Solving Hard Problems with Lots of Computers

Sandy Ryza's Senior Thesis

Spring 2012

Abstract

This paper describes the use of commodity clusters in distributing techniques in combinatorial optimization. First, I use Hadoop to distribute large neighborhood search, a metaheuristic often used to achieve good solutions for scheduling and routing problems. The approach involves a sequence of rounds where processors work independently to improve a solution, with the best solutions duplicated and passed on the subsequent rounds. I test the approach using a solver for the Vehicle Routing Problem with Time Windows, on which it offers improvements with up to 1400 processors. Second, I distribute branch and bound using a master/worker architecture with worker-initiated work stealing. I apply the approach to the Traveling Salesman Problem, achieving non-negligible speedups with up to 80 machines. I pay attenton to the challenges of scaling such an approach, particularly to both the ways in which the work stealing is successful and the ways in which it can improved.

1 Introduction

Combinatorial optimization is a rich field within computer science and operations research. It seeks to provide techniques that find optimal or good solutions for problems with finite search spaces that are often intractable to search exhaustively. Since near the field's inception, research has sprung up that attempts to parallelize its algorithms. Most of early research within parallelizing techniques in combinatorial optimization focuses on multicore architectures with relatively few processors.

Sparked in part by Googles MapReduce[6] distributed computation framework, recent trends in tackling computation on large problems have focused on scaling horizontally across clusters of commodity machines. It has become common for both research institutions and companies across all industries to host large computer clusters on which to perform intense computational tasks. This thesis seeks to explore the challenges associated with parallelizing some of the techniques in combinatorial optimization on such clusters of hundreds or thousands of commodity machines.

I first cover large neighborhood search, a local search technique that uses constraint programming to explore exponentially sized neighborhoods. Approaches to parallelizing local search techniques typically run local search chains independently, paired with mechanisms for propagating good solutions across processors in order to focus computing power towards promising portions of the search space. Often genetic crossover operators are used to both retain attributes of good solutions and promote solution diversity. In this vein, I use a simple approach that runs rounds of independent large neighborhood search chains, and in between rounds discards poor solutions and replaces them with the best ones. This approach lends itself to easy parallelization on MapReduce. The intuition is essentially that a more sophisticated diversity-ensuring approach is not necessary to achieve good results, because large neighborhood search is good at escaping local minima if enough random neighborhoods can be tried.

I then cover branch and bound, a technique for achieving optimal solutions to difficult problems in combinatorial optimization. It refers to a brute force exploration of the search space as a tree, in which large subproblems can be thrown away by using sophisticated bounding techniques to determine beforehand that they do not contain the optimal solution. For reasons that will be discussed below, branch and bound is not an ideal fit for MapReduce, and thus I wrote a custom framework to distribute it, featuring a master/worker architecture which features dynamic work-stealing.

Section 2 covers large neighborhood search. 2.1 explains large neighborhood search, 2.2 explains the approach to parallelizing it in detail, and 2.3 explains the use of Hadoop. 2.4 describes the Vehicle Routing Problem with Time Windows, and the large neighborhood search solver that I wrote for it. 2.5 provides an experimental evaluation, with a discussion of the parameters and results on how well it scales to large numbers of processors.

Section 3 covers branch and bound and the challenges associated with parallelizing it. 3.1 presents the master/worker scheme and work stealing mechanism. 3.2 describes the solver that I wrote for the Traveling Salesman

Problem. 3.3 provides an experimental evaluation, with a discussion of how well the approach scales to large numbers of machines and the factors that prevent it from achieving perfect speedup.

2 Large Neighborhood Search on Hadoop

2.1 Large Neighborhood Search

Large neighborhood search (LNS) is a local search approach used to achieve good solutions to a variety of problems in combinatorial optimization. As a local search algorithm, it works by starting with an initial solution, and explores a neighborhood, or space of modifications to other solutions, for solutions that allow it to improve its objective function. At each iteration, the solution is replaced by a better solution within the previous solution's neighborhood. Large neighborhood search relies on constraint programming to explore neighborhoods that are exponential in size (to the number of variables). It works according to a destruction/construction principle. It destructs a current solution by relaxing constraints on a set of variables, and attempts to construct a better solution by using constraint programming to optimally reassign them. The technique has seen particular success in scheduling and transportation problems[8] [11]. In speeding up a solver that uses LNS, the objectives are twofold: to achieve better solutions, and to reduce the amount of time required to achieve them.

2.2 Population Large Neighborhood Search

To distribute large neighborhood search, I tried a population approach similar to the one described in [14]. In this approach, computation is divided into a sequence of rounds. At the start of each round, a solution is distributed to each of the computation units. The computation units attempt to improve their solution using a local search with a random component. At the end of each round, the k best solutions are passed on to the next round, and the remaining solutions are replaced with the best solution found so far. While in simulated annealing, we accept solutions that increase the objective function, the assumption with large neighborhood search is that we can reach the best or at least a very good solution purely through greedy moves. Because of this, I discarded the component of the original paper that preserves the solution at the end of a round distinct from the best solution found during the round.

