (called workers), apply the updates internally, and then distribute the new state to the workers [3]. Note that the PS is not necessarily a single machine, and there are multiple implementations of this architecture which shard the PS or even blur the distinction between these paradigms [2–4]. The differences between the allreduce and PS implementations are numerous, and result in quasi-symmetrical strengths and weaknesses: while the allreduce implementations frequently are network bandwidth-optimal and excel in scenarios where computing nodes are symmetrical and have balanced workloads, they incur in increased latency on the communication of the updates and may not support asynchronous operation [2]. In turn, PS implementations are prone to inefficient bandwidth usage on either workers or servers, needing special care to ensure that neither component constitutes a bottleneck due to a wrong configuration; however, PSs may exhibit reduced latency since an update needs to take less hops to reach its consumers and can easily take advantage of not having to synchronize the workers among themselves, using relaxed consistency models [2, 3].

Being easily amenable to relaxed consistency models is an especially promising advantage of PS, which has received considerable attention in recent literature. More concretely, one can try and relax the consistency of the intermediate states, relying on iterative convergent nature of ML algorithms to achieve a final acceptable result. For instance, Arbitrary-BSP (A-BSP) and Stale Synchronous Parallel (SSP) are consistency models which offer the possibility of choosing a compromise between the correctness of the intermediate state and relaxing the synchronization barrier, by allowing some of the workers to be a

bounded number of iterations ahead of others [5]. These particular models have been shown to produce benefits in previous weakly-consistent PS implementations [3, 6].

Another recent trend on distributed ML research to increase the communication-computation overlap is to use one-sided communication, as supported by the advent of Remote Direct Memory Access (RDMA) network technologies – this means that a node can directly send information to another potentially without needing any coordination between them, enabling the receiving node to continue computation. This is in contrast with traditional systems, where communication is handled by the operating system and therefore must interrupt an ongoing computation to gain access to the CPU, which imposes a coordination overhead. This and other advantages of RDMA led to a proliferation of work trying to use this communication technology, with success, including allreduce and PS implementations [7, 8].

To the best of our knowledge, and despite all these important advances, there are still no libraries which combine these two promising trends, namely weakly-consistent PS and one-sided communication. Therefore, our thesis aims at filling this gap; with that goal, we present Lapser, a communication library which provides a Parameter Server interface, supporting relaxed consistency models and using one-sided RDMA technology. By combining relaxed consistency with one-sided communication, we obtained a system which has low overheads, being competitive with communication routines which were tailored for the specific application.

To achieve our goal, we faced some technical challenges, such as how to balance the amount of inconsistency we can tolerate in the Machine Learning (ML) training processes; also, implementing SSP with one-sided communication can be non-trivial. For example, using one-sided communication means that a number of data races are possible between the local and the remote machines, and one must be careful in detecting and dealing with these situations – in particular, while one could use coordination protocols to signal that a given piece of memory is safe to access, that would imply to worker synchronization, potentially defeating the purpose of using SSP and asynchronous communication.

Lapser surpasses these challenges by leveraging the insight that some ML algorithms can be structured in such a way that the data follows a single-writer policy, therefore eliminating write-write data races; the remaining data races are dealt with using integrity validation and retry mechanisms; finally, the use of fine-grained consistency control helps to control precisely the consistency guarantees demanded by the application.

We implemented the Lapser library using GASPI, an abstraction over RDMA; then, we tested it in BPMF, a Matrix Factorization application, and evaluated in a supercomputer cluster up to 16 nodes, with realistic datasets. We achieved communication times comparable to previous communication routines that were specifically tailored to take advantage of the problem characteristics, and observed that the use of relaxed consistency can speed up the time to finish 500 iterations by up to 40% (when compared to Lapser using no slack), while having marginal impact on solution quality.

The rest of this document is organized as follows. In chapter 2 we give some introductory background and introduce some key concepts regarding ML computations and distributed systems, and give an overview of existing solutions for the problem of communicating shared state in Machine Learning clusters; we then proceed to chapter 3, where we present the requirements and a solution to respond to them, along with architectural details. Chapter 4 details how to use the Lapser library in an existing application, which is then used in Chapter 5 to show an evaluation of our system against existing hand-optimized solutions on a specific ML task with real-world datasets; and in the final chapter we present a short conclusion and point to future directions to explore.

2

Background and Related Work

Contents

2.1 Machine Learning as a search for parameters

One of the most active fields of research in Machine Learning is called Supervised Machine Learning. The main goal in this field is to, given a set of examples (called a *dataset*), each comprised of attributes and a label, construct a function such that given the attributes it returns a prediction for the label value, even for attributes not found in the original examples. For example, we could try to build a system to predict the daily weather; in this case, each example would pertain to a different day, where the label would be the various possible weathers (sun, rain, storm, etc.) and the attributes would be meteorological information, such as the humidity, air pressure, previous day weather, among others.

The functions used for these predictions take the form of parameterized algorithms – where the outcome is given by a combination of attributes and some parameters (specific to the algorithm, and typically numeric tensors). By using this strategy, the original problem becomes a search for the right parameter values; and this process is often called *training*. There are various ways to do this training, depending on the specific algorithm, but one common example is *Stochastic Gradient Descent* [2]. In this algorithm, we repeatedly pick one of the examples from the given dataset, generate a prediction, compare it with the ground truth (given by the example's label) and we produce a *gradient*. Then, we apply this gradient to the parameters according to a *weight update rule* (or *optimizer*), generating a new version of these parameters (also known as *weights*) – these weight update rules are actively researched in the area of *optimization*, but they can be as simple as multiplying the gradient by a constant and add to the respective parameters (called Learning Rate update rule [2]). This process stops after a fixed number of steps, or when sufficient accuracy is attained, or when the gradients no longer materially affect the parameter values.

As a concrete example, we can use Stochastic Gradient Descent in Low Rank Matrix Factorization, which is a technique used for recommendation systems. The typical example is movie ratings – these systems try to predict the rating a user gives to a certain movie, based on ratings given to other movies and by other users. The underlying idea is that the ratings given by users are determined by certain number of unobserved *features* and how users respond to them; and we try to use the known rankings to indirectly obtain the values of these features. More formally, having M users and N movies, we are given a sparse matrix R of size $M \times N$, where each row u represents a user and each column v represents movie; each position r_{uv} contains the rating user u gave to movie v (or is empty if no rating is known). By using Matrix Factorization, and using K latent features, we want to produce matrices U and V, of sizes $M \times K$ and $K \times N$ (respectively), such that the product of these matrices yields $\hat{R}$, a reasonable approximation of R (according to the known ratings):

$$U \times V = \hat{R} \approx R$$

We can compare the predictions of $\hat{R}$ with the known values of R, and produce the gradients we need to refine the current values of U and V. In this case, we can consider both matrices U and V as "parameters" and the values in R as our dataset, and each value r_{uv} an example from this dataset.

Note that there are other algorithms for Low Rank Matrix Factorization, such as Bayesian Probabilistic Matrix Factorization (BPMF) [9], which naturally use other methods to approximate the solution; the biggest difference to the gradient descent method outlined above in terms of output is that BPMF directly computes new U and V values, instead of updates.

One way of parallelizing these Machine Learning algorithms (and many others that are similar) is to treat the parameter values as a global state that is updated by various computing nodes. Then, we can give each node a subset of the training examples, making the training process distributed using what is called *data parallelism*.

2.2 Sharing the parameters

By distributing the training workload across multiple nodes, it becomes necessary to define how to efficiently communicate and aggregate their local results into a single, shared one. In the following subsections, we will analyze the two most common approaches: using an allreduce collective, and introducing a parameter server.

2.2.1 Allreduce

In the Distributed Systems field, we often employ what are called *collectives* (which is a shorthand name for *collective operations*). These are a set of communication patterns (involving a plurality or totality of computing nodes) that are commonly used in distributed applications; nearly all distributed computations employ at least one of these operations. These operations are used to redistribute data or to consolidate it according to a given function – called a *reduction operation*, such as summation or average. [1]

An example of a collective is the *allreduce* operation. In this collective, we have some data that is present in all nodes, and we wish to consolidate it but, instead of having a single node collect the final result (which would be the *reduce* collective), we want the result to be available to all nodes. Table 2.1 shows a diagram summarizing the effects of an allreduce operation.

Node 0	Node 1	Node 2	Node 3
$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$

(a) Before, each node has a part

Node 0	Node 1	Node 2	Node 3
$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$

(b) After, all nodes have the combined result

Table 2.1: Illustration of the result of the allreduce operation, inspired by Figure 1 of Chan et al. [1]. Note that summation is not the only possible reduction operation.

A common way to implement allreduce is to divide the algorithm in two phases:

1. The *reduce-scatter* phase – Each node receives a part of the data to be reduced, uses the reduction operation to combine the local results with the received data, and sends this partial-reduction result to another node

2. The *allgather* phase – After enough steps of the previous phase, each node will have a fraction of the data containing the combined results of all nodes; now, they repeatedly pass these results, until all nodes have the full reduced data.

This method can be adapted to have better performance, according to the existing network topology. For example, in a ring topology, one can use the *ring allreduce* algorithm, illustrated in Figure 2.1. Note that a given node always sends to the same node, and also always receives from the same node (but different from the one it sends to) – which, in a ring topology, would be its immediate neighbours. This algorithm is bandwidth-optimal, since every part of the parameter space is only sent once and received once, which is the absolute minimum communication necessary to propagate their contribution and also receive the contributions from the other nodes.

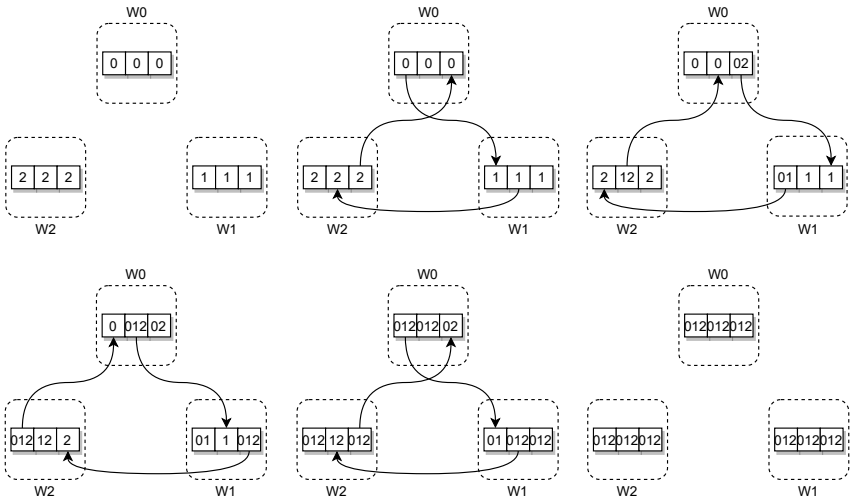


Figure 2.1: Visualization of the communication of the ring allreduce algorithm. Computing nodes are represented in rounded dotted squares, communications are represented as black arrows, and the reduction operation of two digits results in their concatenation. The top three stages belong to the reduce-scatter phase, the bottom to the allgather phase.

As a final note, we can use allreduce in distributed ML to average the gradients across the worker nodes, and then allow the nodes to use the weight update rule with the result of the collective operation.

2.2.2 Parameter Servers

Alternatively, we can use *parameter servers*, which were introduced to distributed Machine Learning by Li et al. [3].

In this approach, each node can have one of either two roles:

- The worker nodes are responsible for the training; each one has a part of the training dataset, and uses it to compute gradients according to the latest version of the parameters.

- The parameter servers receive the gradients and aggregate them, and depending on the implementation, may also apply the weight update rule.

