Neighborhood Composition: A Parallelization of Local Search Algorithms

Handa, Yuichi
Department of Electrical Engineering and Computer Science, Kyushu University

Ono, Hirotaka
Department of Electrical Engineering and Computer Science, Kyushu University

Sadakane, Kunihiko
Department of Electrical Engineering and Computer Science, Kyushu University

Yamashita, Masafumi
Department of Electrical Engineering and Computer Science, Kyushu University

Neighborhood Composition:
A Parallelization of Local Search Algorithms

Yuichi Handa, Hirotaka Ono, Kunihiko Sadakane, and Masafumi Yamashita

Dept. of Electrical Engineering and Computer Science, Kyushu University
{u1,ono,sada,mak}@tcslab.csce.kyushu-u.ac.jp

Abstract. To practically solve NP-hard combinatorial optimization problems, local search algorithms and their parallel implementations on PVM or MPI have been frequently discussed. Since a huge number of neighbors may be examined to discover a locally optimal neighbor in each of local search calls, many of parallelization schemes, excluding so-called the multi-start parallel scheme, try to extract parallelism from a local search by distributing the examinations of neighbors to processors. However, in straightforward implementations, when the next local search starts, all the processors will be assigned to the neighbors of the latest solution, and the results of all (but one) examinations in the previous local search are thus discarded in vain, despite that they would contain useful information on further search.
This paper explores the possibility of extracting information even from unsuccessful neighbor examinations in a systematic way to boost parallel local search algorithms. Our key concept is *neighborhood composition*. We demonstrate how this idea improves parallel implementations on PVM, by taking as examples well-known local search algorithms for the Traveling Salesman Problem.

1 Introduction

Most of combinatorial problems which frequently arise in various real-world situations, such as machine scheduling, vehicle routing and so on, are known to be NP-hard [3], and are believed that there would not exist polynomial time algorithms to find optimal solutions. Many researchers are hence interested in approximation algorithms [8] that can find near-optimal solutions in reasonable time.

Among them are metaheuristics algorithms based on local search very popular [4, 1, 10] because of their simplicity and robustness. A generic outline of a local search algorithm starts with an initial feasible solution and repeats replacing it with a better solution in its neighborhood until no better solution is found in the neighborhood. Although local search algorithms are much faster than exact algorithms, they may still require an exponential time for some instances. Furthermore we have started considering that just a polynomial time algorithm is no longer practical for huge instances in real applications.

Parallel implementations of local search algorithms are promising to satisfy the above requirement and hence many parallel algorithms have been proposed

mainly 1) to reach a better solution and 2) to re.duce the processing time, although they are apparently related with each other. A well-known paradigm called multi-start has every processor independently execute the same algorithm from randomly selected initial solution and returns the best solution among those obtained by the processors, mainly to increase the quality of solution (see e.g., [2]). A parallelized GRASP is an application of multi-start paradigm; while original GRASP randomly generates initial solutions in greedy manner then applies local search for each initial solution, in parallelized GRASP it is considered that several processors do GRASP for different seeds of randomness (e.g., [7]).

As mentioned, a local search algorithm repeats a local search starting with the current solution. Since the area of neighborhood can be huge as the size of instance becomes large, a local search consists of many independent searches (inside neighborhood) for a locally optimal solution. Many parallel implementations thus try to extract parallelism from each of the local searches by distributing the searches (in each of local searches) to processors, mainly to reduce the processing time. However, in straightforward implementations, all the processors will be assigned to the search of neighborhood of the current solution when the next local search starts, and the results of all (but one) searches in the previous local search are thus discarded in vain, despite that they would contain useful information on further search.

This paper explores the possibility of extracting useful information even from unsuccessful neighborhood searches to boost parallel local search algorithms. Our key concept is *neighborhood composition*. In a local search algorithm, a local search starts with a solution x and tries to obtain a better solution y. Suppose that a search i suggests the replacement of a subsolution u_i of x with a v_i to obtain a better solution x_i. Since the size of an instance is huge and search i can explore only very limited neighborhood of x, for many i and j, u_i and u_j do not overlap each other. Obviously, in many problems, x_i can be improved further by replacing u_j in x_i by v_j. This trivial fact is the essence. Suppose that searches i and j are executed in different processors p_i and p_j in a parallel implementation and that x_i achieves the locally optimal solution. Why don't we use u_j, v_j pair to improve the current solution, instead of discarding it? This is our claim.

