Boosting Machine Learning with Weakly Consistent Collectives

Amândio Faustino
IST and INESC-ID
University of Lisbon
Lisbon, Portugal

ABSTRACT

Stale Synchronous Parallel (SSP) is a synchronization model proposed to speed up iterative convergent Machine Learning algorithms in distributed settings. In this model, synchronization among workers is reduced by allowing workers to see different intermediate solutions that can be a bounded number of iterations out of date (bounded staleness). With the advent of Remote Direct Memory Access (RDMA), one-sided communication has become a popular alternative to two-sided communication in asynchronous environments. Although SSP is inherently asynchronous, to the best of our knowledge no SSP solutions are using one-sided communication.

The goal of this thesis is to create a solution to SSP that takes advantage of RDMA's support for one-sided communication and to provide it to application programmers through a new weakly consistent collective abstraction developed using the GASPI API. To this end, we designed and implemented two different solutions ranging from directly adapting an existing synchronous allreduce algorithm to support SSP, to using the ideas behind a Parameter Server architecture, and running the Parameter Server shards directly on the nodes performing the collective.

Our solutions were evaluated on the MareNostrum4 supercomputer, using up to 64 nodes, and evaluated under two implementations of the Matrix Factorization algorithm, one being our own, and the other a real-world implementation. Using our proposed collective we were able to reduce the collective execution time by up to 2.5x when compared to MPI's allreduce, while having minimal impact on the convergence rate of the algorithms tested.

KEYWORDS

Stale Synchronous Parallel, Remote Direct Memory Access, Distributed Computing, Machine Learning

1 INTRODUCTION

Machine Learning (ML) is rapidly becoming one of the most important building blocks of new applications and services. With ML, we are able to extract useful information from large-scale data, across a variety of domains. Examples of ML techniques of growing importance include: recommender systems; image, video and text classification; topic modeling; etc.

From the collection of ML algorithms, we will focus on algorithmic approaches that can be classified as iterative and convergent (e.g., matrix factorization and topic modeling). These algorithms start with an initial guess of the solution and improve on this guess over several iterations. This process is repeated until sufficient convergence has been reached, or after a given number of iterations has been completed. These algorithms were often envisioned to run sequentially on a single machine. However, for these algorithms to have good results, they require the analysis of large amounts of data, which in turn requires a lot of computational power. Therefore, the algorithms need to be brought to a distributed setting.

Distributed implementations normally shard the input data (e.g., user-movie ratings or news documents) across the available workers, and follow the BSP model [4]. According to this model, all the workers share the same intermediate solution, and, in each iteration, each worker is in charge of computing adjustments to the current solution using its assigned input data and the current intermediate solution. At the end of each iteration, workers apply their adjustments to the shared solution and wait for each other, ensuring that in the next iteration all workers see the same, now improved, intermediate solution.

However, this solution has the shortcoming of leading to possibly long wait times, where fast workers are sitting idle waiting for the current iteration to end due to slow workers. To address this problem, a recently proposed model called SSP [4] focuses on using a weak consistency model called bounded staleness for the shared algorithm state. In this model, workers are allowed to see a stale intermediate solutions that can be a bounded number iterations out-of-date. Because of this extra flexibility, instead of synchronizing at the end of each iteration like BSP, SSP allows the fastest worker to be a bounded amount of iterations ahead of the slowest worker. By tuning this bound value, SSP, is able to reduce the synchronization delays (less waiting for slow workers) and communication costs (less communication), which, in turn, despite requiring more iterations to achieve convergence, speeds up the overall convergence speed of the algorithm.

Very often, distributed ML systems are deployed on high performance computing (HPC) clusters, which normally support several state of the art network technologies, most notably RDMA. In an RDMA network, we can create the idea of a PGAS, where we can read, write and allocate memory regions across multiple computers as if we were doing so in a local memory device. Furthermore, in an RDMA setting, we can perform fast one sided communication, meaning that the requesting node can read/write to the destination node by reading/writing directly from/to memory without the intervention of destination node. By doing so, the destination node does not need to use CPU cycles to answer the request and there is no need to insert synchronization points between processes on different machines in order to achieve inter-node communication. To abstract the low level API offered by RDMA, several libraries have been developed. Among them we find the GPI-2[1] library, an implementation of the standard GASPI (Global Address Space Programming Interface) [14] API.

Several implementations of the SSP model have been proposed [4, 7]. In these approaches, communication is performed using the two-sided communication paradigm. With the advent of Remote

[1] Found in their website http://www.gpi-site.com/

Direct Memory Access (RDMA), one-sided communication has become a popular alternative to two-sided communication in asynchronous environments. Although SSP is inherently asynchronous, to the best of our knowledge no SSP solutions are using one-sided communication.

Still in the world of HPC, the available programming models commonly offer a set of abstractions called collectives. Through collectives we can efficiently perform communication among a group of nodes to exchange information. These could be used, for example, between workers to exchange updates to the global solution by using the *allreduce* collective. In an *allreduce*, the data from each worker is combined, for example, by adding all the updates together, and then the final result is obtained in each of the participating nodes. The *allreduce* collective requires the participation of all nodes involved. In case some of the nodes are slow and take longer to contribute, the collective can take too long to finish. In these situations, if the application tolerates some sort of staleness, nodes could be allowed to finish the collective before all nodes have contributed, by using staler versions of missing contributions. SSP is a model that defines limits to this relaxed behavior. However, to the best of our knowledge, there are no *allreduce* collectives based on SSP.

Observing these two research opportunities, we created a solution based on SSP through a new weakly consistent collective built upon the GASPI API to take advantage of RDMA technology. In this setting, we define a weakly consistent collective as a collective that, even though it involves data from all nodes, may compute based on slightly stale versions of that data. Therefore, our proposed primitive has the potential to require a much shorter synchronization time, and even, in many cases, no synchronization at all.