Parameter Servers (PSs) are logically centralized components of the infrastructure, as depicted in Figure 2.2, contrary to the allreduce approach, which is decentralized. If implemented in a simple centralized fashion, the PS approach can easily perform worse than any alternative due to the system bottleneck in terms of communication around the one node serving as a server instance; but early on the need to distribute the parameter server role was evident, and when presenting the framework Li et al. [3] already presented the parameter server as a task to be distributed with various mechanisms to distribute load across instances.

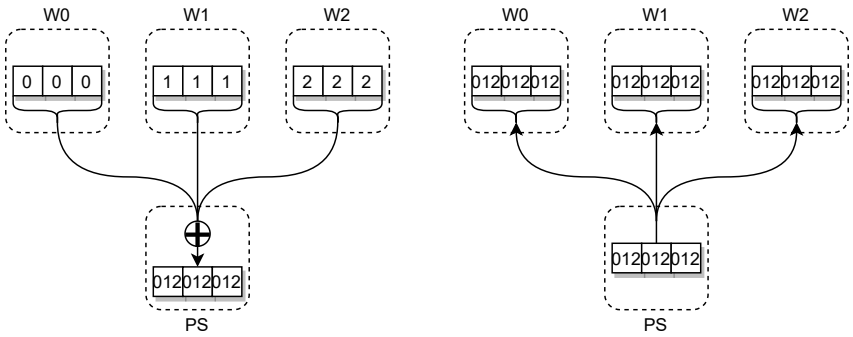


Figure 2.2: Visualization of the communication of the Parameter Server paradigm. Nodes are represented in rounded dotted squares with “PS” being a parameter server instance while “W0”, “W1” and “W2” are computing nodes; communications are represented as black arrows, and $\oplus$ represents the reduction operation. On the left, we have the workers pushing the updates, and on the right, they are pulling the latest state.

The main motivation of the PS framework is to increase the scalability of the distributed systems used in Machine Learning, to enable running workloads with unprecedented amounts of data in the training process; as such, the implementations of this paradigm focused initially on features to help with this goal, such as allowing relaxed consistency models, elastic scalability of workers and servers, and fault tolerance.

2.2.3 Comparing the approaches

We will first analyse these approaches in terms of their bandwidth and latency. In this analysis, we have k the number of worker processes, p the number of parameter servers (lower than or equal to k), and will consider the case where we have a single parameter of size M, which is updated by all k nodes.

In the PS approach, we can divide the parameter in shards of size M/p; which means that each worker, after producing the updates, will have to send p messages of that size, for a total send size of $p \times M/p = M$; and will receive from each parameter server a fraction M/p of updated gradients – for a total received amount of M. The parameter servers, in turn, will receive k updates of size M/p, for a total of $k \times M/p$, and will have to send the same amount of information in the form of updated parameters.

However, if we allow a worker and a parameter server to share a single physical machine (called *colocated mode* in [8]), and assuming we have much faster communication between processes residing in the same machine, we can get a lower bandwidth usage: the workers will send and receive a total of $(p-1) \times M/p$, while parameter servers will have a send and receive size of $(k-1) \times M/p$. However, this means that a single machine now has to handle the traffic of the two processes, totalling $M/p \times [(p-1) + (k-1)] < 2 \times (k-1) \times M/p$ for the sent and received messages.

We will now analyze the allreduce approach when using the ring allreduce algorithm as presented in section 2.2.1. We first note that the parameter will be divided in k chunks of size M/k each. Then, at each stage, we have:

- In the reduce-scatter phase: $k-1$ steps where each node concurrently sends and receives a message of size M/k, all to the same neighbour (and after reception, performs a local reduction operation);
- In the allgather phase: $k-1$ steps where each node concurrently sends and receives a message of size M/k, all to the same neighbour

As such, each node will have a total send size of $2 \times (k-1) \times M/k$, which is the bandwidth lower bound for allreduce [1, 7, 8] and similar to the colocated case of the PS.

One thing that analyzing the bandwidth usage does not reveal is the communication latency of each of these methods, approximated by the number of "hops": while in the ring allreduce algorithm each node has a total of $2 \times (k-1)$ communication steps that do not take place parallel, in the Parameter Server model we have direct communications only – the workers communicate its updates directly to the servers, and when they want the latest state they request it directly from the servers. Additionally, although there are allreduce algorithms that reduce the latency, they are still limited by the lower bound of $\log_2 p$ [1].

Besides these low-level details, another significant difference between these two approaches lies in their interface. The allreduce approach is typically exposed as a function call, where we specify the

location of the data we want to reduce, the reduction operation, where to store the result and the nodes that will participate in this operation. In the Parameter Server (PS) approach, the most common interface (for the workers) is a set of two functions: push and pull. In the push call, we specify which parameters are we trying to update (typically using keys), along with the data; in the pull call, we also specify what we want to fetch from the server. This interface effectively allows for selective parameter updates, which means that in some ML tasks we can effectively send less data in the updates, while in the allreduce approach all nodes must communicate in a symmetrical fashion, meaning that all nodes must send updates even for parameters that remain unchanged. This interface, plus the fact that updates are only communicated with one "hop" also allows for the usage of advanced techniques such as message compression, which would be too time-consuming with allreduce approaches such as the ring allreduce, especially since, in the case of ring allreduce, each step receives updates, then applies the reduction operation, and then sends the result to another neighbour. Thus, if we used compression, we would need to compress and decompress $k-1$ times for each part of M, versus compressing and decompressing once in a parameter server approach. Finally, having a defined push and pull operations instead of a single function call facilitates dealing with unresponsive or late workers seamlessly. While it is possible to implement allreduce variants that mitigate these problems [10], the PS-based approaches can naturally deal with these events by allowing workers to pull even without having the newest updates (effectively allowing for asynchronous training), and eventually reassign the workload of the late workers to existing or new nodes, without ever suspending the work of the PSs.

2.2.4 Parameter server usage in practice

As stated in section 2.2.2, the first paper to introduce the Parameter Server approach as an unified concept is by Li et al. [3], and the primary driver for this approach was to increase the scalability of the training process in Machine Learning. Using a number of architectural decisions – such as using keys with ranges and operations defined on those ranges, asynchronous communication, possibility of using various types of consistency (including BSP and SSP), or user defined filters to assess relevance of updates – and implementation choices – vector clocks, or sparse encoding of updates – led to decreased communication and less time waiting while increasing tolerance to failures, which in turn led to lower convergence times. This approach inspired many implementations, such as TensorFlow [11].

In 2017, researchers at Uber attempted to improve the scalability properties of the distributed implementation of TensorFlow and created Horovod [7], which is the open-source version of an internal tool at the company to run distributed Machine Learning projects. Instead of implementing a Parameter Server architecture, they used *ring allreduce* because of its bandwidth optimality – relevant because many applications in the Machine Learning field are bandwidth constrained. This communication algorithm was already implemented by Baidu researchers in 2017 [12] and applied to Tensorflow [13]; and many sub-

sequent implementations of distributed Machine Learning built upon variations of ring allreduce (such as 2D-Torus allreduce in [14, 15]). The authors of Horovod [7] also introduced the idea of *Fusion Buffers*, which in essence consists of batching updates for multiple small parameters, instead of updating each parameter in a separate collective operation. By this time, the state-of-the-art training speed records were held by systems using the allreduce collective [14–16].

Recently, Jiang et al. [8] introduced a novel approach called BytePS, which unifies the parameter server and *allreduce* approaches. The major insight gathered by the authors was that, in typical Machine Learning workloads, the majority of the work is done using GPUs, leaving CPUs under-utilized. Therefore, they decided to partition the communication workload into two services: the Communication Service (CS) manages the communication between the local GPUs and CPUs, while the Summation Service (SS) receives the updates from CSes (local or remote), aggregates them, and returns the result back to the various CSes; the SS can be performed either by GPUs or the previously under-utilized CPUs. To minimize the communication time while balancing the bandwidth needs of CS and SS running on GPUs, the authors derive formulas to partition the SS workload between GPUs and CPUs. By additionally handling intra-node GPU communication in an efficient way, and balancing the use of the summation service to fully utilize the available bandwidth, they outperformed existing *allreduce* and parameter server implementations.

Later, Thankgakrishnan et al. [4] unveiled Herring, which is a parameter server implementation that, besides taking advantage of some networking infrastructure improvements, presents the idea of *Balanced Fusion Buffers* (*BFB*). These are distributed data structures where, besides aggregating updates to different variables in a single update, each parameter server is responsible for a predetermined fraction of each *BFB*, instead of being directly responsible for a variable. By using this technique, the system guarantees that the aggregation work is evenly distributed among parameter servers.

A significant difference between the mentioned Parameter Server-based approaches is the calculation that is made by the parameter server instances. The parameter server as initially introduced in [3] is responsible for running the (user-specified) optimizer (or weight update rule), having as input the received gradients (computed by the workers); later parameter servers do not attempt to run the full optimizer, because they have become progressively more complex, and so this part of the training process is also calculated by the worker nodes that are equipped with hardware to accelerate this computation. More concretely, in Herring [4] the workers send the gradient values (as in the older parameter servers), but the server only sums the received gradients and returns the result, just as in the allreduce approaches; while the BytePS architecture [8] has two modes of operation: in synchronous training, they also only sum gradients (just as the allreduce and Herring approaches); but to support asynchronous training, they can also sum deltas of the parameter values, obtained after the worker runs the optimizer, allowing the Summation Service (analogous to the parameter server) to keep the parameter values.

In both these implementations (Herring and BytePS), the servers only need to sum the contributions received from the workers; this simplification means that this component is less compute intensive and does not need the acceleration provided by the use of GPUs, usually reserved for the worker processes.

2.3 Staleness and synchronization

One of the strategies used to achieve better worker utilization in parameter servers as introduced by Li et al. [3] is the use of more relaxed consistency models. More specifically, they used the idea of "bounded delay", which can be also called *staleness*.

The paper that introduced the idea of *staleness* in the context of ML computations was Cipar et al. [5], which consists of the idea that we can use state that is relatively recent, instead of requiring the freshest updates from other nodes. This can be used to speed up iterative convergent algorithms, since these algorithms tolerate small intermediate errors (in this case, introduced due to staleness) but nevertheless will converge to a correct solution.

Later, the same authors applied their idea in Ho et al. [17], where they theoretically analyze SSP, and Cui et al. [6], where they test three synchronization approaches with different ML algorithms. The most common approach is what the authors called Bulk Synchronous Parallel (BSP), where at the end of iteration $t-1$ there is a barrier among all threads to exchange information, to ensure that during iteration t (also called clock t) all threads have access to the work that other treads completed at $t-1$. Then, the authors introduce two different models: Arbitrary-BSP (A-BSP) and Stale Synchronous Parallel (SSP). A-BSP comes from the realization that, within a given clock t, threads may be allowed to compute based on slightly stale information – they only have access to information with clock $t-1$, not the intermediate calculations of the current clock. Then, the idea is to increase the work done in each clock, defined as work per clock (WPC). The effect is that, for instance, by doubling WPC (doing twice the amount of work), the number of barriers where threads must wait are halved. Finally, the authors present the SSP model, as a generalization of the previous ones. Here, instead of having explicit barriers at the end of each clock, we have a parameter, called *slack* (s), that controls the maximum difference in clock between the slowest and the fastest threads; a thread in clock t has access to all shared state from a clock that is at least at $t-1-s$, and may have some parts of the state updated up to $t-1$. The advantage of SSP over A-BSP is the convergence speed in the presence of straggler effects (threads that are delayed in relation to others): SSP allows the other threads to make progress (until we hit the slack limit), whereas A-BSP (and BSP), due to the barriers, will not allow this progress. However, in some scenarios A-BSP can outperform SSP, namely if there are no significant stragglers, because it omits some communication phases, whereas SSP might try to communicate even if the state is fresh enough (this is, if the locally available state obeys the bound of $t-1-s$).