2.2.1 Helper Neighborhoods

I use a technique particular to large neighborhood search to attempt to squeeze out additional speedup. Each iteration of LNS chooses a random set of variables to relax and re-optimize. Especially as solutions become further optimized, it becomes increasingly rare that a set of variables will improve the objective function. In parallel runs, non-conflicting sets of relaxed variables may independently provide improvements to the objective function. With the population approach outlined above, if improvements to the same solution are found in parallel, some of the improvements may be lost with solutions that are not passed on. To help alleviate this situation, I save successful relaxation sets (neighborhoods) during the search, and I pass on a list of successful neighborhoods for discarded solutions to the next round. Large neighborhood searchers in the next round try relaxing these sets of variables first in the hope that they will be more likely to provide improvements. To both avoid passing around too much data and reducing the diversity of the solutions by exploring the same neighborhoods, the ”helper neighborhoods” are limited and randomly selected. Only the first E successful neighborhoods are saved by each LNS runner. A random subset of the neighborhoods passed on from all the runners, of size H, is generated for each runner in the next round. To both reduce the amount of data transferred and reduce the exploration time, only the variables that were actually changed in the new solution are sent.

2.3 Using Hadoop

The population approach is easily parallelized using the MapReduce[6] framework. Mappers are given solutions and perform the large neighborhood searches on them, outputting their improved solutions. A single reducer collects all the solutions from the mapper, and determines which solutions (the best ones) will be written out as input to the next round of mappers. Mappers terminate not after having computed for a given length of time, but after reaching an absolute time, determined by the previous reducer. This means that if Hadoop schedules some mappers later than others, they will not hold everything up while they complete.

Using Hadoop over a custom system introduces a degree of overhead. Data between the map and reduce phases is written out to disk, data between the rounds is written out to HDFS, the jar must be sent to each mapper before each run, and a process must be started on the mapper for each run. However, despite these shortcomings for this application, I chose to use Hadoop because of its widespread usage, and techniques that work successfully on it could be useful to organizations with Hadoop clusters installed for other reasons. Results on the overhead incurred by Hadoop are explored in the experimental evaluation section.

2.4 Vehicle Routing Problem with Time Windows

The Vehicle Routing Problem with Time Windows (VRPTW) is a well-known problem in combinatorial optimization and operations research that models the problem of plotting courses for vehicles to deliver goods to a geographically diverse set of customers. A problem instance consists of a central depot, a vehicle capacity, and a set of customers, each with a location, a demand, a service time, and a time window. Vehicles are routed through a set of customers back to the depot such that a set of Hamiltonian circuits through the customer graph is generated, which each customer visited in exactly one circuit. An additional set of constraints make the problem more difficult. The summed demands of the customers visited by a single vehicle may not exceed the vehicle capacity. Each customer may only be visited during its time window, each visit takes time equal to the customers service time, and each edge in the customer graph has an associated travel time. If a vehicle arrives at a customer before the beginning of its time window, it waits until the time window begins. Conventionally the problem features a lexicographic objective function in which the number of vehicles is first minimized, and then the total travel time. Because my interest was primarily in the parallelization issues, I used a simplified version of the conventional objective function in which I only attempted to minimize the total travel time. This allowed me to avoid the complex multi-stage approaches that show up in state-of-the art literature on the problem.

2.4.1 Large Neighborhood Search for VRPTW

For my solver for the VRPTW I use the algorithm used in the paper that coined the term large neighborhood search[11]. At each step, a random set of customers is relaxed, meaning removed from the solution, and precedence constraints between all the other customers remain fixed. The set of customers to relax is chosen using a heuristic that probabilistically prefers customers that are close to each other and on the same route. Constraint programming is then used to insert the relaxed customers back in the optimal way. For variable selection, the solver chooses to insert the customer whose minimum insertion cost is the highest. The value selection heuristic selects insertion points in order of insertion cost. In order to both seek the best solutions quickly and to limit the amount of time spent any one particular set of customers, I use limited discrepancy search [7]. This search strategy only explores the search tree partially, allowing no more than a fixed number of discrepancies from the value selection heuristic.

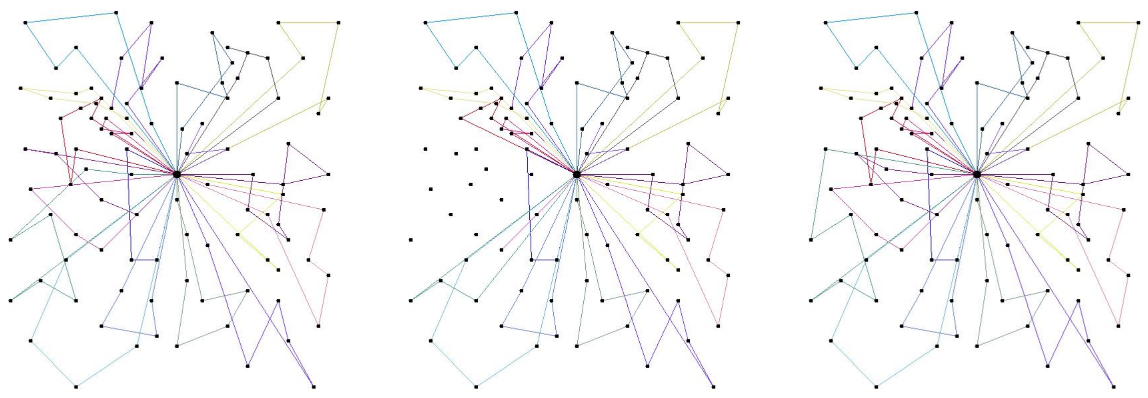

(a) Old Solution (b) Relaxed (c) Improved

Figure 1: Large Neighbohood Search

The neighborhood size is varied throughout the solving, using the method proposed in [3]. In this approach, neighborhood size starts at 1, and is incremented after a given number of consecutive neighborhoods at the size fail to improve the objective function. After reaching a maximum neighborhood size, the process restarts at 1.

Initial solutions were achieved using a greedy nearest neighbor heuristic[13], which builds routes one at a time, choosing customers based on a weighted comparison of distance, temporal proximity, and urgency. Different initial solutions were generated for the different mappers by randomly varying the weights

The solver was written in Java and contains about 1000 lines of code.

