Progressive Load Balancing in Distributed Memory

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200033

Progressive Load Balancing in Distributed Memory

Mitigating Performance and Progress Variability in Iterative Asynchronous Algorithms

Justs ZARINS [a,1], Michèle WEILAND [b]
[a] *School of Informatics, University of Edinburgh, UK*
[b] *EPCC, University of Edinburgh, UK*

Abstract. System performance variability is a significant challenge to scalability of tightly-coupled iterative applications. Asynchronous variants perform better, but an imbalance in progress can result in slower convergence or even failure to converge, as old data is used for updates. In shared memory, this can be countered using progressive load balancing (PLB). We present a distributed memory extension to PLB (DPLB) by running PLB on nodes and adding a balancing layer between nodes. We demonstrate that this method is able to mitigate system performance variation by reducing global progress imbalance 1.08x–4.05x and time to solution variability 1.11x–2.89x. In addition, the method scales without significant overhead to 100 nodes.

Keywords. asynchronous algorithm, load balancing, performance variability, iterative algorithm, system noise

1. Introduction

With the ever increasing scale of high performance computing systems, there comes an array of new challenges. Technical, architectural and economic hurdles need to be overcome in order to build and deploy an exascale machine. However, creating the right hardware is only half the answer; software that can run on it efficiently is an essential part.

Any algorithm or application aiming to run on millions of parallel threads, which may be running at varying speeds, must be able to cope with performance variability. In such a scenario, tightly synchronising algorithms are not a suitable choice. Instead, we consider asynchronous or "chaotic" algorithms [1] which can make progress with stale data if some other thread has stalled. This allows for greater flexibility in adjusting to performance variability. Nevertheless this does not mean variability can be ignored completely. It still affects time required to reach the solution, but in a more complex manner than with synchronous algorithms. Therefore it is interesting to consider load balancing in the context of asynchronous algorithms.

[1]Corresponding Author: E-mail: j.zarins@ed.ac.uk

Note that in this paper we are *not* considering task based parallelism, where asynchrony refers to replacing global synchronisation with point-to-point synchronisation to satisfy data dependencies.

A recent approach [2] called *progressive load balancing* (PLB) was introduced to address the asynchronous context. The method was shown to be able to effectively mitigate the effect of a slow core in a shared memory environment. PLB achieves this by periodically moving work between CPU cores, not in order to equalise iteration rates, but to bound progress imbalance; it is balancing load over time, not instantaneously. In the present paper we build upon PLB and extend it to the distributed memory setting. Specifically, this paper makes the following contributions:

- A description of an implementation strategy extending PLB to distributed memory.
- An evaluation of the application of distributed PLB (DPLB) to balance the load in an iterative asynchronous algorithm.

2. Background

As we move towards exascale computing, synchronisation in applications is an increasingly important issue. Performance of individual cores, sockets and entire nodes that, on paper are identical, is in fact variable. This is due to factors such as energy efficiency and temperature management [3–5], random OS noise [6], and network latency variation [7]. Increasing the number of components that an application is run on also increases the likelihood that performance variation will be encountered; this is an increasing issue when considering exascale [8]. Synchronisation in applications within this hardware context results in large loss of efficiency, even running at the rate of the slowest component. While static load balancing can help, ultimately it is of limited use because the machine's performance can change at runtime.

Given the efficiency limitations of large scale synchronisation, it is natural to consider algorithms that do not rely on strict synchronous execution. In asynchronous algorithms, data synchronisation points are removed to allow the use of stale data if other workers in the system have not made progress due to stalls. There exist various iterative convergent algorithms where this is possible [1,9–12]. Performance in asynchronous algorithms is generally dictated by a tradeoff between iteration rate and convergence rate. The former is usually improved by going asynchronous, but the latter may suffer because of the use of stale values, at the extreme resulting in failure to converge [13]. It follows that system performance variability is still a concern because it results in "progress variability".

The issue of progress variability in asynchronous algorithms has previously been tackled using a load balancing approach called progressive load balancing [2]. It is framed as a load balancing method specific to asynchronous algorithms since they do not require balance to be instantaneous. Instead, balancing is done over time by effectively swapping iteration rates of different problem subdomains. In other words, balance is in a state of dynamic (not static) equilibrium; update rates of problem subdomains keep changing but the difference in number of updates between subdomains is bounded. This leads to a dynamically controlled level of asynchrony in the system, which was shown to be effective at dealing with performance variation in shared memory.

