Parallel Simulated Annealing with MRAnneal

Benjamin Marks, Riley Collins, and Kevin C. Webb
Department of Computer Science, Swarthmore College, Swarthmore, PA
{bmarks1,rcollin4,kwebb}@cs.swarthmore.edu

***Abstract*—Simulated annealing algorithms, which repeatedly make small changes to candidate solutions to find approximately optimal ones, are a common method for approximating solutions to computationally expensive optimization problems. While using multiple machines to perform such computations in parallel is attractive as a means to reduce the running time, execution in a cluster environment requires substantial software infrastructure to cope with the challenges of a distributed system.**

In this paper, we introduce MRAnneal, a framework that simplifies the implementation of parallel simulated annealing algorithms. MRAnneal allows users to explicitly trade-off running time and the quality of approximate solutions by supplying only a small number of automatically tuned parameters. Our experimental results demonstrate that implementing applications using MRAnneal is straightforward and that such implementations yield approximate solutions quickly, even for applications without intuitive serial approximation heuristics.

I. Introduction

Optimization problems manifest in many practical scenarios, such as determining the cheapest route for a salesman, determining the smallest number of people, each with certain skills, needed to perform a task requiring some larger set of skills, or scheduling exams for undergraduates. One characteristic of these problems that makes them so difficult is a large solution space — for each task there may be an exponential number of possible solutions, and iteratively checking each one would take years for even modest input sizes. Generally, increases in computing power have not made finding optimal solutions to many classes of problems (e.g., NP-complete) any more tractable. At its core, an $O(2^n)$ algorithm remains exponential, even with a factor of 100 speedup.

With no known method for solving such optimization problems in polynomial time, much effort has been focused on developing approximation algorithms or randomized algorithms to find acceptable approximate solutions. Simulated annealing [1], repeatedly making small changes to candidate solutions in order to find approximately optimal ones, is one such approximation method. Many simulated annealing algorithms are known to benefit from parallelization [2]. However, given the notorious difficulty of parallel programming [3], it remains challenging for users who are not parallel computing experts to easily utilize parallel simulated annealing to solve computationally intensive optimization problems.

This work presents MRAnneal[1], a framework that simplifies the construction of parallel simulated annealing algorithms. MRAnneal enables users without extensive distributed programming experience to quickly author high-performance parallel simulated annealing software as easily as a serial routine. Any problem for which solutions can be randomly approximated by iteratively making small changes to an existing solution can be easily expressed and automatically parallelized using the MRAnneal framework. Users indicate their desired solution quality (at the cost of run time) using a small set of parameters, and MRAnneal will dynamically adjust its behavior accordingly.

MRAnneal utilizes the MapReduce [4] programming model, making it easily deployable on clusters of commodity hardware via popular, freely-available software (e.g., Hadoop [5]). An execution of MRAnneal consists of multiple rounds in which mappers and reducers iteratively work towards higher quality solutions. Mappers load-balance candidate solutions across multiple reducers, each of which performs simulated annealing to randomly search the solution space in parallel. Reducers yield a subset of the annealed solutions, which are passed to the next round of execution.

II. MRAnneal Design

MRAnneal represents a layer of abstraction between users and the underlying MapReduce runtime system. Before executing, users supply two inputs: a small set of functions that implement their optimization problem's logic and a simple set of parameters. Across several stages, MRAnneal combines these inputs with statistical regression to automatically tune its runtime behavior.

