# Parallelization Primitives for Dynamic Sparse Computations

Tsung-Han Lin
*Harvard University*

Stephen J. Tarsa
*Harvard University*

H.T. Kung
*Harvard University*

## Abstract

We characterize a general class of algorithms common in machine learning, scientific computing, and signal processing, whose computational dependencies are both sparse, and dynamically defined throughout execution. Existing parallel computing runtimes, like MapReduce and GraphLab, are a poor fit for this class because they assume statically defined dependencies for resource allocation and scheduling decisions. As a result, changing load characteristics and straggling compute units degrade performance significantly. However, we show that the sparsity of computational dependencies and these algorithms' natural error tolerance can be exploited to implement a flexible execution model with large efficiency gains, using two simple primitives: *selective push-pull* and *statistical barriers*. With reconstruction for compressive time-lapse MRI as a motivating application, we deploy a large Orthogonal Matching Pursuit (OMP) computation on Amazon's EC2 cluster to demonstrate a 19x speedup over current static execution models.

## 1 Introduction

Mainstream applications, including data mining at web scale, unsupervised machine learning, and high performance computing, often require deploying massive parallel computations in shared computing facilities like Amazon's EC2 cluster. Optimizing scheduling and load balancing decisions on these platforms is a challenge because available resources are dynamically constrained by runtime conditions such as fluctuating customer demand, node failures, and network congestion, which can slow workers and cause "stragglers" to occur regularly [1].

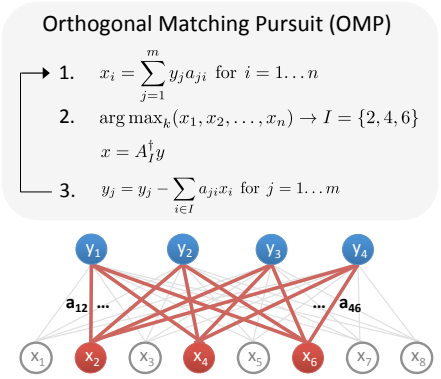


Figure 1: Orthogonal Matching Pursuit (OMP) is an iterative algorithm whose computational dependencies are bipartite, sparse, and dynamically determined throughout execution. As indicated by the red lines, only a subset of variables $x_i$ computed by Stage 1's matrix multiply are used in the least squares and residual calculations of Stages 2 and 3. In this illustration, these active variables have indices 2, 4, and 6.

Consequently, parallel programming models like MapReduce [2] and GraphLab's Gather/Apply/Scatter [3] are popular. These models enable users to leverage parallelism by abstracting away complicated scheduling, recovery, and load balancing issues. For example, MapReduce exploits data parallelism in two conceptually simple phases, map and reduce, for tasks like data summarization. In contrast, Gather/Apply/Scatter is designed for data with a known graph structure, and capitalizes on subgraph parallelism by processing disjoint vertex neighborhoods concurrently. This latter approach fits algorithms like PageRank on large graphs, where distributed variables are updated in an uncoordinated manner, and parallel asynchronous execution is a natural fit.

We seek a program abstraction appropriate for parallelizing dynamic sparse computations in machine learning, scientific computing, and signal processing applica-

This material is based on research sponsored in part by the Intel Corporation, and by the Air Force Research Laboratory under agreement number FA8750-10-2-0180. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the U.S. Government, or the Intel Corporation.

tions. Consider Orthogonal Matching Pursuit (OMP) [4] as a representative example. OMP solves an underconstrained linear system $\mathbf{y} = \mathbf{Ax}$, when the $n \times 1$ vector $\mathbf{x}$ has at most $k$ nonzero components, with $k \ll n$. OMP iterates over three basic stages:

**Stage 1 (Outer loop)** Find potential nonzero components in the current iterate $\mathbf{x}$, by computing correlations between the columns of an $m \times n$ measurement matrix $\mathbf{A} = (a_{ij})$ and the associated residual vector $\mathbf{y} - \mathbf{Ax}$.

**Stage 2 (Inner loop)** Select the largest $k$ components in $\mathbf{x}$ found by Stage 1, exclude others, and estimate $\mathbf{x}$ by least squares from the resulting over-constrained system.

