Solving the straggler problem with bounded staleness

James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee,
Gregory R. Ganger, Garth Gibson, Kimberly Keeton*, Eric Xing
*Carnegie Mellon University, *HP Labs*

Abstract. Many important applications fall into the broad class of *iterative convergent* algorithms. Parallel implementations of these algorithms are naturally expressed using the Bulk Synchronous Parallel (BSP) model of computation. However, implementations using BSP are plagued by the straggler problem, where every transient slowdown of any given thread can delay all other threads. This paper presents the *Stale Synchronous Parallel* (SSP) model as a generalization of BSP that preserves many of its advantages, while avoiding the straggler problem. Algorithms using SSP can execute efficiently, even with significant delays in some threads, addressing the oft-faced straggler problem.

1 Introduction

Machine learning algorithms are an important part of many applications, including document classification, movie recommendations, bioinformatics, and more (Table 1). For instance, collaborative filtering algorithms are used to recommend movies, songs, and other products to users based on their previous taste, purchases, and browsing history. Sparse regression models are applied to genomes to identify the genes most likely to be responsible for certain traits (e.g., Alzheimer's).

As these applications become more ubiquitous, increasingly complex algorithms are being deployed on larger data sets, leading to performance problems. A state-of-the-art document topic modeling algorithm may take many hours to analyze a large corpus. For instance, running Latent Dirichlet Allocation [10] (LDA) over a corpus of 300,000 documents [2] takes about 10 days [17].

To reduce computational time, application and algorithm designers are turning to parallel and distributed implementations running on clusters of servers [17]. While the diversity of these algorithms and applications makes it difficult to create a general-purpose method of parallelizing them, many of them share some important traits. This paper focuses on *iterative convergent algorithms*, the class of algorithms that start with some guess as to the problem solution and proceed through multiple iterations that each improve this guess. The key property that makes this approach work is convergence, which allows such algorithms to find a good solution given any starting state.

Algorithm	**Example applications**
Latent Dirichlet Allocation (LDA)	News classification
Low-rank matrix factorization	Movie/music recommendations
Sparse regression	Genome-wide analysis
Conjugate gradient	Linear system solvers
Principal eigenvector	Web search/page rank
All-pairs shortest path	Mapping and route planning

Table 1: Examples of iterative convergent algorithms, and some of their applications.

Distributed implementations of iterative convergent algorithms tend to follow the Bulk Synchronous Parallel (BSP) computational model. In this model, the application operates on a snapshot of the data produced by the previous iteration, requiring all threads to execute the same iteration at the same time. This per-iteration barrier synchronization causes a straggler problem, which can significantly reduce the performance of these algorithms.

Succinctly, a straggler problem occurs when a small number of threads (the stragglers) take longer than the others to execute a given iteration. Because all threads must be synchronized, all threads will proceed at the speed of the slowest thread in each iteration. This problem grows with increased parallelism: as the number of servers increases, the probability of having a straggler in any given iteration also increases.

Existing systems address straggler problems in a number of ways. For consistent stragglers (e.g., less capable nodes), proper load distribution (e.g., via work stealing) or speculative execution (e.g., [22]) suffices. Speed variation due to transient effects, such as intermittent "background" activities or contention for shared resources, is much more difficult. Some systems, typical in High-Performance Computing, avoid using any hardware or software components that may introduce "jitter". Other systems restrict the communication patterns and interdependence between threads. Yet other systems allow the threads to run asynchronously, avoiding stragglers, but potentially complicating the algorithm.

We propose a middle ground between full synchronization and no synchronization: allowing some threads to proceed ahead of others, by a certain amount. The Stale Synchronous Parallel model (SSP) relaxes consistency and freshness guarantees without completely eliminating them. In many cases, an SSP-based system can behave and perform like a best-effort system. However, it will detect when data becomes *too* unsynchronized, and will partially synchronize threads to avoid unbounded data staleness. Initial experiments with a prototype system, called LazyTables, show promise that SSP can mitigate transient straggler effects.