In more detail, the global problem domain (e.g. a 2D grid for solving Jacobi's algorithm in 2 dimensions) can be split into more subdomains than there are threads. Then each thread will have multiple subdomains to continually update. The update rate per subdomain is inversely proportional to the number of subdomains that a thread owns. Thus the update rate of a particular subdomain can be increased or decreased by removing from or adding to the owning thread's workload respectively. PLB uses this mechanism to periodically move subdomain ownership between threads in order to limit update staleness without wasting CPU time spent on waiting for stragglers. In the resulting pattern some subdomains get updated quicker for a time, using stale values, but later the update rates are changed so that the subdomain that had raced ahead begins to iterate slower and eventually falls behind at which point the process is reversed again. In this paper we will later see how the scheme can be extended to distributed memory with a separate layer to move subdomains between nodes and the PLB mechanism to integrate the subdomains within nodes.

2.1. Related Work

The area of load balancing is an active field of research, however the majority of techniques are developed for, and applied to, synchronous algorithms and so may not transition well to asynchronous algorithms or require significant changes to the techniques. For example, work stealing is a popular and scalable method [14]. Workers process local queues of tasks and when they run out, more work is stolen from work queues of other workers. In this form it cannot be applied to asynchronous algorithms because workers in principle never run out of work and always appear busy, so they would just use continuously more stale values. Work stealing can be applied to semi-synchronous algorithms, where a maximum staleness bound is enforced so the amount of work available per worker does have a limit. However, our experiments showed that the method does not work well in the semi-synchronous case because the system soon reaches a state where there are many starved workers and not much work to steal. Hence we are focusing on techniques that have been shown to be applicable to asynchronous algorithms, which is a key criterion for us.

A few examples of load balancing of asynchronous algorithms in distributed memory exist. For instance, Bahi et al. show a load balancing algorithm applied to a 1D stencil application [15]. This algorithm sends parts of the working array from one worker to a less loaded neighbour. The algorithm is similar to PLB, however it seeks a static load balance while PLB aims to create dynamic equilibrium, which can result in good balance with coarser work adjustments. Additionally, the proposed algorithm is presented in 1 dimension only; an extension to multiple dimensions would be difficult to design and implement.

A more passive balancing approach has been applied to large scale deep learning [10]. The application uses asynchronous stochastic gradient descent as the core algorithm. Groups of synchronous workers are linked together asynchronously to update parameter servers for the model under training. This hybrid asynchronous-synchronous scheme balances statistical and hardware efficiency by tweaking the sizes and the number of synchronous groups, but cannot deal with performance variation changes at runtime.

Another strategy is to ignore performance variance itself and rather deal with the resulting staleness. There are various examples of such algorithmic corrections for stale values applied to asynchronous stochastic gradient descent [16,17]. A limitation of these approaches is a lack of generalisation to other applications.

Our test problem in this work is asynchronous Jacobi's algorithm (see Sec. 4). This problem has been previously examined by Bethune et al. at large scale [18]. They observed large variations in numbers of iterations completed by different processes. This resulted in a significant increase in the number of iterations taken to converge. They also document a case where a single core running at half speed doubled the runtime of a 32k core synchronous run.

3. Extending PLB to Distributed Memory

While PLB was shown to be successful in a shared memory setting, for it to be truly valuable it needs to be able to scale further. In this paper we extend the method and evaluate its effectiveness in a distributed memory setting.

3.1. DPLB

To extend PLB we add a layer that moves work between nodes. This method is referred to as *distributed progressive load balancing* (DPLB). In DPLB we run PLB on each node, and add infrequent work movements across nodes. This extension is important for situations where whole nodes are affected by noise and are significantly slower than others.

The main steps in the algorithm are as follows:

1. Periodically, with a set frequency, nodes find out the average number of updates performed on other nodes.
2. The difference between the highest and lowest averages are compared to a set threshold.
3. If the difference is larger than the threshold, the least progressed node sends a randomly chosen problem subdomain to the node that has advanced the most.
4. The node that has received the subdomain assigns it one of its cores initially, but, since PLB is running on every node, the subdomain gets moved between cores as is required to balance progress on the node.

The implementation details of these steps will vary based on the problem that is being solved and the programming techniques and libraries used, but we will next explain some of the most important implementation considerations for our example case.

