Interacting Agents for Local Search

Deborah K. Weisser
dweisser@cs.cmu.edu

Abstract

Optimization problems are prevalent in many application domains, including circuit layout, load balancing, and job scheduling. Many of these problems are NP-hard, and algorithms for finding exact solutions take exponential time. Approximation algorithms have been developed to find near-optimal solutions in polynomial time. One class of general approximation algorithms is local search, which explores the space of solutions by moving from one state to another according to some neighborhood structure. Local search tends to find good solutions but is generally slow. Its inherent problem is that it searches only locally for a solution that is globally optimal.

We attempt to overcome this shortcoming by using a more global approach to explore many parts of the state space in a controlled way simultaneously by using cooperating agents. The multiple agents may be run on different processors on a parallel machine, or they may be run serially, where agents take turns making moves. Each agent performs a local search algorithm using its own copy of the state space. Periodically the agents interact, and better states are distributed among the agents. In a parallel setting, communication is asynchronous, and efficiency is quite high.

We apply our techniques to two optimization problems: TSP and graph bisection. Our algorithm consistently finds good solutions on many instances of these problems.

1 Introduction

Since optimization problems are NP-hard, solving even small instances directly may take an inordinate amount of time. Approximation schemes, such as local search, are frequently used to find near-optimal solutions in a polynomial amount of time. When the problem size becomes large, even local search takes a long time to find an acceptable solution.

In this paper, we present a multi-agent method for performing local search. In our approach, each agent can be thought of as a processor. Indeed, we have implemented our technique as a parallel program using virtual processors, i.e. as a set of threads that may be run on one or more physical processors. The algorithm can be applied to any local search technique on any optimization problem.

The algorithm works by running multiple copies of the problem independently on different agents. Periodically, the agents interact (asynchronously), and agents that are not doing well copy states from agents that are doing better. We have two versions of the program: One in which the number of agents is constant and one in which the number of agents changes dynamically to minimize the total amount of work performed across all agents. Our algorithms have been implemented to run serially and in parallel.

Our results are promising in terms of solution quality. We achieve better quality solutions to many TSP problems than Lin-Kernighan. In terms of solution quality for the problem of graph bisection, our algorithm generally performs as well or better than spectral methods, Kernighan-Lin and Multi-level Kernighan-Lin.

We discuss previous work related to local search later in this section. In In Section 2, we introduce the optimization problems that we use as test cases for our algorithm and discuss problem-specific previous work. We describe our overall algorithm and problem-specific move operators in Section 3. We present experimental results, including comparisons to other problem-specific approximation algorithms, in Section 4. We discuss our conclusions in Section 5.

1.1 Local Search: Related Work

Local search works by moving from state to state according to some move operator until a stopping criterion is met (in the realm of optimization problems, states correspond to proposed solutions). Simulated annealing [12], the Metropolis algorithm [18], tabu search [6] [8], and hill-climbing [24] are several commonly used local search methods. Simulated annealing always accepts transitions to lower cost states and accepts transitions to higher cost states probabilistically, where the probability of accepting a move to a higher cost state decreases over time. The Metropolis algorithm is a variation of simulated annealing in which the probability of moving to a higher cost state is kept constant. Hill-climbing is another variation in which only cost-decreasing moves are accepted. Other local search techniques include genetic algorithms and tabu search [21].

There are many multi-agent approaches to local search. Some involve all the agents working on one global state [7], while in others each agent has its own copy of the data. Two examples of the latter approach are the independent runs method [17, 22] and the systolic method [1].

The "Go With The Winners" algorithm of Aldous and Vazirani [2] considers the course of agents (called particles in their paper) traversing a tree starting from the root. In one phase, the agents move independently from one level of the tree to the next. At each phase, all agents are at the same level in the tree. Agents that are at leaf nodes are redistributed among agents at non-leaf nodes. Their algorithm finds deep nodes in the tree in time polynomial in h, the height of the tree, and κ, a measure of the imbalance of the tree. Simulated annealing can be mapped to "Go with the winners" by considering successive levels in the tree to correspond to different successively lower temperatures. All agents are at the same level in the tree corresponds to the agents all performing simulated annealing at the same temperature. Their algorithm does not describe an actual implementation.

Our algorithm can be considered as one practical implementation of the "Go With The Winners" algorithm. In our implementation, the nodes of the tree are states, and the traversal of agents from one level to the next is represented by many moves of an agent in the state space.

2 Problem Instances

The problems we consider are graph bisection and TSP. They have both been the focus of much attention, and each has its own problem-specific approximation algorithms.

2.1 Graph Bisection

The goal of graph bisection is to divide a the nodes N of a graph $G = (N, E)$ into equal-sized sets of nodes S and $\overline{S}$ such that the number of edges between each part, or *cut size*, is minimized.

Graph partitioning arises in a variety of settings. Perhaps the most prevalent is that of partitioning work for parallel computation. Some typical problems that, when computed in parallel, rely on good partitioning of data to run efficiently are sparse matrix-vector multiplication, explicit methods for solving PDE's, and Gaussian elimination. In addition, some VLSI layout problems can be solved by using a graph partitioning approach.

