Technische Universität Kaiserslautern – Fachbereich Informatik
Interner Bericht 331/04

Embedding a Chained Lin-Kernighan Algorithm into a Distributed Algorithm

Thomas Fischer* and Peter Merz†

August 03, 2004

The Chained Lin-Kernighan algorithm (CLK) is one of the best heuristics to solve Traveling Salesman Problems (TSP). In this paper a distributed algorithm is proposed, were nodes in a network locally optimize TSP instances by using the CLK algorithm. Within an Evolutionary Algorithm (EA) network-based framework the resulting tours are modified and exchanged with neighboring nodes. We show that the distributed variant finds better tours compared to the original CLK given the same amount of computation time. For instance `fl3795`, the original CLK got stuck in local optima in each of 10 runs, whereas the distributed algorithm found optimal tours in each run requiring less than 10 CPU minutes per node on average in an 8 node setup. For instance `sw24978`, the distributed algorithm had an average solution quality of 0.050% above the optimum, compared to CLK's average solution of 0.119% above the optimum given the same total CPU time (10^4 seconds). Considering the best tours of both variants for this instance, the distributed algorithm is 0.033% above the optimum and the CLK algorithm 0.099%.

1 Introduction

The Traveling Salesman Problem (TSP) is one of the most well known *combinatorial optimization problems*. It describes a salesman's problem of finding the cost-optimal route to a given number of cities (customers) such that each of these cities (customers) is visited exactly once. The problem can be represented by a graph $G = (V, E)$ and a function d that describes the distance between two vertices $v_i, v_j \in V$. For *symmetric* TSPs (STSP) $d_{i,j} = d_{j,i}$ holds always for all i and j. For *asymmetric* TSPs (ATSP) $d_{i,j} \neq d_{j,i}$ holds for at least one pair (v_i, v_j). The optimal solution is a *Hamiltonian cycle* where the sum of the distance values is minimal. Such a cycle is a *permutation* π on the vertices V that minimizes the cost function $C(\pi)$.

$$C(\pi) = \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)} \tag{1}$$

For an instance with n cities ($n = |V|$), there are $\frac{(n-1)!}{2}$ different tours. Although having a simple setup, the TSP is a *NP-hard* problem.

1.1 Algorithms

To solve TSP instances, different different types of algorithms have been developed. *Exact algorithms* enumerate implicitly each possible solution. These algorithms can find a provable optimal solution based on e.g. Cutting Plane [19], Branch-and-Bound [34] or Branch-and-Cut [7] techniques.

*Distributed Algorithms Group, Department of Computer Science, University of Kaiserslautern. fischer@informatik.uni-kl.de
†Distributed Algorithms Group, Department of Computer Science, University of Kaiserslautern. pmerz@informatik.uni-kl.de

Approximation algorithms construct a valid tour providing a guarantee for the worst-case time complexity and the expected tour length. Christofides heuristics [16] finds a tour for Euclidean TSP instances in time $O(k^3)$, where k is the number of vertices in the graph having an odd degree. For this tour the heuristics guarantees that it is for a factor of at most 1.5 above the optimum. A *polynomial time approximation scheme* (PTAS) for Euclidean TSPs has been developed by Arora [11]. This algorithm guarantees to find a tour that is at most for the factor $(1+\epsilon)$ above the optimum requiring a time of $n^{O(1/\epsilon)}$.

Heuristic algorithms perform a non-complete search in the solution space and do not guarantee to find an optimal solution. Their main advantage is that they can find a good sub-optimal solution in much shorter time than exact algorithms. Heuristic algorithms performing a *local search* usually exploit neighborhood relations between nodes. One example is the *k-opt neighborhood* [21, 35], where two tours are called neighbors if one tour can be transformed to the other by exchanging k edges. A tour is called *k-optimal* (k-opt) if the tour cannot be improved by exchanging k edges. Increasing k and performing an exhaustive search for possible exchange moves increases the tour quality, but also requires a fast growing amount of computation time. So, for most applications k is limited to $k \leq 3$.

Lin and Kernighan [36] approached the problem of finding a tradeoff between tour quality and computation cost by introducing an algorithm (LK) where k is kept variable. In each iteration the algorithm performs a sequence of exchange moves (thus increasing k) while considering whether or not a possible move could lead to a better tour. Although this depth search returns only better tours (if available) it might consider internally worse tours as intermediate steps. The search stops when a termination criterion as the *positive gain criterion* (the sum of gains over all steps must always be positive) is fulfilled.

To improve the results of the LK algorithms, early implementations restarted the algorithms with new initial tours iteratively. This approach was superseeded by the Chained LK algorithm (CLK) introduced by Martin, Otto and Felten [37], which follows a simulated annealing (SA) pattern. Instead of restarting with a new tour, the chained variant perturbates a LK-optimized tour by applying a 4-exchange move (*double-bridge move*, DBM) to escape from local optima. This move is cheap to perform as it does not need subtour flips and can change a tour severely. Variations and implementations for CLK were proposed e.g. by Applegate, Bixby, Chvátal and Cook [8] (ABCC-CLK) and Applegate, Bixby and Rohe [9] (ABR-CLK).

Both CLK algorithms support the *Quick-Borůvka tour construction heuristics*. This heuristics is based on the minimum-weight spanning tree (MST) algorithm by Borůvka [15, 38]. The Quick-Borůvka heuristics for TSP tours sorts the vertices of the problem instance by coordinates (if possible) and processes each vertex by adding an adjacent edge to the tour that has a minimal weight and is feasible (no subtour will be generated by adding). As stated in [8, 9], this construction algorithm's tour quality is worse than other greedy algorithms, while requiring less time to build a tour. This algorithm is suited for further Chained Lin-Kernighan processing, too, and can be efficiently implemented by *kd*-trees [14].

Another variation of the original LK algorithm worth mentioning comes from Helsgaun [27]. His algorithm (LKH) incorporates larger and more complex search steps and new features like the α-neighborhood (based on minimal spanning trees or, to be more precise, on 1-trees). According to a comparison of TSP heuristics by Johnson and McGeoch [30] Helsgaun's LK variation is the best LK based implementation in respect to tour quality, but requires a significant larger amount of time compared to other implementations.

For TSP instances that have not yet been solved to optimality a lower bound for the optimal tour length can be estimated. The best known algorithm to find lower bounds is the one from Held and Karp [25, 26]. The Held-Karp Lower Bound (HKLB) is calculated by using a 1-tree with modified edge weights. Johnson *et al.* showed in [31] that for randomly generated instances the optimal tour length is on average less than 0.8% above the Held-Karp bound and for most instances from the TSPLIB [40] collection less than 2%. So, the Held-Karp bound can be used as an evaluation criterion for TSP instances where no optimal solution is known.

1.2 Motivation

Due to improved algorithms and faster computers the size of instances that could be solved has grown steadily since the beginning of TSP research. The first instance solved to optimality was Danzig *et al.*'s 42/49[1]-cities problem [19] in 1954. The latest achievement is the solution for `sw24978` in May 2004. The required amount of computation power can only be supplied by a cluster of nodes. In case of this instance, 96 dual processor machines (2.8 GHz) needed a total of over 80 CPU years to prove optimality with an exact algorithm. In contrast, for heuristic algorithms distributed computation is not common

[1]See `http://www.tsp.gatech.edu/gallery/igraphics/dantzig.html` for details how cities were counted.

yet. This might be due to the fact that heuristic algorithms require less computation time and therefore there is no need for distributed computation compared to exact algorithms so far. But to solve large TSP instances with today's heuristic algorithms it is inevitable from our point of view to distribute computation. Our approach is presented in this paper.

The classical approach for distributed computing is to setup a client-server system, where the central server organizes the workflow, while the clients are not aware of other clients and just perform their computations. This star-like topology is quite popular and is used in e.g. [3] and [6, 42]. The main drawback here is the server as the single point of failure. As every communication is routed through this server its throughput bounds the size of the network and thus the system is not scalable.

This problem is addressed by Peer-to-Peer (P2P) networks. In these networks each node is both server and client for other nodes and ideally all nodes are symmetric. Unlike traditional client-server systems, P2P networks are designed for a dynamic environment where nodes can join and leave at any time, communication is asynchronous and no global information is known. For file sharing, this technology has been applied in e.g. [4] and [17].

Distributed computing is a rather new branch in P2P systems. A prominent project in this area is DREAM [39, 10], which is an distributed environment to simulate evolutionary computation experiments. In this system distributed resource machine (DRM) nodes communicate using epidemic algorithms [20]. The implementation of DREAM is build in layers and allows users to choose the appropriate level of detail and flexibility.

In this paper we present a distributed algorithm for solving Traveling Salesman Problems. This algorithm utilizes an existing Chained Lin-Kernighan algorithm from Applegate *et al.* [2] and embeds it into an evolutionary algorithm that is running distributed over several nodes in a network. With this approach our algorithm finds both better tours given a computation time limit and it converges faster towards an optimal solution compared to the original Chained Lin-Kernighan algorithm. For some instances, where the original Chained Lin-Kernighan algorithm gets stuck in local optima, the distributed algorithm finds an optimal solution. This advancement is due to two enhancements. On one hand, the distributed algorithm adds an additional perturbation step that is variable in its strength. On the other hand, nodes can exchange their local tours with neighboring nodes. Thereby, high quality tours spread through the network and become the base for further local improvements.