The main contributions of this thesis are as follows: (1) we develop a simple to use collective that is able to abstract the synchronization model of SSP; (2) we propose two alternative designs for this collective, both of which take advantage of asynchronous execution of SSP by using one-sided communication; (3) we implement these two variants of the new collective, and evaluate their effectiveness by applying them to several iterative and convergent computations, including a real-world use case in the area of drug discovery. Using our proposed collective we were able to reduce the collective execution time by up to 2.5x when compared to MPI's allreduce, while having minimal impact on the convergence rate of the algorithms that were tested. Section 4

The rest of the document is organized as follows: Section 2 provides an overview of existing work related to the use of staleness in ML as well as the use of RDMA in common approaches to communicate worker updates. In Section 3 we explore the two alternative designs for this collective, and present its API. Section 4 presents the evaluation methodology, details the applications used, and analyzes a set of experimental results. Finally, Section 5 provides a brief conclusion overviewing the contributions of this work and highlights possible directions for future work.

2 RELATED WORK

The use of staleness in ML is a widely used practice [3–5, 7–11]. In these systems, several synchronization models were proposed to reduce the convergence time of iterative convergent algorithms. One synchronization model is the aforementioned SSP [4], where workers are allowed to be a bounded number of iterations apart. Other models [5, 9–11], discard the need for synchronization entirely and instead allow workers to work independently from each other. The lack of synchronization between workers is appealing and has been proven to converge under certain scenarios [10]; however, more synchronous models such as SSP have been shown to converge faster than fully asynchronous systems due to more accurate worker updates [4, 7].

In order to communicate updates, these systems usually resort to using an architecture referred to as Parameter Server (PS) [3–5, 7–9]. In this architecture, one or more specialized nodes hold the current state of the model and are in charge of receiving and applying worker updates to the current state of the model, as well as sharing the current model with the workers.

Other approaches use the widely studied, allreduce collective [12, 13] to perform the exchange of updates directly between the workers. In this approach, the current state of the model is not stored in a central location. Because of this, instead of communicating the new contribution, workers communicate the history of all contributions they computed as a single contribution (corresponding to the reduction of all contributions in the history). By reducing the history of contributions from all workers, we obtain the updated model.

From these two approaches to communicate updates, we have only seen SSP used in conjunction with the PS architecture. All implementations of allreduce seem to follow the synchronous approach of waiting for a new contribution from each worker before returning, as opposed to using stale data when a wait was required.

Regarding the use of RDMA, to the best of our knowledge, there are no PS implementations following the SSP synchronization model and using one-sided communication. Instead, these approaches use the more common two-sided communication. Considering allreduce collectives, traditional implementations also use two-sided communication; however, more recent work proposes algorithms that are able to take advantage of one-sided communication to speed up collective operations. One of these papers [2] proposes a novel approach to collectives by extending GASPI's notification system enabling all processes participating in a shared window (shared memory) to observe the entire notified communication at the window. With this extension, the authors implement the Allreduce and Allgather(V) collectives and achieve 2x-4x performance improvements compared to the best performing MPI implementations for various data distributions. Another approach can be seen in [6] where they propose an algorithm that takes advantage of GASPI's split-phase collectives using n-way dissemination. They are referred to as split-phase because from every call to the collective you can specify that you only want it to progress to the next step (or phase) of the algorithm. This approach was able to outperform the best MPI implementations while reducing the number of communication rounds.

3 SSP_ALLREDUCE

In this section we will describe our weakly consistent collective, which we named *ssp_allreduce*. To create *ssp_allreduce*, we explored

solutions that ranged from directly adapting an existing synchronous allreduce algorithm to support SSP to creating a new collective based on the ideas of the Parameter Server (PS) Architecture.

We propose two possible implementations of an allreduce collective using SSP. Our goal in providing these two solutions is to explore different points in the design space.

- **Hypercube-SSP**. The first solution is based on the Hypercube allreduce algorithm. This solution is intended to be aligned with existing synchronous allreduce collective design.
- **Sharded-SSP**. The second solution uses the ideas behind the PS architecture in a collective by sharding the PS and running PS shards on top of the workers. This is the approach that is normally used with SSP [4, 7, 8].

In the next subsection, we will describe our proposed API. Following that, we describe the two main solutions in more detail.

Throughout the description of these solutions, we will use the following definitions proposed by prior work [4, 7]:
Clock - progress of workers is measured in clocks, where in each clock a certain amount of work is performed (e.g., one iteration).
Age of data - workers produce 1 contribution per clock. The age of that data is defined to be the clock at which that contribution was computed on. When reducing contributions together, the age of the data becomes the smallest age of any of the contributions involved.
Slack - allowed maximum age difference between contributions used when performing a reduction.

3.1 ssp_allreduce API

When designing our API, our goal was to keep it similar to existing synchronous allreduce APIs by only performing specific adjustments to the SSP setting.

One of the adjustments consisted of adding a parameter allowing the programmer to select the desired slack. Another adjustment was the removal of several parameters that cannot change between calls. These correspond to the number of elements in the reduction vector, the size of each element, and the reduction operation to use. Instead, these parameters are passed to an initialization function, which takes care of setting up important data structures to support SSP. The final adjustment was made to allow multiple collectives to execute at once. This was done by adding another parameter to the collective representing the ID associated with the reduction we wish to target. This ID is passed to the initialization function and corresponds to a GASPI segment ID that is created and used by that reduction.

After initialization, the collective can be called iteratively with the function:

```
ssp_allreduce( reduction_result, new_contribution,
               reduce_segment_id, slack )
```

- `reduction_result`: vector that will receive the result of the reduction;
- `new_contribution`: vector containing a new contribution to be added to the reduction;
- `reduce_segment_id`: segment id identifying the reduction to be targeted;
- `slack`: desired slack;

Having described the API, we will now describe our solution in more detail, beginning with Hypercube-SSP.