In the following paragraphs, we discuss several approaches that have been implemented. We compare results of our algorithm to many of these in Section 4.

One method that provides provable bounds on inexact bisections was designed by Leighton and Rao [14] in conjunction with their results on multicommodity max-flow min-cut theorems [13]. Their polynomial-time algorithm for computing max-flow can be converted to polynomial-time algorithm for computing approximate min-cut. It works by repeatedly partitioning the graph along an approximate min-cut and removing the smaller set of nodes until the remaining graph is close to half the size of the original graph.

Spectral methods [20] use the second smallest eigenvalue of the Laplacian matrix of a graph to find a separator. It is a method that finds good solutions in practice, albeit not quickly, and has proven bounds on its effectiveness for meshes that can be embedded in lower dimensional objects. The bottleneck in Spectral partitioning is computing λ_2. It can be done in $O(|N^3|)$ time or approximated in roughly $O((|E|^2)$ time. There is no guarantee on solution quality. This method finds good solutions in practice, but it is slow.

Geometric methods [23, 5, 19] use coordinate information to find a line, plane, circle, etc. to partition nodes of the graph. They work under the assumption that edges tend to lie between nearby vertices. This is true in the case of meshes, for example. One example of this method is the "random circles" of algorithm [5]. In this algorithm, a vertex separator is found with provable bounds on the partition it produces. Roughly speaking, the nodes of the graph are projected onto a carefully chosen $(d+1)$-dimensional sphere, where the graph is in d-dimensions. The sphere is bisected by a random plane, which partitions the vertices into three sets, N_1 and N_2 on either side of the circle, and N_s on the circle. The sphere can be chosen so that with high probability $|N_1|$ and $|N_2| \leq n * \frac{d+1}{d+2}$ nodes, and $|N_s|$ is not too big. While there is not an exact bound on the running time, except that it is polynomial, it runs quickly in practice. There is no guarantee on solution quality for general graphs.

The Kernighan-Lin algorithm [11] works as follows: The nodes of graph G are partitioned into equal-sized sets, *red* and *blue*. The goal is to minimize the number of edges between *red* and *blue*. A phase consists of a series of moves. At the beginning of each phase, all nodes in the graph are unmarked. In each move, an unmarked node a is chosen from *red* at random. It is exchanged with the best unmarked node b from *blue*, i.e. b is chosen from unmarked nodes in *blue* to minimize the cost of G after exchanging a and b. Each iteration of Kernighan-Lin takes $O(|N|^3)$ time. Feducci and Mathias implemented a version of Kernighan-Lin in which each iteration takes time $O(|E|)$. In practice, Kernighan-Lin terminates after 2 – 4 rather expensive iterations. It is outperformed in speed and solution quality by multilevel Kernighan-Lin.

Multilevel graph partitioning algorithms work by clustering vertices, performing a partitioning algorithm on the smaller graph of the vertex clusters. The vertices are clustered by finding maximal matchings of the edges. Each pair of vertices connected by an edge in the matching is clustered. The partitioning algorithm is performed on the smaller graph of vertex clusters, and the resulting partition used as an initial partition for the full graph. This process can be performed recursively on more than two levels. Although it has no proven bounds, in practice Multilevel Kernighan-Lin algorithms [9, 10] (e.g. the METIS and Chaco packages) produce very good partitions extremely quickly. The multilevel approach has also been applied to spectral partitioning [3]. Other partitioning algorithms, including ours, could also be run in a multilevel fashion.

2.2 TSP

In the TSP problem, we are given an instance $I = (N, d(i,j)), |N| = n$ of cities and distances between them. The objective is to find a permutation of all the cities, $T = (t_1, t_2, \ldots, t_n)$ that minimizes a tour starting and ending at the first city in T and visits all the cities in the permutation order, i.e. such that $\sum_{i=1}^{n-1} d(t_i, t_{i+1}) + d(t_n, t_1)$ is minimized.

This problem has been the object of a great deal of interest and effort. There are approximation algorithms that perform well in practice, although results have been proven that give upper bounds on the worst case performance of all approximation algorithms. The Lin-Kernighan algorithm has long been considered to be the best algorithm for solving TSP. It, along with other TSP move operators, is described in conjunction with our move operator in Section 3.1.2. We compare results of our algorithm Lin-Kernighan in Section 4.

Local Search	Graph Bisection	TSP
Input	A graph	A list of cities and a distance function
State	A bisection of nodes into *red* and *blue* sets	A tour of cities
Move operator	Exchange a pair of *red* and *blue* nodes	Reverse a segment of the tour
Cost	Cut size	Tour length

Table 1: Our mapping of local search to the optimization problems. The way in which a move is selected is discussed in Section 3.1.

3 Interacting Agents for Local Search

In this section, we describe our algorithm in some detail. In Section 3.1, we define the move operator our local search technique uses for state transitions for both of the optimization problems we explore. In Section 3.2, we present our technique for using a fixed number of agents to explore the state space in the local search algorithm, and in Section 3.3 we present a variation of the fixed-agent algorithm in which the number of agents changes dynamically over time.