2.5 Experimental Evaluation

I tested on a 91-node Hadoop cluster running CDH3u4, generously provided by Cloudera Inc. Each machine had 16 processors for a total of 1456 slots. The master program ran the Hadoop jobs from one of the machines in the cluster. Tests were run on the extended Solomon benchmarks, a standard set of randomly generated standard instances with up to 1000 customers, split into 5 categories. While I achieved similar results for other problems I tested on, I chose to focus on of the largest instances RC110_1, which had 1000 customers. For the limited discrepancy search, 5 discrepancies were allowed. The randomness parameter for choosing neighborhoods to relax was set to 15. A maximum of 35 consecutive failed iterations were allowed before incrementing the neighborhood size. The maximum neighborhood size was set to 35. The number of rounds for each test was set so that they would complete in under 30 minutes. Unless otherwise stated, the round times were set as 70 seconds, meaning that mappers were set to stop 70 seconds from when the preceding reducer (or initializer) finished. Configurations were judged based on the best solution produced.

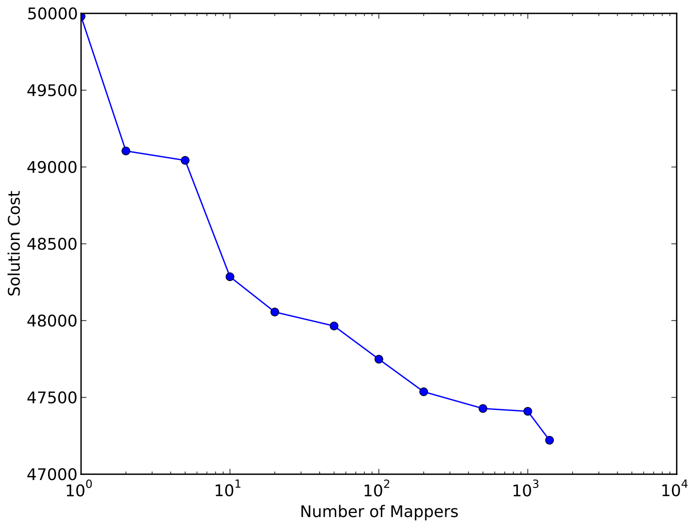


Figure 2: Scaling

Figure 2 shows how well the method scales, i.e. how solution quality is affected by the number of mappers used. Objective function values are taken after a run of 20 rounds, with 20 seconds per round, and population k=1. Each cost is averaged over 4 runs. The quality of the solution continues to increase with the number of mappers all the way to the largest number tested 1400. Note that we would not expect linear improvement to the objective function even in ideal circumstances, as, at better qualities of solution, there are fewer good solutions to available, and it gets possibly exponentially more difficult to improve the objective function.

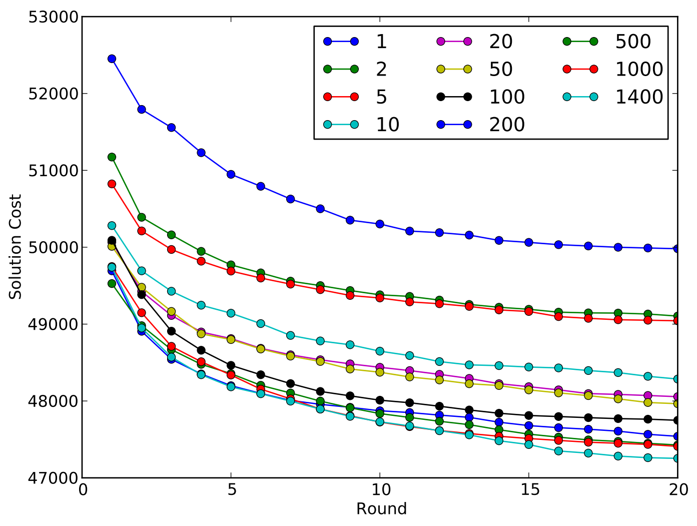


Figure 3: Best Costs as Solving Progresses

Figure 3 depicts the best solutions at the completion of each round with different numbers of mappers. Each point is averaged over 4 runs. Runs with more mappers both descend more steeply at the beginning and continue to improve in later rounds.

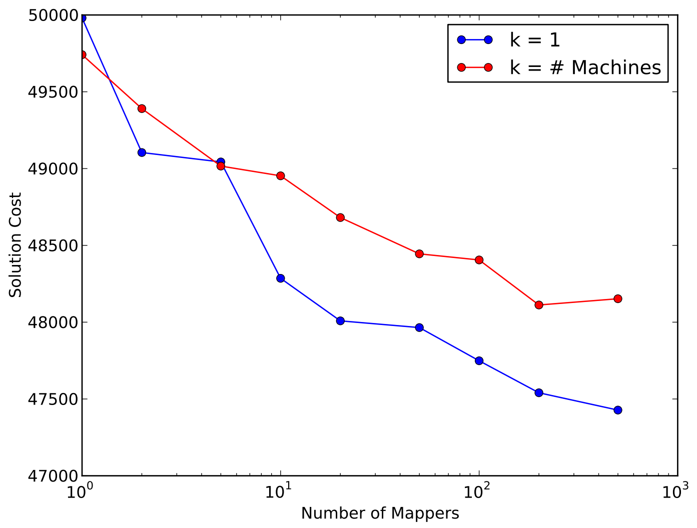


Figure 4: Comparison with Independent Solvers

Figure 4 verifies the value of best solution propagation approach by comparing it to setting the population size k equal the number of machines used, which is equivalent of running the solvers entirely independently in parallel and taking the best result at the end. All points are an average over 4 runs[1] The performance of the MapReduce approach far exceeds that of the independent approach, for which even 500 nodes does not get the objective function below 48000. This is almost always achieved using one fifth of the nodes with the MapReduce approach.