3.2 Hypercube-SSP

For our first solution, we propose Hypercube-SSP. To understand the modification we made to the regular hypercube allreduce algorithm, let us start by taking a closer look at each step of the reduction process. The left hand side of Figure 1 details the communication pattern for the hypercube algorithm using 8 workers. The grid on the left of the figure is composed of several squares, each representing a worker. The y-axis of the grid corresponds to the rank of the worker, and the x-axis to the step of the algorithm. At each step, workers connected by an edge exchange contributions, and reduce the received contribution with the contribution they have sent, resulting in a partial reduction to be sent in the next step. We refer to it as partial as it does not contain all contributions. After doing this process for enough steps, the final reduction produces the reduction result.

To adapt this algorithm to support SSP, workers will remember the contributions received at each step, and provided that these contributions are not too stale, use them instead of waiting for fresh contributions.

In our designs, we are considering solutions based on one-sided communication. This allows for workers to send data to another worker, without requiring the receiving worker to call a receive primitive. Instead, the worker wanting to send the data can write directly to a remote segment of that worker, which we will call **Received data**. Later, the receiving worker, will read from that segment, and check for new data received, which can be done in GASPI using notifications. In algorithm .1, we present the pseudo code for this adaptation.

Algorithm .1: Hypercube-SSP

```
// Algorithm in a hypercube with k dimensions
Input  : new_contribution, slack
Output : reduction_result
begin
    clock ⟵ clock + 1
    min_clock_accepted = clock − slack
    partial_reduction ⟵ new_contribution
    for 0 ≤ k < d do
        comm_worker ⟵ get_comm_worker(k)
        // Send partial reduction
        send(partial_reduction, clock, comm_worker)
        recv_data ⟵ recv_data_vec[comm_worker]
        if recv_data.clock < min_clock_accepted then
            wait_for_update(comm_worker)
        partial_reduction ⟵
          reduce(partial_reduction, recv_data)
    reduction_result ⟵ partial_reduction
```

To explain it in an accessible way, we next present an example of its execution. The example can be found in the middle of figure Figure 1. In the example, we are looking at worker 0, currently with clock 3 in a scenario where slack is set to 1. Slack being 1 means that the worker can use data from the current clock, in this case, clock 3, but it can also use data from the previous clock, clock 2. In the first step, the worker sees that it already received data from the communicating process, and uses it to move on to the second step. At the second step, the worker now finds received data that is stale at clock 2, but still fresh enough to be used in order to move on to the next step. Once it reaches the last step, it finds that the current received data is too stale too be used. In this case, and only in this case, the worker waits until receiving a new update for the current step.

3.3 Sharded-SSP

The next solution is based on the Parameter Server Architecture. Previous work has used the combination of SSP with a PS architecture, [4, 7]. However, because our implementations are based around the use of collectives, we cannot have parameter servers running on dedicated servers. Instead, the work of the PS will be handled directly by the workers. More precisely, the Parameter Server is sharded into as many shards as there are workers, and each of those shards is assigned to a worker. For this solution, we use a different reduction paradigm than in our previous solution. Following the approach used in Parameter Servers, we now expect to receive contributions that can be directly reduced with a current reduction state. We will refer to this solution as `Sharded-SSP`, and also simply as `Sharded`.

To start off this section, let us take a look at how we handle communication. In this solution, workers exchange information by communicating through the PS shards in the following way: workers communicate a new contribution by sharding the contribution into contribution shards, and sending them to the corresponding PS shard; The PS shards, upon receiving parameter updates, apply them to the current reduction state. Then, whenever a shard has received one update from each worker, it sends the new reduction state for that shard, to all workers.

To support this communication scheme, we require some utility memory segments. Both workers and PS shards require 2 memory segments each: one from which they push information, and another segment always ready to receive information. Worker memory segments:

- **Missed contributions**: contains worker contributions not yet sent to the PS shards. This segment is split into shards according to the sharding of the reduction vector.
- **Local Reduction**: contains reduction updates received from the PS shards. Similarly to the above segment, this segment is split into shards according to the sharding of the reduction vector.

Parameter server shard memory segments:

- **Reduction state**: contains the most up-to-date reduction for the assigned shard of the reduction vector.
- **Worker contributions**: contains the last contribution shard received from each worker. This segment has dedicated memory to receive contribution shards from each worker.

In the SSP solutions we mention, workers replace their last contribution directly from the memory of the receiving PS shard. Because of this, it is possible for a worker to replace the last contribution when the receiving PS shard has not yet used it. If a contribution is replaced with a new one before being reduced with the current reduction state, we will effectively lose the previous contribution. While, for most iterative convergent applications, missing contributions is acceptable, we note that our goal with this collective is to broaden its applicability, and because of that we want to support algorithms that cannot afford losing contributions.

In order to address the problem of missing contributions, we were inspired by an idea presented in DOGWILD [11], where workers only communicate with a master (which essentially takes the role of a Parameter server) upon receiving reduction updates from the master. Based on this idea, we decided to create a cyclic dependency between workers and PS Shards, which prevents the loss of updates from workers at the PS shards. In this scheme, workers only send contributions to the PS Shards when they receive reduction updates, and Parameter Server Shards only send reduction updates to a given worker after it has used the latest contribution from that worker. This cycle is illustrated in Figure 2 using arrows.

Because workers are not always sending their latest computed contribution, we need to store them in someway so that they can be sent later, when the PS shards are ready to receive them. We do this as follows: at the start of the call to `ssp_allreduce`, the worker begins by reducing the new contribution with any previously not sent contributions. The result of this reduction corresponds to a new contribution that has the information of both the new contribution and the contributions not yet sent to the PS shards. We will refer to this new contribution as missed contributions. With this idea, when the worker wants to push a new contribution to the PS shards, it will send the missed contributions.