3.1 Move Operators

Local search uses move operators to move from one state to another. Move operations are problem-specific and determine the neighborhood structure of the state space for the problem. We can think of local search as being superimposed upon this neighborhood structure, as local search uses the problem-specific move operator to generate moves.

Move operators may employ heuristics, random choice, or a combination of the two. Different move operators for the same problem may differ from each other in how fast they are (e.g. Lin-Kernighan versus 2-OPT for TSP) and how "connected" the state space is (e.g. the longest path between two states, the probabilities of ever reaching particular states).

Our algorithm requires a quick move operator, or rather it requires that different agents are likely to quickly diverge from the same state to explore the state space independently. In this section, we examine specific move operators for each problem and discuss our selection of move operators.

3.1.1 Graph Bisection

We are given a graph $G = (N, E)$, where N is the set of nodes, E is the set of edges, and we let $n = |N|$ and $e = |E|$. The nodes are always partitioned into two equal-sized sets, which we denote as *red* and *blue*. One common move operator for graph partitioning is a simple node exchange, where moves are proposed by selecting one node at random from *red* and another from *blue*, and exchanging them.

We now present our move operator. Like Kernighan-Lin, each node is in a proposed move once in each block of $\frac{n}{2}$ moves. The requirement that each pair selected be the best possible is relaxed in our move operator, however. The number of steps for each move is O(1).

We define the *frontier* of the partition to be the nodes that contain edges that cross the partition. Let F_b denote the frontier nodes in *blue*, and let F_r denote the frontier nodes in *red*. In each move, we alternate, picking a node at random from F_b (F_r) to exchange with a node from *red* (*blue*).

We add one more restriction by requiring that a move be attempted for each node once in each block of $\frac{n}{2}$ moves attempts. We have found that this requirement significantly improved both the running time and the quality of the final partition.

Our move operator:

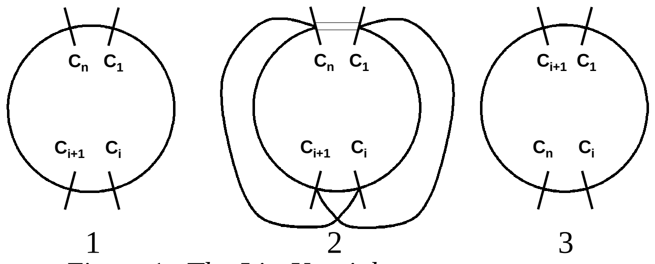

Figure 1: *The Lin-Kernighan move operator*

1. Create sets F_b and F_r
2. Unmark all nodes
3. Repeat until no more moves can be made ($n/4$ times):
   (a) Select an unmarked node $n_1 \in F_b$ (if all nodes in F_b are marked, select an unmarked node from *blue*).
   (b) Select an unmarked node $n_2 \in$ *red*.
   (c) Mark n_1 and n_2.
   (d) Proposed move: Exchange n_1 and n_2
   (e) Repeat the first four steps, now selecting nodes from F_r and *blue*.

3.1.2 TSP

The input to a TSP instance is a graph $G = (N, E, D)$, where N is the set of nodes, E is the set of edges, and $D = (n_1 \in N, n_2 \in N, distance)$ is the set of distances between nodes. The goal is to find a minimum length tour that visits each node exactly once.

Many move operators work by taking a “slice” of consecutive cities out of a tour and reinserting it somewhere else. One example of this type of move operator is 2-opt, which works by selecting two nodes at random, n_1 and n_2, and reversing the slice of the tour beginning and ending with n_1 and n_2. The 3-opt move operator is similar, but it chooses three nodes, n_1, n_2, and n_3, takes the slice surrounded by n_1 and n_2, and inserts it after n_3.

A very effective algorithm for solving TSP is Lin-Kernighan [16]. Lin-Kernighan finds good solutions to TSP problems, usually in 2-4 $O(n^2)$ time iterations. Each iteration of the Lin-Kernighan algorithm is a series of carefully chosen 2-opt moves. At the beginning of an iteration, one city is designated as the starting point. All moves in an iteration involve reversing a segment that ends at the last city in the tour (See Figure 1.) Each city has a neighbor list of it k closest neighbor cities. This list is used in selecting where to break the tour.

Our algorithm relies on a move operator that can be performed quickly so that agents starting at the same state can quickly move to different areas of the state space. We experimented with 2-opt, 3-opt, and some variations, and did not observe any difference in solution quality for a fixed number of moves. We chose to use 2-opt because it is the fastest.

3.2 The Fixed-Agent Algorithm

In this section, we describe our fixed-agent algorithm in detail. In this algorithm, a fixed number of agents are simultaneously executing separate copies of the problem, where we choose the number of agents to be powers of 2. This section addresses the interaction between the agents.

In our exploration of multi-agent techniques for local search, we sought a two-way balance between allowing the agents to explore the state space independently and propagating good states to multiple agents.

Toward that goal, our method is asynchronous, and each agent works independently on its own copy of the problem. Agents periodically interact, and "good" states are copied to other agents. The "grain size" of the algorithm, in this case the number of moves each agent makes before propagating its best state, can be adjusted to suit the problem.