2 Background

This section describes iterative convergent algorithms, the types of applications that use them, and current models for running these algorithms in parallel.

2.1 Iterative convergent algorithms

Iterative convergent algorithms typically search a space of potential solutions (e.g. N-dimensional vectors of real numbers) using an *objective function* that evaluates the goodness of a potential solution. The goal is to find a solution with a large (or in the case of minimization, small) objective value. For some algorithms (e.g., eigenvector and shortest path), the objective function is not explicitly defined or evaluated. Rather, they continue to iterate until the solution does not change (significantly) from iteration to iteration.

These algorithms start with an initial state S_0 with some objective value $f(S_0)$. They proceed through a set of iterations, each one producing a new state S_{n+1} with a potentially improved solution (e.g. greater objective value $f(S_{n+1}) > f(S_n)$). Eventually they reach a stopping condition and output the best known state.

A key property of these algorithms is that they will converge to a good state, even if there are minor errors in their intermediate calculations.

2.2 Bulk synchronous parallel

These algorithms are often parallelized with the Bulk Synchronous Parallel model (BSP). As in the sequential version of the algorithm, BSP applications proceed through a series of iterations. In BSP the algorithm state is stored in a shared data structure (often distributed among the threads) that all threads update during each iteration.

A single iteration of BSP consists of three steps. In the **computation** phase, all threads compute on the previous iteration's output in parallel. In the **communication** phase, threads produce new output, sharing it either by explicit communication, or by writing to a shared data structure. Lastly, in the **synchronization** phase, threads execute a barrier to ensure that they don't begin the next computation step until all other threads have finished the communication step.

BSP provides a simple and easy-to-reason-about model for parallel computation, and can be easily applied to most iterative convergent algorithms.

2.3 Stragglers in BSP

A well-known problem with BSP is the *straggler problem*: because of the frequent and explicit synchronization, each iteration proceeds at the pace of the slowest thread. This problem only gets worse as the level of parallelism is increased. Because of random variations in execution time, as the number of threads increases, the probability that one of them will run unusually slowly in a given iteration increases. As a result, the entire application will be delayed in every iteration.

Stragglers can occur for a number of reasons including heterogeneity of hardware [14], hardware failures [7], imbalanced data distribution among tasks, garbage collection in high-level languages, and even operating system effects [4, 18]. Additionally, there are sometimes algorithmic reasons to introduce a straggler. Many algorithms use an expensive computation as a stopping criterion. Even if this computation is only run on a single thread, other threads will have to wait for it to finish before they can start the next iteration.

2.4 Existing solutions

The High Performance Computing community – which frequently runs applications using the BSP model – has made much progress in eliminating stragglers caused by hardware or operating system effects [12, 13, 18, 23]. While these solutions are very effective at reducing "operating system jitter", they are not intended to solve the more general straggler problem. For instance, they are not applicable to programs written in garbage collected languages, nor do they handle algorithms that inherently cause stragglers during some iterations.

In large-scale networked systems, where variable node performance, unpredictable communication latencies and failures are the norm, researchers have explored relaxing traditional barrier synchronization. For example, Albrecht, et al., describe partial barriers, which allow a fraction of nodes to pass through a barrier by adapting the rate of entry and release from the barrier [6].

Another class of solutions attempts to reduce the need for synchronization by restricting the structure of the communication patterns. GraphLab [15, 16] programs structure the computation as a graph, where data can exist on nodes and edges. All communication occurs along the edges of this graph. If two nodes on the graph are sufficiently far apart they may be updated without synchronization. This model can significantly reduce synchronization in some cases. However, it requires the application programmer to specify the communication pattern explicitly.