[1]Code available at: http://www.cs.swarthmore.edu/mranneal

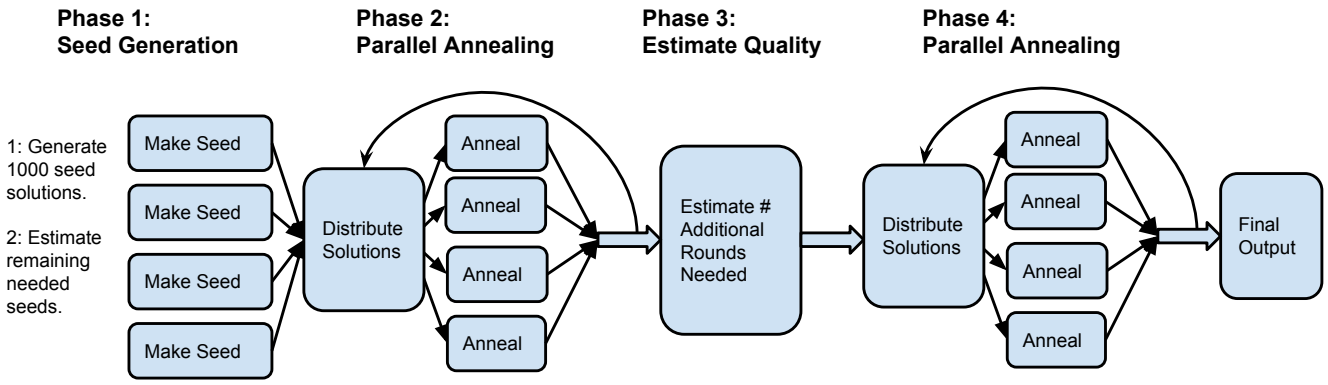

Fig. 1. An overview of the stages in MRAnneal's parallel execution strategy.

Figure 1 depicts an overview of MRAnneal and its parallelization strategy. Given the user's inputs, MRAnneal executes in four phases. It begins by generating initial *seed* solutions, automatically gauging the number of seeds necessary to achieve a diverse spread across the solution space. Next, MRAnneal performs a small number of parallel annealing rounds over the seed solutions, which it uses to estimate how many total annealing rounds will be needed to reach the user's solution quality goal. Finally, it finishes the computation by running the remaining parallel annealing rounds and formatting the output. This section describes the details of each step along with MRAnneal's general architecture, our goals for the system, and the challenges we faced in designing it.

A. User-supplied Functions

MRAnneal's primary design goal is to provide users with an interface that is both straightforward to non-experts and widely applicable to a large class of optimization problems. Towards generality, MRAnneal targets any optimization problem with the following three common characteristics:

1) Calculating any single correct solution to the problem is easy (the solution need not be optimal or even close to it).
2) Any pair of solutions can be quantitatively ranked against one another.
3) Small or incremental changes can be made to a solution (and undone) without sacrificing its correctness.

These three qualities serve to define the user-supplied functions in MRAnneal. Users implement *generate_seed()*, which is called to calculate an initial solution, *score_solution()*, which assigns a numerical score to a solution, and an *anneal()* and *undo_anneal()*, which perturb candidate solutions in search of better scores. Note that an implementation of these functions does not require the user to consider solution distribution, partitioning, or any other aspects of parallel programming.

B. Parameters

Having defined the required functions for a problem of interest, a user can easily influence job execution characteristics via a small set of parameters. To further support simplicity for non-expert users, MRAnneal allows users to explicitly choose a position on the spectrum of running time and solution quality. Enabling users to express this trade-off succinctly is challenging in a distributed systems context, where users are often faced with an overwhelming number of parameters. In early versions, we required users to precisely specify several parameters, including the number of seed solutions to generate, the total number of anneals to perform, and many others, whose impact on performance and solution quality were not immediately obvious.

Rather than requiring the user to determine every aspect of the system, MRAnneal now takes an alternative approach in which the user is only expected to provide two basic parameters and a high-level performance goal. While executing, MRAnneal collects dynamic runtime data and automatically tunes the remaining internal system parameters according to the user's high-level goal.

Table I summarizes MRAnneal's requisite parameters. The first two entries, *num_machines* and *num_results*, specify to MRAnneal the available hardware resources and the desired number of final output results, respectively. The final required parameter, *target_quality*, deter-

TABLE I
SUMMARY OF USER-SUPPLIED PARAMETERS. PARAMETERS WITH A (*) ARE REQUIRED. FOR THOSE NOT REQUIRED, WE SHOW THEIR DEFAULT VALUES IN PARENTHESES.

Parameter	Role in MRAnneal
num_machines*	Number of machines available.
num_results*	Number of results the user wishes to receive in the final output.
target_quality*	Percentage of the estimated potential improvement user wishes to pursue.
min/max_seeds (1,000 / 10,000)	Lower and upper bounds on number of seed solutions to generate.
min/max_rounds (10 / 100)	Lower and upper bounds on number of annealing rounds to perform.
num_anneals (150)	Number of anneals to perform, per round, on each solution.

mines the stopping criteria when generating seed solutions and annealing in parallel. Focusing on the latter, each solution maintains a history of its scores at the end of each round. MRAnneal uses this data to estimate the score for that solution after the maximum number of annealing rounds have been executed. A solution's *potential improvement range* is the difference between the score when the solution was first generated (s_0) and the estimated score after the maximum number of rounds (s_{max}). MRAnneal will stop annealing a solution in round k if:

$$s_k \geq s_0 + \textit{target\_quality} * (s_{max} - s_0).$$

In other words, *target_quality* specifies how much of the potential improvement should be pursued.