3.2. Implementation

In our implementation we target iterative convergent algorithms that can be parallelised by splitting the global problem domain into smaller subdomains. We also assume that data is being exchanged between the subdomains using "halos". Stencil applications match this pattern closely, however the balancing principles presented here are not limited to this class of applications.

Distributed communications are implemented mainly using MPI single sided calls. This communication paradigm is well suited to asynchronous algorithms, since it min-

Table 1. "Cirrus" test system details (www.cirrus.ac.uk)

System type	SGI ICE XA	Topology	Hypercube, 282 nodes
CPU Sockets	2	L3 cache	45 MB
CPU	Intel E5-2695	RAM per CPU	128 GB
Core count per CPU	18	Compiler	GCC 6.2
Clock	2.1 GHz	MPI library	Intel 17.0
Interconnect	FDR Infiniband	Main compilation flags	-O2

imises the need for global synchronisation. Also, the application can be more dynamic because there is no need to match specific sends and receives. Some two sided communication still exists, but only where the matching does not interfere with asynchrony. On node we use OpenMP threading.

Information gathering about work progress of nodes is done using a reduction implemented using RMA operations. Every node publishes a small data structure containing the average progress of its problem subdomains. Other nodes can query these structures with a get operation when global balance is being checked.

An important part of the implementation is moving subdomains between nodes dynamically and adjusting communication targets. To ensure scalability, it is important to avoid introducing a global bottleneck here, for example by using a centralised table of subdomain physical locations. Instead, in our implementation subdomains keep track of just their neighbours' locations. When a subdomain moves, it leaves behind a message with its new host rank (i.e. MPI rank). When its neighbours perform halo exchange, as part of the halo they also receive the message that the subdomain has moved and which is the new rank that should be queried for the desired halos.

The main component facilitating this interaction is metadata appended to halos, specifically an ID and owner rank. Upon retrieval of a halo, the metadata is checked to make sure it is as expected (initial locations of subdomains are known). If the metadata rank is not the same as the rank the halo was received from, the halo and associated subdomain have moved (the rank that does the moving changes the halo metadata to reflect the rank to which it has migrated). Once the new rank is known, an array of halo displacements is retrieved from the target rank. The array is searched to find the physical memory location of the target halo. The halo can now be retrieved and the ID checked to make sure they are correct. Only the communicating neighbours were involved in this transaction, which makes it scalable.

4. Experiments

As our test application we use Jacobi's algorithm applied to the diffusion problem in 2 dimensions. This is an iterative convergent stencil application which is often used when testing asynchronous algorithms due to its simplicity and numerical stability [1]. We use a Gaussian shaped function as the boundary condition along one edge, the others being set to zero. The problem domain is distributed across nodes in 1 dimension, along the x axis.

We used the HPC system Cirrus for our experiments. Hardware and compiler details are listed in Table 1.

While there is inherent noise and imbalance in the system, for some experiments we inject artificial noise to simulate particular scenarios in order to have repeatable experiments. Noise is generated by running an additional background thread that sleeps and busy-waits for set amounts of time. Additionally, the workers' *niceness* is set to a high value so that they have lower priority, thus yielding to the noise generating threads when active.

We chose a noise level of 40% per CPU socket (i.e. the CPU effectively runs at 60% of its normal clock frequency). This value is the mean of worst case clock frequency variations due to manufacturing variability observed in [4] when limiting node power - a factor to consider in future exascale systems with global power constraints. Also, Chunduri et al. report application runtime variability between 1.18x and 1.74x (38% on average) related to network congestion on a production system [19]. For experiments where we slow down a whole node, we chose the same level to make comparisons between experiments more direct. This can happen if both sockets are slow, the node is hot from a previous job or if there is significant network congestion.

Each experiment was repeated multiple times on different sets of nodes. Where possible, a series of experiments with differing settings (e.g. normal, normal plus balancer, normal plus balancer plus noise etc.) were repeated on the same node set so that differences between the experiments would be mainly due to algorithmic differences, instead of node conditions.

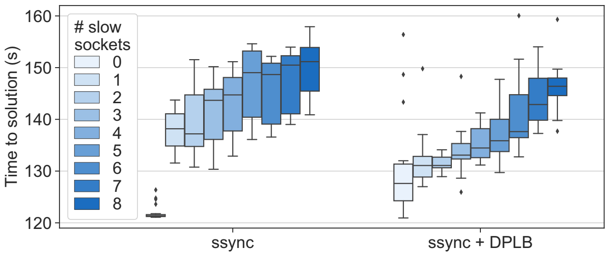
In *time to solution* (TTS) experiments the application runs until the global l_2-norm of the residual, normalised by its initial value, reaches a threshold. We set this threshold at 10^{-3}. Generally the threshold is smaller in real applications, however here we wanted to limit the total execution time and focus on performance metrics rather than the final solution.