Considerable work has been done in describing and enforcing relaxed consistency in distributed replicated services. For example, the TACT model captures continuous consistency using three metrics: numerical error, order error and staleness [21]. Replicas locally buffer updates (each with an optional weight) before requiring remote communication. Numerical error is the maximum weight of writes not seen by a replica. Order error is the maximum weight of not-yet-communicated local writes. Staleness is the maximum time before a replica sees a write accepted by a remote replica. Although the context differs (LazyTables relaxes consistency for iterative parallel computations to speed convergence rather than for replicated data stores), the consistency models have some similarities.

Lastly, it is possible to ignore consistency and synchronization altogether, and rely on a best-effort model for updating shared data. Yahoo! LDA [5] as well as most solutions based around NoSQL databases rely on this model. While this approach can work well in some cases, it may require careful design to ensure that the algorithm is operating correctly.

3 Stale Synchronous Parallel

To address the straggler problem without giving up the benefits of synchronization, we propose a new computational model based on BSP, which we call Stale Synchronous Parallel (SSP). Like BSP, SSP assumes that the program consists of a number of threads, each proceeding through the same number of iterations. During each iteration each thread reads and updates some shared state.

However, programs using the SSP model must be aware of some crucial differences in the consistency model that can affect algorithm design as well as performance. These differences can be described in terms of the following properties [20].

Bounded staleness Data that is read by a thread may be *stale* (missing some recent updates). In other words, there is a delay between when an update operation completes, and when the effects of that update are visible. An application can put an upper bound on how stale the result of each `read()` operation may be. In SSP, this bound is expressed in terms of the number of elapsed iterations. This property is analogous to TACT's numerical error [21].

Read-my-writes If a thread updates a value, all subsequent `read()` operations by that thread will see the update (unless it is overwritten by a later update). In other words, threads see their own updates *immediately*, even if updates from other threads may be delayed (staleness).

Soft synchronization At the end of each iteration, threads execute a "soft barrier". Unlike a full barrier (as in BSP), which blocks until all threads are caught up, a soft barrier blocks the thread until all threads are within a specified range of the current iteration. For instance, a soft barrier with a parameter of "1" finishes when no thread is more than 1 iteration behind the calling thread.

The Stale Synchronous Parallel computational model can be thought of as BSP with the addition of bounded staleness, read-my-writes, and soft synchronization. It is important to note that reads in SSP are not historical queries: the system *may* return fresher data than the specified bound. In fact, it may return fresher updates from some threads, and stale updates from others. Specifically, the system *must* incorporate all updates from the current thread to implement read-my-writes.

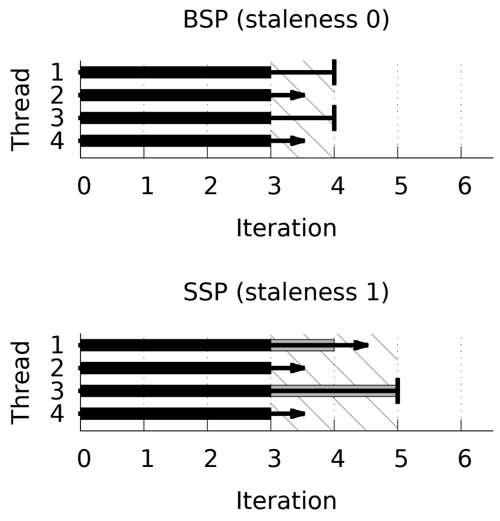


Figure 1: Diagram of Bulk Synchronous Parallel and Stale Synchronous Parallel execution state. The thick black bars indicate data that is visible to all threads. The gray bars indicate data that may be visible, but is not guaranteed. The lines indicate thread progress: arrows are runnable threads, while blocked threads are terminated with vertical lines.

Figure 1 illustrates these freshness properties. The BSP diagram represents an application with 4 threads using the BSP model. In this diagram, threads 2 and 4 are executing in iteration 3, while threads 1 and 3 are blocked waiting for them to finish. When these threads read data, they are guaranteed to see all updates up to the end of iteration 2.