Applying this to the parameter server approach, this means that we can allow the faster workers to progress, even when some other workers may have been subject to some transient delay, or failed entirely and a replacement worker has not yet recovered all the missing iterations. We can apply this ideas to Machine Learning training, because the training algorithm (previously outlined in section 2.1) can be seen as an iterative convergent algorithm, where we *iterate* on the examples of the given dataset, and by successive modifications (according to the computed gradients) we *converge* on the example's label.

However, it should be noted that, while bounding the staleness can preserve the convergence guarantees, using inconsistent models frequently has a negative impact in Machine Learning results [18], by making each iteration of the training algorithm less effective, which may increase the total training time compared to a synchronous solution. Therefore, the slack value needs to be carefully tuned, according to the application (or automatically [19]).

Another way to mitigate the effects of staleness on the iteration quality is to use an aggressive prefetching strategy, instead of a conservative one. An aggressive prefetching strategy, in the context of SSP, means that the workers will try to gather newer versions of the shared state, even if it is fresh enough; a conservative strategy will try to minimize communication, only getting newer versions of the global state when the one stored locally is too old to be within the staleness bounds [3].

Finally, although these limited consistency models are typically associated with the Parameter Server architecture, because the initial papers demonstrated the improvements of staleness using a Parameter Server interface [3,5,6] – there is some research on implementing relaxed consistency on other communication formats, such as allreduce [20], and it presents good performance improvements over existing allreduce implementations.

2.4 One-sided communication using RDMA

In an High-Performance Computing (HPC) cluster, Remote Direct Memory Access (RDMA) is achieved with hardware solutions, such as InfiniBand, which is essentially a Network Interface Controller (NIC) with direct memory access to main memory that implements a communication protocol in hardware, bypassing both the operating systems and the CPU, achieving lower latency and potentially higher bandwidth. Programs can instruct the RDMA NIC to perform multiple operations by posting *verbs* to *queues* maintained by the NIC. There are verbs for multiple functions, such as atomic operations or data transfer with one or two-sided semantics. Directly using RDMA is possible, but can be challenging due to a number of low-level details.

2.4.1 GASPI - a Partitioned Global Address Space API

We decided to use a communication library which simplified the usage of RDMA features with a number of abstractions. Global Address Space Programming Interface (GASPI) [21] is, as the name indicates, an API specially designed to support a Partitioned Global Address Space interface, with a focus on asynchronous, one-sided communication. This means that we can declare **segments** on each node, and they will be available to be read and written to other nodes, without needing intervention of the node where the segment resides. To signal the completion of a given data transfer, we can use **notifications** – for instance, the sending node may issue multiple calls of the `gaspi_write` function and therefore start transferring data to the receiver (making use of RDMA if available); upon completion of all requests, it may send a notification via a `gaspi_notify` call. Meanwhile, the receiving node can continue making progress, undisturbed by the ongoing data transfer; when this node wants to check if any transfer occurred, it can use `gaspi_notify_waitsome` to receive notifications directed at it.

3

Design of Lapser

Contents

In this chapter, we present the architectural and some implementation details, as well as some rationale for the underlying decisions, of Lapser – which stands for **LA**zy **P**arameter **Se**rver with **R**DMA.

3.1 Data model and Application Programming Interface

3.1.1 Data model

The Lapser library is to be used by a given distributed application composed of multiple processes, henceforth called workers. The workers can use Lapser to share items, which are identified by a numeric id, also known as key, and have associated values, which do not have any particular meaning – internally they are treated as arrays of bytes with a fixed size. Each item value also has an associated clock, which is a version number; each new value should update the clock. Each item can only have one worker submitting updates – which is its producer – and may have multiple workers interested in receiving the item – multiple consumers.

3.1.2 Operations

The workers can interact with the items stored in Lapser via an Application Programming Interface (API) inspired in LazyTable [6], which is presented below.

- **lapser_set(in: key, in: new value, in: clock)** This call sets a new value for the parameter identified by **key**, a numeric identifier, declaring the version to be **clock**.
- **lapser_get(in: key, out: destination, in: clock, in: slack)** This function retrieves the value identified by **key**, storing it in the memory location pointed by **destination**. In case the specified item is not available within the specified **slack** considering the current **clock**, this function may wait or indicate to the caller that the request was not fulfilled.
- **lapser_init(in: number of items, in: size of each item, in: items to produce, in: items to consume)** This call initiates the infrastructure and the necessary metadata in order to communicate an upper bound of **number of items** with the specified maximum size. Simultaneously, it declares that the caller which items the caller will get and set.
- **lapser_finish()** Frees the resources associated with the library.

3.1.3 API properties

This API hints at some deviations from the common PS model, which are motivated and analyzed below.

3.1.3.A No support for multiple workers update a single item

The most important variation and restriction is the lack of support for multiple writers of the same item, which means that each item can only have exactly one producer. This can be seen a major roadblock for the adoption of Lapser in some ML applications, although others can benefit from this simplification.

The reasons that led to this architectural decision are multiple; namely: that there already are many systems which can combine contributions from multiple computing nodes, and notable among them is [20], which presents the advantage of SSP with an allreduce implementation. Another reason is that BPMF does not generally need to combine contributions from a multitude of nodes. And, when a subset of items need to be combined, the application programmer can use a multitude of tools to deal with those specific items, such as the above mentioned [20] or even other communication methods like MPI; there is also the possibility of using Lapser to store the individual contributions from each node and aggregate them as needed at the application level. This was necessary during our evaluation; please refer to section 4.3 for a description of a concrete example.

3.1.3.B Omission of update and prefetch operations

Compared with the APIs presented in [6] and [3], the most significant difference is the omission of an update mechanism based on partial contributions (also called deltas), as well as the lack of a prefetch operation (called `refresh` in [6]).

The lack of update primitives reflects the lack of support for items with multiple producers, as well as to avoid the complexity inherent to having to configure and support arbitrary reduction functions. Also, our showcase application BPMF did not produce updates, only new values so this feature became deprioritized as well; the lack of prefetch is also due to lack of applicability in the BPMF program.

3.1.3.C Consistency models and slack

The **lapser_get** call has the following consistency property: if an item is requested with given clock t and a slack of s, then the item age is at least $max(t - s, 0)$. This matches the properties enunciated in section 2.3 needed for SSP.

However, SSP can be seen as a generalization of stronger consistency models, such as A-BSP or even BSP, according to the usage of the slack parameter that is used on the **lapser_get** calls [6]. For example, to get an execution with strong, BSP-like consistency, one simply needs to use slack equal 0 on all the items.

An interesting question arises if one allows the slack to be different for different items. Assuming an iterative convergent program where at each iteration the processes communicate items with either slack s_0 or s_1, it stands to reason that the effective slack will be the lowest between the two – $\min(s_0, s_1)$. If

all the different slacks are above 0, then we effectively have SSP consistency. If $\min(s_0, s_1) = 0$, then there is a communication request which will force the producer to be at least on the same iteration as the consumer, as the consumer will have to wait for the current item value; however, this allows the producer to be ahead of the consumer, and the consumer to use the item values from those ”future” iterations. If there is another item where the roles of the two processes is inverted which has $slack = 0$, then both processes will be forced to be on the same iteration, or at least on consecutive iterations with the process which is ahead waiting for the other to submit a new item version. In this case, that specific item is subject to BSP consistency. The other items with higher slack will have an A-BSP like consistency, because the items with $slack = 0$ will act as iteration barriers, but the items with $slack > 0$ do not need to be communicated or waited for in every iteration, making the WPC depend on their slack.

3.2 System components

In our architecture, we do not have distinct server and worker processes, and we do not currently employ any background threads - every worker performs the functions of a PS during the calls to the library, and from our implementation no process takes a differentiated role, except during the initialization of the library. Besides some metadata for all the items, each process only needs to allocate memory for the items they either produce or consume, instead of the full parameter space. The items are transmitted exclusively using one-sided communication, using GASPI's notifications when we need to wait for the completion of a given item transfer.

3.2.1 Metadata and data segments

The implementation features two main data structures to try to organize the items in memory in a compact way. Each of these data structures uses one GASPI segment, and memory locations inside these segments are expressed as offsets from the beginning of the corresponding segment.

We will now look at each segment separately.

3.2.1.A Metadata segment

The metadata segment contains two different items:

- A global atomic counter
- An array of metadata structures

The global atomic counter is used in the **lapser_init** function call, in order to coordinate the various processes in the writing of the array of metadata structures.

The array of metadata structures contains one structure for each item, with the purpose of indicating the item location, and is fully replicated in all workers. More concretely, each metadata structure contains:

- Key: a numeric item id
- Producer: the rank responsible for the production of the item
- Offset: the memory location of the item in the data segment of the producer

In certain cases, there is the need to also store information about the consumers of the items – the processes which want to **lapser_get** a given item. In these situations, the above-mentioned structure is complemented with:

- Consumers: a bit set indicating the processes interested in receiving the item
- Consumer offsets: an array of offsets, indicating the memory location where each process will want the item stored on their local data segment

3.2.1.B Data segment

Each item is stored in a structure inside an array. Each item structure contains:

- Clock: the stored item version number
- Checksum: a numeric value computed by an hash function which takes into account the item value and the clock value
- Value: an array of bytes to hold the item data

3.2.2 Update propagation algorithm

As mentioned in section 2.4, RDMA provides one-sided reads and writes, so naturally there are two ways of transmitting the items: either the consumer reads from the producer, or the producer writes to the consumer. We named this alternatives ”Pull” and ”Push” variants, respectively.

The Pull variant communicates during a **lapser_get** call – it checks if the locally available version has a clock which satisfies the slack requirements; if not, it will issue `gaspi_read_notify` calls directed at the producer's data segment, until the slack requirements are fulfilled. The Push variant makes the producer call `gaspi_write_notify` whenever new item versions are available in **lapser_set**, regardless of any slack values.

This means that the Push version will always transmit the new value when it is available, and the Pull will only initiate communication if the item is deemed too stale. This is analogous to the different prefetching semantics mentioned in section 2.3: Push follows the aggressive prefetching strategy, whereas Pull is more similar to the conservative prefetching. This difference is visible in the results presented chapter 5, both in terms of time per iteration and achieved results.

3.2.2.A Push variant

As said above, the Push variant uses one-sided communication during the **lapser_set** call. algorithm 3.1 shows a pseudo-code version detailing how it is done.

Algorithm 3.1: Producer algorithm in Push variant for the **lapser_set** call

Input: key, $newValue$, $newClock$
begin
  $item\_metadata \longleftarrow metadata\_segment[key]$
  $item \longleftarrow data\_segment[key]$
  $item.value \longleftarrow newValue$
  $item.checksum \longleftarrow calculateChecksum(newValue, newClock)$
  $item.clock \longleftarrow newClock$
  $notificationId \longleftarrow getNotificationIdForKey(key)$
  foreach $consumer$ *in* $item\_metadata.consumers$ **do**
    `gaspi_write_notify(`$item$`,` $consumer$`,` $notificationId$`)`

The consumers only need to verify if the item as stored in their local data segment is fresh enough and if it matches the checksum; if not, a remote write may be in progress, and therefore we wait for the completion (as signaled by a notification) and repeat the checks; this is structured as shown in algorithm 3.2

3.2.2.B Pull variant

The Pull variant uses the opposite approach – the communication is done via remote reads on the **lapser_get** call, instead of on the **lapser_set** one. Algorithm 3.3 demonstrates how it is done.