The paper is organized as follows. The rest of the introduction presents several related studies. Section 2 introduces the architecture of the distributed algorithm, important algorithmic features, the testbed and available and used parameters. Section 3 presents the results of the simulation runs in detail. Finally, in section 4 conclusions are drawn and perspectives for future work are proposed.

1.3 Distributed Algorithms for large TSPs

Bachem and Wottawa proposed in 1992 an attempt to parallelize heuristics for the TSP on transputers [12]. They used a job-level parallelization running the LK algorithm on different processors. Each run starts with a random generated tour and the locally optimized tours are broadcasted to the other nodes. In a speedup technique called *partial reduction*, edges that appeared in the previous best tour are protected within the LK algorithm after its first iteration. This technique reduces the runtime for about $10 - 50\%$ while keeping the tour quality constant. Another algorithm by Bachem and Wottawa is based on clustering the original instance. So, a global tour through all clusters including entering and leaving cities for each cluster has to be found. In a second step, a path through all cities in a cluster is searched incorporating the two fixed cities. Finding the global tour is divided into two steps. First, the approximate distances between the clusters are calculated based on the clusters' balance point. Second, for close-by clusters convex hulls are used to get the exact distances. For the generation of clusters for Euclidean TSPs the authors use Karp's method [32] to divide the node set recursively into two halfs alternating the coordinate as sorting criterion in each step. A draw-back of clustering is that the shape of the node subsets affects the shape of the final tour. Moving the clusters and improving the tour again is proposed by the authors to circumvent the problem. The resulting tours are $2 - 5\%$ worse than LK tours.

The Asparagos96 system by Gorges-Schleuter [22] is an asynchronous parallel genetic algorithm. In each generation each individual selects a mate from its neighborhood and performs a MPX2 (*maximal preservative crossover*) crossover. The resulting tour which may be damaged is mutated by a double-bridge move and repaired by a 2-repair step (3-repair step for ATSPs). If the offspring has a shorter tour length than its ancestor, the ancestor is superseeded by the offspring. For parallelization the algorithm simulates several populations in parallel. In this case, the best individual of a population can be chosen

as mate for a individual in another population. The average tour quality for instance `fl3795` over 10 runs results in a tour quality of 0.34% above the optimum. The time required for one run is about 17 hours, which would scale to about 3.5 hours on a Alpha 500 MHz.

Baraglia *et al.* introduce a genetic algorithm (GA) [13] using an island model [23], where each node represents an island with a subpopulation. Here, the tours are encoded in a compact form [24] storing only the probability values in a $k \times k$ triangular matrix P (k is the number of cities). The matrix element $p_{i,j}$ represents the probability that the edge (i, j) is part of an individual's tour. In each generation a tour L is constructed using the probability values. This tour L is refined to tour W by the CLK algorithm. The matrix elements' values are increased if the corresponding edge occurs only in W, but not in L, decreased in the opposite cased and remain unchanged if the edge occurs in both or none of the tours. Although the paper's conclusions are meager with respect to numerical data, the supplied plots show that the more processes cooperate the less generations are required for each one to find the optimal solution for an instance. The instances sized analyzed in this paper range from 532 to 1002.

Nguyen *et al.* describe a GA-based algorithm [28], which uses their LK implementation (applying a 5-opt basic move) for local tour improvement. This algorithm can be parallelized by settling subpopulations on cluster machines. The GA algorithm performs in each generation a mutation or crossover operation on one member of each subpopulation. For mutation, a selected tour is mutated by a Random-walk kick and optimized by the LK algorithms for a number of iterations (thus reproducing an Iterated LK). The best intermediate tour will replace the original tour if it was better than the original one. For the crossover operation (MPX3), two parents are selected from the subpopulation and merged. Common subtours from both parents are fixed for the LK algorithm to follow. The resulting tour will replace the worst parent if better. The GA will terminate, if no improvement has been found for a number of iterations. Nguyen *et al.*'s algorithm can compete with Helsgaun's LK regarding the tour quality, but requires significantly less time for larger instances. E.g. for instance `d18512`, the GA-based algorithm requires about 5000 seconds in a 10 node setup for an average tour quality of 645323.8, whereas LKH requires over 100000 seconds for a worse tour (645332.2) in on a single computing node.

A multilevel approach embedding a Chained Lin-Kernighan algorithm has been proposed by Walshaw [41]. He uses a multilevel scheme where a problem instance is iteratively coarsened by matching and merging cities to reduce the problem size. After constructing a tour in the reduced instance, the instance is stepwise uncoarsened again. In each uncoarsening step the current tour is refined by the C^nLK algorithm. The resulting tours are better compared to the plain CLK algorithm given the same computation time. Furthermore, a given tour quality is reached quicker with the multilevel system. In a testbed of 81 instances the multilevel setup MLC$^{N/10}$LK (number of kicks for CLK is one tenth of the number of cities) archives on average slightly better tours and is still 4 times faster than CLK. This algorithm is not parallel or distributed, but is included here for completeness and later comparison.

2 Experimental Setup

2.1 Details on Chained Lin-Kernighan

In this subsection some details regarding the Chained Lin-Kernighan algorithm and its implementation by Applegate *et al.* are presented.

2.1.1 Quick-Borůvka tour construction heuristics

The original Borůvka algorithm [15, 38] is a greedy construction heuristics for minimum-weight spanning trees (MST). This algorithm is based on the cut property, which states that the edge with minimum weight of every cut is part of the MST. In each iteration two steps are performed.The first step selects for each vertex v of the graph $G(V, E)$ the adjacent edge with minimum weight and adds it to a initially empty edge set E'. This edge was the minimum weight edge of the cut $(v, V \backslash \{v\})$. In the second step the nodes of each tree in the forrest (V, E') are contracted and emerging loops and multiple edges are resolved. As in each iteration the edges have to be sorted and the number of vertices is reduced at least by the factor two, the complexity of this algorithm is $O(sort(|E|) \cdot \log |V|)$.

The Quick-Borůvka algorithm in [8, 9] is a TSP tour construction heuristics inspired by the original Borůvka algorithm. For geometric instances, the vertices are sorted by their coordinates to determine the order of processing. The algorithm iterates until a valid tour has been constructed (at most two iterations are required). In each iteration each city which has not yet two adjacent edges in the partial tour is processed. For a city that is processed an edge with the following properties is selected: It has

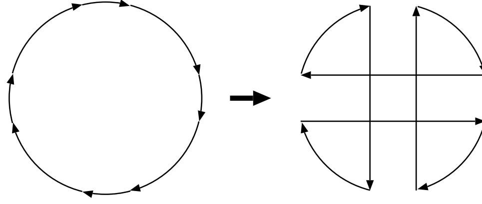


Figure 1: Double-bridge move.

a minimum weight, does not lead to a subtour and is not adjacent to a city that has already two other adjacent edges. This selected edge is inserted into the partial tour.

The authors suggest using a *kd*-tree for efficient implementation. Although the quality of the constructed tour is worse compared to other greedy construction algorithms, it requires less time for building a tour. In [9] results from a comparison of CLK runs based on tour constructed by the HK-Christofides heuristics [16] or Quick-Borůvka, respectively, show that the latter heuristics results in better tours, although the first requires more time for tour construction.

2.1.2 Kicking strategies

As part of any Chained Lin-Kernighan algorithm a tour that was optimized by the Lin-Kernighan algorithm (LK) is perturbated. Martin, Otto and Felten proposed in [37] a double-bridge move (DBM) to "kick" the intermediate tour. The double-bridge move is a 4-exchange move (see figure 1) which is a cheap move as it does not require to flip the order of the cities in any subtour. Depending on the selection of the 4 relevant cities (the other 4 cities are successors of them) this move can have a strong impact on the tour. Additionally, a double-bridge move cannot be reversed by a simple set of LK moves.

For the selection of the 4 cities Applegate *et al.* propose four kicking strategies. These strategies differ in computation cost to find four cities and the impact on the tour quality.

Random In this kicking strategy the relevant cities are selected at random. This strategy degenerates the tour, but might help to leave a local optimum. The variator of the distributed algorithm explained later uses this strategy, too.

Geometric For this kicking strategy the relevant cities are chosen from the k nearest neighbors of a selected city v. Small k cause the kick to be local, whereas for large k the kick becomes similar to the Random kick. For geometric instances, *kd*-trees can be used to find the nearest neighbors. For non-geometric instances, the cities have to be sorted to get a nearest neighbor list.

Close A subset of the cities of size βn (β is a parameter, n is the number of cities) is chosen. From this subset the six cities nearest to a chosen city v (first relevant city) are used to choose the three other relevant cities. Larger β make it more likely that cities close to v are part of the subset and thus used for a local kick.