The neighborhood composition gives us a concrete idea how to realize this idea on PVM or MPI. It works on the master/slave model. First a master processor divides the solution space and distributes them to slave processors. Then each slave searches the solution space independently. The master keeps the current best solution found in the whole search, and the slaves keep their own current solutions. When a slave finds an improved solution, it sends the difference between its own current solution structure and the improved solution structure, as *improvement information*, to the master. The master then tries to apply the improvement information to its own current solution. If the trial is successful, the master gets a new improved solution and sends it to the slaves. Otherwise if it is not, the section of solution space assigned to the slave is updated (because the slave's current solution may be completely different from the current solution).

Since our method does not need synchronize the slaves, each slave can search its own neighborhood while other slaves communicate with the master, which hide the communication overhead.

To confirm the availability of our method, we choose the Traveling Salesman Problem [6] (TSP, for short) and 2OPT (or Or-OPT) and Lin-Kernighan [5] neighborhood local search algorithms as a model problem and its algorithms. We implemented these algorithms on PVM, and then conducted computational experiments. The reasons why we adopt TSP and these neighborhood set are as follows: 1) TSP is one of the most well-known problems, and we can easily obtain many benchmark problems from TSPLIB[1]. Also many algorithms based on local search are proposed. 2) 2OPT, Or-OPT and Lin-Kernighan neighborhoods in TSP are suitable to explain our idea visually. Through the computational experiments, we see that our parallelized local searches achieve good performance in many cases; that is, the processing time of parallelized algorithm is much smaller than the original one.

2 TSP and Local Search

In this section, we introduce some basic ideas and notations used throughout this paper. Although we should actually give more general definitions and notations for combinatorial optimization and local search algorithms, we restrict our explanations to TSP and its local search problems due to the space limitation. One can easily apply them to many other problems and many other local search algorithms.

TSP is described as follows: Given a set of n cities and an $n \times n$ distance matrix D, where d_{ij} denotes the distance from city i to city j, with $i, j = 1, ..., n$, find a tour that visits each city exactly once, and is of minimum total length.

In order to solve TSP, many local search based algorithms are proposed. A local search starts from an initial solution σ and repeats replacing σ with a better solution in its *neighborhood* $N(\sigma)$ until no better solution is found in $N(\sigma)$, where $N(\sigma)$ is a set of solutions obtainable by slight perturbations. The local search from an initial solution σ_0, in which the neighborhood N is used, is formally described as follow.

Algorithm Local Search(N, σ_0)

step1. Set $\sigma := \sigma_0$.
step2. Search a feasible solution $\sigma' \in N(\sigma)$ such that $cost(\sigma') < cost(\sigma)$ (SEARCH). If such σ' exists, set $\sigma := \sigma'$ (IMPROVE) and return to step2. Otherwise go to step3.
**step3.** Output σ and stop.

Obviously, the performance of local search algorithm depends on which neighborhood we use. If we adopt wider neighborhood, the local search may find the

---

[1] http://www.crpc.rice.edu/softlib/tsplib/
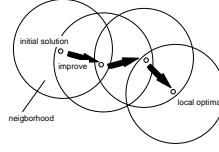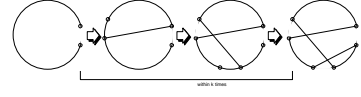
Fig. 1. Local Search

better solutions, however, it probably requires more computational time. In this paper, we consider local search algorithms for TSP whose neighborhood types are 2OPT, 3OPT, and Or-OPT. A 2OPT move deletes two edges, thus breaking the tour into two paths, and then reconnects those paths in the other possible way (Fig.2). Thus, A 2OPT neighborhood of a solution (tour) σ is defined as a set of tours that can be obtained by 2OPT moves from tour σ. Similarly, 3OPT move and neighborhood are defined by the exchange replaces up to three edges of the current tour. An Or-OPT neighborhood is a subset of 3OPT (Fig.3). The Lin-Kernighan heuristic allows the replacement of an arbitrary number of edges in moving from a tour to neighboring tour, where again a complex greedy criterion is used in order to permit the search to go to an unbounded depth without an exponential blowup. The Lin-Kernighan heuristic is generally considered to be one of the most effective methods for TSP.
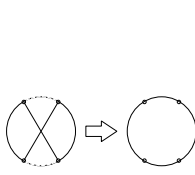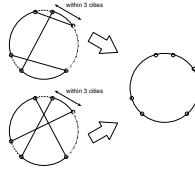
Fig. 2. 2OPT move   **Fig. 3.** Or-OPT move   **Fig. 4.** k LK-OPT move

All of these are the most basic types of neighborhoods for TSP, and indeed quite a many number of local search and metaheuristics algorithms are proposed and studied[1]. Also, these neigborhood structures can be seen in many other (NP-hard) combinatorial problems. Indeed, GAP(general assignment problem)'s basic neigborhoods, swap-neigborhood, ejection-neigborhood have quite similar structures to 2OPT and LK-OPT of TSP, respectively.