The SSP diagram shows the same application, but using the SSP model with a fixed staleness of 1. In this diagram, threads 2 and 4 are still executing in iteration 3. However, because they are willing to use stale data, threads 1 and 3 did not have to wait for the other threads to complete that iteration. Thread 1 is currently executing in iteration 4. Thread 3 is blocked at the start of iteration 5 because it requires data from iteration 3 to continue.

3.1 LazyTables Prototype

We built a prototype system called LazyTables that implements the Stale Synchronous Parallel model to support distributed machine learning applications. LazyTables provides the abstraction of a set of shared sparse matrices that all processes access as a "parameter server" to store intermediate results.These matrices are stored in memory, distributed across a set of servers.

LazyTables provides an API similar to the Piccolo [19] system. It provides read and update operations including `get()`, `get_row()`, `put()` and `increment()`. While Piccolo provides support for a generic `update()` operation, LazyTables currently supports only `increment()`, which is sufficient for our test cases. Generic updates are necessary for some other algorithms such as all-pairs shortest-path, which uses "min" as the update operator.

4 Experiments with masking stragglers

The main goal of our initial experiments is to demonstrate that the Stale Synchronous Parallel model can mask the effects of stragglers on performance. Additionally, we show example algorithms that are "staleness tolerant", and exploiting staleness can improve their convergence behavior. However, a detailed examination of these latter points is left as future work.

Our LazyTables prototype is written in C++, using ZeroMQ [3] for asynchronous communication. Data, on both the servers and the client caches, is stored in RAM using the C++ standard template library. All experiments are conducted on virtual machines in the CMU OpenCirrus [9] cluster. These VMs are configured with 8 cores and 15GB of RAM. No other applications or VMs are running on the hosts concurrently with the experiments.

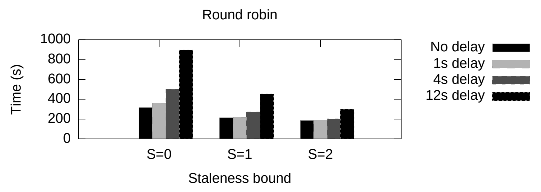
4.1 Stragglers

To demonstrate the effect of stragglers, we conducted two experiments that introduced delays into LazyTables. The first experiment simulates the effects of an "algorithmic" delay, where an expensive calculation (e.g. stopping conditions) must be done regularly. In every iteration we chose one thread to perform the "calculation" and forced it to sleep. The choice was made in a round-robin fashion so that the sleeps would be balanced between the threads over the course of the execution.

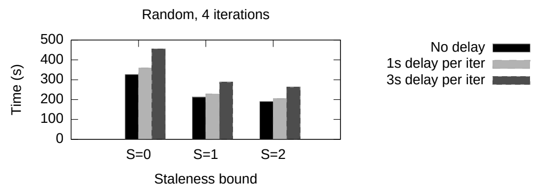
The results of this experiment are shown in the top of Figure 2(a). Observe that with a staleness bound of 0 (first group of bars), even a small delay causes significant overall delay in the application. However, as more staleness is introduced, the system is able to mask ever larger delays.

The second experiment simulates the effects of external interference due to things like network contention, transient hardware errors, operating system activity, and garbage collection. To simulate these effects, we inserted random sleeps into some iterations of the program. When a thread is in a normal state, it may choose, with probability $1/8$ to enter a delayed state for 4 iterations. When in a delayed state, at the end of every iteration it sleeps for some number of seconds.

Figure 2(b) shows the results of this experiment. Observe that with increased staleness, the effects the delay can be substantially reduced.



(a) Round-robin, single-iteration delay



(b) Randomized, 4-iteration delay

Figure 2: Time to execute 50 iterations of LDA using the aan_short data set.