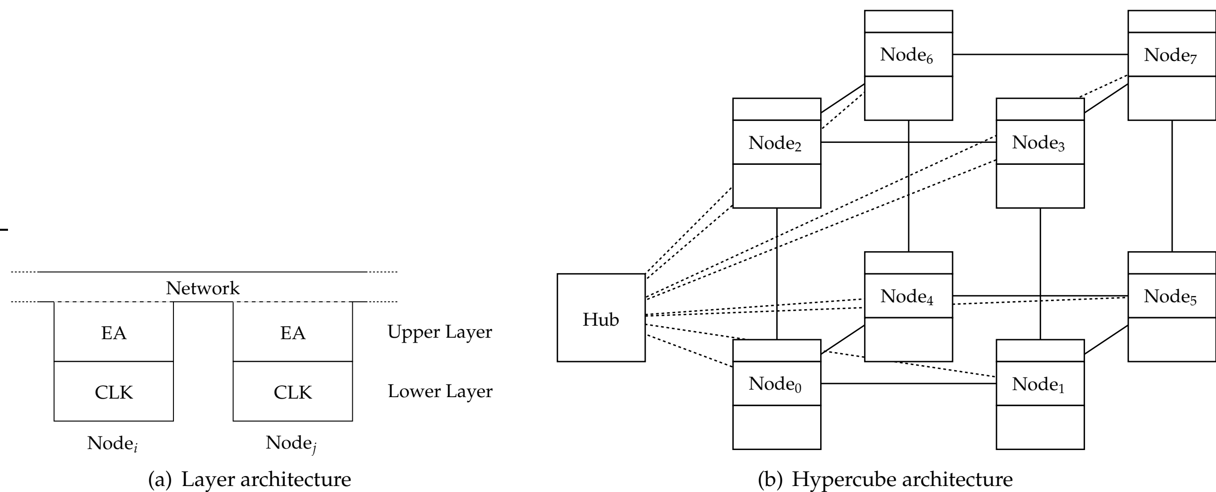
Random-walk Starting from the first relevant city v three independent random walks of a given length are performed on a neighborhood structure. The end points are the missing relevant points. The neighborhood structure for each city includes the three closest cities in each of the four geometric quadrants. Shorter random walks make the kick local, for large k the kick becomes similar to the Random kick.

2.2 System Architecture

The implementation of a node in this distributed system consists of two layers (figure 2(a)). The lower layer is a Chained LK algorithm, which provides a simple interface as abstraction of the underlying algorithms. The upper layer is an evolutionary algorithm (EA) that utilizes the CLK for local optimization. Additionally, it includes network functionality to communicate with other nodes.

2.2.1 Lower Layer

The Chained LK implementation (CLK) for the lower layer was taken from the Concorde package (`co031219`) provided by Applegate, Bixby, Chvátal and Cook [2, 8]. This implementation is well-known in the TSP community and has been used by other researchers (e.g. Walshaw in [41]).



(a) Layer architecture (b) Hypercube architecture

Figure 2: System Architecture.

The C source code was enhanced with a JNI[2] framework to provide the following functions to the upper layer:

- Construction of an initial tour using the Quick-Borůvka heuristics.
- Optimization of a given tour using the Chained Lin-Kernighan heuristics. Internally this function used the default values from the original implementation except for the kicking type, which could be set from outside.

2.2.2 Upper Layer

The upper layer is a Java program with network communication capabilities. One instance of this program was running on each cluster machine for the experiments here, but there is no restriction on the number of programs per cluster computer from the system itself. In the following text, "node" is used interchangeable as "program on a cluster computer" and "cluster computer".

The communication capability is used to send and receive tours from neighboring nodes. For the simulations presented here the nodes' neighborhood was determined by a hypercube topology (see figure 2(b)). To setup the topology each node contacted a dedicated hub to get of list of neighboring nodes. Unlike client-server systems this hub was only used during the setup phase and was not involved into any computation or message routing.

Sending and receiving tours is controlled by the EA of each node. To be able to receive incoming tours asynchronously while the CLK algorithm is running, each node starts a dedicated receiving thread. Sending or broadcasting tours, respectively, is performed by the main thread that calls the CLK algorithm, too.

2.3 Algorithm

A node's upper layer represents a stepping stone model [33] from a genetic algorithm's perspective. Each island maintains only a single individual (tour), which is, depending on the EA, reiteratively mutated, locally optimized and sent to neighboring islands or nodes, respectively.

Simplified, the algorithm is structured as follows. In each iteration the node's tour is perturbated by one or several random double-bridge moves (see section 2.3.1 for details). This perturbated tour is optimized by the Chained Lin-Kernighan algorithm. Thereafter, the new tour is compared with all tours received meanwhile from other nodes. The best tour is stored as the new node's tour. If this tour was the result of the local CLK function it is broadcasted to all neighboring nodes.

The pseudo code for this algorithm is shown in algorithm 1.

The termination criterion as represented by the function TERMINATIONDETECTED can be the occurrence of one of the following events:

[2]Java Native Interface. Allows to embed native source code such as C into Java.

Algorithm 1 Pseudo code for the upper layer.

```
function DistributedAlgorithm
  s := InitialTour;
  s_best := ChainedLinKernighan(s);
  while not TerminationDetected do
    s := Perturbate(s_best);
    s := ChainedLinKernighan(s);
    S_received := AllReceivedTours;
    s_best := SelectBestTour(S_received ∪ {s});
    if s_best = s then
      BroadcastToNeighbours(s_best);
    else
      iterationsWithoutImprovement + +;
    end if
  end while
```

- If applicable, the known optimum for the tour length has been found by the local CLK function.
- A notification message has been received that another cluster node has found an optimum solution.
- A predefined time limit has been reached.
- A predefined number of CLK calls has been reached.

Due to different runtimes on the nodes at the end of a simulation more and more nodes might become inactive. Thereby the network topology degenerates and the neighborhood of each nodes decreases. As there is no global control, the best result of this simulation has to be collected from the local output of each node independently from the simulation itself.

2.3.1 Variation and Restarting

Nodes perturbate their current best tour before optimizing it by using the CLK algorithm. This is done to leave local optima, where the CLK algorithm got stuck. But the strength of the perturbation has to be chosen carefully. A perturbation that is too weak might not help to leave the current local optimum, but a too strong perturbation might damage the tour too heavily causing a loss of quality. So, as a compromise the used strategy begins with a weak perturbation and increases its strength if no better tours are found. If subsequent strength increase does not help, the current tour is discarded and a new initial tour will be constructed.

The perturbation is performed by a variator module, that variates a given tour. Within this module a random double-bridge move is executed, where the four relevant cities are chosen randomly. In the used EA, one or several random double-bridge moves are performed depending on the history of the CLK algorithm. Whenever the CLK function does not find a better tour than the previous best tour, a dedicated counter is increased. This counter gets resetted when a better tour has been found or received from another node. The value of the counter determines the number of double-bridge moves applied to a tour as given in equation 2, controlled by a parameter c_v. It is intended that a greater number of perturbation moves will modify the tour, so that it will leave the current local optimum.

$$variatorStrength = \lfloor \frac{iterationsWithoutImprovement}{c_v} \rfloor + 1 \tag{2}$$

If multiple perturbation moves do not help to change the tour significantly, the current tour will be discarded and a new tour will be constructed. This event occurs if the number of iterations without improvements reached the value of a parameter c_r.

Both the strategy for the number of perturbation moves and the reset event can be parameterized. See algorithm 2 for details on the code.

Algorithm 2 Pseudo code for the perturbation step.

```
function PERTURBATE(s)
  if iterationsWithoutImprovement > c_r then
    RESETCOUNTERS;
    return INITIALTOUR;
  else
    variatorStrength := ⌊iterationsWithoutImprovement / c_v⌋ + 1;
    return VARIATOR.VARIATE(s, variatorStrenth);
  end if
```

Instance	Size	Held-Karp Bound	Optimal Tour Length
C1k.1	1000	11330836	11376735
E1k.1	1000	22839568	22985695
fl1577	1577	21886	22249
pr2392	2392	373490	378032
pcb3038	3038	136588	137694
fl3795	3795	28477	28772
fnl4461	4461	181569	182566
fi10639	10639	520527	*520383*⋆
usa13509	13509	19851464	19982859
sw24978	24978	855528	855597
pla33810	33810	66050535	*66005185*⋆
pla85900	85900	142383704	*142307500*⋆

Table 1: Testbed Instances. Instances marked with a star (*) are not solved to optimum yet, the values represent the length of the best known tour.

2.4 Testbed

For our analysis a set of instances from various sources has been selected. The instance sizes range from 1000 to 85900 cities and are the same as used in other research projects.

- From Reinelt's TSPLIB [40] the following instances were taken: `fl1577`, `fl3795` (both clustered instances), `pr2392`, `pcb3038` (both drilling problems), `fnl4461` (map of East Germany), `usa13509` (map of the United States), `pla33810` and `pla85900` (both programmed logic array). The number in the instance names denotes the number of cities in the instances.
- From the 8th DIMACS challenge [1] the random instances `C1k.1` and `E1k.1` were used. In both instances 1000 cities are arranged in a square using Euclidean distances. For instance `E1k.1`, the cities are randomly uniform distributed. For instance `C1k.1`, the cities are normally distributed around one of 10 cluster centers. For details on construction see [30].
- From the collection of national TSPs [5]: `fi10639` (map of Finnland) and `sw24978` (map of Sweden). The number in the instance names denotes the number of cities in the instances. No optimal solutions for the instances `fi10639` is currently known. An optimal solution for `sw24978` has been recently found (March 2003, approved in May 2004).