3 Parallelization of Local Search

As described in Section 2, local search has two important phases, SEARCH and IMPROVE. In our parallelization, these two phases are imposed by different processors: IMPROVE is done by a master processor, and SEARCH is by slave

processors; a rough idea of acceleration is that the master processor maintains the current best solution (do IMPROVE) and slave processors share SEARCH operations.

For this purpose, we consider to divide the neighborhood in SEARCH, and to assign them to slave processors. Such a division is easy to design if all the slave processors are synchronized. One simple way is as follows: The master divides search space, and assigns them to slaves. Each slave executes SEARCH operation for the assigned space. Once a slave finds a better solution, it sends the result to the master. Then it performs IMPROVE by replacing the current solution with the received solutions, and divides the new search space for the improved solution again. In this method, search space is divided into all n slaves, so that the time spent by SEARCH will be ideally reduced to $1/n$. In fact, this method however needs much time in communication between the master and slaves. Moreover, only a result of one slave is used for updating the master's current solution; most of SEARCHs by slave processors are in vain.

To overcome this, we propose the following method:

Algorithm Master (IMPROVE)

step1. Set $\sigma_{master} := \sigma_0$ and divide $N(\sigma_{master})$ into N_1, ..., N_n, then send σ_{master} and N_i to each slave $slave[i]$.

step2. Wait. When receiving a data from slave, go to step3.

step3. Improve σ_{master} with the improvement information received from $slave[i]$, if possible(CHECK).

step4. Divide $N(\sigma_{master})$ and send the σ_{master} and new N_i to $slave[i]$ from which the improvement information were received just before. Return to step2.

Algorithm Slave (SEARCH)

step1. Receive the initial solution σ_0 and N_i. $\sigma_{slave[i]} := \sigma_0$.

step2. If there is a better solution σ' in N_i, send to the master the improvement information of σ'.

step3. Wait. When receiving σ_{master} and N_i from the master, replace N_i and set $\sigma_{slave[i]} := \sigma_{master}$, and go to step2.

Improvement information in the above description is defined as the difference between the original solution and its improved solution. For example, in case of 2OPT, improvement information is represented by two edges in the improved solution but not in the original solution.

As an example, suppose that there are one master and three slaves available. First, the master and slaves have the same initial solution. Slaves begin to search the same neighborhood of the initial solution from different neighbor (Fig.5).

If one slave finds a better solution, the slave sends to the master not the better solution but the improvement information. Receiving the improvement information, the master CHECKs if it is consistent for the current solution (i.e., the improvement keeps the solution feasible). If CHECK is yes, the master improves the current solution based on it. The master then returns the new current

solution to only the slave that generated the improvement information; The other slaves (Slave A and C at Fig. 6) do not receive the new solution. That is, these slaves continue to their own neighborhood.
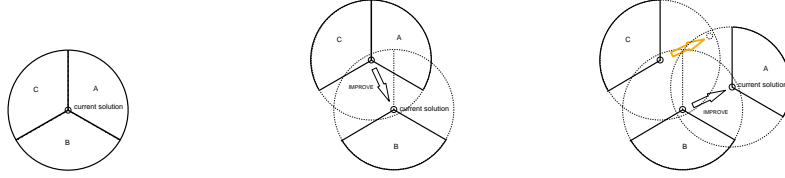
Fig. 5. Each slave begins a search in a different space

Fig. 6. Slave B finds a better solution

Fig. 7. Slave A finds a better solution (CHECK is needed)

Here, it should be noted that the solution kept by the master may be different from ones of the slaves. If the neighborhood structure of the slave's solution is quite similar to the one of master's solution, its improvement information can be applied and the improvement is successful (Fig. 9). Otherwise, the improvement information cannot be applied and it fails (Fig. 10). Actually, since our algorithm is based on local-search manner, we can expect that the neighborhood structures are not so different. We call this devise of improving solutions *neighborhood composition*.

Note that our parallelized algorithm does not need synchronization, which causes the following properties: As the defect, the master needs to do the extra task CHECK for every improvement of slave's solution. However, the overheads become negligible because while one slave communicates with the master and the master do CHECK, other slaves can continue SEARCH.

4 Experimental Results

To evaluate the performance of this parallelization method, we implement parallel algorithms based on a local search algorithm with basic neighborhood structures. The algorithms which we have parallelized are the following 2 types:

A. searching 2OPT and OrOPT neighborhood [2]
B. searching Lin-Kernighan neighborhood [3]

As a problem instance, we used a standard TSP instance, att532, pr1002, nrw1379 in TSPLIB.