We use a problem size of 1000 by 1000 values per core. Thus, in the 15 node experiments the global problem size is $6k$ by $90k$ and in the 100 node experiments it is $6k$ by $600k$.

5. Evaluations

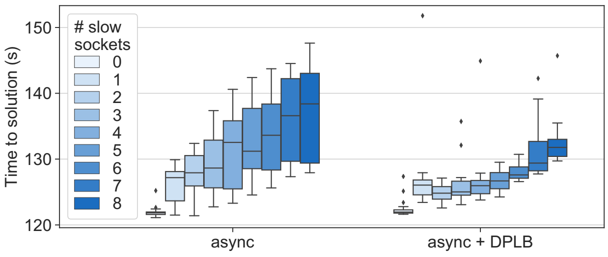
In this section we present an experimental evaluation of DPLB acting on semi-synchronous and asynchronous Jacobi. Figures 1 to 3 show time to solution results comparing performance before and after applying DPLB (each bar represents 9 to 20 data points). Less time and smaller variance is better. These figures include tables of iteration rates (in units of 1000 iterations per second per node) and staleness (most stale halo encountered). The specific settings of PLB parameters were chosen to be the same as in [2] because these were found to give good performance ($nPairs = 6$, $lowThresh = 2$, $highThresh = 6$). DPLB performs balancing across nodes every 0.5 seconds with PLB balancing each CPU socket separately every 0.001 seconds. The semi-synchronous staleness bound is set to 30 and in both the semi-synchronous and the asynchronous experiments each core initially holds 4 problem subdomains.

To test the method we run Jacobi on 15 nodes while growing the number of nodes with a slow CPU socket. In order to survey the range of possible noise scenarios, a portion of the experiments has noise placed randomly and at fixed locations. For the latter we picked "worst case" and "best case" noise placement, based on the problem that is being



	sync(4)		ssync(4) + DPLB	
# slow	stale	iter rate	stale	iter rate
0	31	10.80	34	10.75
1	31	9.86	39	10.44
2	31	9.34	39	10.38
3	31	9.25	39	10.30
4	31	9.20	40	10.18
5	31	8.97	40	10.01
6	31	9.02	39	9.96
7	31	8.90	40	9.70
8	31	8.76	39	9.49

Figure 1. Effect of DPLB on semi-synchronous Jacobi running on 15 nodes. Color indicates the number of CPU sockets running 40% slower. The table shows median values.



	async(4)		async(4) + DPLB	
# slow	stale	iter rate	stale	iter rate
0	1860	10.85	175	10.86
1	3122	10.75	2587	10.78
2	3339	10.65	2070	10.75
3	3493	10.55	2732	10.69
4	3628	10.45	2507	10.62
5	3593	10.35	2725	10.53
6	3657	10.26	2876	10.45
7	3723	10.16	3455	10.33
8	3731	10.06	3152	10.22

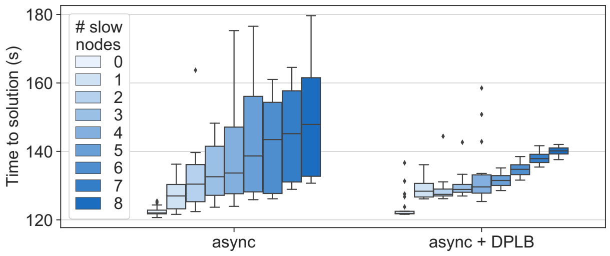
Figure 2. Effect of DPLB on asynchronous Jacobi running on 15 nodes. Color indicates the number of CPU sockets running 40% slower. The table shows median values.

solved. The initial conditions put a Gaussian shaped source in the middle of the problem domain, so updates in the middle contribute more towards reducing the residual than the edges. Thus we add noise to components that are initially responsible for the middle of the problem domain to get worst case performance and add noise to edges to get best case performance.