2.5 Runs & Parameters

Each simulation setup was performed 10 times. The number of runs was limited due to time constraints. For further analysis average values were calculated.

2.5.1 Chained Lin-Kernighan

The program `linkern` that is part of the `concorde` package has been used. For the first part of the analysis, no modifications were made on the source code. The resulting values were used for comparison with later results from the distributed algorithm.

The following parameters were set via the command line:

- The kicking type was set to one of the four valid types ($0-3$) using the switch `-K`.
- The number of kicks was set to a large integer (`-R` 50000000) to observe the long time behavior of the program.
- The time limit was set to 10^4 CPU seconds for instances with less than 10^4 cities and 10^5 CPU seconds for larger instances.
- For instances with known optimum, this optimum was set as termination criterion (switch `-h`).
- Each run was repeated 10 times (switch `-r`)
- Each program run was initialized by a pseudorandom integer (switch `-s`).

2.5.2 Distributed Chained Lin-Kernighan

The distributed algorithm was tested with different setup values. The parameters below were the same for all setups and defined in configuration files:

- The number of CLK calls has been limited to 262144, which was equally to ∞ as it was never reached.
- The time limit was set to 10^3 CPU seconds per node for instances with less than 10^4 cities and 10^4 CPU seconds per node for larger instances.
- For instances with known optimum, this optimum was set as termination criterion.
- The number of CLK calls before increasing the variator strength (c_v) was set to 64.
- The number of CLK calls before resetting the tour (c_r) was set to 256.
- Each node was initialized by a pseudorandom integer.

Some parameters have been changed in different simulations to observe effects of different values. Below a list of these parameters is shown with values that were used.

- 8 cluster nodes connected in a hypercube topology were used as default. Additionally, a single node repeated some simulations to check the influence of parallelization.
- The kicking type was set to one of the four valid types (Random, Geometric, Close and Random-walk). As Random-walk performed best during initial simulations, simulations for larger instances were primarily done with this kicking strategy.

As the software currently does not support to make independent runs subsequently, the 10 runs were started manually.

2.6 Hardware

The cluster used for this analysis consisted of eight computer nodes with one 3.0 GHz SMT processor (Pentium 4) and 512 MB RAM each running Linux 2.6. The nodes were connected in a switched Ethernet with 1 Gbps.

3 Experimental Results

3.1 Chained Lin-Kernighan

The Chained Lin-Kernighan program `linkern` from the Concorde package (ABCC-CLK, for short CLK) was used to solve all TSP instances from the testbed. On each instance all four kicking strategies were applied.

For instances with a size above 3000 cities, CLK could not find an optimum at all in any run. For smaller instances, the Random kicking strategy had the most successful runs (23/40). Table 3 shows in columns marked with "CLK" the number of successful runs for a given instance and kicking strategy.

Instance	Kicking strategy: Random 100 sec	Random 10^4 sec	Geometric 100 sec	Geometric 10^4 sec	Close 100 sec	Close 10^4 sec	Random-Walk 100 sec	Random-Walk 10^4 sec
C1k.1	0.013%	0.007%	0.013%	OPT	0.031%	0.020%	0.005%	0.002%
E1k.1	0.035%	0.020%	0.068%	0.043%	0.035%	0.028%	0.024%	0.016%
fl1577	0.569%	0.275%	1.206%	0.992%	1.102%	0.661%	0.670%	0.594%
pr2392	0.275%	0.050%	0.361%	0.283%	0.237%	0.105%	0.237%	0.093%
pcb3038	0.150%	0.070%	0.156%	0.081%	0.175%	0.077%	0.103%	0.060%
fl3795	0.567%	0.567%	0.801%	0.579%	0.884%	0.581%	0.643%	0.524%
fnl4461	0.121%	0.048%	0.089%	0.054%	0.093%	0.041%	0.098%	0.041%
fi10639	0.318%	0.160%	0.245%	0.138%	0.287%	0.144%	0.217%	0.106%
usa13509	0.268%	0.130%	0.234%	0.127%	0.229%	0.129%	0.204%	0.112%
sw24978	0.488%	0.153%	0.308%	0.140%	0.342%	0.136%	0.307%	0.122%
pla33810	0.508%	0.168%	0.563%	0.358%	0.592%	0.372%	0.519%	0.287%
pla85900	0.544%	0.209%	0.442%	0.232%	0.419%	0.231%	0.334%	0.160%

Table 2: Distance of the average tour length compared to known optimum (Held-Karp bound for instances fi10639, pla33810 and pla85900) for CLK-ABCC after 100 and 10^4 CPU seconds, respectively. Compare to table 4 on page 14.

For the smaller instances (C1k.1, E1k.1, fl1577 and pr2392), the Geometric kicking performs worst regarding both approximation performance and average tour quality after reaching the time limit (10^4 CPU seconds). The other three kicking strategies perform equally well on these instances whereas the Random and the Random-walk strategy show a small advantage. A similar behavior is shown for instance pcb3038, although the performances are not grouped that strict. For instance fl3795, the Close kicking strategy has slow approximation towards the optimum, whereas the Random and the Random-Walk strategy perform best. Larger instances (fnl4461, fi10639, usa13509, sw24978 and pla85900) again perform with the Random kicking strategy worst. Here, the other strategies perform equally well with an advantage for the Random-walk strategy. In contrast, for instance pla33810 the Random kicking strategy performs best, while the Random-Walk strategy is only the second best choice.

For a graphical overview see figures 3 and 4. Table 2 shows the proximity of the average tours to the optimum (or Held-Karp lower bound) for different instances and kicking strategies after a given periode of time.

A statistical analysis has been performed to compare the four kicking strategies with regard to the tour lengths for the five largest instances (fi10639, usa13509, sw24978, pla33810 and pla85900) after 100 and 10^4 CPU seconds. Here, the confidence intervals were set by the error bounds of $\overline{X} \pm \frac{s_X}{\sqrt{n}}$, where s_X is the bias-corrected sample variance and $\overline{X}$ the average over $n = 10$ runs. This corresponds to a significance level of 68.3%. No significant difference between the kicking strategies has been found, as each confidence interval for a given instance, time and kicking strategy overlaps with the corresponding confidence intervals of the other kicking strategies.

3.2 Distributed Chained Lin-Kernighan

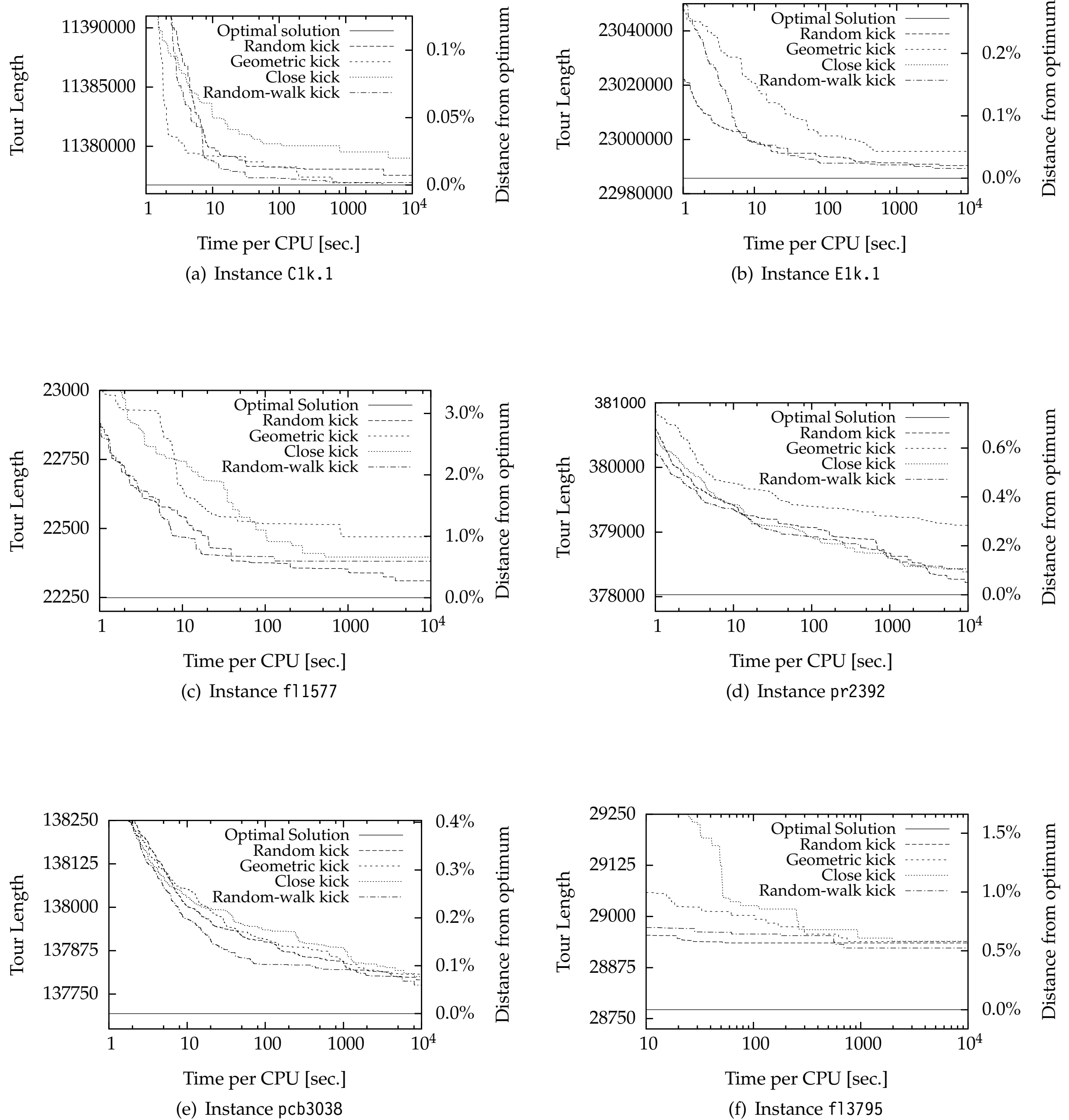
As the Random-walk kicking strategy performs good or even best for most instances and is the default kicking strategy in linkern, simulations with the distributed algorithms were performed primarily with this kicking strategy. For smaller instances (less than 10^4 cities) all four kicking strategies were applied.