Table 1 shows the run times of the original and the parallelized algorithms "A" and "B". As shown above, the communication between master and slave and CHECK phase on master are needed when using this parallelization method, so paralleled with a few slaves may slow the process. However, adding the slaves,
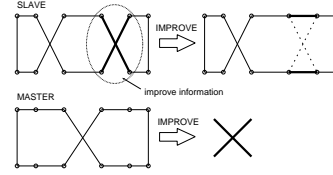
---

[2] `http://www-or.amp.i.kyoto-u.ac.jp/members/ibaraki/today/tsp1.c`
[3] `http://tcslab.csce.kyushu-u.ac.jp/ũ1/program/lkh3_1.c`
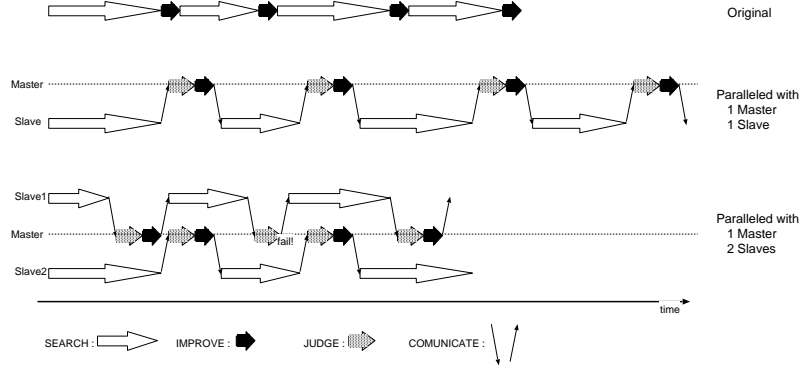
Fig. 8. comparison of Non-Parallelized, Parallelized by 1 master and 1 slave and Parallelized by 1 master and 2 slave
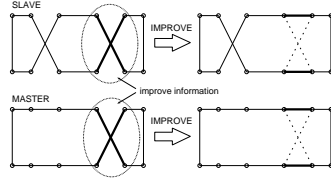
Fig. 9. The master's solution has the same two edges that the slave cut for IMPROVE

Fig. 10. The master's solution does not have the same two edges that the slave cut for IMPROVE

these overheads are hidden and the process gets faster. IMPROVE arises as many in "A" as in "B". The overhead (communication and CHECK) is proportional to the number of IMPROVE, so the overheads in parallelizing "A" and "B" are almost the same. But in "B" one IMPROVE takes less time than in "A", which means the rate of SEARCH time in the whole time is small, so the parallelization is less effective in "B". Nevertheless using more than six slaves, we can make "B" faster than the original.

¿From the table 2, it is observed that the failure ratio of IMPROVE becomes higher as the number of slaves increases. The reason may be that the solution structures of slaves easily vary if the number of slaves is large; the similarity is lost. For the same number of slaves, the failure ratio is higher in 5 edges cut and reconnected. The larger changes makes the master's and slave's solution structure more different from each other. The high rate of failure makes the effect of the parallelization smaller.

Figure 11 plots the result by the two method. The horizontal axis represents the run time and the vertical axis gives the costs obtained by the algorithms. The symbol "∘" represents the result of the original Lin-Kernighan algorithm. The symbol "×" and "+" imply the result by Lin-Kernighan algorithm paral-

lelized with "Neighborhood Composition" and with multi-start, and the number above the symbol means the number of slave processors. The symbol "∗" implies the result using both two parallelizing methods, and has "(x, y)" above it. "y" means the number of groups, and each group has one master processor and "x" slave processors. Lin-Kernighan algorithm parallelized with "Neighborhood Composition" is executed by each group, then the best solution of all the groups is adopted. Parallelizing with "Neighborhood Composition" makes the local search faster, and parallelizing by multi-start makes its solution better. By combining two methods we can get highly precise solution in short time.

		original	2 slaves	4 slaves	6 slaves	8 slaves
att532	2,OrOPT	62.83(1.00)	35.98(0.57)	25.16(0.40)	20.43(0.32)	20.36(0.32)
	3-LK-OPT	4.65(1.00)	11.79(2.59)	4.02(0.86)	2.75(0.59)	2.36(0.51)
	5-LK-OPT	4.70(1.00)	12.90(3.17)	4.06(0.86)	3.19(0.68)	2.38(0.51)
pr1002	2,OrOPT	349.42(1.00)	199.03(0.56)	118.58(0.34)	98.75(0.28)	83.19(0.24)
	3-LK-OPT	21.11(1.00)	81.80(3.87)	21.58(1.02)	13.22(0.62)	10.33(0.49)
	5-LK-OPT	21.68(1.00)	89.85(4.14)	24.12(1.11)	13.40(0.62)	10.62(0.48)
nrw1379	2,OrOPT	881.49(1.00)	494.68(0.56)	309.04(0.35)	221.07(0.25)	203.28(0.23)
	3-LK-OPT	34.04(1.00)	133.36(3.92)	34.76(1.02)	22.10(0.62)	17.72(0.52)
	5-LK-OPT	35.42(1.00)	134.96(3.81)	39.95(1.12)	23.83(0.67)	18.26(0.52)