C. Phase 1: Generating Seed Solutions

Given an implementation of the user-supplied functions and the required parameters, MRAnneal can begin the first phase of its execution: seed solution generation. "Seed solutions" represent unoptimized solutions that will serve as inputs to future annealing steps. While seed generation generally accounts for a relatively short portion of MRAnneal's overall execution time, the phase is important, as it ultimately provides the foundational corpus of all the solutions MRAnneal will examine.

During this phase, MRAnneal must determine how many seed solutions to generate. Producing too few seeds limits the diversity of the solution pool and reduces MRAnneal's coverage of the global solution space. On the other hand, too many seed solutions lead to an inefficient duplication of effort in subsequent phases.

By default, MRAnneal generates between 1,000 and 10,000 seed solutions. As it generates and scores the first 1,000 solutions, MRAnneal records the range of the solutions' scores as each new solution is added to the set. Having tracked the first 1,000 solutions, it performs a logarithmic regression to predict how the score range would change if up to 9,000 additional seed solutions were generated, choosing how many additional solutions to generate based on the user's *target_quality* percentage. That is, if the user asked for a target quality of 80%, MRAnneal will estimate what the score range would be after generating 10,000 solutions, compare this range to the observed range covered by the first two generated solutions, estimate the number of seed solutions needed to be generated to capture 80% of the increase in coverage, and generate that many seed solutions.

D. Phase 2: Annealing I

Following seed generation, MRAnneal begins its first annealing phase, performing a series of map and reduce rounds. Within a round, mappers assign keys to solutions to ensure that each reducer's workload remains balanced. The underlying MapReduce runtime shuffles the solutions to reducers, which work independently to evaluate each solution and, if annealing is predicted to improve solution quality, repeatedly perturb and score their assigned solutions with the user-supplied *anneal()* and *score_solution()* functions.

As each solution is passed between rounds, it includes with it a list of its scores in prior rounds. Before assigning a solution to a reducer for the next round of annealing, the mapper fits a curve to the solution's score history. If the curve indicates that additional annealing is unlikely to produce any further improvement, MRAnneal will declare the solution to be *finalized* and discontinue annealing it. MRAnneal discards finalized solutions whose scores are not high enough to have any hope of being among the top *num_results* in the final output set.

As each solution is perturbed, MRAnneal considers, and potentially saves, lower scoring solutions. This prevents solutions from getting "stuck" at a suboptimal local maximum. When annealing results in a lower score, the modified solution will be kept at most 25% of the time[2]. Even though a lower-scoring solution may be retained, MRAnneal records the best solutions seen overall to avoid "forgetting" a good earlier solution by the time the computation ends. The best two derivatives of each solution are candidates to be passed on to the next round.

[2] $P[\text{keep lower score}] = 0.25 * \frac{new\_score}{old\_score}$

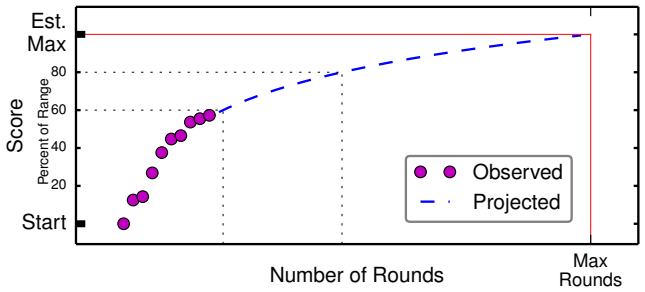

Fig. 2. Illustration of MRAnneal's round estimation algorithm. A logarithmic regression over phase 2's solution scores predicts the improvement for annealing over additional rounds, up to the maximum. MRAnneal chooses the number of rounds needed to achieve the user-specified *target_quality* percentage of this improvement. A higher percentage leads to higher scores, at a cost of longer running time.