The final issue we had to tackle was how to synchronize the contributions received by the PS shards. We have achieved this through our push policy, i.e., the conditions required to push updates, and an extra memory buffer. First, we will explain the push policy. Here is how the mechanism works: when a worker has contributions to push, it only pushes the contribution after knowing that all shards are ready to receive the new contributions; On the PS shard side, the shard only sends updates after processing a new contribution from each worker. From the combination of these two conditions, we ensure that the reduction update sent by each shard to a given worker results from the same set of contributions. However, this does not mean that all reduction updates reach the worker at the same time, so it would still be possible for a worker to see some shards with a previous update and others having new reduction updates. To address this issue, we have an extra memory segment, which we will call **stable_local_reduction**. This segment is updated by the worker once it receives an update from all PS shards, and the worker will only use that memory segment to perform local reductions.

To finish the design we will describe how we integrated the PS shard execution with the execution of the worker, inside the execution of the collective. A possible approach would be to run the parameter server as a separate thread inside the worker, and thus mimic as if the parameter server shard was running on a dedicated

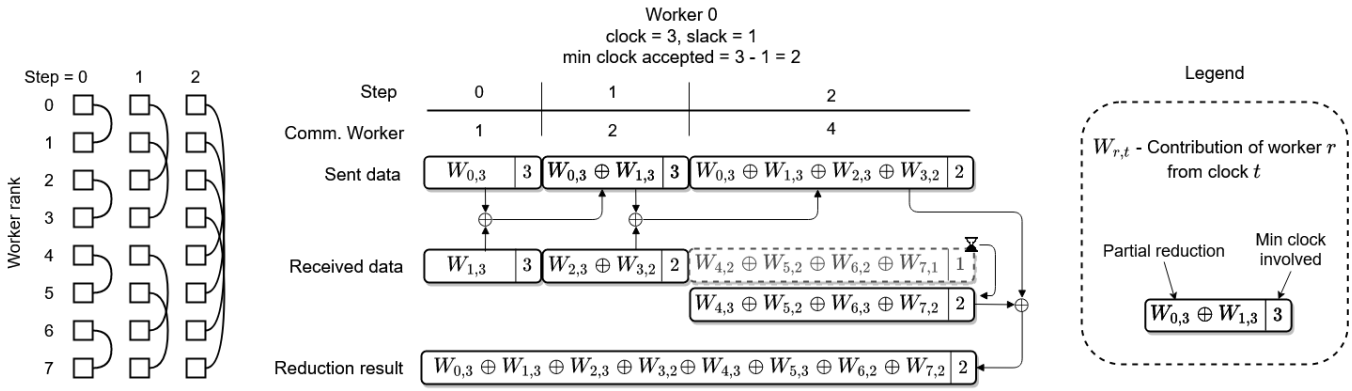

Figure 1: On the left hand side of the figure we see the communication pattern in hypercube using 8 processes. And at the right hand side we have an example of Hypercube SSP.

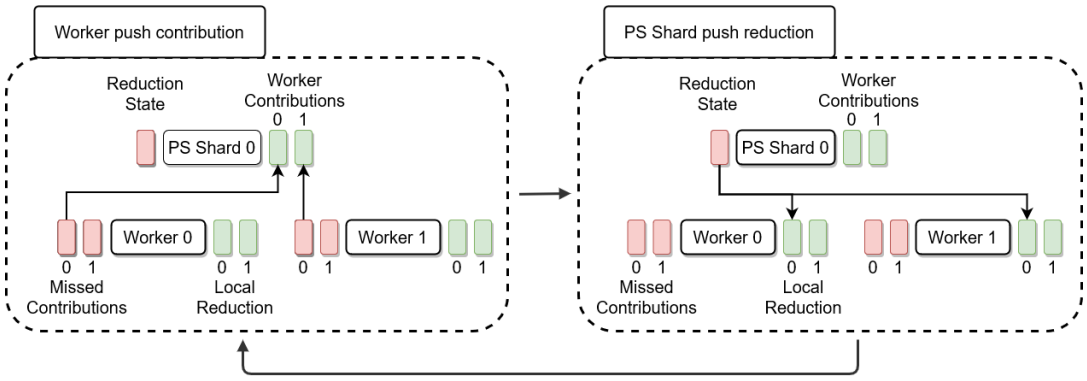

Figure 2: Communication pattern on Sharded SSP in a setup with 2 Workers. Given this configuration, the reduction vector is split in two shards, shard 0 and shard 1, each assigned to a PS shard. In the figure, we focus on shard 0. On the left of the figure, we see Workers pushing their missed contributions regarding shard 0 to their assigned memory on PS shard 0. On the right-hand side of the figure, we see the PS shard 0 pushing an updated reduction state, for the shard it is handling, to all workers.

server. However, in order to be able to have a more fair comparison between our approach and existing allreduce algorithms, we decided to avoid threads in the presented solution, and benefit from threads in a variation of this solution which we describe in subsection 3.4.

To avoid using threads in this design, we decided to interleave the operations of the worker with the operations of the Parameter Server as follows:

(1) Worker push contribution – Worker starts by reducing the new contribution with any previously missed contributions, updating the missed contributions. After this, the worker checks for received updates from all PS shards. If the process received one update from each PS shard, it pushes the unseen contributions to the PS shards. Otherwise, the push is skipped.
(2) PS shard reduce – PS shard (associated with the worker) reduces received contribution shards with the current reduction state of the shard. Then, it waits for more updates if continuing means we enter a deadlock. After that, it checks if the current reduction state received one update from each worker, since the last push. If it did, it pushes the current state of the reduction to all workers. Otherwise, the push is skipped.
(3) Worker update local reduction – Worker checks for received reduction updates from PS shards. If an update was received from each PS shard, replace the current stable local reduction, stable_local_reduction, with the updated reductions received. If the current stable_local_reduction is staler than slack, wait for reduction updates before continuing. Finally, reduce the stable_local_reduction with any unsent contributions, and return the result to the user.