[1]Except for 200 and 500 for the k = # machines, which are only an average over 2 runs.

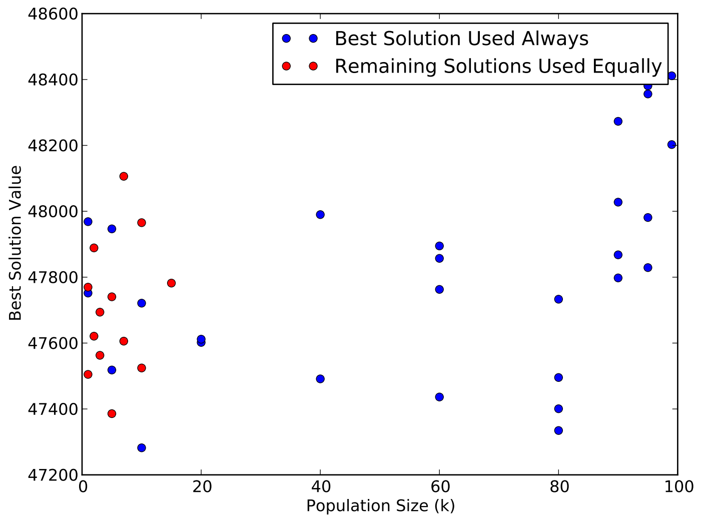


Figure 5: Population Size and Policies

To determine the impact of population size on solution quality, I fixed the number of mappers at 100 and varied the population size, k. Additionally, I evaluated two policies for replacing the worse solutions at the end of each round. In the first, all discarded solutions are replaced by the best solution. This means that with the number of machines as 7 and k=3, a round's output with solutions ranked 1 through 7 would result in a next round's input of $\{1, 2, 3, 1, 1, 1, 1\}$. In the second, the discarded solutions are replaced by the remaining solutions evenly. This means that in the same situation, the next round's input would be $\{1, 2, 3, 1, 2, 3, 1\}$. As shown in Figure 5, neither the policy nor the population size seem to appear very relevant in determining solution quality. While k=80 for policy 1 appears to have a slight advantage over the other configurations, I was unable to produce similar results with a similar ratio and policy on higher or lower numbers of mappers. As we would expect, when k gets very close to the number of machines, solution quality begins to degrade, because good solutions aren't being significantly propagated.

The irrelevance of population size is an interesting result that somewhats confirms the assumptions of the approach. A higher k should preserve a greater diversity of solutions, which should help to avoid becoming trapped in local optima. With a large neighborhood search, diversity of solutions may be less important, as a larger portion of the search space can be jumped to from any current solution. By trying a sufficient number of different relaxation sets, it is often possible to find an improving solution. Perhaps the reason that high diversity is helpful with 100 mappers but not higher numbers is that high numbers of mappers can achieve enough diversity is achieved within each round purely by exploring so many relaxation sets concurrently.

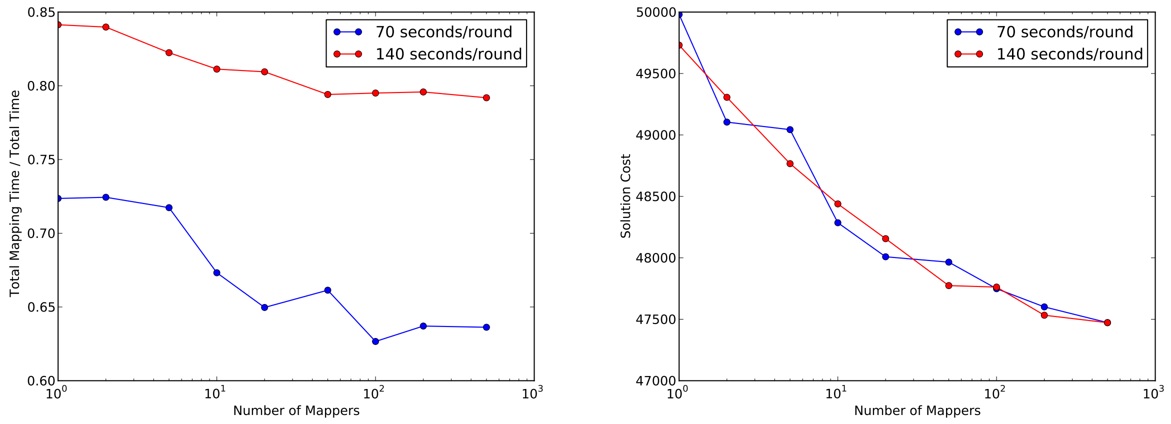
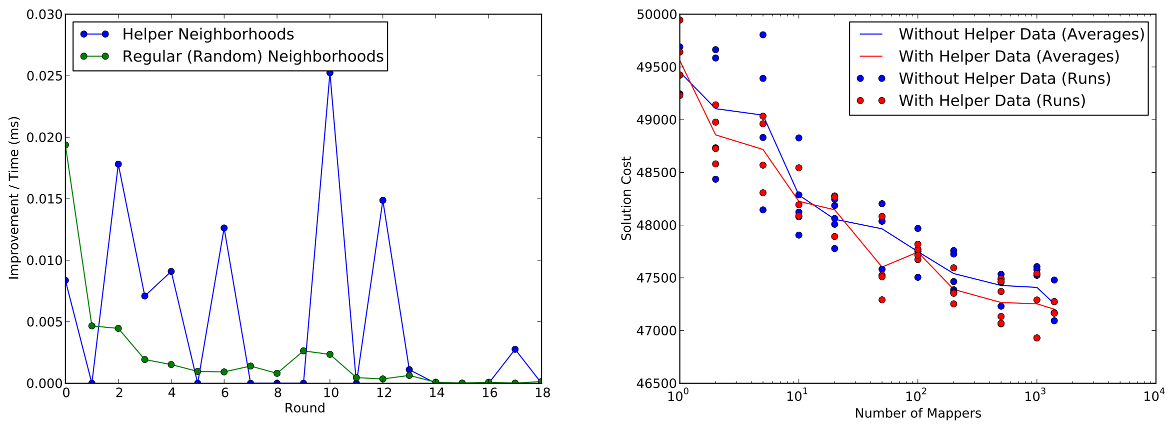