At the end of an annealing round, each reducer yields its top solutions, passing one solution on to the next round for each input soluion. Completion of a round serves as MRAnneal's communication and synchronization point, in which the better solutions are passed on to the next round's mappers for load balancing, and the process repeats. In this first annealing phase, MRAnneal anneals for ten rounds, recording the growth in solutions' scores as the rounds progress. The rate of growth provides the data with which MRAnneal can predict how many additional rounds are necessary to achieve the user's solution quality goal.

E. Phase 3: Solution Quality Estimation

Across optimization problems, the growth in solution quality scores follows a general pattern: in early annealing rounds, solution quality typically grows quickly, but at some point the growth tapers off, with further annealing yielding diminishing returns. An insight behind MRAnneal is that we can statistically forecast the solution score growth, with satisfactory accuracy, by fitting a logarithmic curve to phase two's recorded score data. Having such a projection enables MRAnneal to estimate where on the curve it will be, thus allowing it to execute only as many additional rounds as is necessary to reach the user's solution quality goal.

Like the estimation done for seed solution generation, MRAnneal performs a logarithmic regression over the maximum solution scores that it encountered during phase 2's first ten annealing rounds. It analyzes the curve to determine, up to the maximum number of remaining annealing rounds, how many additional rounds are necessary to reach the user's target percentage of potential improvement. The process of score estimation, target score derivation, and rounds calculation is depicted in Figure 2.

Having calculated the number of additional rounds needed, MRAnneal then invokes the final phase, in which in anneals for that many more rounds.

F. Phase 4: Annealing II

The second annealing stage proceeds similarly to phase 2. The only notable difference is that the length of phase 4 varies depending on the observed improvements to the solutions and the user's target percentage.

G. Output Processing

The fields and contents of a solution are determined by the user (via *generate_seed*) and are opaque to MRAnneal. Often, users of MRAnneal will pass solutions between anneal rounds in formats that are not easily understood by humans. While users could certainly write their own post-processing scripts to manipulate output into something more friendly, MRAnneal's interface exports an optional *format_result()* function which, if defined, is called once per final output solution to manipulate the result into a human-readable form.

III. Implementation

Our MRAnneal implementation takes advantage of the MRJob [6] MapReduce library for Python, which simplifies job execution for several MapReduce runtime environments (e.g., local machine, private Hadoop [5] cluster, or Amazon's Elastic MapReduce [7]). This section describes MRAnneal's efforts to improve parallel performance and demonstrates that it can be utilized to succinctly model several practical example problems.

A. Performance Considerations

Even in parallel, simulated annealing is time consuming and computationally intensive, with most of the runtime spent annealing and scoring solutions. For some optimization problems, a large amount of time may also be devoted to generating seed solutions. MRAnneal aims to make the execution of any function that is called repetitively in a loop as fast as possible, since such loops are where the majority of time will be spent.

Initialization. As an optimization to eliminate costly (re-)initialization, MRAnneal permits users to define *init_seed()* and *init_anneal()* functions that get called exactly once prior to entering tight loops. Such functions allow, for instance, the seed solution generator to read an input file once (e.g., problem description) and store its contents in memory for the duration of its execution.

While the idea of including *init* functions is not novel, MRAnneal's *init* functions differ from those of other MapReduce libraries in that they require considerably less knowledge about the underlying MapReduce implementation. That is, a user of MRAnneal need not worry about how to manipulate MRAnneal objects or structures, as may be the case with other systems (e.g., Hadoop Context objects or MRJob runner data members). MRAnneal's *init* functions can be imported from any context, and the returned result is passed transparently to subsequent calls to seed generation, annealing, or scoring functions. This optimization eliminates the need for re-initialization at every call. In many problems, *init* functions provide substantial performance benefits. For example, in an instance of Max Cut running with 50 machines, we achieved a 4x speedup by eliminating redundant initializations.

Partitioning. To maximize its usage of the available computing resources, MRAnneal must evenly distribute seed solutions across reducers during each annealing round. With an imbalanced distribution, some reducers might execute far longer than others, leaving parts of the cluster idle. Thus, MRAnneal exerts fine-grain control over the MapReduce partitioning function to systematically balance the load of solutions across reducers. Specifically, we take advantage Hadoop's *KeyFieldBasedPartitioner*, as it allows us to specify that if keys are of the form x, y, then all keys with the same x value are sent to the same reducer. Mappers distribute results to reducers in a round robin fashion, ensuring that all keys destined for the same reducer are grouped by the same y value, resulting in a single invocation of *reduce()*.