3.2.1 Parameters for Multi-Agent Approach

In addition to the number of agents, a multi-agent algorithm has other parameters that control the interaction between agents.

- *neighborhood_size*: Each agent as a set neighbors, *neighborhood_size* agents which are chosen at random.
- *step_size*: Every agent communicates its best state information with its *neighborhood_size* neighbors after each completion of *step_size* move attempts.

The neighborhood size reflects the degree of the underlying communication network between agents. If the neighborhood size is large, then a "good" state will quickly be copied by all the agents. The tradeoff is that with a large number of neighbors, a single agent with a very good state will spend a lot of time copying it to other agents.

If we set *neighborhood_size* to be $\log k$, where k is the number of agents, the expected amount of time for the best state to reach all the agents is less than $(\log_{\frac{\log k}{2}} \frac{k}{2} + 2) * step\_size$:

Let g be the lowest cost state among all the agents. We want to determine the number of moves required for g to be distributed to all k agents.
The number of moves required for g to reach the first $\frac{k}{2}$ agents:

$$\begin{aligned} &\text{prob[neighbor of a processor with } g \text{ does not yet have } g] > \frac{1}{2} \\ \Rightarrow\quad &\text{E[number of neighbors to receive } g \text{ for the first time for each agent sending } g] > \frac{\log k}{2} \\ \Rightarrow\quad &\text{E[number of moves for } \geq \frac{k}{2} \text{ agents to receive } g] < \log_{\frac{\log k}{2}} \frac{k}{2} \end{aligned}$$

The number of moves required for g to reach the last $\frac{k}{2}$ agents:

$$\begin{aligned} &\text{At least } \frac{k}{2} \text{ agents distribute } g \text{ to at most } \frac{1}{2} \text{ agents} \\ \Rightarrow\quad &\text{(using a coupon-collector argument) E[number of moves to reach all agents]} \leq 2 \\ \Rightarrow\quad &\text{E[number of moves to distribute } g] \leq 2 * \frac{k}{2} \log k = k \log k \end{aligned}$$

Thus E[total number of moves to distribute g to k agents] $< (\log_{\frac{\log k}{2}} \frac{k}{2} + 2) * s$.

There is a similar tradeoff with *step_size*. As *step_size* becomes large, the algorithm approaches a set of independent runs, and only one agent ends up contributing to finding the best solution. If *step_size* is small, the algorithm may lose some of the value of multiple agents, since the agents will often be working on related states. The setting of *step_size* is completely problem dependent, however.
The internal variables of each agent:

- Input problem
- Current state and its cost
- Best state encountered so far by this agent (Its cost is *local_best_cost*. Cost of previous best state is *old_local_best_cost*.)

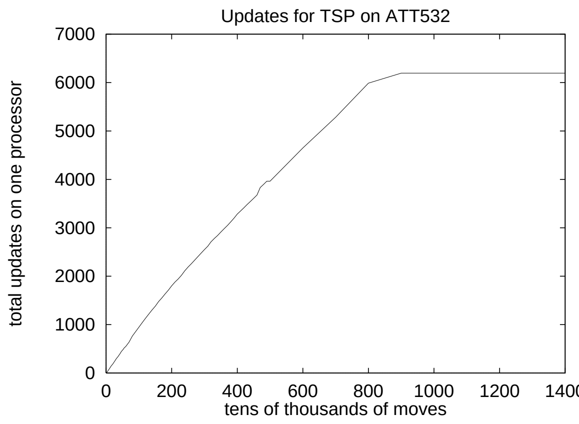

Figure 2: *Total updates versus number of moves on a tsp problem*

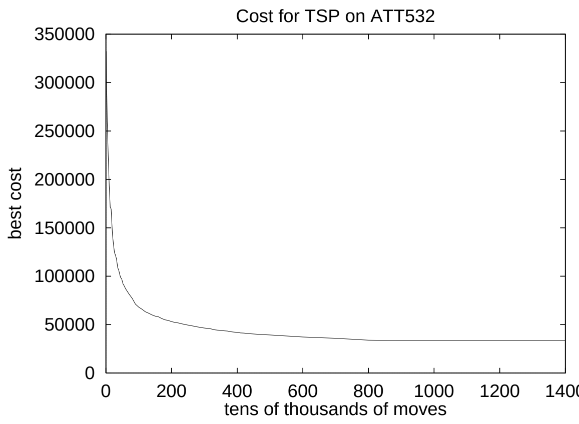

Figure 3: *Best cost versus number of moves on a tsp problem*

- *nbr_best_cost*, cost of best state of a neighboring agent
- *best_nbr_id*, id of agent that owns state with cost *nbr_best_cost*

Each agent performs the following algorithm:

1. Repeat *step_size* times:
   (a) Propose (and perhaps accept) a move.
   (b) (Performed every time)
   If *nbr_best_cost* $<\beta *$ *local_best_cost* (*nbr_best_cost* has been updated by another agent that has found a state that is better by at least a factor of β, where the parameter $\beta \leq 1$) then copy state of the agent that sent *nbr_best_cost*.
2. Update *nbr_best_cost* and *best_nbr_id* of neighbors (communication phase).