**Stage 3 (Update)** Compute the new residual using the estimated $\mathbf{x}$; **Iterate**.

This algorithm's dependencies can be expressed with a bipartite graphical model whose vertices represent variables $y_j$ and $x_i$, and whose edges have values $a_{ij}$, as in Figure 1. While all variables are used for the matrix-vector multiplication in Stage 1, only a subset of these are carried into Stages 2 and 3. As the computation proceeds and OMP's estimate is iteratively refined, the precise set of variables needed by Stage 2 changes, so we say these variables and their dependencies are dynamic.

This structure, in which an outer loop identifies a small number of important variables, and dependencies to an inner loop are sparse and dynamic, appears in many algorithms, like K-SVD [5], CoSaMP [6], StOMP [7], SMP [8], SuPrEM [9], etc. The algorithms themselves underlie applications in machine learning and signal processing, like sparse feature extraction for deep learning and hierarchical inference, and compressive sensing signal recovery, respectively. In the former application, $\mathbf{A}$ represents an overcomplete dictionary matrix, and in the latter, a compressive sensing measurement matrix. We call attention to the fact that this dynamic sparsity is intrinsic to the *algorithm*, and is not a property of the application data structure, as in traditional sparse matrix or PageRank computations operating under a static computational model.

Both MapReduce and Gather/Apply/Scatter are suboptimal abstractions for dynamic sparse computations. Though only a few values from Stage 1 are needed by Stages 2 and 3, MapReduce asserts dense all-to-all communication between mappers and reducers, and enforces a rigid synchronization barrier on all variables. Similarly, Gather/Apply/Scatter partitions graph edges among workers and assumes a static graph throughout execution, leading to extra work when nodes pull data from their entire neighborhood, instead of the small subset associated with active variables. GraphLab also requires all synchronization barriers to be fully specified at the outset, and, like MapReduce, is not malleable enough for the dynamic dependencies of our algorithms.

Dynamic sparse computations mandate a more flexible execution model. For instance, an efficient implementation of OMP should compute and synchronize only on those values necessary for computation. This is difficult when the identities of these variables are unknown at the outset. However, we will exploit both the limited number of active variables, and the natural error tolerance of iterative sparse estimations, to implement a flexible execution model.

We accomplish this using two simple primitives: *selective push-pull* and *statistical barriers*. Selective push-pull isolates computations to active variables involved in the inner loop by using a vertex-initiated "ping-pong" execution flow, eliminating the need to track active dynamic edge sets. Meanwhile, statistical barriers use a statistical metric to relax synchronization requirements. They exploit the fact that a large fraction of task completion (e.g., in OMP Stage 1) is often sufficient to capture $k$ of $n$ important values, when $k \ll n$. Should a value be missed in one iteration of the algorithm, it can be picked up in subsequent iterations. These two primitives work together to dynamically focus computations on small portions of a much larger, densely connected dependency graph, realizing major efficiency gains.

With reconstruction of compressive time-lapse MRI images as a motivating application, we extend GraphLab to implement selective push-pull, deploy a large OMP computation on Amazon's EC2 cluster, and demonstrate a 19x speedup over the current static runtime system. We then use an event-driven simulation and straggler statistics from the literature to show that statistical barriers improve both average and worst case computation times, despite the possibility of additional OMP iterations. Finally, we use our EC2 implementation to scale previous OMP-based image reconstruction for compressive time-lapse MRI applications with additional processors.

This paper is organized as follows: in Section 2, we review GraphLab and MapReduce; Section 3 presents our motivating application, compressive-sensing based time-lapse MRI reconstruction; Section 4 discusses our strategy for parallelizing sparse estimation algorithms and describes the two supporting primitives; finally, Section 5 presents performance results.

## 2 Review of Parallel Program Abstractions

To provide background information on parallel programming abstractions, we first consider two popular examples.

### 2.1 GraphLab

GraphLab consists of both a low level runtime system for distributing and executing parallel jobs, and a high level

program abstraction called Gather/Apply/Scatter. The runtime provides interfaces for loading, distributing, and computing on data with a static graph structure, and uses MPI for inter-processor communication.

Users define generic "vertex programs" that are bound to each vertex, which implement Gather/Apply/Scatter. At initialization, GraphLab first constructs a graph and associates input data values with vertices and edges. Then, the edge set is partitioned, and edge and vertex data is distributed to workers, where it is held in main memory. Partitioning the edge set load-balances operations in dense vertex neighborhoods, but means that vertices can be "split" when their edge sets are spread across workers. When this happens, workers running a vetex program will compute partial results using their local edge sets, and the runtime system will merge and synchronize values. Throughout, computation is driven forward by signaling vertex programs to execute Gather/-Apply/Scatter, while distributed locks ensure that one vertex per neighborhood executes at a time.