On the other hand, the **lapser_set** call becomes extremely simple, as it only has to make the updated item available in the data segment, meaning that a simple copy suffices; the corresponding algorithm is shown in algorithm 3.4.

Note how neither the **lapser_set** and **lapser_get** calls in the Pull variant needed explicitly to know the full set of consumers of a given item, whereas the Push variant requires that information in the **lapser_set** call; this is why the consumer information on the metadata segment is optional.

Algorithm 3.2: Consumer algorithm in Push variant for the **lapser_get** call

Input: key, $requestedClock$, $slack$
Result: $value$
begin
  $item\_metadata \longleftarrow metadata\_segment[key]$
  $item \longleftarrow data\_segment[key]$
  $notificationId \longleftarrow getNotificationIdForKey(key)$
  $localClock \longleftarrow item.clock$
  $value \longleftarrow item.value$
  $localChecksum \longleftarrow calculateChecksum(value, localClock)$
  while $requestedClock \leq localClock + slack \ \ \& \ \ localChecksum \neq item.checksum$ **do**
    `gaspi_notify_wait(`$notificationId$`)`
    $localClock \longleftarrow item.clock$
    $value \longleftarrow item.value$
    $localChecksum \longleftarrow calculateChecksum(value, localClock)$

3.2.3 Data structure initialization

The metadata segment initialization begins with every worker allocating an equally sized GASPI segment to hold the metadata and partitioning the space on the newly-allocated segment for each process proportional to the items that they are responsible for producing. To do that, each process increases an atomic counter located at the beginning of the metadata segment of a pre-configured process (by default, the process with rank 0), using a `gaspi_atomic_fetch_add` operation, which gives the number stored at the location prior to the increment. That result is used as the offset to the beginning of the metadata region that is allocated to the worker in all metadata segments, the local and all remote ones.

For example, imagine that we have 2 workers, $W0$ and $W1$, and $W0$ is tasked with producing items with keys 0 and 2, while $W1$ is the producer of items 1 and 3. In this case, both workers need a metadata region capable of storing two metadata structures – therefore they will try to increase the atomic counter by 2. The first process to increment will get the metadata structures starting at 0 up to 2; the last process will get from structure 2 up to 4 (non-inclusive). Note that the first metadata structure is not reserved to the item with key 0 or 1; it can have any key. Therefore, the metadata structure array can arrive at two different orderings: $[0, 2, 1, 3]$ if $W0$ is the first to increment, or $[1, 3, 0, 2]$ otherwise (assuming that each worker writes the metadata structure by its key order).

After locating the beginning of its metadata region, each worker fills the metadata structures for the items they produce; and then they propagate this information to all the other processes using remote writes.

If consumer information is needed, the next step is, for each item that the worker wants to consume, to signal interest on the bit set of the producer of the item (in the metadata segment). The bit set is written using an atomic write operation, to avoid write-write data races between two consumers of the

Algorithm 3.3: Consumer algorithm in Pull variant for the **lapser_get** call

Input: key, $requestedClock$, $slack$
Result: $value$
begin
  $item\_metadata \longleftarrow metadata\_segment[key]$
  $item \longleftarrow data\_segment[key]$
  $notificationId \longleftarrow getNotificationIdForKey(key)$
  $producer \longleftarrow item\_metadata.producer$
  $localClock \longleftarrow item.clock$
  $value \longleftarrow item.value$
  $localChecksum \longleftarrow calculateChecksum(value, localClock)$
  while $requestedClock \leq localClock + slack \ \& \ localChecksum \neq item.checksum$ **do**
    `gaspi_read_notify(`$item$`,` $producer$`,` $notificationId$`)`
    `gaspi_notify_wait(`$notificationId$`)`
    $localClock \longleftarrow item.clock$
    $value \longleftarrow item.value$
    $localChecksum \longleftarrow calculateChecksum(value, localClock)$

Algorithm 3.4: Consumer algorithm in Pull variant for the **lapser_set** call

Input: key, $newValue$, $newClock$
begin
  $item \longleftarrow data\_segment[key]$
  $item.value \longleftarrow newValue$
  $item.checksum \longleftarrow calculateChecksum(newValue, newClock)$
  $item.clock \longleftarrow newClock$

same item which might try to concurrently set their bits on the same bit set. They also indicate what is the predicted offset in their data segment that will be reserved for the item in question, and transmit it to the producer.

Then the processes proceed to allocating the data segment. Each worker allocates only the necessary space to to store the item structures for the items they produce and consume. After a successful allocation, they initialize these structures with default values. Note that the item structure order has to be consistent with the offsets declared in the metadata segment.

3.3 Technical challenges

During the implementation of this system, some technical challenges were met, and solutions were devised to surpass them. The two most significant challenges are presented below, along with the strategy followed to deal with them.

3.3.1 Improving data race handling

In asynchronous distributed memory systems, data races are ever-present. During implementation, some papers on analogous systems where consulted to check for approaches about how to deal with them, and we found Pilaf [22], a key-value store accelerated with RDMA. Their solution to data races is to avoid the existence of write-write races, and to have mechanisms to check the consistency of reads in order to detect and mitigate read-write data races. As our system avoids write-write data races, their solution for read-write data races seemed sensible and therefore we implemented consistency checks via the usage of a checksum function with the result stored within the item structure, as mentioned before in section 3.2.1.

There are some critical details about the checksum and its storage in memory. One of them is what information is needed to be protected by the checksum – initially, I was not including the clock in the calculations. This meant that a partial overwrite of the clock field in the structure would an old version of an item pass as a new one, undetected by the checksum, as illustrated in fig. 3.1.

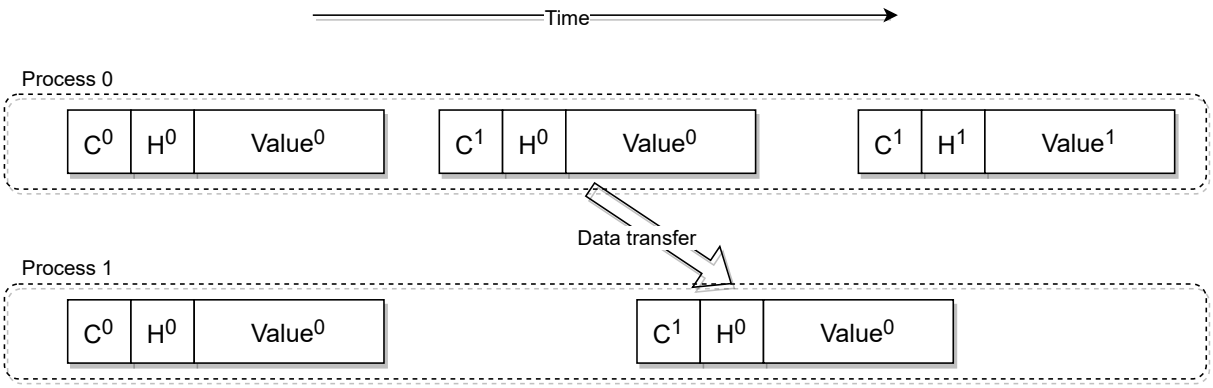


Figure 3.1: A depiction of a data race between two processes. Process 0 produces an item, where C is the clock, H the checksum, and $Value$ represents the item data; the superscripts represent the clock. Process 1 is a consumer which per chance read the item after the clock was updated but before the storage of the rest of the information; as the checksum only validated the data, the result is that the structure composed of C^1, H^0 and $Value^0$ is (incorrectly) considered valid.

The immediate solution is to include the clock version in the checksum. Another part of the solution is to store the checksum after the new item value is already fully copied, and after that then copy the new clock into the item storage. By first writing the item value before the other data, it is assured that any concurrent will be able to detect the data race, because the checksum will not be valid.

3.3.2 Limited notification id space

As previously said in section 2.4.1, coordination with GASPI uses notifications, with an numeric id and an associated value; one can specify a range of ids to check or wait for. The GASPI implementations used during the development, testing, and evaluation of the library impose a limitation on the notification

identifiers of at most 16 bits, meaning that they could range from 0 to 65535 ($2^{16} - 1$). Also, it is documented that notification 0 has a special behaviour[1], which further reduces the number of usable ids.

Ideally, to implement communication as described in section 3.2.2, we would be able to assign one notification id to each item, according to their key, to allow each process to specifically wait for the items they have requested. However, due to the small number of notification ids, it is needed to reuse notification ids, and write code that tolerates the possibility of never receiving a notification because it was already consumed by another, concurrent operation. Also, it is necessary to avoid having items being mapped to notification id 0, so that the caller doesn't spuriously think the data transfer is done when it should wait more time for the completion of the underlying operation.

In the current implementation, the notification id corresponding to a given key is the following: take the 15 least significant bits, shift one place to the left, and set the least significant bit to 1. The reason to use the least significant bits is that they have more variation than the most significant bits; setting the least significant bit to one completely avoids using the notification id 0, at the expense of reducing the total id space to half.

[1] Waiting for notification id 0 returns immediately indicating success, even if no such notification was sent [21]

4

Case study: Bayesian Probablistic Matrix Factorization over Lapser

Contents

When adapting BPMF to use Lapser, we started off from the implementation of the GASPI variant, because it followed a point-to-point approach when transmitting the items of U and V. This chapter details the changes needed in the BPMF application in order to use our library.

4.1 Overview of the original application

The application has three main data structures, which need to be shared in a distributed implementation: a) the known ratings matrix R; b) the current approximation for the U and V matrices; and c) other data derived from each of these matrices.

This last data consists of summary statistics about the latent features stored in U and V, such as the covariance of each feature and the sum of all the rows; this information serves as parameters of the hyper-parameter distribution. From now on they will be referred to as hyperparameters.

Since the R matrix is immutable for the purposes of Matrix Factorization (MF), we only need to be concerned with how to communicate the last two data structures, which depends on how is the workload is distributed among processes. In this specific implementation, each of the matrices U and V are sharded among processes, and each process is responsible for computing the new values of their shard; therefore, each row of U and V has only a single producer. Note that while computing their designated U or V rows, they also compute their part of the hyperparameters.

After computing their share of the new values and hyperparameters, this data is shared among processes in distinct ways. The hyperparameters need to be aggregated, and to that end an allreduce collective is used (with the sum reduction function). The U and V matrices are stored in separate GASPI segments and replicated across processes using remote writes. The application takes care of only issuing the writes that are needed – if a process does not need to access a given user or movie, the corresponding row will not be sent by its producer. To know which items a given process needs, a map data structure is created, mapping the row number to a list of the processes which do need to access it. After all computation is finished, notifications are issued to ensure that the matrix data has arrived to all processes, and so the application is ready to continue to the next step.

4.2 Updating *U* and *V*

4.2.1 Data layout in Lapser

The first step in using Lapser on any program is to map the application entities that need to be communicated into items to be stored, and find a convenient key scheme to identify the items. The mapping we used was directly inspired in LazyTable [6]: each row of U and V is an item, and the row number

is the key. Due to implementation limitations, only one instance of Lapser can be active in any process at a given time, therefore there was a need to disambiguate between the U and V row numbers – the solution found was to use an ”adjustment” variable, so that the U items could occupy the key range $[0, adjustment[$ and the V items have key ids starting from the ”adjustment” value (and naturally the adjustment value is the total number of rows in U).