Note that the algorithm is asynchronous and that transmission of states takes place only on demand.

Figures 2 and 3, show the total number of updates between agents and the best cost found so far versus the total number of moves. The number of updates increases at a constant rate until the cost levels off, at which point, both the cost and the number of updates level off rather suddenly. We only show a small tail here. To find the best states it can, our algorithm is typically run for many more moves, and the running time is dominated by the tail, where agents are clustered at a few states until the next breakthrough.

Notice that since *step_size* is the number of move *attempts* between best cost communications, as the algorithm progresses, the number of *accepted* moves between best cost communications decreases since the rate of move acceptances decreases. This is desirable because in the beginning, all the agents are making progress and exploring different parts of the state space, whereas at the end, just one accepted move could start a breakthrough.

3.3 The Dynamic Multi-Agent Algorithm

In our goal to minimize the total amount of work performed, as opposed to the total amount of time taken when running on a parallel machine, we have designed an algorithm that uses varying numbers of agents at different points in the execution. By dynamically increasing and reducing the number of agents, we can reduce the total amount of work performed across all the agents for very good solutions. Although we have found the very best solutions using a fixed number of agents, the dynamic agent algorithm finds very good solutions after a relatively small number of moves.

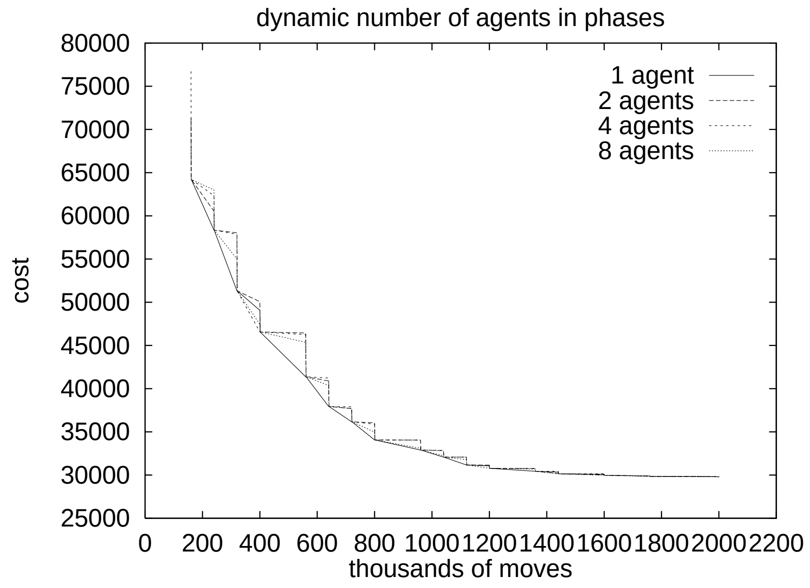

Figure 4: *Dynamic algorithm with 1, 2, 4, and 8 agents, where each set of agents starts phase i with the best state across all sets of agents from phase $i-1$.*

Figure 4 shows a somewhat idealized example of the dynamic-agent algorithm. The execution is divided into phases. Four different sets of agents (of 1, 2, 4, and 8 agents) work independently within each phase, i.e. the agents within a set may communicate with each other, but sets of agents do not share information. At the end of each phase, the best state found among all the sets (where "best" is defined as the state with the lowest cost) is copied to each set. We compare and results of the two dynamic multi-agent algorithms and the n agent algorithm in Section 4.

4 Experimental Results

We now present the results of our program. Our algorithm was implemented in C and run on SPARC stations, using up to 128 agents. All the results in this chapter describe runs on a single processor. We ran each instance of our program several times with different random seeds and report the average cost.

In discussing different variations of our algorithm, using a fixed number of agents versus a dynamic number of agents, we use graphs to present our data. The x-axis is the *total* number of moves summed up over all agents, and the y-axis is the solution cost. In the graphs for the fixed-agent version, we plot one line for each set of p agents.

We use the number of moves as a measure of the amount of work done in an attempt to generalize our results. This measure is slightly biased in favor of more agents because it disregards the time required to copy states between agents.

We compare our results with those of other methods in Section 4.1. We then present data where the number of agents is fixed versus self-adjusting in Section 4.2. We discuss our parallel implementation in Section 4.3.

In all cases, the local search technique we use is simulated annealing.

4.1 Comparison With Other Methods

In this section, we compare our best results with those of other algorithms. We obtained all our results by running in "production mode" i.e. we ran our program on each problem instance only three times and report the average cost.

Graph	Nodes	Edges	Best other	Our algorithm
*AIRFOIL1	4253	12289	81 (GS)	81
**AIRFOIL2	4720	27444	147 (MKLm)	129
**AIRFOIL3	15606	45878	158 (G)	140
*BCSPWR05	443	590	10 (GS)	10
*BCSPWR09	1723	2394	9 (GS)	9
BCSPWR10	5300	8271	30 (MKLc, GS)	69
**BCSSTK14	1806	30824	781 (GS)	778
*BCSSTK15	3948	56934	1474 (MKLc)	1474
*EPPSTEIN	547	1566	40 (MKLc, MKLm)	40
*PARC	1240	3355	21 (MKLc, GS)	21
*TAPIR	1024	2846	23 (MKLc, GS)	23

Table 2: A comparison of our algorithm on graph partitioning with other methods. GS = Geometric Spectral, MKLc = Chaco Multilevel Kernighan-Lin, MKLm = Metis Multilevel Kernighan-Lin, G = geometric. * = tie best result, ** = beat best result. The cost is the number of edges that cross the partition.