Figure 1 shows results for the semi-synchronous version. Without balancing, the time to solution gradually increases; we also observed instances of 200%–260% slow-down when 6, 7 or 8 sockets were noisy. DPLB mitigates the noise noticeably for all noise counts, and avoids the large outliers at higher noisy socket counts. Since progress imbalance is capped, the performance difference comes from DPLB sustaining a higher iteration rate. The balanced version's median TTS is reduced by 3–10%, except for the noiseless case where the unbalanced version is 5% faster on average. We note that the current implementation allows the staleness bound to be overstepped slightly due to sub-domain updates occurring while some subdomains are being transferred between nodes.

Results for the asynchronous version can be seen in Figure 2. In the table it can be seen that iteration rate is not affected adversely by DPLB and halo staleness is reduced 1.08x–1.61x. As a result, the balanced asynchronous version converges quicker for every noise setting, with a median reduction of up to 6%. TTS of the balanced version is larger than that of the noiseless case, but this is to be expected even with perfect balancing since slow components take away the total amount of available compute power in the system. Furthermore, the worst case TTS grows at a higher rate without DPLB, which implies reduced scalability. With DPLB the worst case TTS remains mostly flat until noise is added to 5 or more sockets.

Because the asynchronous version shows better performance than the semi-synchronous version overall, we test it further by slowing down whole nodes, not just



	async(4)		async(4) + DPLB	
# slow	stale	iter rate	stale	iter rate
0	1880	10.85	181	10.86
1	3794	10.65	3051	10.64
2	4755	10.46	1272	10.53
3	4617	10.26	1257	10.45
4	5377	10.07	1782	10.29
5	5591	9.87	1680	10.13
6	5575	9.68	1559	9.90
7	5581	9.47	1377	9.67
8	5709	9.33	1707	9.47

Figure 3. Effect of DPLB on asynchronous Jacobi running on 15 nodes. Color indicates the number of nodes running 40% slower. The table shows median values.

individual CPU sockets. This can occur if a job is assigned a hot node or if there is a lot of network communication from other jobs going through the node's links. These results can be seen in Figure 3. The overall patterns are similar to the previous case, but more pronounced. Balancing reduces median TTS by up to 6% again, but the reduction in worst case TTS and staleness is significantly higher at 1.24x–4.05x.

An important feature to emphasise is the excellent reduction in performance variability due to DPLB. Table 2 shows, for each noisy component count, the ratio of the unbalanced version's spread of TTS (distance between the boxplots' whiskers) against that of the balanced version. The last column of the table shows this ratio applied to the spread of TTS across all counts of noisy components, i.e. between the highest top whisker and lowest bottom whisker in each category. The change for the semi-synchronous code varies between 0.06x (the balanced version is more variable) and 4.00x (the balanced version is less variable). However, for the asynchronous code, balancing always reduces variance, ranging from 1.14x and 11.07x. If the number of noisy components is not set at any particular value, the balanced versions range from 1.11x to 2.89x less variant. This increased consistency in runtime is crucial for time sensitive applications, e.g. predicting the path of a hurricane using weather simulation. It is also important in cases such as application scheduling on shared compute resources, benchmarking and keeping within budget of HPC resources.

As a final test, we ran our code on 100 nodes (3600 cores) with highly variable noise settings from run to run in order to simulate a real life scenario. For each individual run we selected a random set of nodes to be noisy; the size of the set was also chosen randomly between 0 and 15. The level of slowdown on each node in the set was chosen randomly between 15% and 40%. We obtained 42 data points with the asynchronous Jacobi code and another 42 with asynchronous Jacobi plus DPLB. The results can be seen in Table 3. Both versions performed very similarly. The test problem, when increased in size, proved to be highly resilient to random noise so adding load balancing in this case did not reduce time further. However, other inputs can be more sensitive to noise and this experiment shows that DPLB has no significant overhead in this setting and it scales perfectly well.

On the whole, the results of the asynchronous algorithm with DPLB show greatly reduced variance in TTS and variability in update progress of problem subdomains. In addition, the worst case noise scenario TTS is less when DPLB is added while the best case noise scenario is slightly higher. These observations taken together indicate that smoothing noise is beneficial in the majority of the time. While reducing progress imbalance occurring in a less critical part of the problem domain results in a small increase in TTS,

Table 2. Runtime variability ratios

# slow	ssync, socket	async, socket	async, node
0	0.06	1.14	8.12
1	1.21	1.86	1.47
2	4.00	2.42	3.50
3	2.19	3.55	3.88
4	1.81	4.24	6.26
5	1.03	3.39	7.65
6	0.83	4.37	5.06
7	0.89	1.51	5.71
8	1.71	3.41	11.07
extremes	1.11	1.51	2.89

Table 3. Runtime comparison on 100 nodes

(seconds)	mean	min	max	std. dev.
async	118.1	118.2	120.9	1.3
async + DPLB	118.1	115.2	120.7	1.4

not reducing imbalance in a more critical part results in a much larger increase in TTS. On average, the risk of excessive runtime and progress imbalance of an asynchronous algorithm can be noticeably reduced with DPLB.