Figure 6: Round Time

Figure 6 investigates the time overhead incurred by Hadoop. It plots the efficiency of the solver at different numbers of processors, which I define as the ratio between the actual time spent solving, measured as the sum of the times spent inside all mappers of all rounds, and the total time taken multiplied by the number of mappers. Each result is averaged over 4 runs[2]. With round times doubled to 140 seconds instead of 70, the ratio is higher and degrades less steeply. However, this has little impact on actual results, as evidenced by the runs summarized in Figure 6. For the runs with 140-second rounds, the number of rounds was reduced from 20 to 11 to preserve total time. The benefit of additional time allowed for solving is counterbalanced by the benefit of propagating the current best solution more often.



(a) Helpfulness of Helper Data in a Single Run

(b) Best Solutions with and without Helper Neighborhoods

Figure 7: Helper Neighborhoods

To evaluate the benefit of passing along successful neighborhoods to the next round, I compared runs with different numbers of mappers and turned on and off helper neighborhoods (Figure 7a). Each cost is averaged over 4 runs[3]. In my implementation, helper neighborhoods make sense with a k value of 1, as successful neighborhoods lose relevance when drawn from a more diverse set of solutions. When on, up to 100 successful neighborhoods are passed on from

[2]Except for 500 for the 140 seconds, which is only an average over 2 runs.
[3]Except for 1000 and 1400 with helper neighborhoods, which are only an average over 3 runs.

the mappers to the reducers, and up to 300 are assigned to try for each mapper. The inclusion of helper neighborhoods appears to confer a slight advantage. Because the curves are close, the results from individual runs are provided. Larger numbers of helper neighborhoods than 300 (400 and 600 were tested) did not confer additional benefits. Figure 7b compares the ratio of time to objective function improvement for both regular random and helper neighborhoods, for the best solution at each round, as the solver progresses for a run using 100 mappers. The first round is excluded because no helper neighborhoods are assigned to it. In many of the rounds, time spent on helper neighborhoods is far better spent than time on regular neighborhoods.

The inclusion of helper neighborhoods, and the associated additional data passed around, did not appear to incur significant overhead. The ratio of actual time inside mappers compared to ideal time was not significantly different between runs with helper neighborhoods and runs without. The total time taken was not significantly different either.

2.6 Related Work

Perron and Shaw parallelized large neighborhood search on a network design problem, using a single machine with 4 cores.

Bartodziej, Derigs, and Vogel[2] also propose a parallel approach with large neighborhood search, using the pb.net framework and focusing on the Vehicle Routing Problem with Pickup and Delivery and Time Windows (VRPPDTW). They rely on 9 different destruction subheuristics to promote solution diversity. They focus most of their analysis on a single machine with 4 cores, but also test on a cluster with 25 machines using a single core each.

Rochat and Taillard[10] parallelize local search for the VRPTW with an approach featuring a master which holds a pool of good routes. It builds solutions out of them to assign to worker and assigns them to workers, which improve them with local search and then submit them back to be incorporated into the pool. Little attention is given to scaling or the parallel aspect.

A number of efforts have used similar multiple-round approaches to distribute local search technques with MapReduce. Radenski[9] distributed Simulated Annealing algorithms for the Traveling Salesman problem.

A multiple-round MapReduce approach has seen use with genetic algorithms on a variety of problems. Crossover operators are applied to solutions from the previous round during the reduce phase to generate solutions for the next round's mappers to work on.

3 Distributed Branch and Bound

Branch and bound refers to the approach of exploring a search space exhaustively as a tree. At each node in the tree, variables are fixed, so the children of each node are subproblems and each leaf is a possible solution. Bounding techniques avoid a brute force exploration of entire space by proving entire subtrees unable to contain the optimal solution. At first glance, parallelizing branch and bound seems trivial. Different parts of the search tree can be explored independently with no information beyond the cost of the current optimal solution. The difficulty lies in distributing load, as good bounding algorithms can allow quick discovery that the entire subtree allocated to a particular machine is worthless. Branch and bound is thus not amenable to static partitioning, as required by distributed computation frameworks such as MapReduce, because it is impossible to predict the amount of work required on a subtree before actually carrying it out. An uneven distribution of work leaves processors idle. Thus, an approach that allows for dynamic reassignment of work is required.

3.1 Master/Worker Scheme

The system features a single master and numerous workers, each running on a separate machine. The master begins the computation with a breadth-first search on the search tree to generate work for the workers. Having generated enough subtrees, it serializes them and distributes them to the workers. Workers work on the tree assigned to them until they have exhausted it, and then initiate the work stealing procedure to get more work, continuing until a work stealing response reports that no more work remains. The frontier of the search tree is stored explicitly on the worker nodes so that multiple threads can work on it concurrently. The search is terminated when no workers have work nodes remaining, that is they have all issued work stealing requests to the master and are in a state of waiting for it to respond.