The best results were achieved with a distributed algorithm variant running on 8 nodes and using double-bridge move variator (independent from the CLK algorithm), which is the default setup for the following discussion. For comparison, several instances were analyzed using a distributed algorithm that was restricted on 1 node, running without DBMs or running with both restrictions (see sections 3.2.2 and 3.2.3).

As shown in table 3 the distributed algorithm finds the optimal solution for most instances up to fnl4461 in at least one run, for many instances in all 10 runs. In cases were not all runs were successful within 1000 CPU seconds, the results were already close to the optimum. E.g. for the instance fl3795 with the Close kick strategy the 10th run found an optimal tour after 1044 CPU seconds.

Compared to the successfulness of CLK runs, the distributed algorithm has only in one case (fl1577 with Random kicking) fewer successful runs. The distributed algorithm can handle instances (e.g. fl3795) very well (39/40 runs successful over all kicking strategies) whereas the standard CLK fails every time within its time bound.

The approximation towards the optimum is faster with the distributed algorithm (DistCLK) compared

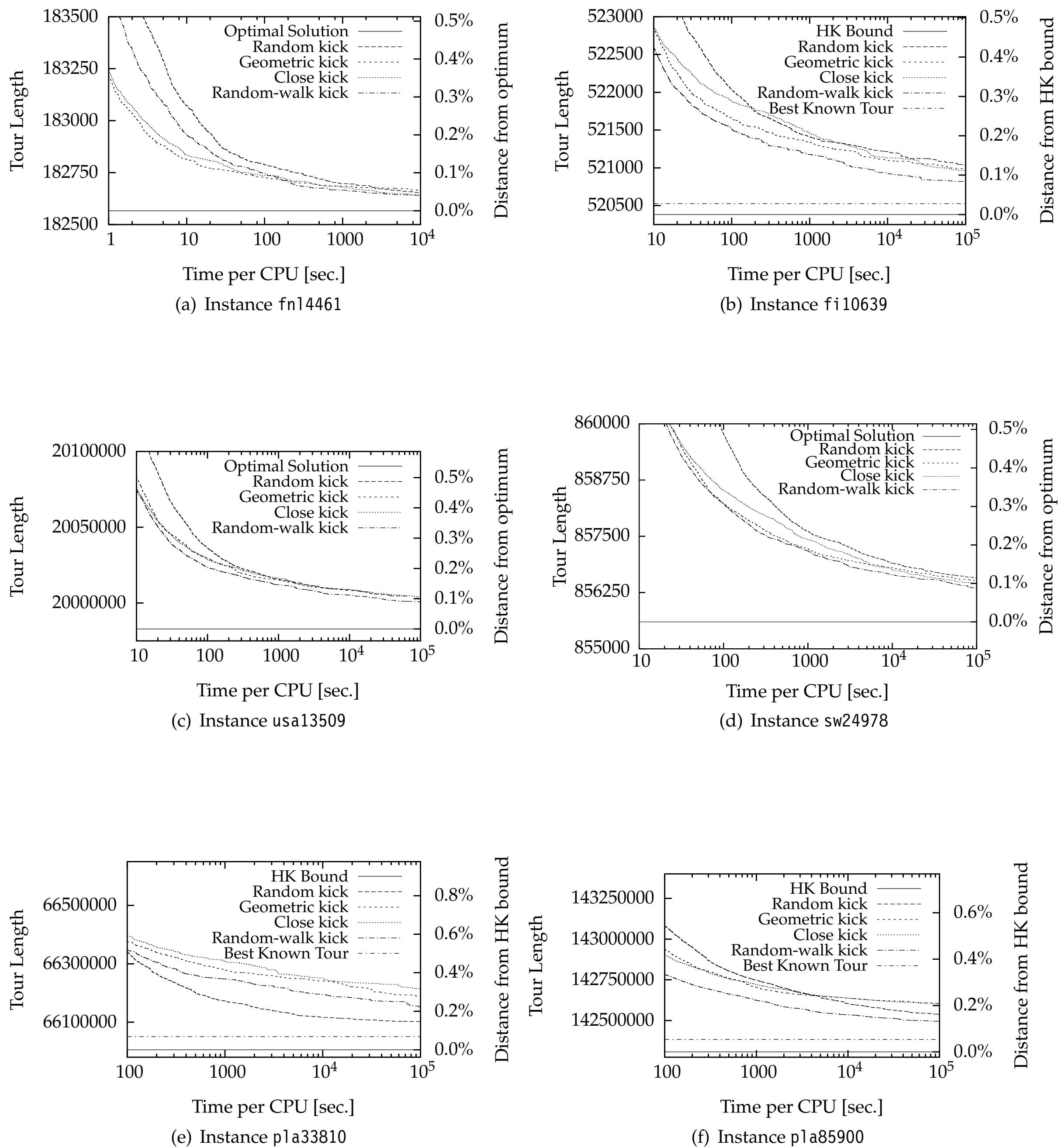


(a) Instance `C1k.1` (b) Instance `E1k.1`

(c) Instance `fl1577` (d) Instance `pr2392`

(e) Instance `pcb3038` (f) Instance `fl3795`

Figure 3: Relation between tour length and solution quality for the Chained Lin-Kernighan algorithm from Applegate *et al.* (Part 1).



(a) Instance `fnl4461`

(b) Instance `fi10639`

(c) Instance `usa13509`

(d) Instance `sw24978`

(e) Instance `pla33810`

(f) Instance `pla85900`

Figure 4: Relation between tour length and solution quality for the Chained Lin-Kernighan algorithm from Applegate *et al.* (Part 2).

Instance	Random CLK	Random DistCLK	Geometric CLK	Geometric DistCLK	Close CLK	Close DistCLK	Random-Walk CLK	Random-Walk DistCLK
	Kicking strategy							
`C1k.1`	6/10	**10/10**	10/10	10/10	4/10	**10/10**	9/10	**10/10**
`E1k.1`	3/10	**10/10**	0/10	**10/10**	0/10	**10/10**	3/10	**10/10**
`fl1577`	**5/10**	3/10	0/10	**9/10**	0/10	**8/10**	0/10	**8/10**
`pr2392`	9/10	**10/10**	0/10	**10/10**	2/10	**10/10**	4/10	**10/10**
`pcb3038`	0/10	**5/10**	0/10	**4/10**	0/10	**5/10**	0/10	**7/10**
`fl3795`	0/10	**10/10**	0/10	**9/10**	0/10	**9/10**	0/10	**10/10**
`fnl4461`	0/10	0/10	0/10	0/10	0/10	**1/10**	0/10	**1/10**

Table 3: Number of CLK runs that found the optimum within a given time bound. For CLK, the limit was set to 10^4 seconds, for DistCLK, the limit was set to 10^3 seconds as here 8 nodes were solving in parallel. Larger instances were omitted as both algorithms did not find optimal solutions for them.