4.2.2 Producer and consumer mappings

As said before, the original BPMF application stored a map detailing, for each row of U and V, which workers needed to have access to the updated values of that row. This is different from what the library requested in the **lapser_init** call (as presented in section 3.1.2), which asks to each process which items they produce, and which items they consume. First, it is necessary what should be considered an item – and the choice was each row of U and column of V, which correspond to the predicted latent feature values of each user u and each movie v, respectively. Having clarified that, then we need to answer the question ”which items am I responsible for producing?”, which was easy because the application assigned a continuous section of U and V to each process, and that information was easily accessed.

The more difficult question is ”which items do I want to consume?”. To compute new values for a user u, we need the predicted latent features of the movies that they have rated; and for each movie v, we need the predicted latent features of all the users that have rated it. This amounts to consulting the R matrix and check which positions are non-zero. The final code to compute which items are needed by each item is shown in listing 4.1.

Lines 5 and 6 show how the items to produce are determined. The function `std::iota` fills the range between the first two arguments with monotonically increasing numbers, starting with the number passed as the third argument. `Sys::from` is a function that indicates what is the first row that this worker must compute, while `Sys::num(procid)` on line 5 returns the number of rows to compute. Note the use of the adjustment variable – as the `to_produce` list will have to hold keys, we have to transform rows by using this variable, as explained above.

Lines 10 to 24 give an example of how to determine the keys to consume. For each row we have to produce (line 12) we traverse the corresponding column in the ratings matrix, in the code denoted as `M` (line 13), and for each non-zero position (which is what the `InnerIterator` iterates on line 13) we insert the row into a set (line 17), unless the item is on our range to produce (line 16). Lines 19 to 23 repeat the same logic, but for the test set (named `Pavg` on line 19). Finally, line 25 stores the keys in the `to_consume` list. The use of a set was necessary to deduplicate items, which is common in this application. For instance, if two different users rate the same movie, and if we were processing the user matrix, that movie will be inserted twice into the set, but only stored once.

```
1  void SYS::build_conn(SYS& to)
2  {
3      [...]
4
5      to_produce.resize(Sys::num(procid));
6      std::iota(to_produce.begin(), to_produce.end(), Sys::from() + adjustment);
7
8      [...]
9
10     std::unordered_set<Key> temp;
11     temp.reserve(to_produce.size()); // to give a memory size estimate
12     for(int idx=Sys::from(); idx<Sys::to(); ++idx) {
13         for (SparseMatrixD::InnerIterator it(M, idx); it; ++it)
14         {
15             // do not include self produced items
16             if(to.from() <= it.row() && it.row() < to.to()) { continue; }
17             temp.insert(it.row() + other_offset);
18         }
19         for (SparseMatrixD::InnerIterator it(Pavg, idx); it; ++it)
20         {
21             if(to.from() <= it.row() && it.row() < to.to()) { continue; }
22             temp.insert(it.row() + other_offset);
23         }
24     }
25     to_consume.assign(temp.begin(), temp.end());
26
27     [...]
28
29     lapser_init(total_items, sizeof(double)*num_latent,
30                 global_to_produce.data(), global_to_produce.size(),
31                 global_to_consume.data(), global_to_consume.size(),
32                 Sys::update_freq);
33 }
```

Listing 4.1: Code excerpt to determine the items to communicate

However, there is a subtlety here – imagine we are processing the U matrix; in this case, the `to_produce` list will hold items in the U matrix, but the `to_consume` list will have items corresponding to the M matrix. That is why, instead of using `adjustment` when inserting into the set, we use `other_offset`; also, the check on line 16 uses not the limits of the U matrix (that would be `Sys::from()` and `Sys::to()`, which are used on line 12), we use the limits of the other matrix we have to compute, which is the sole argument to the function (and therefore we use `to.from()` and `to.to()`).

Finally, lines 29 to 32 initialize the Lapser library. Note how the arguments for indicating the items to consume and produce are not the previously used variables, but instead they have a `global_` prefix. This is because there is the need to join multiple lists of entities to produce and to consume; for instance, the users and movies to produce, which are computed separately (each on their `build_conn` call).

4.2.3 Storing and updating *U* and *V*

As previously mentioned, the application used GASPI segments to both communicate and serve as the primary storage of the U and V matrices; and the rest of the application takes advantage of that fact, and assumes that the items matrices are contiguously stored in an array – which is not the case when using Lapser. We opted by allocating these arrays, and when communicating either read from our write to these arrays, keeping the necessary changes limited to the communication functions.

In concrete terms, there was a function responsible for the transmission of each row, called as soon

as it was computed, named `send_item`; the original version is shown in listing 4.2, and the modified version in listing 4.3

```
void GASPI_Sys::send_item(int i)
{
    BPMF_COUNTER("send_item");
    for (int k = 0; k < Sys::nprocs; k++)
    {
        if (!do_send(k)) continue;
        if (!conn(i, k)) continue;
        auto offset = i * num_latent * sizeof(double);
        auto size = num_latent * sizeof(double);
        SUCCESS_OR_RETRY(
            gaspi_write(items_seg, offset, k,
                        items_seg, offset, size, 0, GASPI_BLOCK));
    }
}
```

Listing 4.2: Original function to communicate items from U and V

In the original version, we can see there are some checks to confirm if process `k` needs the item, which are completely hidden away in Lapser.

```
void SYS::send_item(int i)
{
    BPMF_COUNTER("send_item");
    auto size = num_latent * sizeof(double);
    auto offset = i * size;

    lapser_set(i + adjustment, ((Byte*) items_ptr) + offset, size, iter+1);
}
```

Listing 4.3: Function to communicate items from U and V using Lapser

In this code excerpt, we observe again the usage of the `adjustment` variable used to convert from a row number `i` to the corresponding key in Lapser. Also, in the second argument we can see a reference to the array holding the matrix – stored in the variable `items_ptr` – and the calculation of offset for the memory location of the `i`-th row.

As for ensuring that the matrices were already fully received, the original version had each process send notifications to every other process, and later wait for notifications arriving from every other process, as shown in listing 4.4.

```
    {
        BPMF_COUNTER("notify");

        for (int k = 0; k < Sys::nprocs; k++)
        {
            if (!do_send(k)) continue;
            SUCCESS_OR_RETRY(
                gaspi_notify(items_seg, k, Sys::procid, iter+1, 0,
                             GASPI_BLOCK));
        }

    }
    [...]
    {
        BPMF_COUNTER("sync");
        auto start = tick();

        for (int k = 0; k < Sys::nprocs; k++)
        {
            if (!do_recv(k)) continue;
            gaspi_notification_id_t id;
            gaspi_notification_t val = 0;
            SUCCESS_OR_DIE(
                gaspi_notify_waitsome(items_seg, 0, Sys::nprocs, &id,
                                      GASPI_BLOCK));
            SUCCESS_OR_DIE(gaspi_notify_reset(items_seg, id, &val));
            auto stop = tick();
            sync_time[id] += stop - start;
        }
    }
```

Listing 4.4: Code excerpt to notify the arrival of U and V data from the original version

Instead of waiting for notifications, we need to copy data from the Lapser library into the items array. Using the previously determined `to_consume` list, we copy all the relevant rows to the correct offset. Listing 4.5 shows how it is done.

```
    {
        BPMF_COUNTER("get_item");
        #pragma omp parallel for
        for (size_t i=0; i<to_consume.size(); ++i) {
            Key k = to_consume[i];
            int row = k - adjustment;
            size_t item_offset = row * num_latent;

            auto res = lapser_get(k, &items_ptr[item_offset],
                                  sizeof(double)*num_latent, iter+1,
                                  Sys::update_freq);
            assert(0 == res);
        }
    }
```

Listing 4.5: Code excerpt to read the items from U and V when using Lapser

Note that the `to_consume` list contains key values, not the row numbers, and therefore instead of adding the `adjustment` variable, we need to subtract it.

4.3 Updating Sum, Covariance and Norm

The previous section deals with transmitting the matrices U and V, but that is not everything that needs to be communicated among processes. As mentioned before, there are some properties – that we called hyperparameters – that need a different treatment. More concretely, they are partially computed for each u, and then they need to be summed over the complete U (and likewise for V). As these

computations are divided among the various processes, there is the need to sum the partial results, and make that aggregation available to everyone – which is precisely a scenario where an allreduce collective is appropriate; and in fact, all previous communication implementations used an `MPI_Allreduce` for this communication step.

We could have used the same strategy for our adaptation, but then it would be impossible to experiment with different slack values for these items; also, an allreduce call constitutes a synchronization barrier, rendering SSP unreachable.

Therefore, we used the strategy already outlined at section 3.1.3.A: a) every process stores their contribution in a different item; b) then, when needed, they read all the items that contribute to the intended value; c) finally, aggregate the read values with an appropriate aggregation function. This simple strategy revealed feasible for this specific application; note that the total volume of data is relatively low – assuming we have K latent features and P processes, the total size of this data is $P \times (K^2 + K + 1) \times sizeof(double)$; for the examples shown in table 5.2, that corresponds to $16 \times (128^2 + 128 + 1) \times 8\ bytes = 180,608\ bytes$. This procedure is shown for one of the hyperparameters in listing 4.6.

```
{
    BPMF_COUNTER("set_sumcovnorm");
    lapser_set(sums_to_produce[0], sum.data(), item_size, iter+1);

    [...]
}
[...]
{
    BPMF_COUNTER("get_sumcovnorm");
    VectorNd temp_sum;
    for(int i=0; i<sums_to_consume.size(); ++i) {
        lapser_get(sums_to_consume[i], temp_sum.data(), item_size, iter+1, 0);
        sum += temp_sum;
    }

    [...]
}
```

Listing 4.6: Code excerpt to store the contribution of sum and then aggregate the contributions from all the workers

Note how the keys pertaining to this hyperparameter are stored in the variables `sums_to_produce` and `sums_to_consume`. All hyperparameters follow this pattern – since their keys cannot collide with the ones used for U and V, it is necessary to be careful with reserving enough ids for each hyperparameter, which is done in one of the omitted parts of listing 4.1; later, all the "`to_consume`" and "`to_produce`" lists are joined in a `global_to_produce` and `global_to_consume`, as mentioned in section 4.2.2.

4.4 Consistency model of BPMF with Lapser

As mentioned in section 2.3, the usage of relaxed consistent models can be beneficial for the overall speed of the algorithm, but it can also degrade the final result, and each iteration may become less effective – this depends on the algorithms characteristics. In some cases, it is possible that the algorithm

tolerates no slack, due to some properties that must be maintained.

We believe this is the case of the BPMF with regard to the hyperparameters. After having substituted the allreduce collective with our library, we experimented setting multiple slack values. Unfortunately, the application did not support slack values above 0 for these items – when we tried, the application raised a fatal error due to some mathematical property checks. When comparing the hyperparameter values with or without using slack, we observed a strong divergence in their numeric values.

Fortunately, as it is possible to use different slack values for different items in Lapser, we could continue using higher slacks for the items corresponding to the items in U and V, and 0 to the items that require it. This is already done in listing 4.6 – the last argument in the `gssp_get` function corresponds to slack, and instead of using a variable (like we did in listing 4.5), we used the constant `0`.

This, however, means that we can never take advantage of all the potential benefits of SSP; this is because every process needs to consume an item from every other process using no slack (a scenario analyzed in section 3.1.3.C); however, we can still take advantage of the benefits of A-BSP for the U and V matrices.

5

Evaluation

Contents

In this chapter, we conduct an experimental evaluation of our new library, and its effect on the performance of the example application we presented in the previous chapter. The main measure of performance when training ML algorithms is the time taken during training to reach a given error level. Therefore, with the evaluation of the work, we aim to answer the following questions:

- How does the performance of BPMF using our solution compares with the existing BPMF versions already using hand-optimized 1-sided communication methods?
- How does our library respond to the variation of the configurations parameters – namely slack – in terms of time and quality of the solutions?
- How does the communication strategy (Push versus Pull) affect the performance of our primitives?