Storage. Finally, MRAnneal must be efficient in how it stores solutions in memory. At the end of a round, each reducer outputs the same number of solutions passed to it at the beginning of the round. We use Hadoop's sorting functionality to ensure that each reducer knows k, the number of solutions it is responsible for annealing in that round, before it begins annealing them. Further, we implement functionality similar to that of the Python *heapq*'s *n_largest* function, which allows us to efficiently keep track of our best k solutions that we intend to yield. Together, these limit the memory overhead of MRAnneal during annealing.

B. Example Problems

Here, we demonstrate MRAnneal's flexibility in expressing solutions to several example optimization problems. We believe that these examples show how easily users can harness the power of MRAnneal after defining only a few, short functions. Table II shows the number of lines of Python code in our implementation for four example problems. Our Max-Cut, Job Scheduling, and TSP solvers were each written in under an hour and solve abstract versions of their respective problems. Exam Scheduling is slightly larger, as it accounts for concrete constraints of scheduling exams at Swarthmore College.

TABLE II
NUMBER OF SOURCE LINES OF CODE USED TO IMPLEMENT EXAMPLE MRANNEAL SOLUTIONS.

Example Problem	Lines of Code
Job Scheduling	40
Traveling Salesman	44
Maximum Cut	63
Exam Scheduling	228

Job Scheduling. Given a set of jobs, each associated with a completion time, and a set of machines, find an assignment of jobs to machines that minimizes the total completion time. Many possible assignments of jobs to machines are randomly generated; reducers in parallel swap job assignments and calculate the completion time.

Max Cut. Given a graph whose edges are annotated with weights, partition the vertices in the graph into two sets such that the sum of the weights of the edges across the set is maximized. A MapReduce instance generates many partitions of vertices in parallel; in subsequent instances, reducers repeatedly swap the assignment of a single vertex and calculate the weight across the cut.

Traveling Salesman. Given a set of cities and costs of traveling between each pair, determine the cheapest ordering of cities such that every city is visited exactly once. Multiple reducers in parallel produce an initial ordering and randomly swap pairs of cities.

Exam Scheduling. Given a set of students and classes, determine an exam schedule such that no student is scheduled for two exams at the same time and exams are scheduled relative to each other to optimize some criteria (such as average distance between exams). Multiple reducers generate sets of compatible classes to be scheduled at the same time; these groups are then randomly ordered many times in parallel.

Our exam scheduling implementation is longer than the others because it is capable of scheduling final exams at Swarthmore College. Much of the additional code lies in the *score_solution* implementation as a result

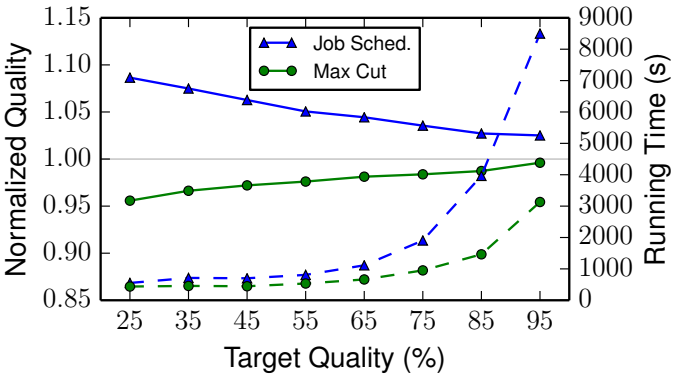

Fig. 3. Performance of MRAnneal, varying the *target_quality* parameter, for Job Scheduling and Max Cut. The left axis (solid lines) shows solution quality, normalized against the ground-truth optimal value (1.0). The right axis (dashed lines) shows running time, in seconds.