The Thrift [12] RPC and serialization framework is used for all communication between nodes.

3.1.1 Upper Bound Propagation

Each time a node discovers a solution that is better than the best it knows about so far, it sends the value of this solution to the master. The master then broadcasts this value to the other vassal nodes so they can use it as an upper bound.

3.1.2 Work Stealing

The system uses a centralized work stealing scheme in which workers submit stealing requests to the master, who fetches the work from other workers and passes it back. As a simple measure to balance load, the master cycles through the workers it asks for work from, skipping any that are currently attempting to steal work. In rare cases, the master may go to worker who has just run out of work, and it will have to try another. When asked for work, a worker donates the top node in its stored frontier of the search tree. To avoid passing work around that would take longer to transfer than to complete, a worker wont donate a tree node that is closer than a given number of nodes to the bottom of the tree. To delay work stealing, a number of starter search nodes are not initially distributed by the master, and are given out in response to the first work requests.

3.2 Branch and Bound for the Traveling Salesman Problem

My solver for the traveling salesman problem uses a simple depth first search to explore the search tree. It prunes search subtrees based on a number of criteria. First, a partial solutions is discarded if it contains any edges that cross other edges. Then, it is discarded if it can be improved with a 2-opt or 3-opt move. Lastly the Held and Karp one-tree bound is used. This bounding relaxation technique constructs a modified version of a minimum spanning tree on the remaining un-fixed nodes, using an iterative scheme that assigns weights to nodes/edges to force the spanning tree towards a tighter bound. The Held and Karp weights for a search node are passed down to its children, but not serialized along with the node data when transferred over the network.

3.3 Experimental Evaluation

I tested using 80 machines on the Brown compute grid, using two cores for each worker. The machines were split between the cluster's "ang" and "dblade" machines. The "ang" machines are equipped with Operon 250 processors and have 12GB of RAM. The "dblade" machines are Dell PowerEdge 1855 blade systems, with Xeon 2.8 processors and 4GB of RAM. The master ran on a separate machine outside the grid. Tests were conducted on two standard TSP instances, eil51, and eil76, using 10, 20, 40, 50, 60, 70, and 80 machines at a time. To both keep running times in check and allow for a closer comparison, only the first 50 cities in the instances were used. Distances were rounded to the nearest integer. An initial upper bound of 500 was used in all cases. Nodes deeper than 6 away from the bottom of the tree (i.e. containing less than or equal to 6 unfixed cities) were not donated.

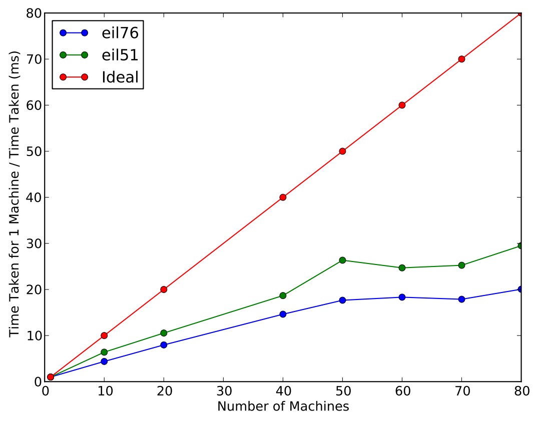
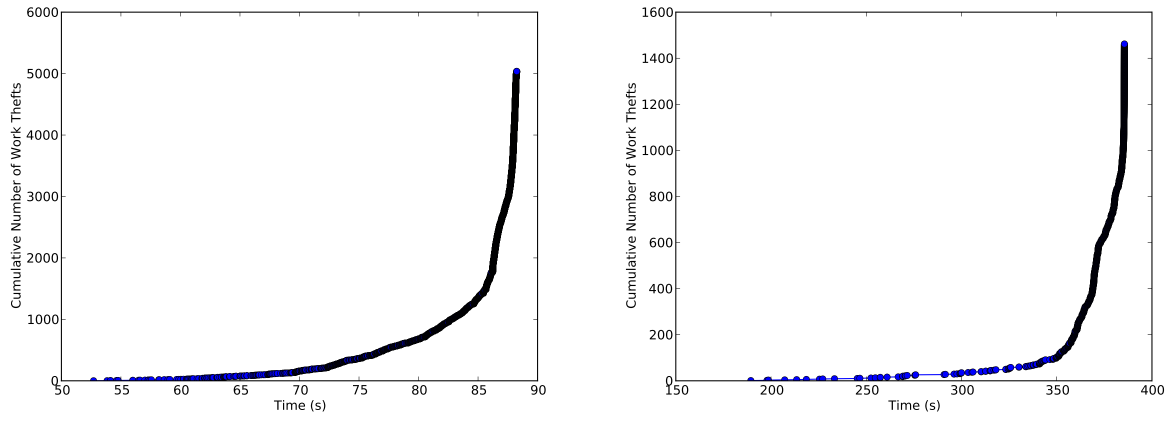


Figure 8: Scaling for eil76 and eil51. Each point averaged over 3 runs.

Figure 8 shows how the speed of the solver increases with the number of machines used. The speedup differs on the different instances. On eil51, steady speedup is achieved up to 80 nodes. On eil76, the speedup is much slower, and begins to degrade with around 50 machines. For the same number of machines, eil76 completes faster than eil51 in all cases, taking 27 minutes vs. 78 minutes for 1 machine. Because the same number of cities are used, the size of the unpruned search trees are identical. This means that more or larger subtrees are pruned for eil76, implying that the subtrees distributed to different nodes are less likely to be even. This leads to more work stealing, which slows down the process. Implicit in this conclusion is that the inclusion of more powerful bounding techniques, such as the comb cuts used in state of the art TSP solvers, might degrade performance for similar reasons.