5.1 Experimental setup

To answer these questions, we need to rely on an experimental infrastructure with a specific kind of setup. First and foremost, we need RDMA capable hardware to fully evaluate the capabilities of asynchronous, zero-copy communications as provided by GASPI. Also, since we want to test the scalability of the resulting system, while being the closest possible to the target computing environment, we aimed to test the solution in an HPC cluster. Therefore, the results presented in this chapter are taken from a cluster at Imec[1], more specifically from a computing pool with a total of 30 nodes, each node having an Intel Xeon E5 dual-socket CPU with 36 cores supporting 72 hyperthreads and able to access a total of 256 gigabytes of RAM, connected via Mellanox InfiniBand cards capable of 56 Gb/s with RDMA capabilites. These machines run the Ubuntu distribution version 20.04.3, using the Linux kernel version 5.4.0.

In terms of software, we used the GPI-2 library[2], an open-source GASPI implementation as a communication library. Also, the existing communication options in the BPMF application rely on the MPI library for communication; we used the Intel 2021 MPI implementation [3] in our experiments.

As an input to the BPMF application, we used two sets of movie recommendation data from the MovieLens datasets [23], more concretely the ml-100k and ml-20m versions, henceforth named "Movielens-tiny" and "Movielens", respectively. The MovieLens datasets are frequently used when testing Matrix Factorization applications [24, 25].

To evaluate the accuracy of the algorithm, 20% of each dataset is reserved as a test dataset, with the remaining 80% being used for training purposes. Table 5.1 summarizes the characteristics of these datasets.

[1] https://www.imec-int.com/en
[2] http://www.gpi-site.com/
[3] https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html

Characteristic	Dataset	
	Movielens-tiny	Movielens
Number of ratings	100000	20000263
Number of users	943	138493
Number of movies	1682	26744
Zeros/Non-zeros	14.9	184.2

Table 5.1: Dataset characteristics. Number of ratings corresponds to the number of non-zero items in the R matrix, number of users and movies to the number of rows in U and columns in V respectively.

5.2 Experimental results

The next sections will be devoted to analyzing the performance characteristics of Lapser when varying multiple parameters, as well as discuss the results we obtained.

5.2.1 Comparison with existing parameter sharing methods

In our first experiment we give a broad comparison across all the implementations that we tested, by measuring how they perform in terms of the time they take to execute 500 iterations, when using 16 machines (with the specifications detailed in section 5.1), with one worker per machine. Table 5.2 summarizes the results.

Version	Dataset				Notes
	Movielens-tiny		Movielens		
	RMSE	Time	RMSE	Time	
GPI	0.924747	17.4 s	0.770619	274.2 s	Uses GPI-2, issuing `gaspi_write` as soon as the results are ready
MPI Put	0.924747	16.8 s	0.770619	415.3 s	Uses MPI Put calls, which uses RDMA communication, similar to GPI version
MPI ISend	0.924747	33.1 s	0.770619	291.4 s	Uses MPI asynchronous communication, with a background thread to coordinate the communication
MPI Bcast	0.924747	29.6 s	0.770619	766.5 s	Uses MPI_Bcast calls
MPI Reduce	0.758193	169.3 s	0.758902	13857.9 s	Uses MPI_Reduce calls
MPI Allreduce	0.758193	171.0 s	N/A	N/A	Allreduce using MPI
Lapser Pull 0	0.924747	32.4 s	0.770619	409.6 s	
Lapser Pull 4	0.923466	30.2 s	0.773364	296.6 s	
Lapser Push 0	0.924747	20.3 s	0.770619	285.7 s	
Lapser Push 4	0.924074	22.0 s	0.770651	284.1 s	

Table 5.2: Time to complete 500 iterations in BPMF with various communication methods and error of the final solution. There are no results for the allreduce version with the largest dataset, due to an argument error (signed integer overflow due to the sheer size of the state being communicated).

The first observation is that, among the original communication versions, there is a large performance gap, over 10 times between the fastest and slowest versions. This is caused by the characteristics of

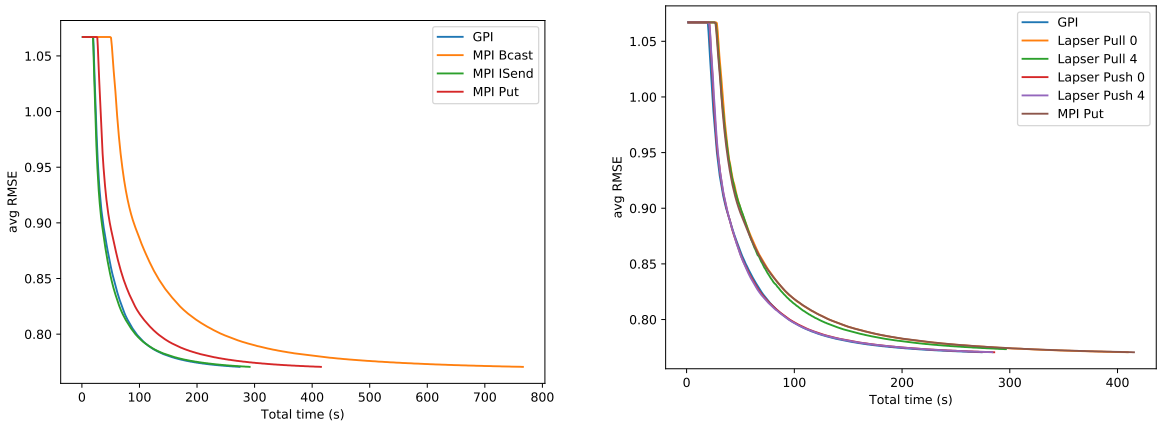
these implementations – some use two-sided synchronous collective communication, while others use one-sided asynchronous and point-to-point communication – which naturally translates into different performance characteristics.

The achieved Root Mean Squared Error (RMSE) among original versions is different, with the reduce-based versions achieving solutions with lower error. These versions are actually running a modified BPMF algorithm (and also don't communicate the U and V matrices directly), which may explain the differing results.

The error of the Lapser based versions when not using slack is equal to the many of the other alternatives, which is expected – with no slack the library follows a fully consistent model, therefore performing the same computations. This is not true when we use slack, as we may get items that are not exactly of the demanded age, and that is why the versions with slack present different final RMSE. However, it is surprising that the error is lower when using slack on the smaller dataset; in the larger example, the inverse is true, which aligns with the expectation that slack worsens the iteration quality due to the potential usage of stale values.

Finally, 3 of the 4 Lapser variants shown in the table have a total time which is, at most, 8% higher than the best solution, and Lapser Push is only worse than the very best version from the application authors. This points to the conclusion that our library is in fact competitive against tailor-made, hand-tuned communication mechanisms written by the application authors.

To validate this claim and better understand the evolution of the error values, we show in fig. 5.1 the evolution of the quality of the solution found over time when running 500 iterations, showing BPMF using our library for communication and also a subset of the original versions (the 4 best performing as shown in table 5.2).



(a) Former communication options

(b) Our communication options, with GPI and MPI Put for comparison

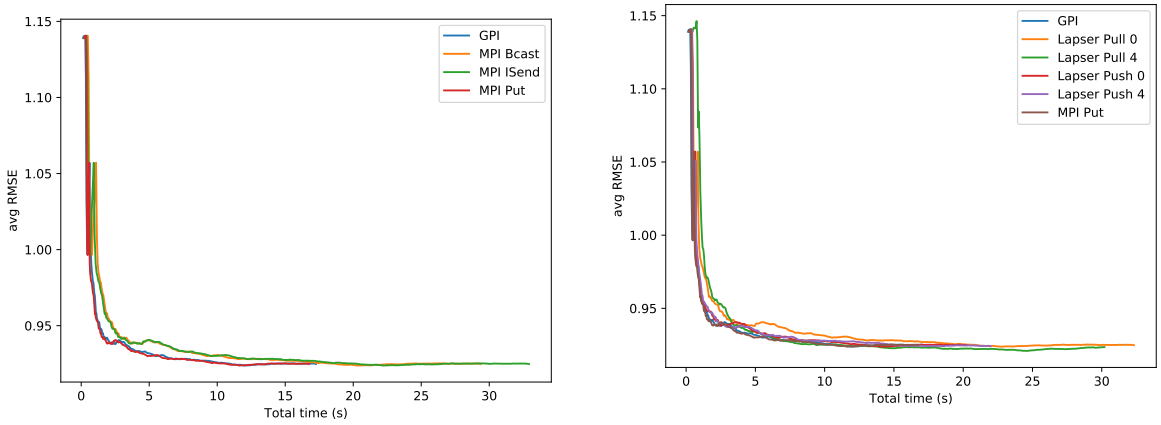
Figure 5.1: RMSE along time of various solutions processing the larger Movielens dataset

Focusing first on fig. 5.1a, we can see that the MPI Bcast version (in orange) is noticeably slower than the rest, by a factor of (almost) 2. This may be due to the usage of collective 2-sided communication which broadcasts all of U and V to all processes, even those that do not need that information.

Of the remaining three, we can see that the MPI Put version (in red) takes 33% more time to finish the 500 iterations, although the difference in RMSE at any given time is low, when compared to the best performing solutions. It is surprising that this communication method, which uses 1-sided communication backed by RDMA, has a noticeable difference from the GPI version (which follows the same communication strategy); although we have not yet found a definitive explanation for this, we speculate that this might be due to this communication method being relatively new in MPI, as this communication interface traditionally focused on 2-sided communication methods and synchronous collectives.

Another surprising result is that MPI ISend, matches the performance of the GPI version. MPI ISend uses asynchronous two-sided communication – and the good performance of this solution may be due the optimized implementation they use. In particular, this version uses a background thread which reads the requests from the computing threads and also deals with receiving items from the other workers; in comparison, the GPI version just issues a `gaspi_write` on the computing thread.

Turning our attention to fig. 5.1b, we can see that the Lapser versions have distinct performance profiles: while the Lapser Push is close to the GPI version (its lines overlap with the GPI one), Lapser Pull is similar to MPI Put. Another observation is that the Lapser Pull with slack equal to 4 has a slightly better performance than with no slack, as its line (in green) can be seen below the line for no slack (in orange); the difference of Lapser Push when varying slack is not visible. We study in further depth the effect of varying slack in section 5.2.3.