Having mentioned the particulars of the design, in algorithm .2, we describe the pseudo code for the algorithm.

Algorithm .2: Sharded-Sync-SSP

Input : new_contribution, slack
Output : reduction_result

begin

```
clock ⟵ clock + 1
min_clock_accepted = clock − slack
// 1 - Worker Push contribution
missed_contr ⟵
  reduction(missed_contr, new_contribution)
if can_push_to_shards then
    push(missed_contr, clock)
// 2 - PS shard reduce
check_worker_updates()
if reduction_state.clock < min_clock_accepted then
    wait_for_remaining_worker_updates()
if recv_updates_from_all_workers then
    reduction_state ⟵ apply_worker_updates()
    push_reduction_to_workers(reduction_state)
// 3 - Worker update local reduction state
check_ps_shard_updates()
if stable_local_reduction.clock < min_clock_accepted
  then
    wait_for_remaining_ps_shard_updates()
if recv_updates_from_all_workers then
    stable_local_reduction ⟵
      copy_local_reduction()
reduction_result ⟵
  reduction(stable_local_reduction, missed_contributions)
```

3.4 Sharded-SSP Variants

In this section we propose two variants to the base algorithm. In the first variant workers are allowed to use local reductions where different shards can hold different contributions between them. In other words, local reduction shards are no longer required to be synchronized so that they always have the same contributions. Note that, in order to respect SSP guarantees we still enforce that each local shard is not too stale. Given this relaxation of the synchronization requirements, we propose the following algorithm, which we call **Sharded-Async**.

This new algorithm takes advantage of three benefits allowed by the weaker set of synchronization assumptions. One of the benefits is that it now allows the worker to use reduction updates directly from the local reduction segment. In other words, it does not need to use the previous **stable_local_reduction** segment, where all shards have the same contributions.

The second benefit is the fact that we can increase the push frequency at the worker, enabling contributions to reach PS shards faster. Previously, the worker had to wait to receive a reduction update from all PS shards before being able to push. To avoid this unnecessary wait, PS shards can communicate with each worker directly, in order to inform the workers that they have processed the last contribution that was sent, and are ready to receive a new one.

The final benefit is the possibility to increase the push frequency at the PS shard, allowing reduction updates to reach workers faster. With this added flexibility, we decided to have the PS shard push reduction updates whenever the age of the reduction state increased, as this corresponds to the minimum change to the reduction state required to prevent a worker from waiting for fresh data.

Another variant we implemented concerned the possibility of using multiple threads per worker process. Multithreading is a natural fit for this design because it enables the PS shard to execute in parallel with the worker. This mimics the effects of having the PS shard being executed on a dedicated server, which is co-located on the same machine as the worker.

We will refer to these variants as "Threaded", and will append the label "Thread" to the name of their base solutions, thus adding the following two designs to our space of solutions: **Sharded-Thread**, and **Sharded-Async-Thread**.

4 EXPERIMENTAL EVALUATION

In this chapter we evaluate both `Hypercube-SSP` and `Sharded-SSP`, as well as its variants. During the evaluation we will focus on different properties of the solutions, namely: execution time of the collective; average age of contributions used in reductions; time waiting for fresh contributions; and finally, the impact of slack on all the previous properties.

To evaluate our collective, we will be using two Matrix Factorization (MF) implementations. One of them is our own implementation of Matrix Factorization using Stochastic Gradient Descent. The other implementation corresponds to the implementation made by Vander Aa et al. [1], where they use Bayesian Probabilistic Matrix Factorization (BPMF), in the context of an algorithm for drug discovery.

The evaluation is split into three parts. In the first part, we make a side by side comparison of the two solutions we developed. In this part, we will be determining which of these solutions performs best, both in terms of the collective execution speed, as well as the ability to use (on average) fresher data. For these experiments we will be running our implementations on a relatively small workload, running 200 iterations on our own MF implementation (and not until convergence, since that is not the main point of comparison for this first part). Once we compare the two solutions, we fix the best-performing implementation, and, in the second part, we analyse how the use of slack affects the quality of the solution reached by running the same experiments, but, this time, until the solution converges. Finally in the third part, we use our best collective implementation in the BPMF implementation, to obtain preliminary results of the use of our collective in a real-world application.

Next, we discuss the baselines to which we will compare our collective. There will be two classes of baselines. One of them corresponds to the performance of our collective when we consider the BSP synchronization model, or in other words, use slack equal to zero. In addition to this more direct comparison to the same codebase with a particular parameterization, we also compare to

two other, existing allreduce implementations: Intel's MPI Library (impi/2018.4) and an implementation of the segmented pipeline ring on top of GPI-2 (which we will refer to as Ring GPI-2), developed by a member of the project in which this thesis takes place.

To conduct our experiments we used the Marenostrum4 cluster[2]. Nodes are interconnected through Intel OmniPath HFI Silicon. This network supports RDMA technology and allows for 100 Gbit/s of bandwidth between each pair of nodes. Except otherwise stated, we will be running experiments using 32 nodes, each with 48 cores, and assign one gaspi process per node. Both our applications have thread support and we assigned 1 thread for each available core.