Figure 3 shows a "swimlane diagram" of the first 100s of the LDA application for four different configurations. (a) shows a synchronous execution (staleness 0) with no delay. The vertical alignment of the bars is caused by threads executing the same iteration at the same time. (b) represents a synchronous execution with a 4s delay round-robined between the threads. Like in the previous diagram, all threads begin executing an iteration at the same time. However, one thread is delayed in each iteration, visible by the diagonal (upper-left to lower-right) pattern of elongated bars. Other threads must wait for the delayed thread to finish before they can start their next iteration. As a result, fewer iterations are completed in the 100s window shown, as seen in the smaller number of stripes.

The bottom two diagrams show the asynchronous case (staleness 1). (c) is the case with no delay. Unlike in the synchronous case, threads are not waiting for one another to finish before starting their next iteration. This is visible in the raggedness of the vertical lines in the diagram. Even without the artificial staleness, the reduced synchronization improves iteration speed. (d) shows the asynchronous execution with a 4s delay. Here, each thread can proceed at its own pace, within the staleness bounds, significantly reducing the impact of the stragglers.

4.2 Performance and convergence

Figure 4 shows the convergence behavior of the LDA algorithm over time. These results were generated using a cluster of 32 machines with 8 cores each, processing the "20 Newsgroups" data set [1]. These results demonstrate that LDA will converge when running partially asynchronously. Furthermore, increased staleness improves convergence performance: the staleness=3 setting

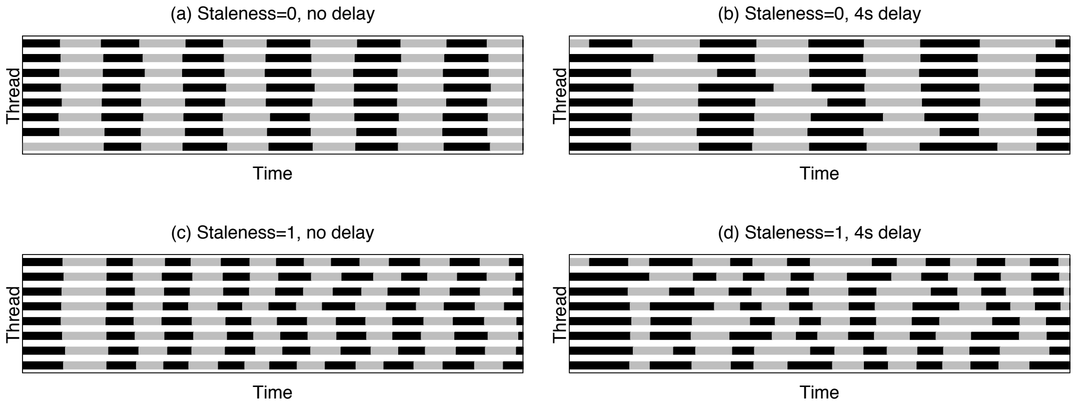


Figure 3: Swimlane diagram of the first 100s of execution. Alternating gray and black bars indicate progress from one iteration to the next. The frequency of stripes indicates iteration speed, so more stripes corresponds to faster iteration execution.

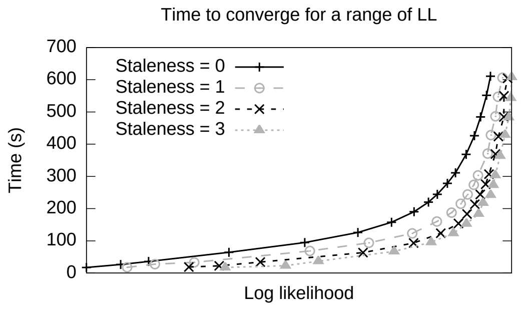


Figure 4: Time needed for LDA to converge to a particular log likelihood value. Log-likelihood measures the probability of the current solution — a higher log-likelihood means a higher probability, and thus better, solution.

often takes half the time to reach a particular log likelihood, compared to the synchronous setting. However, the bounds on staleness are important for the algorithm to converge correctly. As the staleness setting was increased past 4 the convergence behavior began to degrade (not shown in graph). We ran a similar experiment for a sparse regression algorithm (LASSO), and saw similar results to LDA (omitted due to space constraints). We are also experimenting with low-rank matrix factorization.