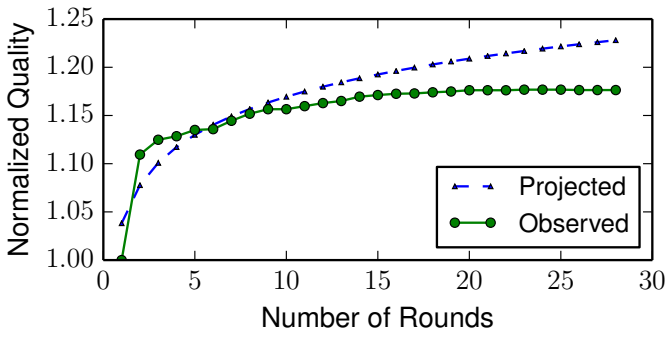

Fig. 4. An example of the solution quality estimation curve in an instance of a Max Cut problem. Note that the observed values roughly follow a logarithmic curve. We observe a similar pattern across the other optimization problem types.

of feedback from our registrar's office. For example, Swarthmore allows students with three exams in a row to reschedule one, which can be logistically difficult for professors. As a result, the scoring metric calculates determines many students have three exams in a row and penalizes accordingly. Other logistical details raised by the Registrar, such as assigning each exam an appropriately-sized room or accounting for special requests by faculty, are handled in our implementation.

IV. Evaluation

This section explores MRAnneal's ability to provide high quality solutions quickly by leveraging parallel computations. We execute our experiments across a heterogeneous cluster of high-end, commodity desktop machines running Hadoop 2.4.1 on Linux in the labs of Swarthmore College's computer science department. As ground truth for solution quality comparisons, we draw on TSPLIB [8] and the Biq Mac Library [9] for solved instances of TSP and Max-Cut.

A. Quality vs. Running time

We begin by examining the user's ability to control the trade-off between solution quality and running time, via MRAnneal's *target_quality* parameter. Recall that the *target_quality* parameter specifies the user's solution improvement goal, out of the estimated potential improvement from running additional annealing rounds. It is not a target for the percentage of the globally optimal solution's score (which is often not known).

The experiment executes two example problems, each executing across 50 machines: an instance of Max Cut from the Biq Mac collection and a randomly generated instance of Job Scheduling (assigning 500 jobs over 20 machines). We vary the *target_quality* parameter from 25% to 95% in increments of 10%. All other parameters use the defaults, as described in Table I.

Figure 3 quantifies the relationship between *target_quality*, solution quality (solid lines, left axis), and running time (dashed lines, right axis). Solution scores are normalized as a proportion of the optimal value. For the Max Cut instance, the optimal weight is published [9], and for Job Scheduling, we quantify optimal using a lower bound heuristic of the completion time as if the jobs were perfectly evenly distributed over all machines. For Max-Cut, higher weights are better, whereas for Job Scheduling, shorter completion times are preferred.

The results show the expected trends: MRAnneal achieves score solution improvements that are linearly proportional to the chosen *target_quality*, and the running time grows approximately exponentially, increasing sharply beyond 70%. Thus, the user controls the running time by setting *target_quality* around 60-70% for a fast, good solution, or around 85-95% for better solution at the cost of additional time. MRAnneal exhibits similar trends for Exam Scheduling and TSP.

B. Estimating Annealing Rounds

Next, we evaluate MRAnneal's ability to estimate the growth in solution quality during the third execution phase. Accurately projecting the growth allows MRAnneal to terminate the computation after the appropriate number of rounds to reach the user's target quality.

Figure 4 displays an example that compares the observed score results to the logarithmic curve that was estimated for an instance of the Max Cut problem. We find that two general trends observed in this graph match those of the other optimization problems:

1) We see large gains in the solution quality during the first few rounds; annealing rapidly improves upon the initial seed solutions. In subsequent rounds, the rate of growth slows.
2) Because of the large initial growth, MRAnneal tends to overestimate the magnitude of solution quality growth.

MRAnneal's overestimation leads to conservative behavior in selecting the number of additional rounds to anneal, running a few more rounds than necessary. That is, reaching the user's target quality percentage of the achievable score increase is *easier* if maximum achievable score turns out to be lower than we estimate it to be. We believe it to be preferable to err on the side of running longer, to ensure that the user's target is met.

C. Parallel Performance

We now consider the effect of increasing the number of machines, showing that users of MRAnneal can expect reasonable speedups as they increase the degree of parallelism. Figure 5 illustrates the running times for MRAnneal when applied to TSP and Exam Scheduling with a fixed target quality of 75%, varying the number of reducers annealing in parallel. For each problem, the running times shown on the Y-axis are normalized to the running time of five reducers.