Gather/Apply/Scatter's three phases proceed as follows: a vertex program gathers, or pulls values from its neighbors, then applies an update to its local value, and finally signals neighbors to begin execution, scattering the computation. Subgraph parallelism is realized when vertex programs in different neighborhoods execute concurrently. The gather phase pulls values toward the active vertex, as opposed to pushing them from neighbor vertices, as a straightforward way to assert that all updates are received before the apply phase.

## 2.2 MapReduce

MapReduce is another popular program abstraction for parallelization, supported primarily by the Hadoop MapReduce distribution, and also by GraphLab. The map phase organizes inputs into key/value pairs using multiple parallel mappers. The reduce phase computes on sets of values in a key-parallel fashion. All-to-all communication occurs between mappers and reducers.

## 2.3 Execution Models

Both abstractions assume statically defined variables and dependencies. In the case of MapReduce, a dependency graph is implied by the arrangement of mappers and reducers: mappers are independent of each other, as are reducers, and dependencies exists from all mappers to all reducers. For GraphLab, the graph is also static, but is user-defined, based on known interrelationships in input data.

Both abstractions allow limited synchronization flexibility in their execution models. MapReduce assumes that full synchronization is needed between map and reduce stages, and requires all mappers to check in at a barrier. GraphLab provides three modes: fully synchronous, fully asynchronous, and partially asynchronous (i.e. "asynchronous serializable" in GraphLab terminology). This allows users to impose barriers on all, none, or subsets of vertices, but requires that these choices be made prior to execution. To respond to runtime conditions that cause stragglers, both systems provision additional idle machines to recompute straggling tasks.

# 3 Motivating Application Scenario: OMP for Compressive Time-Lapse MRI

Our application vehicle is image reconstruction for compressive time-lapse MRI [10], a technique that applies compressive sensing theory to reduce sampling rates during clinical image acquisition. Results include finer temporal resolution for time-lapse MRI scans [11], and lower procedure time for patients [12]. After acquisition, full resolution images are reconstructed using a compressive sensing decoding algorithm such as OMP. In the past, researchers have pursued faster decoding in this specific context to yield more "clinically useful" patient procedures [12], so we adopt the goal of reducing decoding time in this work. In Section 5.2, we use signal dimensions and sampling rates from data in [11] to compare results.

OMP's formula is shown in Figure 1. The algorithm solves the minimization:

$$\min \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2 \quad \text{s.t.} \quad \|\mathbf{x}|_0 \leq k$$

where $\mathbf{y}$ is an $m \times 1$ measurement vector, $\mathbf{A}$ an $m \times n$ measurement matrix, and $\mathbf{x}$ a sparse signal vector with at most $k$ non-zero entries. OMP iteratively refines an estimate of $\mathbf{x}$'s support to reduce residual error, terminating after satisfying an error threshold or exceeding an iteration cap. OMP's original form greedily selects one component per iteration over exactly $k$ iterations. It has since generalized, e.g., in [6], to select and update multiple components per iteration. In this paper, we use "OMP" to refer to this more-general form. As noted perviously, OMP's outer loop computes matrix-vector multiplications, and its inner loop computes least squares solutions for the support set; this structure, where an inner loop computes on a small set of variables dynamically selected by an outer loop is also present in [7], [8], [9].

# 4 Dynamical Primitives

We use two primitives that *work together* to focus computations on dynamically determined subgraphs of our

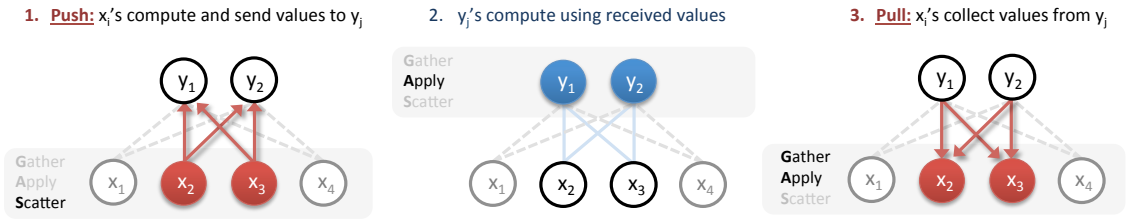


Figure 2: An illustration of selective push-pull, which excludes inactive variables and performs computations only on a selected subgraph (denoted by solid arrows and bolded nodes) to minimize unnecessary work. This primitive is equivalent to three rounds of the Gather/Apply/Scatter abstraction of GraphLab, with an additional "self-signaling" phase.