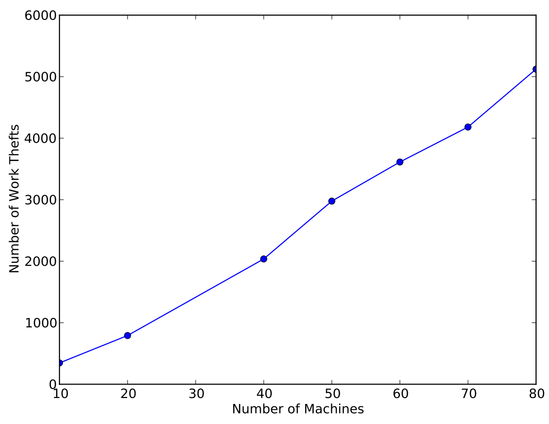


(a) eil76 with 80 machines

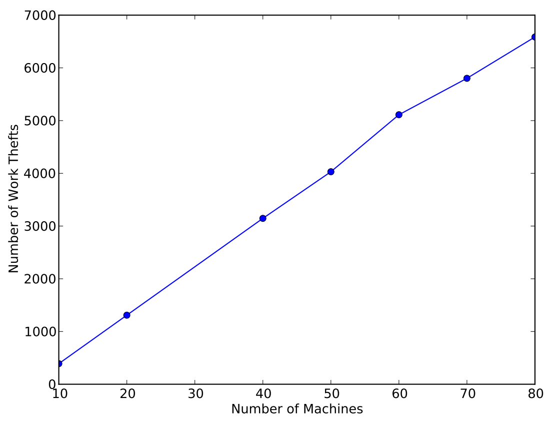
(b) eil51 with 20 machines

Figure 9: Work Thefts as Time Progresses

The primary obstacle to scaling lies in the master as a bottleneck. Delay is experienced when worker nodes run out of nodes and must wait for the master to return more to them. Figure 9 shows a count of work stealing requests to the master as the search progresses for two sample runs. Requests that are answered with the starter nodes stored at the master are not shown. At the beginning of the solving, the master receives sparse requests for work - for the run with eil51 with 20 machines not, a work theft is not carried out until the halfway through the run. As workers begin to run out of the original work they were assigned, the work stealing picks up. Near the end, work becomes scarce, and the search nodes stolen are farther down the tree, meaning that they will take less time to be exhausted. It is possible that the same work is transferred between nodes multiple times. Finally, a flurry of work steal requests bombards the master in the last few seconds of the search.



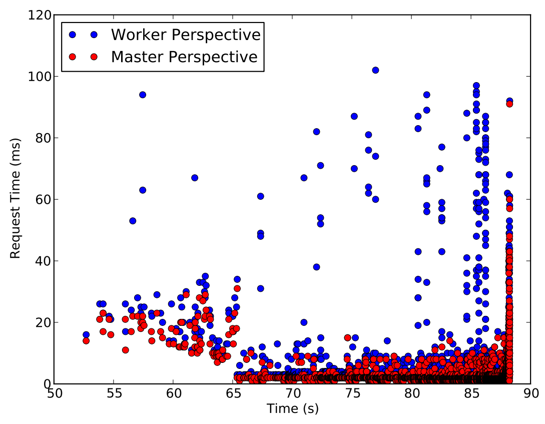
(a) eil76



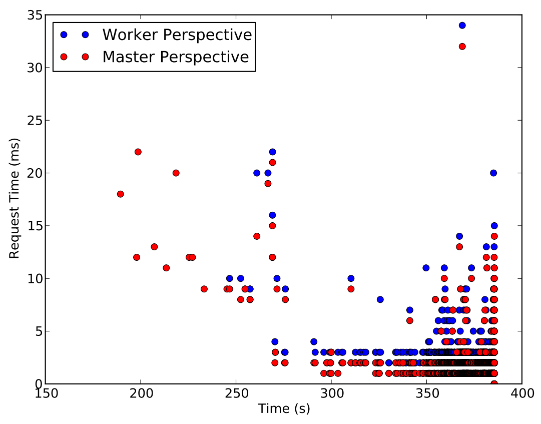
(b) eil51

Figure 10: Total Work Thefts by Number of Machines

Figure 10 tracks how the amount of work stealing scales with the number of machines used. The number of thefts appears to increase linearly with the number of machines. This is a good result, as it means that feedbacks don't incite a larger scarcity of work than we would expect as the number of machines climbs. The number of times in which the master goes out to a worker and it has no work did not increase with time or number of machines for any of the analyzed runs.



(a) eil76 with 80 machines



(b) eil51 with 20 machines

Figure 11: Work Theft Latencies as Time Progresses

Figure 11 helps to illuminate the effect of increased work stealing requests to the master. It displays the time taken by work stealing requests as the search progresses. The blue dots represent the time taken between when a worker issues a steal request and when it receives its response from the master. The red dots represent the time taken between when the master receives a steal request and when it sends back its response. Note that this includes time spent waiting on workers to donate work. The difference between them serves as an indicator of the impact of bandwidth as a bottleneck. The right shows a healthier run with eil51 and only 20 machines. The same amount of time is taken by both client and server at the beginning, implying negligible transport delay. As time progresses, thefts pick up, and bandwidth becomes more scarce, the differences between worker time and master time increase. The left shows a less healthy run on eil76 with 80 machines. In this run, latencies from the worker side are much higher than latencies from the master side. The vertical columns indicate points of blockage in which a number of workers are waiting for a response. Note that these graphs are based on absolute times logged at the different machines, and thus are susceptible to clock skew. I don't entirely understand why latencies are a little higher on both graphs earlier on. Perhaps it has something to do with warming up code paths with the JVM?