Adding more mappers and more reducers can substantially decrease the running time, despite any additional initialization and communication costs across machines. Note that the benefit is not uniform across problems. In particular, problems that are more computationally intensive show a larger benefit from additional machines; as the amount of work to be done grows, the additional overhead of adding one more machine decreases in comparison to the amount of work handled by that machine. At some point, the benefit of parallelization is outweighed by the cost of additional communication, and the running time levels off. We find that as few as 10 machines is sufficient to achieve a considerable reduction in running time, however, especially for computationally intensive problems, additional machines could significantly speed up the computation. Our Exam Scheduling problem's execution time improved from 30,300 seconds (5 reducers) to 5,729 seconds (70 reducers), and TSP decreased from 2,442 seconds to 1,024 seconds.

D. Parallel Solution Diversity

In Phase 1, MRAnneal uses logarithmic regression to try to determine the number of seed solutions needed to achieve sufficient sampling of the solution space. One might be concerned that, over the course of many rounds, this diversity could decrease as a few particularly good solutions "hog" all available spots in the next round.

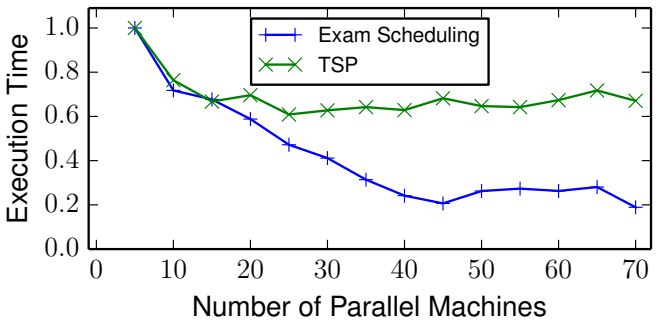

Fig. 5. Initially, increasing the number of machines results in a decrease in runtime; eventually, the runtime levels out as the benefit of increased parallelization is outweighed by higher communication costs. Computationally expensive problems show larger benefits from parallelization. Execution times are expressed as a proportion of the runtime with *five* machines.

To quantify solution diversity, we instrumented MRAnneal to tag each solution with a unique identifier during generation. As it was annealed, the identifier was propagated to all derivatives of that solution. We then ran each implemented algorithm, varying the target quality between 25% and 95% in increments of 10, for a total of 32 runs.

Our results show that diversity did not decrease substantially, even over many rounds. In each run, we requested the best five solutions. In 26 of the runs, all returned solutions were derived from different seeds, and in remaining runs, the number of returned solutions derived from the same seed only exceeded two once.

V. Related Work

NP-Complete MapReduce. Since the publication of MapReduce [4], several projects have attempted to exploit its parallelism to solve computationally expensive (e.g., NP-complete) problems. Many seek to provide exact solutions, whether by brute force [10], or more commonly, with a divide and conquer approach that solves sub-problems in parallel. This approach has been used to test graph isomorphism [11], solve K-Center and K-Median [12], compute spanning trees [13], and find the largest clique in a graph [14]. While divide and conquer techniques are effective, finding exact solutions still requires exponential time, whereas MRAnneal uses randomization, allowing users to trade off running time and solution quality.

Other projects have similarly used MapReduce to find approximate solutions to specific problems, but we are not aware of any other general framework similar to MRAnneal that allows users to easily express optimization problems such that they can be automatically broken into distributed MapReduce sub-problems. Ebrahimi implements a solver for linear programs to evaluate approximation algorithms for NP Complete problems [15]. Chierichetti et al. describe a parallel algorithm to solve Max K-Cover that performs comparably to the greedy approach [16].

Parallel Simulated Annealing. Using simulated annealing to find solutions to complex problems is not novel. Kirkpatrick et al. proposed using simulated annealing as a means to solve optimization problems [1]. Like MRAnneal, more recent work has aimed to improve the running time of simulated annealing by executing the search in parallel. Such work has demonstrated that for specific problems such as job scheduling [2] and traveling salesmen [17], executing simulated annealing in parallel has the potential to greatly improve performance. While these and others [18] describe general metaheuristic strategies that for parallelizing simulated annealing, they implement or evaluate only specific problems on a small set of local machines using specialized software for communication. MRAnneal takes advantage of commonly-deployed MapReduce software (i.e., Hadoop [5]) running on commodity infrastructure, making it easily portable and adaptable to many problems.