algorithms' full dependency graphs. In the outer computational loop, where nonzero variables are identified, *statistical barriers* drive execution forward when a portion of tasks reach the barrier. Synchronization is based on a statistical criterion, such as percent completed, that captures when those few important values are likely to have been computed. In the inner loop, where computation involves only these selected variables, we use *selective push-pull* to dynamically compute on the associated subgraph.

## 4.1 Selective Push-Pull

*Selective push-pull* constrains computation to a subgraph associated with $k$ active vertices from the original set of $n$ vertices, temporarily eliminating the other $n-k$ inactive vertices. During computation, subsets of variables are activated by comparing their values to some criterion, such as a threshold that is predefined or broadcast by a master machine. For OMP, we use this latter method, and update the threshold to be the vertex value of the $k^{th}$ largest $x_i$.

Selective push-pull is a general primitive that can naturally express commonly-applied "ping-pong" operations over bipartite graphs. In these operations, computations flow from one side of the bipartite graph to the other, and then back. For instance, referring to Figure 2, selective push-pull is used to implement two matrix-vector multiplications, computing $i+1^{\text{st}}$ iterates of vectors $\mathbf{x}$ and $\mathbf{y}$: $\mathbf{y^{(i+1)}} = \mathbf{A}\mathbf{x^{(i)}}$ and $\mathbf{x^{(i+1)}} = \mathbf{A^T}\mathbf{y^{(i+1)}}$, with the entries of matrix $\mathbf{A}$ as edge values. The primitive is initiated on the $x_i$ denoted in red. For every connected edge, an $x_i$ multiplies its vertex value with the edge value, and sends the result to neighbors $y_j$. The $y_j$'s will then sum received products and use the result to update their vertex values. Finally, $x_i$'s pull in neighbors' values, multiplying them by the associated edge values. Using this simple primitive, the entire conjugate gradient least squares method can be expressed, which implements OMP's inner loop.

Runtime efficiency of dynamic subgraph selection is improved using selective push-pull because it does not functionally require edge selection in the graph. Under the normal Gather/Apply/Scatter model, a "pull"-only paradigm, dynamic subgraph selection must index active connected edges for every $y_i$ vertex. This solution is not scalable to large graphs due to storage and computation costs. The key idea of selective push-pull is to avoid initiating edge computations from the $y_i$, so $y_i$'s do not need to maintain and compare against lists of activated $x_i$'s.

We implement selective push-pull on top of GraphLab with only minimal modifications. A selective push-pull can itself be expressed with three rounds of Gather/-Apply/Scatter, as shown in Figure 2. The first round executes on the $x_i$'s, where Gather and Apply are passthrough functions, and Scatter is used to push a value to $y_j$'s and signal them for execution. The second round executes on $y_j$'s, where Apply is used to update the vertex value based on pushed values, while Gather and Scatter are passthrough functions. In the third round, $x_i$'s execute normally, pulling values from $y_j$'s to update their vertex values using Gather and Apply. Our implementation uses barrier synchronization to guarantee the execution order of the three rounds, and the only modification to GraphLab is to allow the $x_i$'s to schedule themselves for the third round, at the end of the first round. We therefore add a "self-signaling" phase into the programming abstraction after Scatter.

## 4.2 Statistical Barriers

To optimally implement dynamic sparse computations, we need a flexible synchronization method that imposes barriers on subsets of variables, as determined during computation. We implement this model by introducing a *statistical barrier* primitive that synchronizes on a dynamic subset of variables at the barrier, as determined by some statistical metric. This primitive, for instance, can require that a user-defined portion (e.g., 90%) of tasks complete for execution to continue, imposing synchronization in a statistical sense. Fully synchronous and fully asynchronous execution can be implemented with special cases of the statistical barrier, using threshold values of 100% and 0% of tasks, respectively.