4.1 Hypercube-SSP

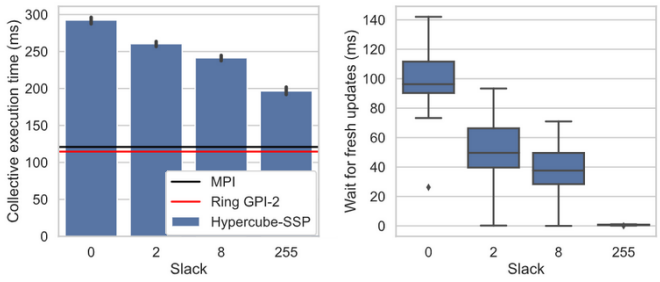

Figure 3: Evaluation of the Hypercube-SSP solution

In Figure 3, on the left plot, we see the collective execution time for the Hypercube-SSP solution. From the plot, we can see that this solution performs significantly worse than both MPI and the Ring GPI-2 implementation. In fact, even in the configuration for the slack value with the lowest execution time, this solution is still around 1.75x worse than our baseline synchronous collectives. This was expected, considering that the Hypercube communication pattern is best suited for small vectors, and in this evaluation, we were using a vector of considerable size. On the other hand, we see better performance on both Ring GPI-2 and MPI that use allreduce algorithms suited for large vectors. Regardless of the absolute performance, we see the collective benefiting from higher values of slack by being able to reduce, and even completely eliminate, the time waiting for fresh updates (as shown in the right plot).

As we mentioned, we did not expect this solution to have stellar performance. Instead, we designed it as a first attempt at adapting an existing step based allreduce algorithm to use SSP and determine if SSP would reduce their execution time (which we confirmed). For this reason, in the rest of the evaluation, we focus on the best performing `Sharded` solution and its variants.

4.2 Sharded-SSP

On the left plot of Figure 4, we find the execution time for the `Sharded-SSP` solution and its variants. From this plot, we can see that all of our variants are able to reduce the collective execution time below the execution time of both MPI and Ring GPI-2. If we now take a look at the plot on the right, we see that, similarly to the previous solution, as we increase the value of slack the time waiting for fresh updates decreases, and this is even completely removed at max slack.

Let us now take a closer look at the performance of each variant. First, we will analyse the collectives that do not rely on running the PS as a separate thread – `Sharded` and `Sharded_Async`. Beginning with slack equal to 0 we see that both variations starts off requiring more time than both of our baselines. however after a certain slack, they are both able to surpass our baselines. In the figure we see this happening at slack 8 for `Sharded`, and slack 255 for `Sharded_Async`.

Next, we analyze the variants of the previous solutions where the PS shard is executed as a thread – `Sharded_Thread` and `Sharded_Async_Thread`. These variants produced interesting results: not only did they achieve execution times that are much lower than the previous variants (for all values of slack), but they were also able to achieve lower execution times than MPI and Ring GPI-2 in a synchronous environment – corresponding to the case where slack is zero.

At their lowest execution time, `Sharded_Thread` and `Sharded_Async_Thread` are, respectively, 57.9% and 59.3% faster than MPI's allreduce, and 55,5% and 56.9% faster than Ring GPI-2. We attribute these improvements to two main factors: first, and contrary to most synchronous collectives, the `Sharded` solution does not enforce a specific order of reduction for contributions, and, instead, contributions are applied in the order they arrive; second, because the PS shard is always running in the background, newly received contributions can be applied to the reduction state even before the associated worker has called the collective.

From this experiment, we have seen that the "Threaded" variants are our fastest implementations. Knowing this, we now attempt to get a better understanding of the age of the data used by each of these two variants. To this end, we will analyze how the age of local reductions at workers evolves during this execution.

4.3 Age of data used

In this subsection, we introduce the term *age difference* to quantify the staleness of reductions used by a worker. As an example, if a worker is at clock 3 and uses a local reduction with clock 2, we say that the reduction used has age difference of $3 - 2 = 1$.

Figure 5 depicts how the average age difference evolves with the number of iterations for the "Threaded" variants. Note that the plot regarding `Sharded_Async_Thread` describes the age difference per shard. Considering the fact that we have 32 workers, this implies that we have 32 shards, and so for each iteration we get 32 different age differences. Since we have 200 iterations we have $200 \times 32 \approx 6400$ entries in the plot.

By construction, `Sharded_Async_Thread` can push reduction updates faster than `Sharded_Thread`. That means that, at the time the collective is called, workers can use reduction updates that are fresher when compared to `Sharded_Thread`. We can see this is the case by comparing the slopes of both solutions for the same slack value. In general, the lines for `Sharded_Thread` has a steeper slope than `Sharded_Async_Thread`, meaning that `Sharded_Thread` reaches a given age difference faster than `Sharded_Async_Thread`. Considering a slack value of 8, `Sharded_Thread` reaches max slack at around the 35th iteration, whereas for `Sharded_Async_Thread` this is reached at around the $3,000th$ ⟨shard,iteration⟩ or $3000/32 \approx$

[2]https://www.bsc.es/marenostrum/marenostrum

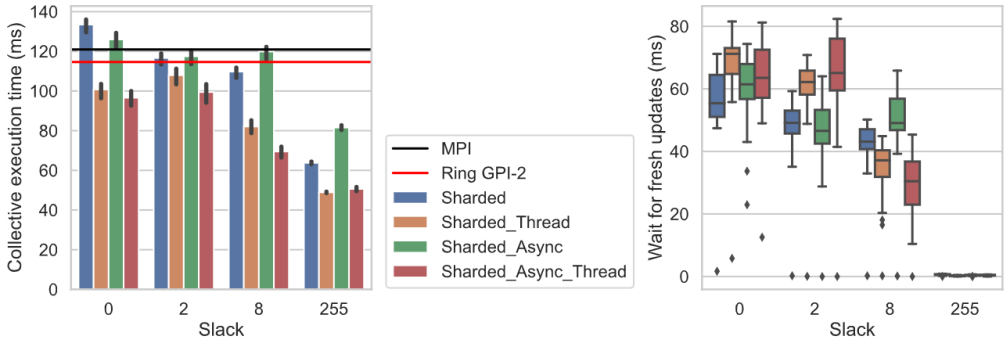

Figure 4: Evaluation of the Sharded-SSP solution and its variations using 48 cores per process.