with the original algorithm. As seen in figures 5 and 6 the distributed version is clearly better than CLK for most instances. For the instance `fl1577`, CLK gets stuck after about 150 seconds in local optima (9 runs in 22395, 1 run in 22256) that it cannot leave within the time bound. The distributed variant, however, finds the optimum in 9 out of 10 runs in less then 300 CPU seconds per node, only one run needed more than 2000 CPU seconds. Instance `pr2392` is quite easy to solve for DistCLK, as all 10 runs find an optimal tour after at most 260 CPU seconds per node. In contrast, CLK finds an optimal tour in only 4 out of 10 runs and those 4 runs require 3853 CPU seconds on the average. The average tour length over all 10 CLK runs after 10^4 CPU seconds for instance `pcb3038` was about 137776. DistCLK performed much better, reaching this tour quality already after 20 CPU seconds per node and the optimal tour quality after 3100 seconds (9 out of 10 runs found the optimum after at most 1723 seconds). On instance `fl3795` CLK got stuck in local optima (9 runs in 28935, 1 run in 28813) again, here after 111 seconds on average. The distributed version solves the problem to optimality after at most 569 CPU seconds per node. For instance `fnl4461`, CLK could have found better tours if given more time, as the 10 runs had found 9 different quality levels after reaching the time bound. The smallest instance in this testbed, for which no optimum is known, is instance `fi10639`. The best known tour from Helsgaun is about 0.028% above the Held-Karp bound. For the distributed algorithm, the average tour after 10^4 CPU seconds per node is 0.050% above this bound, the best run's tour is only 0.036% above. In contrast, the original CLK has an average tour quality of 0.084% above the lower bound after 10^5 CPU seconds. The best CLK tour, however, can keep up with the distributed variant as it is only 0.034% above the Held-Karp lower bound for instance `fi10639`. Although not reaching the optimal tour quality, but showing a good performance is the distributed algorithm at `usa13509`. After 10^4 CPU seconds the average tour quality is only 0.021% above the optimum compared to CLK having a final tour quality of 0.090%. Judging after the plot, it might be possible for the distributed algorithm to reach the optimum within 10^5 seconds. A similar behavior can be observed with instance `sw24978`. The distributed algorithm has an average tour quality of 0.050% over the optimum after reaching its time bound (best is 0.033%), whereas CLK has an average tour quality of 0.119% (best is 0.099%). So, on average the distributed version is twice as close to the optimum as CLK, the best tour is even three times closer. The final average tour length of the CLK algorithm (20002859.9) is already reached after 355.8 CPU seconds per node by the distributed algorithm. Considering that 8 nodes were cooperating on that problem, this gives an speed-up factor of 35. For instance `pla33810`, the average tour quality after the corresponding time limit of 10^4 seconds per CPU is 0.149% above the Held-Karp bound for the distributed algorithm. The CLK algorithm has a final average tour quality of 0.221% above the Held-Karp bound. This tour quality (length 66150904.5) is reached by the distributed algorithm after 670 CPU seconds per node on average. This corresponds to a speed-up factor of 14.9. As the Random kicking strategy performs best for the CLK algorithm for instance `pla33810` (see figure 4(e)), this kicking strategy was applied to the distributed algorithm, too. Here, both algorithms perform equally well on the long run regarding tour quality per total CPU time. The average tour quality for the distributed algorithm after 10^3 CPU seconds per node was 0.217% above the Held-Karp bound, after 10^4 CPU seconds 0.145%. In comparison, the CLK algorithm reached a tour quality of 0.168% above the Held-Karp bound after 10^4 CPU seconds and a tour quality of 0.147% after 10^5 CPU seconds. For instance `pla85900`, the tour qualities for CLK were 0.160% and 0.132% above the Held-Karp bound after 10^4 and 10^5 CPU seconds, respectively. For the distributed algorithm, the tour qualities were 0.181% and 0.128% above the Held-Karp bound after 10^3 and 10^4 CPU seconds per node, respectively. Both algorithms perform for this instance equally well, too. The final tour quality of the CLK algorithm (after 10^5 CPU seconds) was reached by the distributed algorithm after 7448.6 CPU

Instance	Kicking strategy: Random 10 sec	Random 10^3 sec	Geometric 10 sec	Geometric 10^3 sec	Close 10 sec	Close 10^3 sec	Random-Walk 10 sec	Random-Walk 10^3 sec
`C1k.1`	OPT	OPT					OPT	OPT
`E1k.1`	0.018%	OPT					◇	OPT
`fl1577`	◇	0.022%	1.244%	0.002%	0.771%	0.004%	◇	0.006%
`pr2392`	◇	OPT	0.306%	OPT	0.227%	OPT	0.152%	OPT
`pcb3038`	◇	0.007%	0.136%	0.010%	0.127%	0.005%	◇	0.004%
`fl3795`	◇	OPT	◇	0.014	◇	0.019%	◇	OPT
`fnl4461`	◇	0.025%	◇	0.015%	◇	0.008%	◇	0.013%
`fi10639`	◇	0.113%	◇	0.086%	◇	0.072%	◇	0.116%
`usa13509`							◇	0.062%
`sw24978`	◇	0.171%	◇	0.116%			◇	0.116%
`pla33810`	◇	0.217%					◇	0.126%
`pla85900`							◇	0.182%

Table 4: Distance of the average tour length compared to known optimum (Held-Karp bound for instances `fi10639`, `pla33810` and `pla85900`) for DistCLK after 10 and 10^3 CPU seconds per node, respectively. For cells marks with ◇, there is no data available as the algorithm did not return any tour at this point of time. For empty cells, no simulation was performed due to time constraints. Compare to table 2 on page 10.

seconds per node, which corresponds to a speed-up of factor 13.4.

For a graphical overview see figures 5 and 6. Table 4 shows the closeness of the average tours to the optimum (or Held-Karp lower bound) for different instances and kicking strategies after a given periode of time.

3.2.1 Variator Strength and Restarts

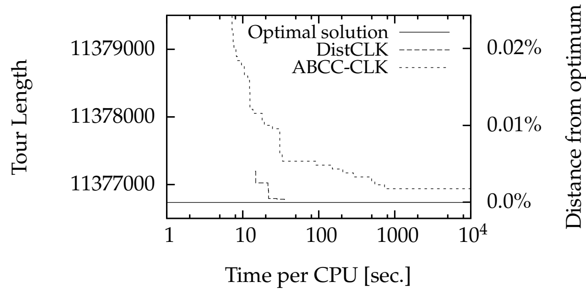
The variation and restarting strategy described in section 2.3.1 (page 7) might behave different depending on random processes for the same instance in different runs. The following three example runs are selected out of ten simulation runs with instance `fi10639` with 8 nodes and the Random-Walk kicking strategy.

For run A only a weak perturbation was enough to enable the CLK algorithm to find a better tour. During the first 4952 CPU seconds 51 improving tours were found by the nodes, thus not requiring any increase of the variation level or even a restart. As after about 6600 seconds no new improvements were made, within a small time frame all eight nodes were set to variation level 2. Before requiring any further increase, a better tour was found (7858 seconds) by a node. As this tour was broadcasted in the net and improving the local best tours, the local variation level counters got set back to the default value, too. After about 9500 seconds another variation level increase was required as no new tour was found meanwhile. Short after the increase a better tour was found, which was improved only once before the time bound was reached. The final tour's length was 520627 which is 0.047% above the Held-Karp bound.

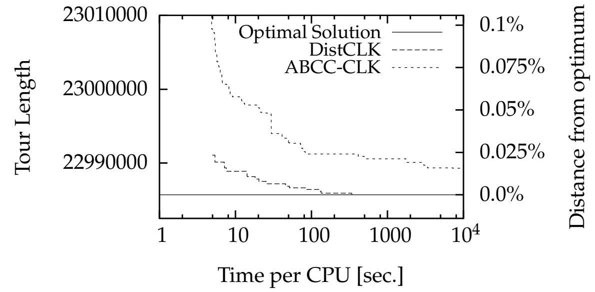
In contrast for run B a strong perturbation was applied during simulation, but the tour quality was not improved thereby. During the first 2503 CPU seconds 48 improving tours were found by the nodes. Here, the nodes exchanged their tours often enough not requiring to increase their variation level or even to restart. As no better tours were found the variation level (see equation 2) increased to 2 on all nodes after about 4140 seconds, to level 3 after 5830 seconds and to level 4 after about 7500 seconds. Variation level 5 was reached, too, but superseeded soon after (about 9400 seconds) by restarts with new tours on every node. Within the remaining time no better tour had been found and the tour found after a quarter of the given time was finally the best tour at all. This final tour's length was 520662 which is 0.054% above the Held-Karp bound.

Run C shows that strong perturbations might help. For the first 3396 CPU seconds 45 improving tours were found by the nodes. Again, during this phase no increase of the variation level or even a restart was necessary. Like in run B the variation levels were increased sequentially: After about 5020 seconds to level 2, after about 6700 seconds to level 3 and after 8370 seconds to level 4. In contrast to run B a better tour was found by a node after 9337 seconds preventing a further increase of the variation level. This tour was improved four more times resulting in a final tour of length 520584 (0.039% above Held-Karp bound).