Table 1. Run time (s) of original and parallelized "2,OrOPT" and "k-LK-OPT"

|		2 slaves	4 slaves	6 slaves	8 slaves
2,OrOPT	2 edges	2.44	4.79	7.51	8.86
	3 edges	13.52	33.18	38.39	42.88
5-LK-OPT	2 edges	4.58	18.67	29.16	36.39
	3 edges	4.25	18.49	29.06	37.08
	4 edges	4.70	19.07	30.96	39.12
	5 edges	6.72	24.40	37.58	46.25

Table 2. comparison of ratio (%) of failing to improve with the improve solution at parallelized 5-LinKernighan (Instance = att532)

5   Conclusion

In this paper, we proposed a parallelization method for local search algorithm, which is applicable to many combinatorial optimization algorithms. The purpose of our parallelization is to reduce the run time. We then implemented typical parallelized local search algorithms for TSP, and conducted computational experiments. The result of experiment shows that our parallelization method greatly reduces the run time: our method or the idea of our method are potentially useful for many combinatorial optimization problems and many local search algorithms. As a future work, we need to confirm that our parallelization method is useful for more sophisticated algorithms, and compare other parallelization methods.

Acknowledgements
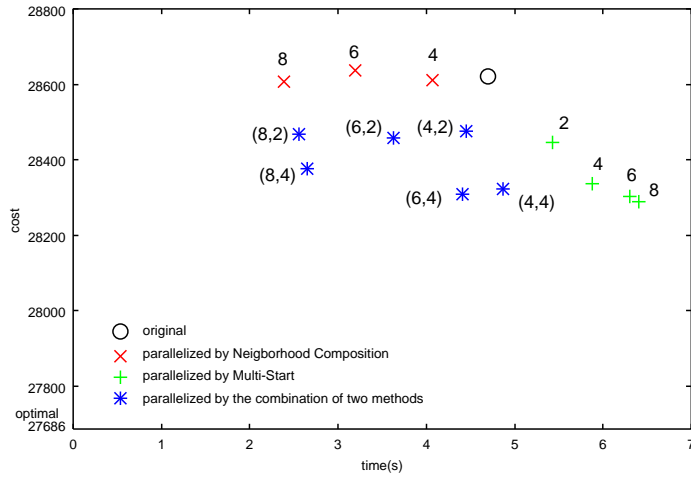
Fig. 11. comparison of the parallelized by Neigborhood Composition and by Multi-Start and by both two method (5-LKH, Instance = att532)

References

1. E. Aarts and J. K. Lenstra, *Local Search in Combinatorial Optimization*, John Wiley & Son, 1997.
2. Y. Asahiro, M. Ishibashi, and M. Yamashita, Independent and Cooperative Parallel Search Methods for the Generalized Assignment Problem Optimization Methods and Software, Vol.18, No.2, pp.129-141, Apr. 2003.
3. M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, 1979.
4. P.E. Gill, W. Murray and M.H. Wright, Practical Optimization. Academic Press. 1981.
5. K.Helsgaun, An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic, *European Journal of Operational Research* 126 (1), 106-130, 2000.
6. E. L. Lawler, J. K. Lenstra, A. H. G Rinnooy Kan and D. B. Shmoys, The Traveling Salesman Problem, A Guided Tour of Combinatorial Optimization, John Wiley and Sons 1985
7. S. Vandewalle, R. V. Driesschie, and R. Piessens, The parallel performance of standard parabolic marching schemes, *International Journals of Super Scomputing,* 3(1):1-29, 1991.
8. V. V. Vazirani, Approximation Algorithms, SpringerVerlag, 2001.
9. A. S. Wagner, H. V. Sreekantaswamy, S. T. Chanson, Performance Models for the Processor Farm Paradigm, IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 5, May 1997.
10. M. Yagiura and T. Ibaraki, "On Metaheuristic Algorithms for Combinatorial Optimization Problems," Systems and Computers in Japan, Vol. 32, Issue 3, 2001, pp. 33-55.