(a) Former communication options

(b) Our communication options, with GPI and MPI Put for comparison

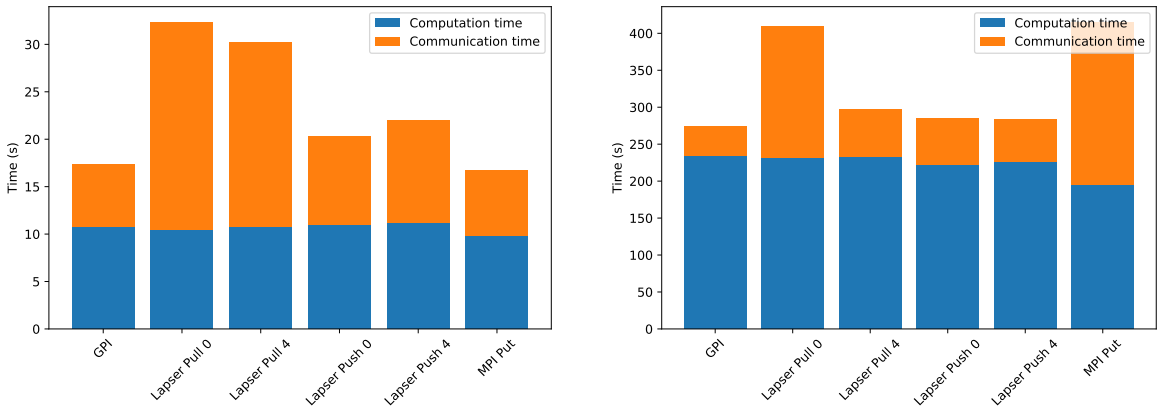
Figure 5.2: RMSE along time of various solutions processing the smaller Movielens-tiny dataset

Looking how these BPMF versions performed on the smaller dataset on fig. 5.2, we see some differ-

ences. Starting with fig. 5.2a, we can see that the various solutions are significantly closer now in terms of reaching a lower error, with every implementation reaching a RMSE lower than 0.95 in approximately five seconds. However, the time to complete all the iterations varies by a factor of 2, similar to before. Surprisingly, the MPI ISend is the version taking the longest time to complete in this dataset; this can be explained by the higher complexity, which represents an upfront cost that may be amortized when there is enough communication happening, but this does not happen to be the case for this dataset.

Figure 5.2b shows a similar picure, with all implementations being fairly close in performance; however, the Lapser Pull versions (the green and orange lines) still have worse performance than the other version, taking more time to finish and having slightly higher RMSE at most points in time (except at the end).

Another metric of interest is the ratio of communication to computation. By using the profiling facilities in BPMF, we can see that in the larger example the majority of time seems to be spent on computation, not communication, while in the smaller example the time taken is more balanced, as shown in fig. 5.3.



(a) Movielens-tiny

(b) Movielens

Figure 5.3: Communication time vs computation time

Between datasets, we can see that the fraction of time taken by communication is larger in the smaller dataset than in the larger one, which is consistent with the fact that the larger dataset has a much larger size.

Another interesting fact is that there is some variation in the amount of computation time, particularly when comparing MPI Put against the other versions; we believe this may be an artifact from the measurement. Nevertheless, it does not affect the rest of our analysis.

We can also observe that the gains from slack seem to be more pronounced in the larger dataset for Lapser Pull, showing a reduction of $1/4$ in communication time when augmenting the slack from 0 to 4. This is an expected behaviour, as Lapser Pull will only communicate when the items are too stale, and with slack at four it happens roughly every 4 iterations.

5.2.2 Performance difference of Push and Pull

As is clear from the tables and graphs above, there clearly is a performance difference between Push and Pull, with the former having better performance out of the box.

The reason for this fact can be explained by analyzing the description provided in section 3.2.2. Push starts the data transfer as soon as possible allowing the RDMA infrastructure to handle the communication while the workers continue with the computations, overlapping computation and communication. The Pull version is unable to do so, because it only communicates when the workers try to read – and by that point no computation is being performed, leading to worse total time per iteration. The use of a prefetch primitive could help in boosting the overlap in the Pull variant; however, in the BPMF application it is not possible to use such strategy, because there is nothing being done between the production and the consumption of the items.

Another factor which affects the performance of the Pull version is that all the items are retrieved at the same time – which means there is a sudden increase in usage of the RDMA infrastructure, leading to contention on this shared resource.

5.2.3 Impact of staleness on the performance of Lapser

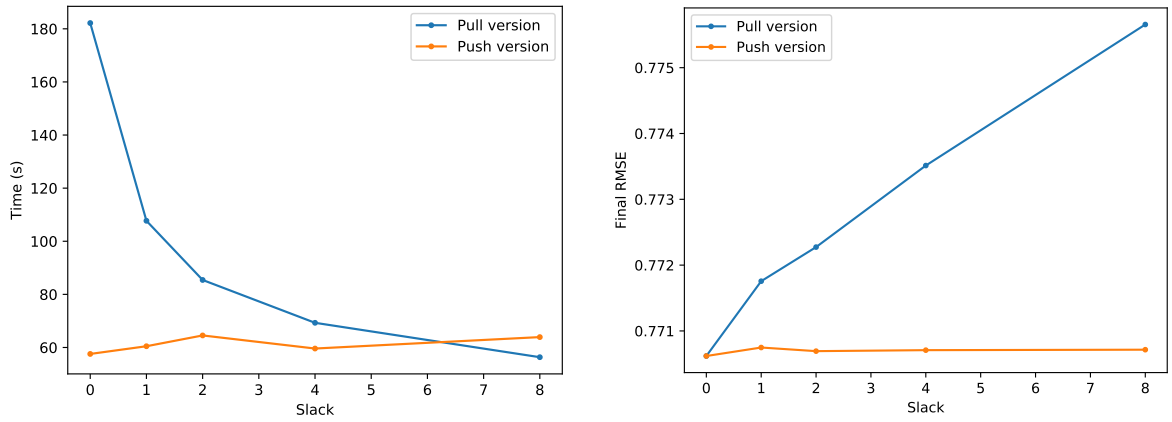
From the numbers referred in the previous section, we can observe that the slack parameter seems to help the Pull variant to improve significantly in the larger dataset (a speedup of 1.4), whereas the gains for the Push version are not significant; on the smaller dataset the gains where greatly reduced or even negative (for the Push version).

Next, we explored further how slack variations affect performance, by measuring a new execution run with the same configuration previously used but with more variety of slack values (and only using the larger Movielens dataset).

Looking at the final time and achieved error, we can clearly see in fig. 5.4a that only the Pull variant is benefiting from slack, and the other version is mostly unaffected.

The fact that the Push implementation is not benefiting can be explained by remembering that it never omits sending any item regardless of the slack; slack can only have a positive effect on time if there are stragglers that force the faster processes to wait for them, instead of allowing them to proceed to the next iterations. In this application, there is a barrier which can not be relaxed with slack – in the form of the communication of the communication of the hyperparameters (vide section 4.4) – rendering the primary benefit of the Push version moot. The Pull version is also affected by this, but because it avoids communication if the items are fresh enough, it presents performance gains from the reduced communication.

Also, fig. 5.4b shows that the final RMSE of the solution is barely affected by the slack increases



(a) Cumulative time devoted to communication

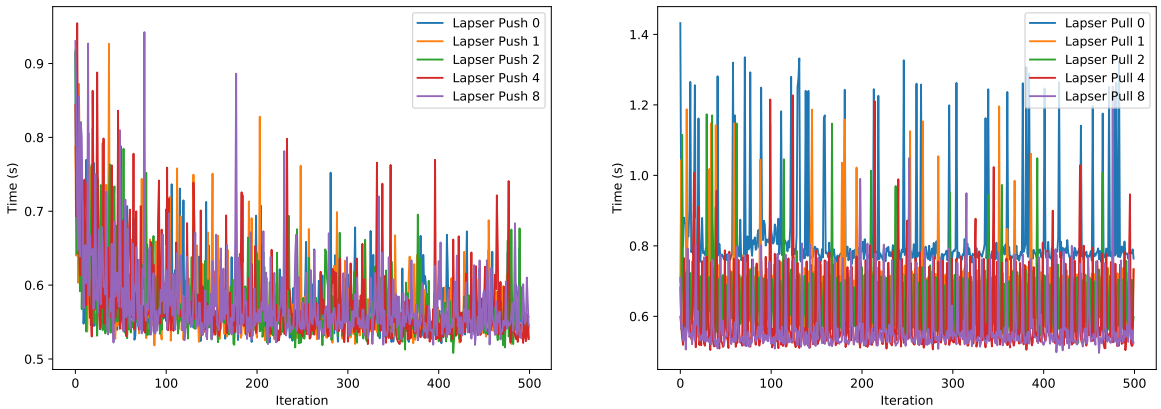
(b) Final RMSE

Figure 5.4: Evolution of speed and solution quality with varying slack on the largest dataset

– despite the apparently strong positive correlation between slack and final RMSE, note that these variations are in the order of the third decimal place of the final measured error. This points to the conclusion that the underlying BPMF algorithm seems to have strong convergence even in the face of (limited) inconsistency in the U and V matrices.

Slack effects on each iteration

Figure 5.5 shows the time taken by each of the 500 iterations when varying the slack on each of the variants.



(a) Push variants

(b) Pull variants

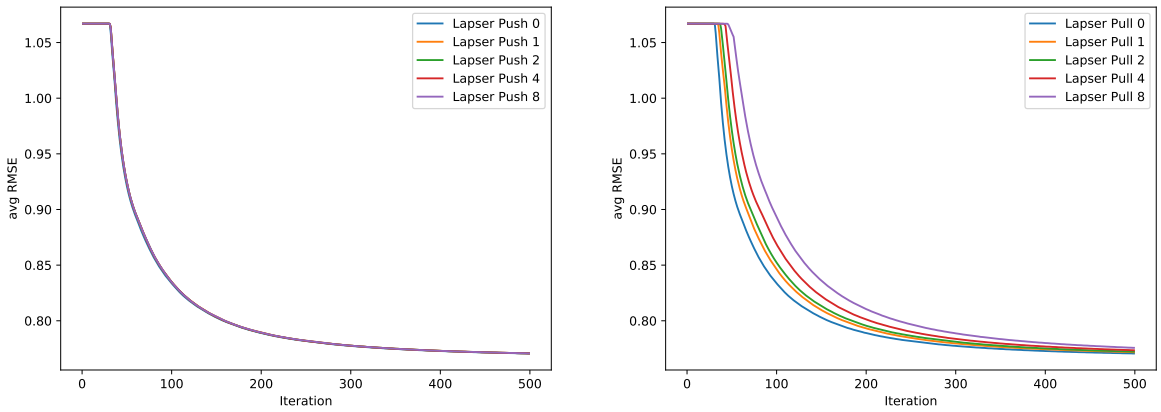
Figure 5.5: Time taken by each iteration, when varying slack, while processing the larger dataset. The numbers in the legend identify the slack used.

There is no discernible pattern in fig. 5.5a, except for some outliers, which appear as spikes in time. While we do not have a definitive explanation for these, they may be due to loss of notifications due

to notification collision – which makes a waiting thread wait until a predefined timeout, instead of being notified that the data transfer has completed.

Turning our attention to fig. 5.5b, we can see a pattern emerging: when we use slack, the plot seems to create a solid bar, and, when we don't, it seems to always have a time per iteration at the top of the band. This can be explained by noting that, when we use slack, each iteration either communicates the items in U and V or not; and when they communicate, the time taken in that iteration increases, and when not the time is lower; graphically this appears as the thick bar we can see, where the top is formed by the iterations where the workers communicate and the bottom corresponds to no communication. When there is no slack, all iterations feature communication, and that forms a blue ceiling which is visible in fig. 5.5b.

Next, we analyze the impact of slack on the iteration quality; therefore, we plotted the evolution of the solution quality per iteration, obtaining fig. 5.6.