3.4 Future Work

I considered a number of approaches to improving the system. The process used to choose which worker to steal work from is not very sophisticated, and is oblivious to some easily available data that might act as an indicator of which workers have the most spare work. The master could build a picture of the most burdened nodes by incorporating information on the depths of the nodes transferred, as well as the upper bounds discovered for them using the bounding techniques. A more even distribution of work could reduce the number of work stealing requests, which would improve the overall efficiency of the system.

Second, the master could store work throughout the search. When it would steal work from a worker in response to a work stealing request, it could keep some of the work for itself by splitting/exploring the search tree. This would allow it to serve future work requests without having to go out to other workers.

Third, workers could initiate work stealing before they've entirely run out of work, so that their CPUs would not be idle while they're waiting for a response.

Last, a multiple or hiererarchical master approach could split the master's burden. Each master could be responsible for a subset of the workers, and only when its subset was running out of work would it need to communicate with the other. Alternatively, workers could direct their stealing requests towards a random master, and it would be able to go out to any of the workers for work. In this case, masters would not need to communicate other than to decide when computation had finished.

While I didn't have time to implement and test these approaches, my work provides a starting point for evaluating the efficacy of improvements.

3.5 Related Work

Neither the application of computer clusters to branch and bound nor the architecture is particularly novel. [5] offers a good overview of existing systems and techniques.

Xie and Davenport[15] focus on parallel branch and bound for the IBM Blue Gene supercomputer. In their architecture, workers actively send subtrees to the master when they perceive that they are working on a large subtree. They test on scheduling problems, and achieve improvements up to 256 cores.

DryadOpt[4] takes a different approach, distributing branch and bound on Dryad, a generalized Hadoop-like system developed for research at Microsoft. Solving is partitioned into rounds. In each round, machines work independently on sets of nodes assigned to them. In between rounds, nodes are shuffled to ensure an even distribution. They apply their system to the Steiner Tree Problem and achieve near-linear speedup.

A similar study from a systems perspective is provided in Aida and Osumi in [1]. They evaluate a hierarchical master approach in which multiple clusters communicate with a single master, and apply their system to the Bilinear Matrix Inequality Eigenvalue Problem. They compare load balancing strategies, but omit a study of work stealing patterns and results on how their system scales.

4 Conclusion

In this paper I proposed two approaches to distributing two approaches in combinatorial optimization across commodity clusters. In the first, a multi-round population intensification approach is used to distribute large neighborhood search with MapReduce. Experimental results on Hadoop verify that proposed approach is effective at leveraging large clusters to aid large neighborhood search solvers. Solution quality produced in a given amount of time increases for up to 1400 processors, and outperforms parallelization using purely independent chains. Most variation in values of the population size parameter k have insignificant impact on results. The same goes for round length. The forwarding of helper neighborhoods, sets of variables to relax that achieved improvements in discarded solutions, has a modest but discernible benefit. The approach is applicable to any local search with a random component, although perhaps particularly well suited to large neighborhood search for its ability to break through local minima within a search. The helper neighborhoods technique can be applied to any parallel large neighbohood search solver.

In the second, a master/worker architecture with work stealing is used to distribute branch and bound. Experimental results verified that the proposed approach was successful at distributing branch and bound across commodity machines, scaling well up to 80 machines. Scaling results differed significantly for different instances, with those whose search spaces were more easily pruned experiencing less speedup. Analysis revealed the increase in transport delays at high volumes of work stealing near the end of the search, but also did not rule out CPU as a bottleneck.

References

[1] K. Aida and T. Osumi. A Case Study in Running a Parallel Branch and Bound Application on the Grid. IEEE Computer Society, 2005.

[2] P. Bartodziej, U. Derigs, and U. Vogel. On the Potentials of Parallelizing Large Neighbourhood Search for Rich Vehicle Routing Problems. *Lecture Notes in Computer Science*, 6073, 2010.

[3] R. Bent and P. van Hentenryck. A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows, 2004.

[4] M. Budiu, D. Delling, and R. F. Werneck. DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines. pages 1278–1289. IEEE International, 2011.

[5] T. Crainic, B. Le Cun, and C. Roucairol. *Parallel Branch and Bound Algorithms*, chapter 1. John Wiley and Sons, Inc., 2006.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] W. D. Harvey and M. L. Ginsburg. Limited Discrepancy Search.

[8] D. Pacino and P. van Hentenryck. Large Neighborhood Search and Adaptive Randomized Decompositions for Flexible Jobshop Scheduling. pages 1997–2002, 2011.

[9] A. Radenski. Distributed simulated annealing with mapreduce. *Lecture Notes in Computer Science*, 7284:466–476, 2012.

[10] Y. Rochat and E. D. Taillard. Probabilistic Diversification and Intensification in Local Search for Vehicle Routing.

[11] P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. *Lecture Notes in Computer Science*, 1520(5):417–431, 1998.

[12] M. Slee, A. Agarwal, and M Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. 2007.

[13] M. Solomon. Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints. *Operations Research*, 35(2):254–264, 1987.

[14] P. Van Hentenryck and Y. Vergados. Population Based Simulated Annealing for Traveling Tournaments. *AAAI Press*, (1):267–272, 2007.

[15] F. Xie and A. Davenport. Massively Parallel Constraint Programming for Supercomputers: Challenges and Initial Results. pages 334–338, 2010.