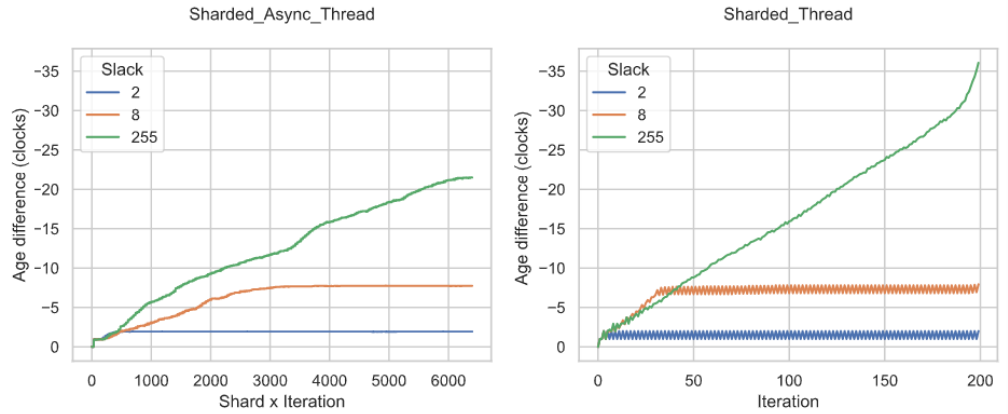

Figure 5: Age difference of contributions used in "Threaded" variation of Sharded-SSP using 48 cores per worker.

93th iteration. Similarly, for a slack value of 255, we see that `Sharded_Async_Thread` reaches an age difference of 20 at the $5500/32 \approx 172$th iteration, while `Sharded_Thread` reaches it at the 125th iteration.

From these plots, we can also see an interesting point that we have not emphasized yet. When using slack, we can reduce the time waiting for fresh updates; however, if a given worker is constantly slower than the others, other workers will iterate faster, and, eventually, will be slack clocks away from the slowest worker. After this point, slack is not beneficial, because it cannot further reduce the time waiting for updates. In other words, this means that we enter an execution resembling an execution with slack 0. However, until reaching that point the collective already benefited from being able to do the previous iterations much faster.

This is actually what is happening in our experiment. If we look at the time waiting for fresh updates in this experiment using slack 0, we can see that some outliers barely wait for data. This tells us that there are workers that are so late to the collective, that all other workers are waiting for it. We can clearly see this effect in the executions using slack 2 and slack 8, where after some time, the slack is reached and the delay stays at this value until the end of the execution.

From this initial analysis, we concluded that the "Threaded" variants are the fastest of the `Sharded` solutions and that both "Threaded" variants reach similar execution times. However, `Sharded_Async_Thread` can use fresher data. For this reason, we take `Sharded_Async_Thread` to be our best solution, and in the next section, we will focus on this variant and study its impact on convergence.

4.4 Impact of slack on time to reach convergence

From the previous experiments, we have seen that we were able to progress through our iterations faster by using stale data. However, because we are using stale data, we may require more iterations to reach a certain error. In this section, we will evaluate the impact of slack on the time required for our own MF to converge. For these experiments, we say that the model converges when its error reaches the error value measured after 3, 000 iterations, using a slack of 0. The results from our experiments can be found in Figure 6.

From our experiment, we can see that with slack 255, we are able to reach the same error as slack 0 using approximately 75 fewer seconds, which corresponds to being 10% faster. We can also see the effect that we highlighted in subsection 4.3, namely that, at a certain point, max slack is reached, and after that, the collective

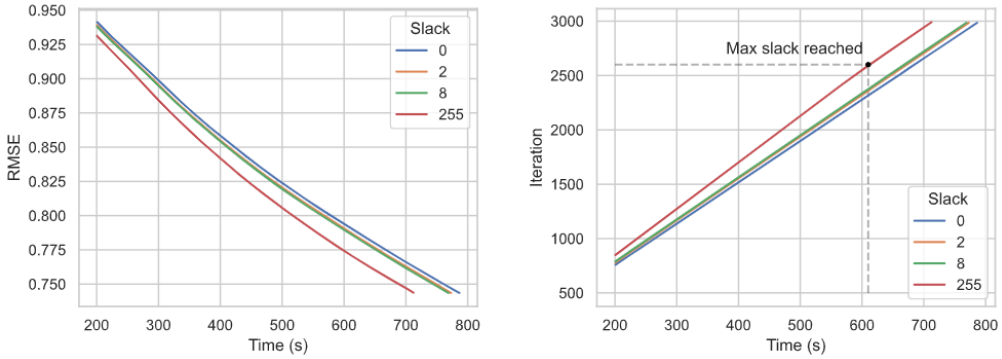

Figure 6: Evaluation of the impact of slack on time to reach convergence using `Sharded_Async_Thread`.

behaves as if it was synchronous. However, at that point it already benefited from being able to do the previous iterations much faster. We can see this point for the execution using a slack of 255 on the right plot. This point occurs at around 610 seconds, where, after that point, we can see the rate of iterations becoming parallel to the rate of iterations of slack 0, indicating that it progresses as fast as a synchronous collective.

Another thing to point out is the fact that not only have we reached a similar error in less time, but also the execution with slack 255 only required 5 additional iterations when compared to slack 0. This result may seem surprising because we are using very stale contributions. However, the iteration effectiveness, or the improvement of the error per iteration, does not seem to be particularly affected. We believe this is happening because, in between iterations, the age difference between the worker and the contributions that are used increases relatively smoothly and slowly. We can see that this is the case by looking at how the age difference evolves over time in the left plot of Figure 5 for the first 200 iterations of this execution. As such, even though the age difference of contributions is rather high, the age used in one iteration is not too different from the previous one. This translates into the fact that, in between iterations, workers do not observe their previous reduction state change significantly due to the use of stale data.

4.5 BPMF

In this final section we will evaluate BPMF using the `Sharded_Thread` variant and using different values of slack. We choose this variant because, BPMF does not support using a reduction result where different shards are comprised of different contributions.