Others have parallelized simulated annealing for common technologies like MPI [19], [20], OpenMP [21], and GPU computation [22]. We are aware of only one other project to employ MapReduce [17]. It targets applying a genetic algorithm to the traveling salesman problem, whereas MRAnneal provides a general framework that benefits from statistics-based performance tuning.

VI. Conclusion

MRAnneal provides a straightforward and flexible framework for performing simulated annealing in parallel. By abstracting away the technicalities of MapReduce, application designers are able to focus on the details of their particular optimization problem rather than the thorny issues of distributed computation and inter-machine communication. MRAnneal's simple interface allows users to model complex problems in relatively few lines of additional code in a manner that resembles a serial implementation. Its regression-based parameter estimation enables users to easily and explicitly trade off solution quality and running time. Thus, users can not only rapidly implement solvers, but also quickly receive their desired results.

References

[1] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi *et al.*, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[2] D. J. Ram, T. Sreenivas, and K. G. Subramaniam, "Parallel Simulated Annealing Algorithms," *Journal of Parallel and Distributed Computing*, vol. 37, no. 2, pp. 207–212, 1996.

[3] A. Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained," http://www.rgoarchitects.com/Files/fallacies.pdf, 2006.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*. USENIX, 2004.

[5] Apache, "Hadoop project," https://hadoop.apache.org.

[6] "MRJob Project," https://pythonhosted.org/mrjob.

[7] Amazon, "Elastic MapReduce," http://aws.amazon.com/elasticmapreduce.

[8] G. Reinelt, "TSPLIB–A Traveling Salesman Problem Library," *ORSA Journal on Computing*, vol. 3, no. 4, 1991.

[9] A. Wiegele, "Biq Mac Library–A Collection of Max-Cut and Quadratic 0-1 Programming Instances of Medium Size," Tech. Rep., 2007.

[10] D. Visariya, "NP-Hard Problems Using Map-Reduce," Master's thesis, Rochester Institute of Technology, 2013.

[11] W. Lin, X. Xiao, and G. Ghinita, "Large-scale Frequent Subgraph Mining in MapReduce," in *ICDE*. IEEE, 2014.

[12] A. Ene, S. Im, and B. Moseley, "Fast Clustering using MapReduce," in *KDD*. ACM, 2011.

[13] A. Andoni, A. Nikolov, K. Onak, and G. Yaroslavtsev, "Parallel Algorithms for Geometric Graph Problems," in *STOC*. ACM, 2014.

[14] J. Xiang, C. Guo, and A. Aboulnaga, "Scalable Maximum Clique Computation Using MapReduce," in *ICDE*. IEEE, 2013.

[15] M. Ebrahimi, G. Weikum, and R. Gemulla, "Solving Linear Programs in MapReduce," Master's thesis, Universitat des Saarlandes, 2011.

[16] F. Chierichetti, R. Kumar, and A. Tomkins, "Max-cover in Map-Reduce," in *WWW*. ACM, 2010.

[17] A. Radenski, "Distributed Simulated Annealing with Mapreduce," in *European Conference on Applications of Evolutionary Computation*. Springer, 2012.

[18] T. G. Crainic and M. Toulouse, "Parallel meta-heuristics," in *Handbook of Metaheuristics*. Springer, 2010.

[19] D.-J. Chen, C.-Y. Lee, C.-H. Park, and P. Mendes, "Parallelizing Simulated Annealing Algorithms Based on High-Performance Computer," *Journal of Global Optimization*, vol. 39, no. 2, 2007.

[20] G. Kliewer and S. Tschoeke, "A General Parallel Simulated Annealing Library and its Application in Airline Industry," in *IPDPS*. IEEE, 2000.

[21] A. Debudaj-Grabysz and R. Rabenseifner, "Nesting OpenMP in MPI to Implement a Hybrid Communication Method of Parallel Simulated Annealing on a Cluster of SMP Nodes." in *12th European PVM/MPI Users Group Meeting*. Springer, 2005.

[22] A. Ferreiro, J. García, J. López-Salas, and C. Vázquez, "An Efficient Implementation of Parallel Simulated Annealing Algorithm on GPUs," *Journal of Global Optimization*, vol. 57, no. 3, 2013.