5 Open questions

Table 1 lists a number of algorithms that we believe can tolerate staleness in their computations. Section 4.2 provides evidence that two of these applications can, in fact, tolerate staleness, and their performance is improved as a result. However, this list is not exhaustive. A formal classification of staleness tolerant algorithms would be an important contribution. We are investigating these properties both in the formal context of a convergence proof, as well as trying to develop informal "rules of thumb" for identifying staleness-tolerant algorithms.

Another interesting question involves the definition of "staleness". Due to the nature of the algorithms we are targeting, SSP defines staleness in terms of iteration count. However, many algorithms do not proceed in strict iterations. Even among those that do, other notions of staleness may be relevant. For instance, an application may want to read a value, ensuring that the result is no more than 10% different from the most up-to-date value. Speculative execution could be used to allow threads to proceed, while repeating work when values actually did differ by more than the requested amount.

6 Conclusions

We propose Stale Synchronous Parallel as a new model for parallel computation. SSP allows applications to specify a freshness requirement when reading shared data. By exploiting the application's tolerance for staleness, a system implementing SSP can significantly reduce the effects of stragglers on execution time. Initial experiments with a parameter server prototype, called LazyTables, show that SSP can significantly improve performance and is worth further development and study.

Acknowledgements. We thank the member companies of the PDL Consortium and the CMU Big Learning research group for their interest, insights, feedback, and support. This work is supported in part by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC) and hardware donations from Intel and NetApp.

References

[1] 20 newsgroups data set. http://qwone.com/ jason/20Newsgroups/.

[2] New york times data set. http://www.ldc.upenn.edu/.

[3] ZeroMQ: the intelligent transport layer. http://www.zeromq.org/.

[4] *The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale*, 2006.

[5] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

[6] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose synchronization for large-scale networked systems. In *USENIX Annual Tech*, pages 301–314, 2006.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, pages 1–16, 2010.

[8] Apache Mahout, http://mahout.apache.org.

[9] A. I. Avetisyan, R. Campbell, I. Gupta, M. T. Heath, S. Y. Ko, G. R. Ganger, M. A. Kozuch, D. O'Hallaron, M. Kunze, T. T. Kwan, K. Lai, M. Lyons, D. S. Milojicic, H. Y. Lee, Y. C. Soh, N. K. Ming, J.-Y. Luke, and H. Namgoong. Open cirrus: A global cloud computing testbed. *Computer*, 43(4):35–43, Apr. 2010.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[11] J. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 321–328, June 2011.

[12] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, 1996.

[13] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti. The impact of system design parameters on application noise sensitivity. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 146–155, Washington, DC, USA, 2010. IEEE Computer Society.

[14] E. Krevat, J. Tucek, and G. R. Ganger. Disks are like snowflakes: no two are alike. In *Proceedings of the USENIX conference on Hot topics in operating systems*, pages 14–14, 2011.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[16] Y. Low, G. Joseph, K. Aapo, D. Bickson, C. Guestrin, and M. Hellerstein, Joseph. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[17] D. Newman, A. U. Asuncion, P. Smyth, and M. Welling. Distributed inference for latent dirichlet allocation. In *NIPS*, 2007.

[18] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 55–, 2003.

[19] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, pages 1–14, 2010.

[20] D. Terry. Replicated data consistency explained through baseball. Technical Report MSR-TR-2011-137, Microsoft Research, October 2011.

[21] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, 2002.

[22] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, pages 29–42, December 2008.

[23] R. Zajcew, P. Roy, D. L. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. An osf/1 unix for massively parallel multicomputers. In *USENIX Winter*, pages 449–468, 1993.