Our use of statistical barriers exploits iterative sparse

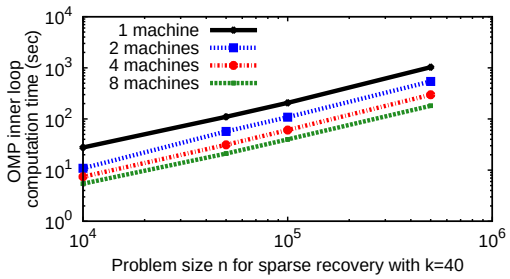


Figure 3: Computation time of OMP's inner loop under Gather/Apply/-Scatter based on a static graph, measured as the number of machines on EC2 is scaled from 1 to 8, for different problem sizes. Running time scales with the graph size, reaching 181 seconds for 8 machines when $n = 500,000$.

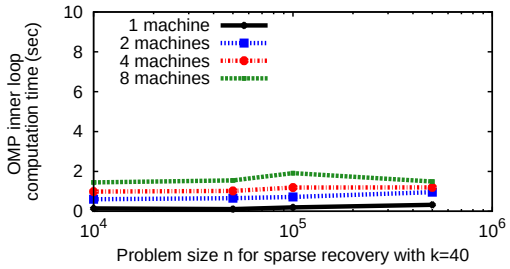


Figure 4: Computation time for OMP's inner loop using selective push-pull, measured as the number of EC2 machines is scaled from 1 to 8, for different problem sizes. Running time scales with the number of active variables, which is constant in the experiment, independent of graph size $n$.

estimations' partial resilience to out-of-order execution. When the portion of completed tasks specified by the barrier is enough to capture all active variables, correct execution order is realized; when some values are erroneously excluded, errors can be corrected in subsequent iterations of the algorithm. For example, after the OMP Stage 1 outer loop, the estimated signal can be refined using an incomplete list of nonzero components. The OMP computation will have another chance to capture left-out components in later iterations. Therefore, provided no component is systematically excluded from computation, a statistical guarantee is sufficient for proper execution order, though the total number of iterations may increase to meet the same accuracy objective.

This statistical barrier primitive is similar in some aspects to early phase termination [13], an optimization technique used to increase overall utilization during parallel computations. If a set of parallelizable tasks is mismatched to the number of processors, early phase termination abandons unfinished tasks when utilization drops below a threshold. Idle time is reduced, as is the total amount of computation performed, and profiling ensures that induced errors are within acceptable limits. Though sharing the same objective of sacrificing some computations to improve overall efficiency, our primitive is used to realize asynchronous execution when precise dependencies are unavailable to the scheduler, but sparsity and error tolerance can be exploited to capture them statistically.

# 5 Experiments and Results

We deploy our modified GraphLab runtime on Amazon's EC2 cluster [14] to evaluate the performance of our execution model. Selective push-pull is fully implemented, and we measure performance on several large OMP computations as the number of machines is scaled. Statistical barriers require more significant modifications to the GraphLab runtime, so we emulate performance by combining timing measurements from EC2 with straggler statistics reported in the literature, in an event-driven simulation. Completion time is computed by implementing OMP, while omitting results from straggling tasks according to the simulated barrier criterion.

The EC2 cloud is a representative data center, from which computing resources can be leased on-demand. Each EC2 instance in our configuration is a 64-bit virtual machine providing one 1.0 GHz core to GraphLab, with 7.5 GB of memory per virtual machine, and a 500 Mbps ethernet link connecting instances. Instances can be requested within a specific geographic region, though physical node location is transparent to the user.

## 5.1 Selective Push-Pull

Figures 3 and 4 compare the running time of OMP's least squares inner loop using both selective push-pull and a baseline static GraphLab runtime, as the number of processors is scaled from 1 to 8. We vary the input signal size from $n = 10,000$ to $500,000$, while keeping $m = 200$ and $k = 40$ constant throughout experiments. This means that the number of active $x_i$'s is always 40, and the least squares problem has the same size regardless of the input signal size $n$. This relationship is true to sparse coding used in feature extraction, and in compressive sensing recovery, wherein $k$ and $n$ are generally independent parameters.