For this experiment, we tried to use the same slack values as before. However, we quickly realized that the algorithm was not converging when we tried slack values higher than 2. For this reason our evaluation now only experiments with slack 0, 1 and 2.

In this solution, we required 2 reductions which we will call L, R. In Figure 7, we can see the collective execution time for each of the reductions.

From these results we can conclude that, in general, the collective execution time decreases with slack, and that at max slack tested, slack equal to 2, the collective reaches execution times lower than

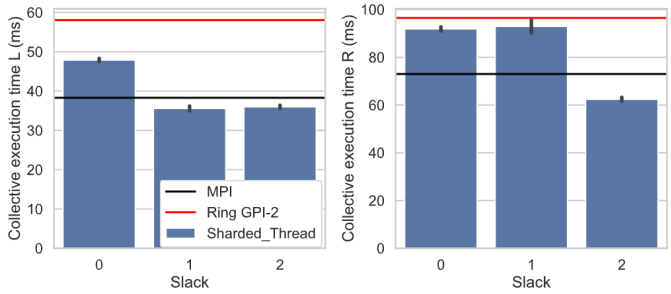

Figure 7: Evaluation of `Sharded_Thread` on the BPMF application.

both baselines. We can also see that, regardless of the slack value, our solution is below the Ring GPI-2 baseline.

From this preliminary analysis, we believe that, there is potential benefit from incorporating SSP with BPMF, but further analysis is required.

5 CONCLUSIONS AND FUTURE WORK

This thesis proposed a new weakly consistent collective abstraction that follows the SSP synchronization model. To implement this collective we explored two alternative designs, `Hypercube-SSP` and `Sharded-SSP`, both of which take advantage of the asynchronous execution of SSP by using one-sided communication. In the `Hypercube-SSP` design, we directly adapted an existing synchronous allreduce algorithm to support SSP. For the `Sharded-SSP` design, we used the ideas behind the PS architecture in a collective by sharding the PS and running PS shards directly on the nodes calling the collective. Besides designing these two variants, we were involved in the design of an adaptation to the BPMF algorithm to support the use of an allreduce collective, allowing us to test our collective in this algorithm. We implemented these variants of the new collective using the GASPI API, and ran experiments on the MareNostrum4 supercomputer, and evaluated under two implementations of the Matrix Factorization algorithm, one being our own, and the other a real-world implementation. In these experiments we focused on different properties of the solutions, namely: execution time of the collective; average age of contributions used

in reductions; time waiting for fresh contributions; and finally, the impact of slack on all the previous properties. Using our proposed collective we were able to reduce the execution time by up to 2.5x when compared to MPI's allreduce, while having minimal impact on the convergence rate of the algorithms tested.

Despite the interesting and promising results, there are many ideas for future work. For instance, we are interested in knowing the performance of our proposed collectives in other real world applications such as in Deep Learning. A second idea is to further optimize our implementation of the `Sharded` solution. From our experiments, we saw that the "Threaded" variants of `Sharded` were able to outperform MPI's allreduce, even in a setting using slack 0. However, our implementation had some unnecessary computation to support SSP that could be removed, allowing the collective to reach even lower execution times. Another possible improvement is to take advantage of threads during the processing of shards at the worker. This happens both at the start of the call to the collective, when computing the updated state for missed contributions, and also when the worker is computing the final reduction result. All of these computations happen on a single shard at a time, and they can all be done in parallel.

Another route would be to adapt other existing synchronous allreduce collectives such as the segmented-ring to work in the SSP model. Our solution uses an all-to-all communication and, as we increase the number of nodes, the strain in the network increases. The segmented-ring however was proved to be bandwidth optimal. For a very large number of nodes, this adaptation could perform better than the best solution we presented.

Another possible future work avenue is applying the SSP synchronization model to collectives other than the allreduce. Some interesting collectives worth considering are: reduce, broadcast and allgather.

REFERENCES

[1] Tom Vander Aa, Imen Chakroun, and Tom Haber. 2017. Distributed Bayesian Probabilistic Matrix Factorization. In *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*. 1030–1039.

[2] Muhammed Abdullah Al Ahad, Christian Simmendinger, Roman Iakymchuk, Erwin Laure, and Stefano Markidis. 2018. Efficient Algorithms for Collective Operations with Notified Communication in Shared Windows. In *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI, PAW-ATM@SC 2018, Dallas, TX, USA, November 16, 2018*. 1–10.

[3] Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 571–582.

[4] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. 37–48.

[5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. 1232–1240.

[6] Vanessa End, Ramin Yahyapour, Christian Simmendinger, and Thomas Alrutz. 2015. Adaption of the n-way Dissemination Algorithm for GASPI Split-Phase Allreduce.

[7] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 1223–1231.

[8] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. 2017. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. 629–647.

[9] Janis Keuper and Franz-Josef Pfreundt. 2015. Asynchronous parallel stochastic gradient descent: a numeric core for scalable distributed machine learning algorithms. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC 2015, Austin, Texas, USA, November 15, 2015*. 1:1–1:11.

[10] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. 2011. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. *CoRR* (2011).

[11] Cyprien Noel and Simon Osindero. 2014. Dogwild! – Distributed Hogwild for CPU GPU.

[12] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.* 69, 2 (2009), 117–124.

[13] Rolf Rabenseifner. 2004. Optimization of Collective Reduction Operations. In *Computational Science - ICCS 2004, 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part I*. 1–9. https://doi.org/10.1007/978-3-540-24685-5_1

[14] Christian Simmendinger, Mirko Rahn, and Daniel Gruenewald. 2015. The GASPI API: A Failure Tolerant PGAS API for Asynchronous Dataflow on Heterogeneous Architectures. In *Sustained Simulation Performance 2014*, Michael M. Resch, Wolfgang Bez, Erich Focht, Hiroaki Kobayashi, and Nisarg Patel (Eds.). Springer International Publishing, Cham, 17–32.