Name	Size	Optimal	Lin-Kernighan	Our algorithm
lin318	318	42029	42586	42097
att532	532	27686	27944	27867
fl3795	3795	28772	30871	29308
fnl4461	4461	182566	184582	192545
pcb442	442	50778	51187	50985
pr2392	2392	378032	384120	393868
pcb3038	3038	137694	139374	146292

Table 3: A comparison of our algorithm on TSP to Lin-Kernighan. The cost is the length of the shortest tour.

Table 2 is a comparison of results of our algorithm on graph partitioning problems to the best result among: spectral (S), geometric random circles (G), multilevel Kernighan-Lin (MKL), and spectral geometric (GS). For the multilevel Kernighan-Lin data, we used both the Metis and the Chaco packages. Both of the multilevel Kernighan-Lin programs run significantly faster than the others, including ours.

Table 3 contains a comparison of our algorithm to Lin-Kernighan for TSP. They were comparable in running time.

4.2 Fixed Versus Dynamic Number of Agents

In this section we present graphs of problems run using both a fixed number of agents, as well as changing the number of agents over time dynamically. For each set of n agents, we plot the cost after a certain number of moves. Figure 5 has graph bisection results, and Figure 6 has results for TSP. The x-axis is the total number of moves (attempted and perhaps performed) summed over all agents. The y-axis is the cost. We plot lines for sets of $2, 4, 8, 16, 32, 64,$ and 128 agents. We also plot a line for the dynamic-agent algorithm, which we discuss in the next section.

In both Figure 5 and Figure 6, the pattern is similar: Earlier in the execution, the versions with fewer agents have the lowest costs, but by then end, the versions with more agents find states with lower costs. The dynamic-agent algorithm quickly finds states of good cost. In case of TSP, the fixed-agent versions with fewer agents always outperforms the dynamic-agent algorithm in the end, albeit not by much.

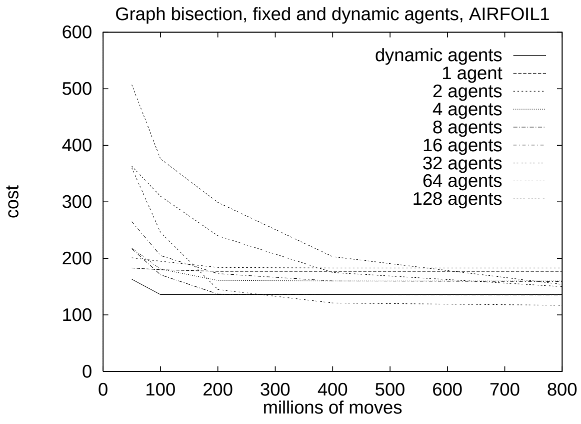

Figure 5: *graph bisection*

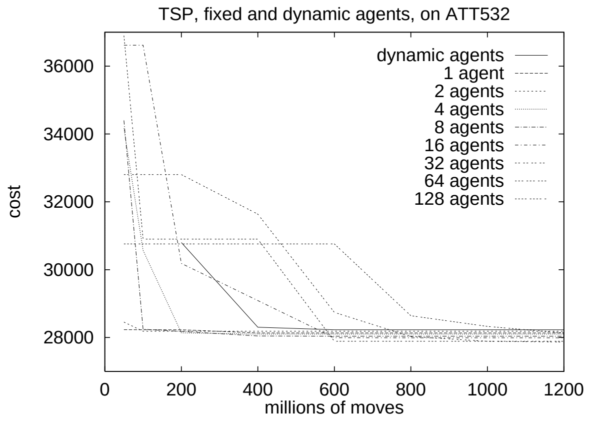

Figure 6: *tsp*

The dynamic-agent algorithm has an advantage over the fixed-agent algorithm in that one need not know the optimal number of agents to use in advance. As the results show, the dynamic-agent algorithm adjusts itself to suit the problem to get almost the same behavior of the best choice for a fixed agent algorithm, and sometimes it even does better.

4.3 The Parallel Implementation

In this section we discuss our parallel implementation and the implications of performance measurements. We implemented a parallel version of our algorithm in the obvious way: each agent is mapped to a processor. One copy of the input problem is shared by all the processors (each processor could have its own copy of the input, but that may become less practical when the input size is large). Each processor stores its own state information. We implemented our algorithm in Split-C [4] using Active Messages [25] on the Thinking Machines CM-5 [15]. We show results for our implementation on TSP and graph partitioning.