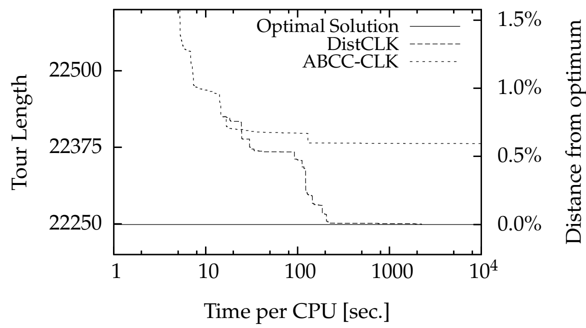
The tour qualities of the all runs with the same parameters were between 520563 (0.035%) and 521002



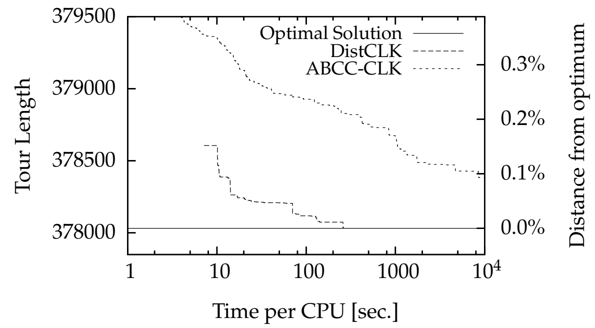
(a) Instance `C1k.1` with Random-walk kick



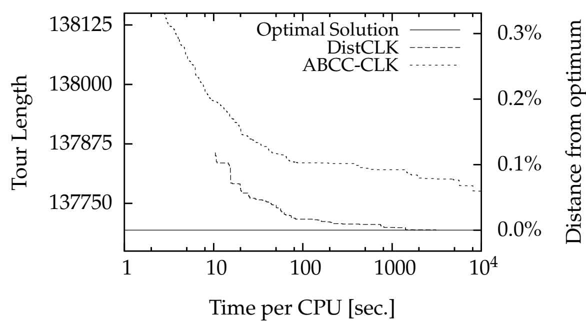
(b) Instance `E1k.1` with Random-walk kick



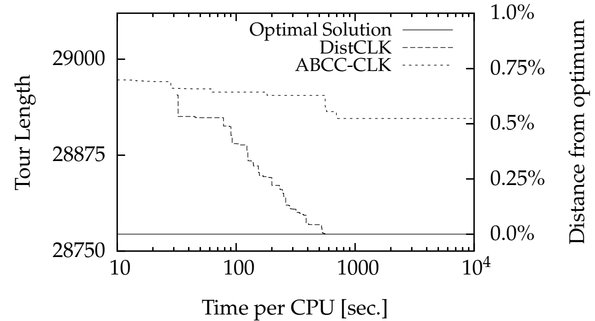
(c) Instance `fl1577` with Random-walk kick



(d) Instance `pr2392` with Random-walk kick

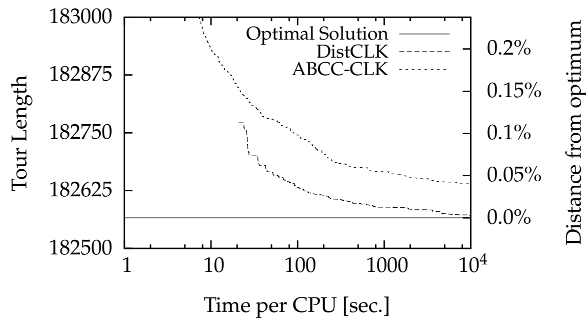


(e) Instance `pcb3038` with Random-walk kick

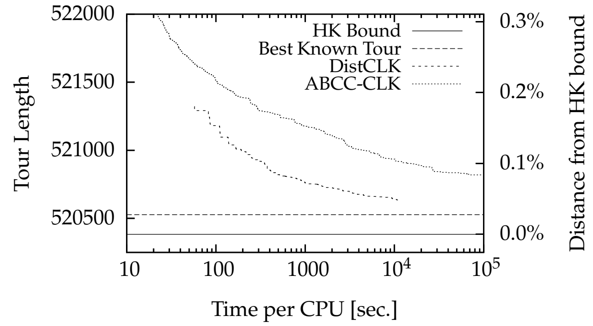


(f) Instance `fl3795` with Random-walk kick

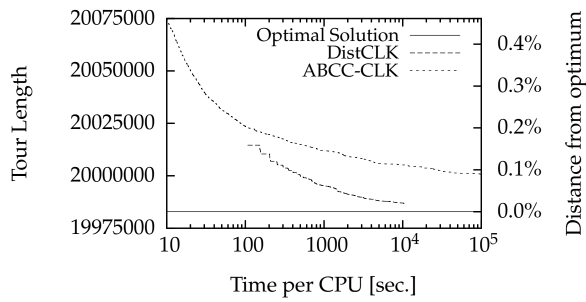
Figure 5: Relation between tour length and solution quality for the Distributed Chained Lin-Kernighan algorithm (DistCLK) compared with the results from the original CLK (ABCC-CLK) (Part 1).



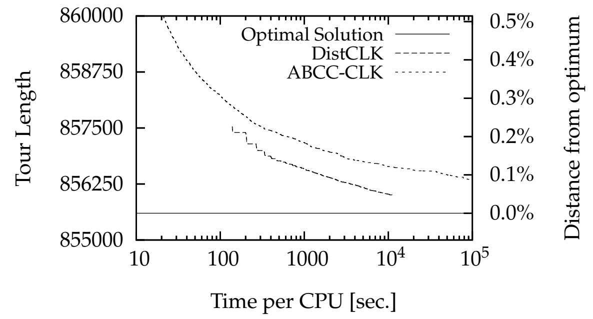
(a) Instance fnl4461 with Random-walk kick



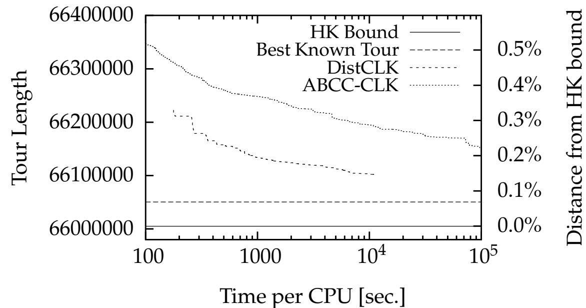
(b) Instance fi10639 with Random-walk kick



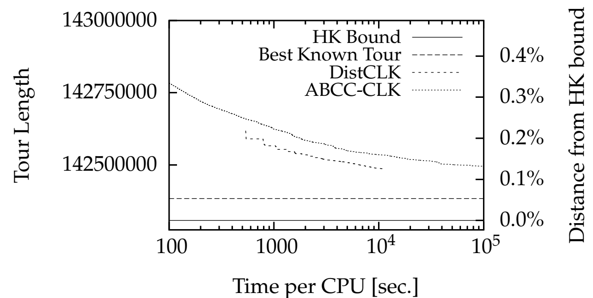
(c) Instance usa13509 with Random-walk kick



(d) Instance sw24978 with Random-walk kick



(e) Instance pla33810 with Random-walk kick



(f) Instance pla85900 with Random-walk kick

Figure 6: Relation between tour length and solution quality for the Distributed Chained Lin-Kernighan algorithm (DistCLK) compared with the results from the original CLK (ABCC-CLK) (Part 2).

Distance to Optimum	CPU time per node [sec] ABCC-CLK	1 node	8 nodes	Speed-up Factor
0.10%	8510.7	246.2	10.7	23.01
0.05%	–	421.1	24.2	17.40
0.00%	–	937.1	262.2	3.57

Table 5: Speedup with instance `pr2392`. Average over 10 runs each.

Distance to Optimum	CPU time per node [sec] ABCC-CLK	1 node	8 nodes	Speed-up Factor
0.50%	–	336.9	78.4	4.30
0.25%	–	1153.3	199.8	5.77
0.00%	–	4223.7	569.0	7.42

Table 6: Speedup with instance `fl3795`. Average over 10 runs each.

(0.119%). So, none of the examples above is an extreme. Due to differences in random processes a variable strategy for perturbation and restarting is required.

3.2.2 Effects of Parallelization

To compare the effects of parallelization the subset of the test instances were run in setups with both 1 and 8 nodes, while keeping other setup parameters constant (e.g. the Random-walk kicking strategy). Between two local CLK search steps the already described variable strength double-bridge move perturbation was performed (see section 2.3.1). In case of the 8 node variant the locally improved tours were exchanged between neighboring nodes. Simulation results show, that the distributed algorithm can scale well with the number of nodes.

In figure 7(a) a comparison between the original Chained LK algorithm and the distributed algorithm running on 1 or 8 nodes, respectively, for instance `pr2392` is shown. At the beginning, the variant with 8 nodes is more than twice as fast as expected from parallelization. It reaches a tour quality level of 0.1% above the optimum after 10.7 CPU seconds per node compared to 246.2 seconds for the single node variant (speed-up factor 23.01). The original CLK algorithm reaches this level after 8510.7 CPU seconds. For the quality level of 0.05% above the optimum the 8 node variant is still two times faster than the single node variant (speed-up factor 17.4), in respect to CPU seconds. Here, ABCC-CLK does not reach this level as well as the optimum within the given 10^5 second time limit. The parallel variant with 8 nodes requires about a quarter of the time of the single node variant (speed-up factor 3.57), which stands in strong contrast to the previous two quality levels (see table 5 for details). This behavior depends on three runs in the parallel variant, that need between 110 and 260 seconds, while the other seven runs require less than 43 seconds to find the optimum. Thereby the medians over the optimum finding times for both variants are 71.2 seconds versus 596.5 seconds (factor 8.38) which suits the expectations from parallelization.

Figure 7(b) shows the averaged CPU time plots for the original CLK algorithm and the distributed variant running on 1 and 8 nodes, respectively, solving instance `fl3795`. Here, the approximation to the optimum over time is smoother that instance `pr2392`'s approximation. For comparison of the distributed variants with 1 or 8 nodes again three different quality levels were selected. The required time to find a tour that is 0.5% above the optimum the single node variant requires 337 secondes, whereas the parallel variant requires 78 CPU seconds. Here, the speed-up factor is only about 4 for using 8 nodes. The speed-up factor gets better, the closer the tour qualities get to the optimum. For a quality level of 0.25% above the optimum, the required CPU seconds are 1153 versus 200 (factor 5.77). To reach the optimum solution, the single node variant requires 4224 CPU seconds on average. Having again a good speed-up factor of 7.42 the parallel variant requires 569 seconds per node.