6. Conclusions

We have presented a method (DPLB) for applying progressive load balancing to an asynchronous algorithm in a distributed memory setting by adding periodic movement of work between nodes and running PLB on the nodes. Evaluation of DPLB showed that it is able to mitigate system performance variation by a reduction of 1.08x–4.05x in global progress imbalance and by 1.11x–2.89x in time to solution variability. We did not observe any significant overheads even when running on 100 nodes. In future work we plan to apply DPLB to other asynchronous iterative algorithms where there is scope for splitting the problem domain and moving it between computing units, for example the Schwarz method or stochastic gradient descent (SGD). This technique improves the resilience of asynchronous algorithms to noise and hence increase their value as components for meeting the exascale challenge.

Acknowledgments

This work was supported by grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism (pervasiveparallelism.inf.ed.ac.uk) from the UK Engineering and Physical Sciences Research Council (EPSRC). This work used the Cirrus UK National Tier-2 HPC Service at EPCC (www.cirrus.ac.uk) funded by the University of Edinburgh and EPSRC (EP/P020267/1).

References

[1] D. Chazan and W. Miranker, “Chaotic relaxation,” *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199–222, apr 1969.

[2] J. Zarins and M. Weiland, “Progressive load balancing of asynchronous algorithms,” in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3’17. ACM, 2017, pp. 5:1–5:9.

[3] D. Hackenberg *et al.*, “An energy efficiency feature survey of the intel Haswell processor,” in *International Parallel and Distributed Processing Symposium Workshop*. IEEE, May 2015, pp. 896–904.

[4] Y. Inadomi *et al.*, “Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 78.

[5] A. Porterfield *et al.*, “Application runtime variability and power optimization for exascale computers,” in *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2015, p. 3.

[6] 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 *Supercomputing Conference*. ACM/IEEE, 2003, pp. 55–55.

[7] R. Underwood, J. Anderson, and A. Apon, “Measuring network latency variation impacts to high performance computing application performance,” in *Proceedings of the International Conference on Performance Engineering*. ACM/SPEC, 2018, pp. 68–79.

[8] R. Lucas *et al.*, “DOE ASCAC Subcommittee Report February 10, 2014,” 2014.

[9] H. Anzt *et al.*, “A block-asynchronous relaxation method for graphics processing units,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1613–1626, dec 2013.

[10] T. Kurth *et al.*, “Deep learning at 15PF: Supervised and semi-supervised classification for scientific data,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’17. ACM, 2017, pp. 7:1–7:11.

[11] D. A. Donzis and K. Aditya, “Asynchronous finite-difference schemes for partial differential equations,” *Journal of Computational Physics*, vol. 274, pp. 370–392, oct 2014.

[12] F. Magoulès, D. B. Szyld, and C. Venet, “Asynchronous optimized Schwarz methods with and without overlap,” *Numerische Mathematik*, vol. 137, no. 1, pp. 199–227, Sep 2017.

[13] D. P. Bertsekas and J. N. Tsitsiklis, “Some aspects of parallel and distributed iterative algorithms – a survey,” *Automatica*, vol. 27, no. 1, pp. 3–21, 1991.

[14] J. Dinan *et al.*, “Scalable work stealing,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009, pp. 1–11.

[15] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, “Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 4, pp. 289–299, Apr. 2005.

[16] I. Mitliagkas *et al.*, “Asynchrony begets momentum, with an application to deep learning,” in *54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2016, pp. 997–1004.

[17] S. Zheng *et al.*, “Asynchronous stochastic gradient descent with delay compensation,” in *International Conference on Machine Learning*, 2017, pp. 4120–4129.

[18] I. Bethune *et al.*, “Performance analysis of asynchronous Jacobi’s method implemented in MPI, SHMEM and OpenMP,” *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 1, pp. 97–111, Feb. 2014.

[19] S. Chunduri *et al.*, “A generalized statistics-based model for predicting network-induced variability,” in *10th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS’19)*, ser. SC’19, November 2019.