The two usual ways that parallel programs lose efficiency are from communication overhead or from synchronization due to lack of parallelism or load imbalance. Because our implementation is asynchronous and there is a single agent running on each processor, the only significant source of parallelism overhead could be from communication. There are two circumstances under which communication occurs:

- After a processor has proposed *step_size* moves, it shares its best state information (the cost and processor id only, not the actual state) with its *neighborhood_size* neighbors, and
- After each move, a processor checks to see if *nbr_best_cost* is less than *local_best_cost*. If so, it copies the entire state from *best_nbr_id*.

Our measurements indicate that the communication overhead in our implementation is very low independent of the problem being solved. For example, in Figure 7, which shows the percentage of time spent in communication in the ATT532 instance of the TSP problem, the overhead is extremely small, less than .2%.

Although these traditional sources of parallelism overhead are low in our implementation, there is potentially a more subtle source of inefficiency, since parallel agents may be wasting time searching uninteresting portions of the state space. Our experience is quite the opposite, i.e. that having multiple agents working mostly independently is such a good idea that it is useful on both a single processor and multiple processors. Figure 8 shows a near-perfect speedup for the ATT532 instance of TSP.

Figure 9 demonstrates the vagaries of reporting speedup in a meaningful way for the BCSSTK14 instance of graph bisection. We plot the number of agents versus the number of moves to reach four different cutoff costs for an example problem. The communication overhead is so low that the number of moves is almost directly proportional to the running time. Although it appears that performance degrades when we use 64

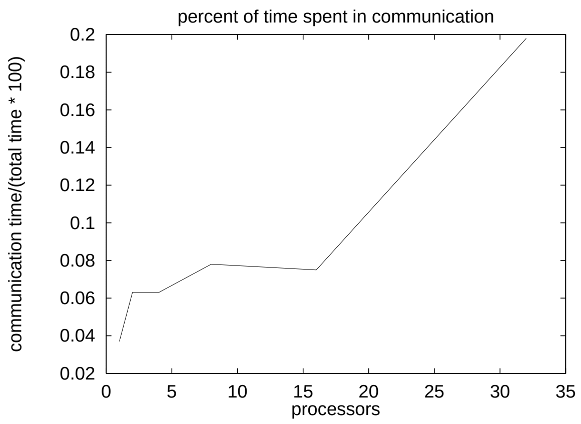

Figure 7: *Percent of time spent in communication in the ATT532 instance of TSP.*

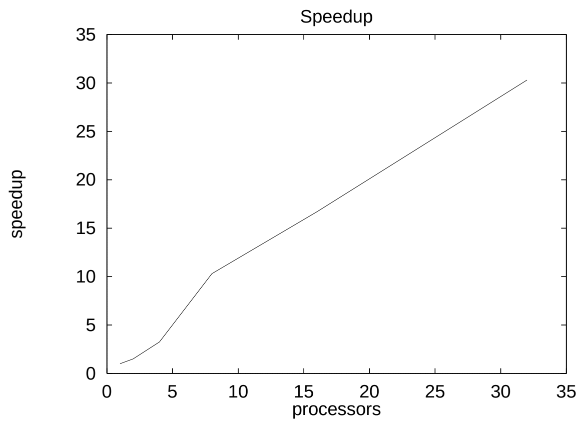

Figure 8: *Speedup for the ATT532 instance of TSP.*

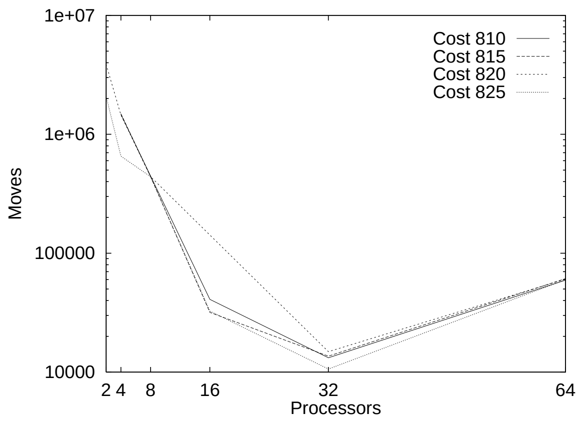

Figure 9: *Agents versus number of moves per agent with different cutoff costs for the BCSSTK14 instance of graph bisection.*

agents, we obtain a final partition with 64 agents that cannot be found using fewer agents. If we chose a lower cutoff cost, the running time would be lowest with 64 agents, but when the cutoff cost goes much below 810, the two-agent run never terminates. In general, as the number of agents increases, the quality of the final partition also increases.

5 Conclusions

In this paper we presented a new multi-agent algorithm for local search on optimization problems, which arise in many settings, including circuit layout, load balancing on parallel machines, and job scheduling. We attempt to improve upon local search by using cooperating agents to explore the state space more globally. The agents work independently on separate copies of the problem. Periodically the agents interact, and better states (states with lower cost) are distributed among the agents. Our algorithm can be used on any local search technique on any optimization problem.

Parameters to our algorithm include number of moves between interactions among agents, the degree of virtual parallelism, and the topology of the neighborhood structure of the agents, and termination criteria. We explore the parameter space extensively. We have found settings for these parameters that allow agents to explore different areas of the state space yet communicate frequently enough that time spent in fruitless sections of the state space is limited.