With GraphLab's static graph, the computation time for the inner loop grows linearly with $n$ due to the growth in graph size. Although completion time can be reduced by using multiple machines, it is still a function of $n$. As shown in Figure 3, the computation spans 181 seconds on 8 machines for $n = 500,000$. In contrast, with selective push-pull, the computation is focused on a much smaller set of active variables. As shown in Figure 4, it takes less than 2 seconds to complete, and the running time is constant relative to $n$. This shows that selective push-pull succeeds in only triggering necessary computation, and can be essential for large scale sparse computation

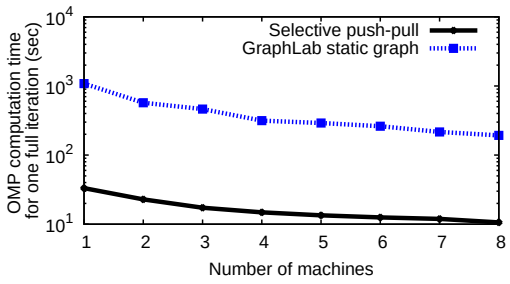


Figure 5: The computation time for running a full iteration of OMP when $n = 500,000$. With selective push-pull, the computations are focused on active variables, therefore outperforming the baseline GraphLab implementation, which assumes a static graph structure.

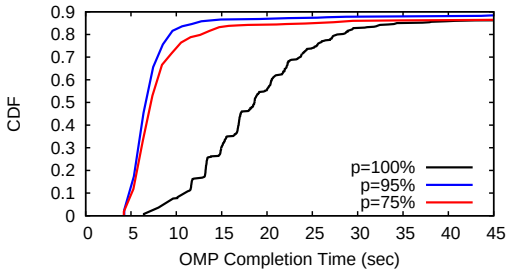


Figure 6: Completion times for $n = 10,000, m = 200$ OMP using statistical barriers that require $p\%$ of tasks to complete. By trimming stragglers without invoking many additional iterations, a $p = 95\%$ statistical barrier saves 2.5x time at the median, and 4x in the worst case.

problems.

In Figure 4, we note that the inner loop computation time is higher when we use more machines in this case, because inter-machine communication cost dominates. Since communication overheads may vary greatly on EC2, we also see a bump for the 8-machine curve at $n = 100,000$. This suggests that, for problem sizes where selective push-pull drastically reduces the amount of computation, the inner loop should be migrated to a smaller number of machines. For simplicity in the experiments reported in this paper, we kept the same number of machines, despite a minor performance hit.

Figure 5 shows the time to compute a full iteration of OMP for $n = 500,000$. With selective push-pull, the inner loop cost is greatly reduced. As a result, the outer loop computation, which involves all variables, dominates. This means that computation time is decreased as the number of machines increases. Overall, selective push-pull improves our baseline GraphLab implementation with a static graph structure by 19x on 8 machines.

## 5.2 Evaluation for Compressive MRI

Next, we compare our GraphLab implementation of OMP with selective push-pull enabled, to the results reported for compressive time-lapse MRI in [11]. With dimensions $n = 35,000$, $m = 7000$, and $k = 2333$, we report a decoding time of 334s using $8 \times 1.0$ GHz cores. This compares to previous results of 534s on a dual-core 2.4 GHz processor. Though we use more compute cores, Figure 5 shows that we can scale the computation to further reduce reconstruction time, possibly by leasing additional virtual machines on-demand. This contrasts with the implementation of [11], which is limited by the performance of a single platform.

## 5.3 Statistical Barriers

Finally, using an event-driven simulation and the straggler statistics reported from a year's worth of MapReduce tasks in a production data center [1], we evaluate statistical barrier performance. We run a large number of OMP computations to termination based on a fixed accuracy criterion (residual $\leq 0.01$). During each iteration, a statistical barrier is used to push the outer loop computation forward once a user-specified percentage of $x_i$'s have completed. We measure overall completion time, which includes savings from straggler mitigation, as well as costs due to possible additional OMP iterations.

In [1], researchers report 25% of all parallel computations experiencing a high proportion (15+%) of straggling tasks, defined as 1.5x median completion time. Of these stragglers, 80% complete in 2.5x the median, while 10% require more than 10x median completion time, showing just how bad the straggler problem can be in the wild. Using these statistics, we fit the median completion time to that measured on a single EC2 instance. Maximum completion time is capped to 15x the median, and we adopt a pessimistic rate of 30% computations affected by stragglers. Tasks are assigned to $w = 10$ machines, and are load balanced. We assign the slowest tasks to a single machine. This not only makes the gains from statistical barriers easily understood, but reflects slowdowns that cause machine-specific performance anomalies, such as location-related network delays. Finally, we randomize task execution order so that the computation is not systematically biased.