(a) Push variants

(b) Pull variants

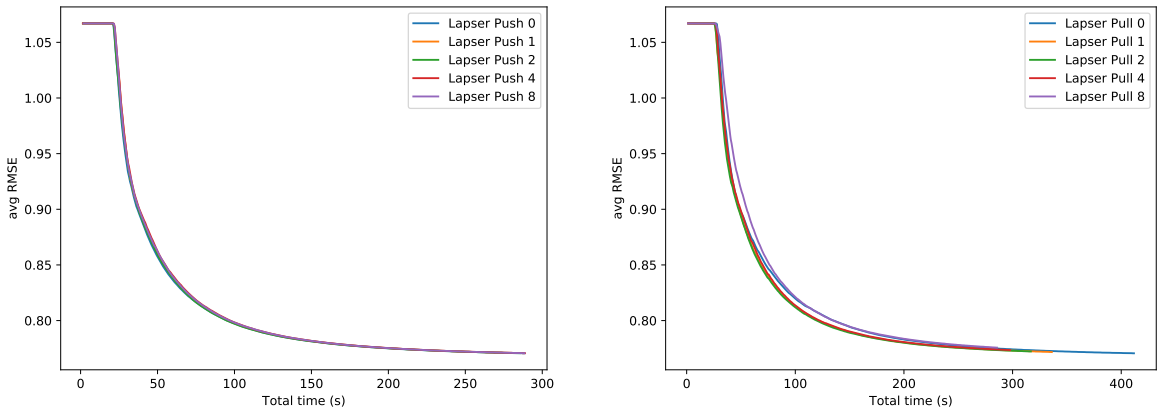
Figure 5.6: RMSE achieved by each iteration, when varying slack, when processing the larger dataset. The numbers in the legend identify the slack used.

On the one hand, looking at the Push variant, we see that slack has virtually no effect on solution quality, which suggests that it is actually not being used. This is consistent with the previous figures which depict no significant variation in any metric when using this Lapser implementation.

On the other hand, the Pull variant shows a noticeable variation on RMSE, and larger amounts of slack have larger error, which is the expected behaviour (as described on section 2.3). However, these differences are reduced when reaching the end of the convergence, around iterations 400 to 500, as opposed to iteration 0 to 200, where significant progress is attained.

So far, when increasing slack in the Pull variant, we have higher RMSE, but less communication happening in more iterations, yielding less total time; which means there is tradeoff, and therefore using some intermediate slack value may be the best choice. This is corroborated when looking at fig. 5.7b,

where we can see that Lapser Pull seems to have two different groups: one formed by the use of slacks 2 and 4, and the rest; these groups are formed using as criterion the proximity of the lines in the plot.



(a) Push variants

(b) Pull variants

Figure 5.7: RMSE achieved per time, when varying slack. The numbers in the legend identify the slack used.

Another interesting observation is that initially using no slack was yielding similar performance to using slack 2 and 4, but around iteration 100 the blue line (which represents Lapser Pull 0) leaves this ”group” and joins the other composed of slacks 1 and 8, marking a performance degradation.

Finally, as Lapser Push showed no variations in either time or error when varying slack, fig. 5.7a naturally shows no difference in performance (when measured as the time to reach a given RMSE) when varying this option.

5.2.4 System scalability

For any distributed system, there is the question of how does it scale with the number of nodes. Unfortunately, we did not had access to a large HPC super-computer cluster to run our experiments with a larger number of nodes. Figure 5.8 shows the time taken to complete 500 iterations when factorizing the Movielens matrix of the most competitive communication solutions, when varying the number of computing nodes and workers.

Every implementation presented in the figure presents parallel lines, suggesting equal scalability properties, with two exceptions. The first exception is Lapser Pull 0, which shows a slower decrease in time than the other implementations. This can be an artifact of the Pull algorithm – with slack at 0, all the consumers of a given item will try to read it from the producer; with a larger number of nodes, that means that the U and V matrices are more divided among workers, but if a given movie is frequently rated by users (to give an example), it is probable that the corresponding row in V will be necessary to a majority of workers, causing contention on the producer interface.

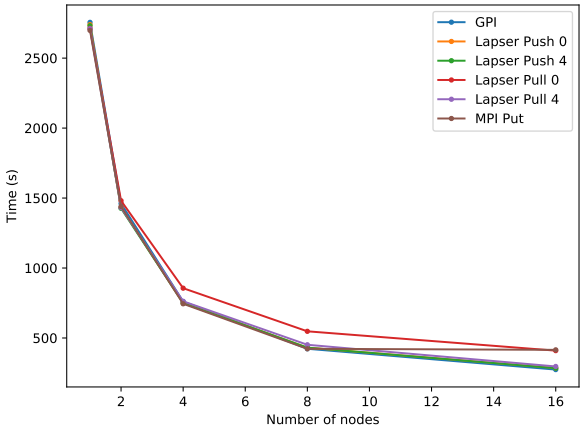


Figure 5.8: Time to complete 500 iterations with varying number of nodes when processing the larger Movielens dataset. Each node only has one worker.

The second exception is the MPI Put version, that seems to not derive any improvement from increasing the number of nodes from 8 to 16; in fact, while the other implementations enjoyed a speedup ranging from 1.3 to 1.5, MPI Put only had a mostly insignificant speedup of 1.01. We believe this lack of speedup is related to the bad performance of the MPI Put explored in fig. 5.1a.

6

Conclusion

Contents

This thesis presented Lapser, a lightweight, asynchronous and highly-scalable Parameter Server, which features tunable relaxed consistency using one-sided communication; to the best of our knowledge, this is a combination that did not yet exist in the literature. In the implementation we used two different approaches, named Push and Pull, each with their different strategy to propagate the parameters from the producer to the consumers, and different behaviours with regard to the freshness of the provided parameters. These implementations were benchmarked in a Matrix Factorization (MF) application against communication routines written and optimized by the application authors, and the results show that they achieve comparable performance both in terms of time and accuracy.

6.1 System Limitations and Future Work

The biggest limitation of our work is the support only for single-producer items, which means some ML applications may not be able to benefit from Lapser.

Another limitation is the lack of range-based operations, and more concretely the lack of mechanisms to enforce multi-item consistency – which might be needed for sensitive items in some ML algorithms.

Our work could benefit from looking at distributed systems with a similar architecture, such as RDMA-enabled distributed key-value stores, namely Pilaf [22] and Herd [26]; the former was used as an inspiration for the data race avoidance strategy, but further benefits could emerge from studying these systems.

In the shorter-term, multiple improvements to the underlying implementations of Lapser will be made, such as allowing multiple concurrent instances of Lapser in the same worker, and providing direct memory access to the item storage to avoid memory copies, for cases when the application authors want to further increase the performance of their application. Other possible implementation changes are improvements the notification id usage, and studying forms of simplifying the initialization, in particular of the metadata segment.

Finally, a performance improvement to be explored is to modify Lapser Push to use a conservative approach to communication, instead of aggressively propagating the new items as soon as they are submitted.

Bibliography

[1] E. Chan, M. Heimlich, A. Purkayastha, and R. v. d. Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/cpe.1206

[2] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *arXiv:1802.09941 [cs]*, Sep. 2018. [Online]. Available: http://arxiv.org/abs/1802.09941

[3] M. Li, Andersen, David G., Park, Jun Woo, Smola, Alexander J., Ahmed, Amr, Josifovski, Vanja, Long, James, J. Shekita, Eugene, and Su, Bor-Yiing, "Scaling Distributed Machine Learning with the Parameter Server," in *Proceedings of the 2014 International Conference on Big Data Science and Computing - BigDataScience '14*. Beijing, China: ACM Press, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2640087.2644155

[4] I. Thangakrishnan, D. Cavdar, C. Karakus, P. Ghai, Y. Selivonchyk, and C. Pruce, "Herring: rethinking the parameter server at scale for the cloud," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. Atlanta, Georgia: IEEE Press, Nov. 2020, pp. 1–13.

[5] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing, "Solving the Straggler Problem with Bounded Staleness," in *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. Santa Ana Pueblo, NM: USENIX Association, May 2013. [Online]. Available: https://www.usenix.org/conference/hotos13/session/cipar

[6] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting Bounded Staleness to Speed Up Big Data Analytics," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 37–48. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/cui

[7] A. Sergeev and M. Del Balso, “Horovod: fast and easy distributed deep learning in TensorFlow,” *arXiv:1802.05799 [cs, stat]*, Feb. 2018. [Online]. Available: http://arxiv.org/abs/1802.05799

[8] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, “A Unified Architecture for Accelerating Distributed {DNN} Training in Heterogeneous GPU/CPU Clusters,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 463–479. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/jiang

[9] R. Salakhutdinov and A. Mnih, “Bayesian probabilistic matrix factorization using Markov chain Monte Carlo,” in *Proceedings of the 25th international conference on Machine learning - ICML ’08*. Helsinki, Finland: ACM Press, 2008, pp. 880–887. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1390156.1390267

[10] Q. Luo, J. He, Y. Zhuo, and X. Qian, “Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 401–416. [Online]. Available: https://doi.org/10.1145/3373376.3378499

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: A system for large-scale machine learning,” *arXiv:1605.08695 [cs]*, May 2016. [Online]. Available: http://arxiv.org/abs/1605.08695

[12] A. Gibiansky and G. Diamos, “baidu-research/baidu-allreduce,” 2017. [Online]. Available: https://github.com/baidu-research/baidu-allreduce

[13] A. Gibiansky and J. Hestness, “baidu-research/tensorflow-allreduce,” 2017. [Online]. Available: https://github.com/baidu-research/tensorflow-allreduce

[14] H. Mikami, H. Suganuma, P. U-chupala, Y. Tanaka, and Y. Kageyama, “Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash,” *arXiv:1811.05233 [cs]*, Mar. 2019. [Online]. Available: http://arxiv.org/abs/1811.05233

[15] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, “Image Classification at Supercomputer Scale,” *arXiv:1811.06992 [cs, stat]*, Dec. 2018. [Online]. Available: http://arxiv.org/abs/1811.06992

[16] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour,” *arXiv:1706.02677 [cs]*, Apr. 2018. [Online]. Available: http://arxiv.org/abs/1706.02677

[17] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," *Advances in Neural Information Processing Systems*, vol. 26, pp. 1223–1231, 2013. [Online]. Available: https://proceedings.neurips.cc/paper/2013/hash/b7bb35b9c6ca2aee2df08cf09d7016c2-Abstract.html

[18] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting Distributed Synchronous SGD," *arXiv:1604.00981 [cs]*, Mar. 2017. [Online]. Available: http://arxiv.org/abs/1604.00981

[19] X. Zhao, A. An, J. Liu, and B. X. Chen, "Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. Dallas, TX, USA: IEEE, Jul. 2019, pp. 1507–1517. [Online]. Available: https://ieeexplore.ieee.org/document/8885215/

[20] A. Faustino, "Boosting Machine Learning with Weakly Consistent Collectives," Master's thesis, Instituto Superior Técnico - Universidade de Lisboa, Lisbon, Portugal, Jan. 2021. [Online]. Available: https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/1128253548922061

[21] C. Simmendinger, M. Rahn, and D. Gruenewald, "The GASPI API: A Failure Tolerant PGAS API for Asynchronous Dataflow on Heterogeneous Architectures," in *Sustained Simulation Performance 2014*, M. M. Resch, W. Bez, E. Focht, H. Kobayashi, and N. Patel, Eds. Cham: Springer International Publishing, 2015, pp. 17–32.

[22] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 103–114.

[23] F. M. Harper and J. A. Konstan, "The MovieLens Datasets: History and Context," *ACM Transactions on Interactive Intelligent Systems*, vol. 5, no. 4, pp. 1–19, Jan. 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2827872

[24] T. Vander Aa, I. Chakroun, and T. Haber, "Distributed Bayesian Probabilistic Matrix Factorization," *Procedia Computer Science*, vol. 108, pp. 1030–1039, Jan. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187705091730501X

[25] J. Geng, D. Li, and S. Wang, "Rima: An RDMA-Accelerated Model-Parallelized Solution to Large-Scale Matrix Factorization," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, Apr. 2019, pp. 100–111, iSSN: 2375-026X.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of the 2014 ACM conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: Association for Computing Machinery, Aug. 2014, pp. 295–306. [Online]. Available: https://doi.org/10.1145/2619239.2626299