Each of the three problems we examined - TSP, graph bisection, and a VLSI layout problem - has associated move operators (e.g. 2-opt for TSP, random node exchange for graph bisection). We explored many move operators and selected those which performed well in terms of solution quality.

We measure the cost of running our algorithm by the total number of moves performed across all agents. We have implemented two versions of our algorithm and compared their performance in terms of solution quality and number of moves, one in which the number of agents remains fixed throughout the execution of the algorithm, and one in which the number of agents changes dynamically. Although both perform reliably well, there are situations in which one version is preferred to another, which we discuss in Section 4. The fixed-agent algorithm is probably the best choice if minimizing the number of moves is not a priority and if there is at least some idea of the optimal number of agents. The dynamic-agent algorithm has the advantage that it adjusts the number of agents to suit the problem, obtaining behavior very similar to the fixed-agent algorithm, and sometimes outperforming it. In addition, the dynamic-agent version tends to find good solutions consistently more quickly than the fixed-agent version.

In a parallel setting, each agent is a separate processor. The input data can either be shared or stored locally. Communication is asynchronous, and efficiency is high. The percentage of time spent in communication is less than .2%.

In fact, this algorithm grew out of a parallel implementation of local search. The super-linear speedup of that implementation implied that the sequential algorithm could be improved upon, perhaps by exploring the state space more globally. Some degree of independent search is valuable, but there are many practical issues which had to be resolved to determine how much independence was useful.

References

[1] E. H. L. Aarts and J.H.M. Korst. Boltzmann machines as a model for parallel annealing. In *Algorithmica*, vol.6(3):437–65, 1991.

[2] D. Aldous and U. Vazirani. "Go with the winners" algorithms. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science (Cat. No.94CH35717)*, pages 492–501, IEEE, 1994. AN4859059.

[3] Barnard and Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, 1993.

[4] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von .Eicken, and K. Yelick. Parallel Programming in Split-C., In *Supercomputing '93*, pages 262–273, Portland Oregon, November 1993.

[5] J. Gilbert, G. Miller, and S. Teng. Geometric mesh partitioning: implementation and experiments. In *Proceedings of the Ninth International Parallel Processing Symposium (Cat. No.95TH8052)*, pages 418–27, IEEE, 1995.

[6] F. Glover. Future paths for integer programming and links to artificial intelligence. In *Computers and Operation Research*, 13/5, pages 533–549, 1986.

[7] D. R. Greening. Parallel simulated annealing techniques. In *Physica D*, pages 293–306, June 1990.

[8] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *RUTCOR Research Report*, pages 43–87, 1987.

[9] B. Hendrickson and R. Leland. The Chaco user's guide, Version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, 1993.

[10] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages 113–122, 1995.

[11] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. In *Bell Systems Technical Journal*, vol.49:291–307, 1970.

[12] P. J. M. van (Peter J. M.) Laarhoven and E. H. L. Aarts. *Simulated annealing : theory and applications*. Mathematics and its applications, Kluwer Academic, 1989.

[13] T. Leighton and S. Rao. An Approximate Max-Flow Min-Cut Theorem for Uniform Multicommodity Flow Problems with Applications to Approximation Algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431, IEEE Computer Society Press, October 1988.

[14] T. Leighton and S. Rao. Multicommodity Max-Flow Min-Cut Theorems and their Use in Designing Approximation Algorithms. November 1996.

[15] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, and R. Zak. The Network Architecture of the CM-5. In *Symposium on Parallel and Distributed Algorithms '92*, pages 272–285, June 1992.

[16] S. Lin and B. Kernighan. An Effective Heuristic Algorithm for the Travelling Salesman Problem. In *Operations Research*, vol.21:498–516, 1973.

[17] M. Luby and W. Ertel. Optimal parallelization of Las Vegas algorithms. In *STACS 94. 11th Annual Symposium on Theoretical Aspects of Computer Science Proceedings*, pages 463–74. Springer-Verlag, 1994.

[18] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machines. In *Journal of Chemical Physics*, 21:1087–1091, 1953.

[19] G. Miller, S. Teng, W. Thurston, and S. Vavasis. Automatic mesh partitioning. In *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, 1993.

[20] A. Pothen, H. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. In *SIAM Journal on Matrix Analysis and Applications*, vol.11:430–452, 1990.

[21] Colin R. Reeves. *Modern heuristic techniques for combinatorial problems*. Halsted Press, 1993.

[22] R. Shonkwiler, F. Ghannadian, and C.O. Alford. Parallel simulated annealing for the n-queen problem. In *Proceedings of Seventh International Parallel Processing Symposium (Cat. No.93TH0513-2)*, pages 690–4, IEEE, 1993.

[23] R. E. Tarjan and R. Lipton. A separator theorem for planar graphs. In *SIAM Journal of Applied Math*, vol.36:177–189, 1979.

[24] C. A. Tovey. Hill climbing with multiple local optima. In *SIAM Journal on Algebraic and Discrete Methods*, 6(3):384–393, 1985.

[25] T. von Eicken, D. Culler, S. Golstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.