Figure 6 shows the empirical cumulative distribution functions (CDFs) of OMP completion times using statistical barriers requiring $p = 75\%, 90\%$, or $100\%$ tasks to complete for an outer-loop computation size of $n = 100,000$, and $k = 40$. All computations achieve the same decoding accuracy, though some require more iterations as a result of the statistical barrier. The baseline method uses a rigid barrier ($p = 100\%$) and has a median completion time of 17s, roughly matching the 16s completion time observed using 8 machines on EC2. However, stragglers hurt, and slow the OMP computation by 10x in the worst case. In comparison, both statistical barriers have better median completion times of 6 seconds.

We see the effect of extra iterations with the aggressive $p = 75\%$ barrier, which performs worse in general than the $p = 95\%$ barrier, due to extra computations. However, by trimming the worst stragglers, the 95% barrier realizes a 2.5x improvement in the average completion time, and a 4x improvement in the worst case. This simulation shows that, when a small number of stragglers affect the computation, most of the gains from statistical barriers can be captured by setting $p$ to a large value such as $p = 90\%$.

## 6 Conclusion

In this paper, we describe a class of iterative algorithms, which exhibit a sparse set of active variables and dependencies during computation. We showed that this class is suboptimally served by current programming abstractions, whose assumptions of statically defined variables and dependencies lead to a huge performance hit. Exploiting the dynamic sparse structures in these algorithms, we defined two new primitives – selective push-pull, and statistical barriers – that work together to regain lost efficiency by implementing a flexible execution model. Performance gains were then demonstrated in bipartite graphs of practical importance. Although these primitives were described in this paper for this specific class of computations, we believe that they will prove useful to a wider range of applications in which efficiency can be gained by dynamic, sparse graph computations, or by optimistic straggler mitigation.

## References


[1] 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 9th USENIX conference on operating systems design and implementation (OSDI'10)*, USENIX, 2010.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX conference on operating systems design and implementation (OSDI'12)*, USENIX, 2012.

[4] J. A. Tropp, "Greed is good: Algorithmic results for sparse approximation," *IEEE Transactions on Information Theory*, vol. 50, no. 10, pp. 2231–2242, 2004.

[5] M. Elad and M. Aharon, "Image denoising via sparse and redundant representations over learned dictionaries," *IEEE Transactions on Image Processing*, vol. 15, no. 12, pp. 3736–3745, 2006.

[6] D. Needell and J. A. Tropp, "CoSaMP: Iterative signal recovery from incomplete and inaccurate samples," *Applied and Computational Harmonic Analysis*, vol. 26, no. 3, pp. 301–321, 2009.

[7] D. L. Donoho, Y. Tsaig, I. Drori, and J.-L. Starck, "Sparse solution of underdetermined linear equations by stagewise orthogonal matching pursuit," *IEEE Transactions on Information Theory*, vol. 58, no. 2, pp. 1094–1121, 2012.

[8] P. Indyk and M. Ruzic, "Near-optimal sparse recovery in the l1 norm," in *IEEE 49th Annual Symposium on Foundations of Computer Science (FOCS'08)*, pp. 199–207, IEEE, 2008.

[9] M. Akcakaya, J. Park, and V. Tarokh, "A coding theory approach to noisy compressive sensing using low density frames," *IEEE Transactions on Signal Processing*, vol. 59, no. 11, pp. 5369–5379, 2011.

[10] M. Lustig, D. Donoho, and J. M. Pauly, "Sparse MRI: The application of compressed sensing for rapid mr imaging," *Magnetic Resonance in Medicine*, vol. 58, no. 6, pp. 1182–1195, 2007.

[11] M. Usman, C. Prieto, F. Odille, D. Atkinson, T. Schaeffter, and P. Batchelor, "A computationally efficient OMP-based compressed sensing reconstruction for dynamic MRI," *Physics in Medicine and Biology*, vol. 56, no. 7, p. N99, 2011.

[12] S. Vasanawala, M. Murphy, M. Alley, P. Lai, K. Keutzer, J. Pauly, and M. Lustig, "Practical parallel imaging compressed sensing mri: Summary of two years of experience in accelerating body mri of pediatric patients," in *2011 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pp. 1039–1043, IEEE, 2011.

[13] M. C. Rinard, "Using early phase termination to eliminate load imbalances at barrier synchronization points," in *ACM SIGPLAN Notices*, vol. 42, pp. 369–386, ACM, 2007.

[14] http://aws.amazon.com/ec2/.