As for instance `fi10639` (figure 8) no optimal solution is known, the Held-Karp bound was used to measure tour qualities. The first quality level of 0.12% above the Held-Karp bound for this instance was reached after 1183 CPU seconds in the one node variant, compared to the eight node variant requiring 189 seconds. This is a speed-up of 6.27, which is improved subsequently. The tour quality of 0.10% is reached in average after 2672 seconds versus 351 seconds (speed-up factor 7.62). Finally, the quality level of 0.08% required a computation time of 6961 seconds for the singe node variant an 723 seconds for the parallel variant resulting in a speed-up factor of 9.63.

Distance to Optimum	CPU time per node [sec] ABCC-CLK	1 node	8 nodes	Speed-up Factor
0.12%	3912.6	1183.4	188.8	6.27
0.10%	15183.3	2671.7	350.6	7.62
0.08%	–	6960.5	723.0	9.63

Table 7: Speedup with instance `fi10639`. Average over 10 runs each.

As shown above parallelization works for this distributed algorithm when comparing single versus multiple node variants. Especially for larger instances with long running times the speed-up factor may be optimal regarding used CPU time.

3.2.3 Effects of the Variator

The Variator in the distributed algorithm may be seen redundant, as the Chained Lin-Kernighan algorithm already provides a set of four different DBM moves to choose from. This assumption gets contradicted by analysises that have been performed to examine this claim.

Exemplary, instances `pr2392` (figure 7(a)) and `fi10639` (figure 8) were solved in simulations with and without Variator in the distributed algorithm with 1 and 8 nodes. These instances are large enough to make long-term observation possible. For further discussion instance `fi10639` has been selected as representative.

In a setup were the distributed algorithm runs on only one node and no variator is used to perturbate the intermediate tours, this algorithm resembles the same performance as the original CLK algorithm. This behavior is expectable, as in this case the distributed algorithm does the same as the original algorithm. In figure 8 the performance of both simulations is visualized by the two top lines labeled with "DistCLK (none, 1 node)" and "ABCC-CLK". For comparison the distributed algorithms were performed in a third setup on one node, but this time the double-bridge move variator was enabled (label "DistCLK (DBM, 1 node)" in the plot). Right from the start, this setup performs clearly better than both the original CLK and the distributed algorithm without random double-bridge move (DBM) variator. After 10^4 CPU seconds comparing the average over 10 runs the third setup is only 0.073% away from the Held-Karp lower bound. In contrast, the distributed algorithm without variator is 0.110% above this bound and the original CLK 0.106% above. The 8 node variant required about 1700 CPU seconds to reach the quality level the single CPU variants reached at the end of the time limit. These results are better for the setup with variator as the number suggest, as the actual optimal tour is longer as the lower bound.

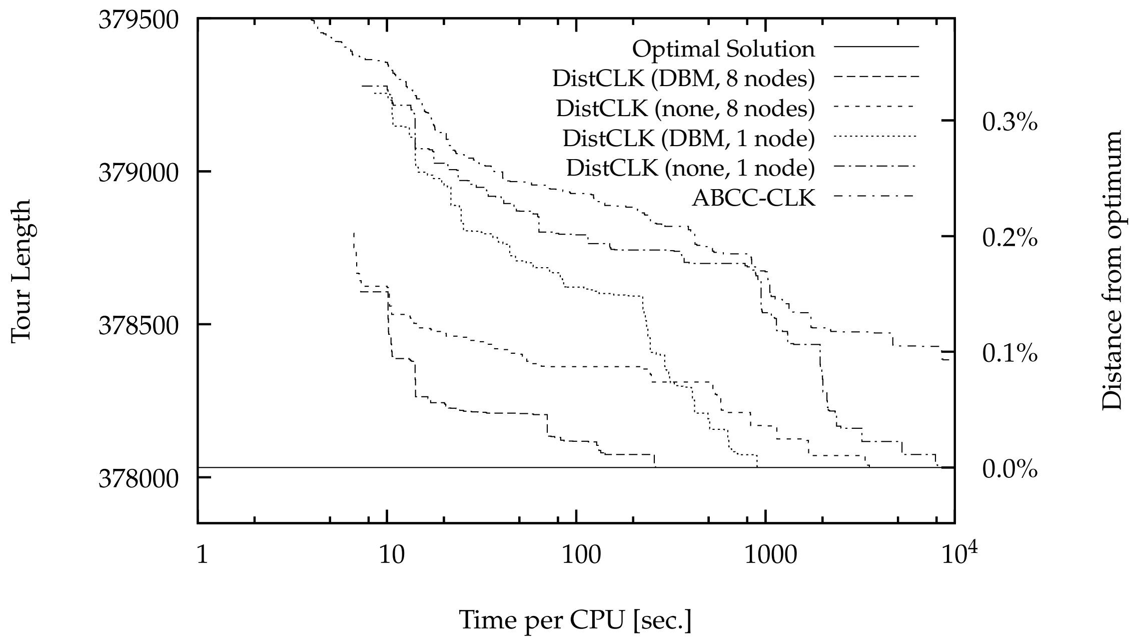
The perturbation provided by the variator module is effective in a multi node setup, too. Below the results of two simulations are compared: The distributed algorithm was run in the first case with its DBM variator (labeled "DistCLK (DBM, 8 nodes)") and in the second case without DBM variator (labeled "DistCLK (none, 8 nodes)"). During the first two CPU minutes both variants perform equally good. After that initial periode it becomes clear that the DBM variant performs better in closing towards the optimal solution. The final tour lengths are 0.080% for the variant without variator and 0.050% above the Held-Karp bound for the DBM variant. The latter variant required only 753.4 CPU seconds per node on average to reach the first variant's final tour quality after 10^4 CPU seconds.

The examples discussed above show that the extra perturbation performed within the distributed algorithm has indeed a positive effect on the simulation results. It could be integrated into the original CLK algorithm as it does improve the solution quality significantly even in single node setups.

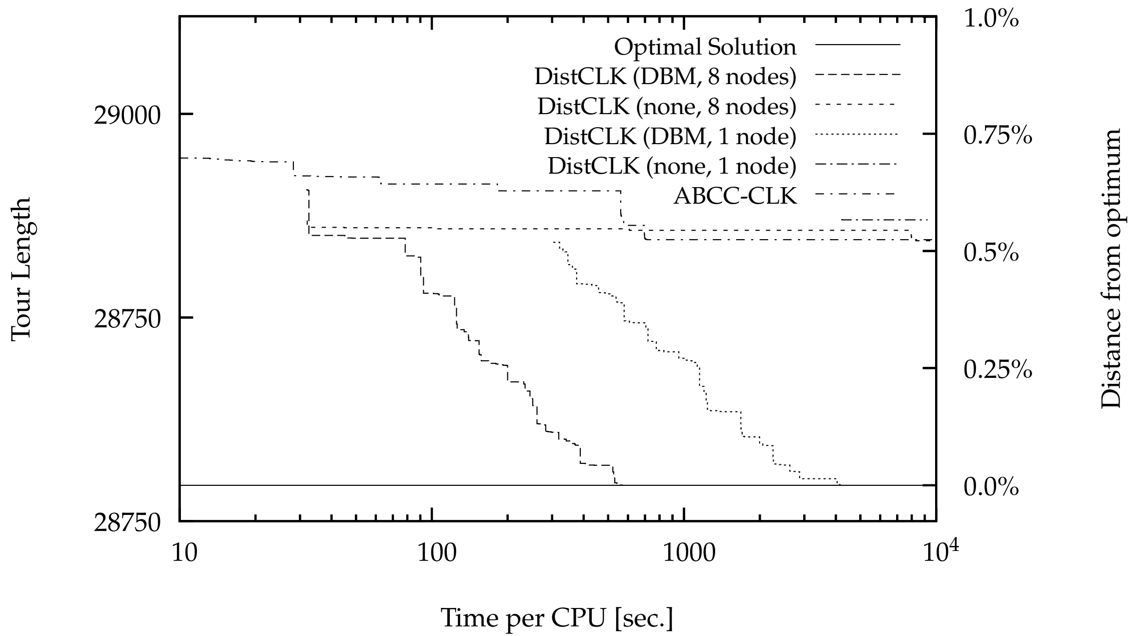
3.3 Comparison with Related Work

For comparison with other TSP solvers, the running times of selected instances have been normalized on a 500 MHz Alpha processor as standardized for the 8th DIMACS Implementation Challenge for the TSP [30, 1]. The computational data for the following presentation and comparison of other TSP solver below has been taken from the same source.

Helsgaun's LK LKH by Helsgaun [27] is a Lin-Kernighan algorithm that differs from the original LK algorithm in several aspects. It uses sequential 5-exchange step operating on neighborhood restricted on 5 members and based on a α-nearness. The α-values are calculated by using a one-trees on a modified weight matrix (π-values).



(a) Instance pr2392



(b) Instance fl3795

Figure 7: Effects of parallelization running the distributed algorithms on a different number of nodes and optional perturbation for instances pr2392 and fl3795. For instance pr2392, plots of the distributed algorithm ("DistCLK") marked with "DBM" used the variable strength double-bridge move variator as perturbation, plots marked with "none" were performed without perturbation.

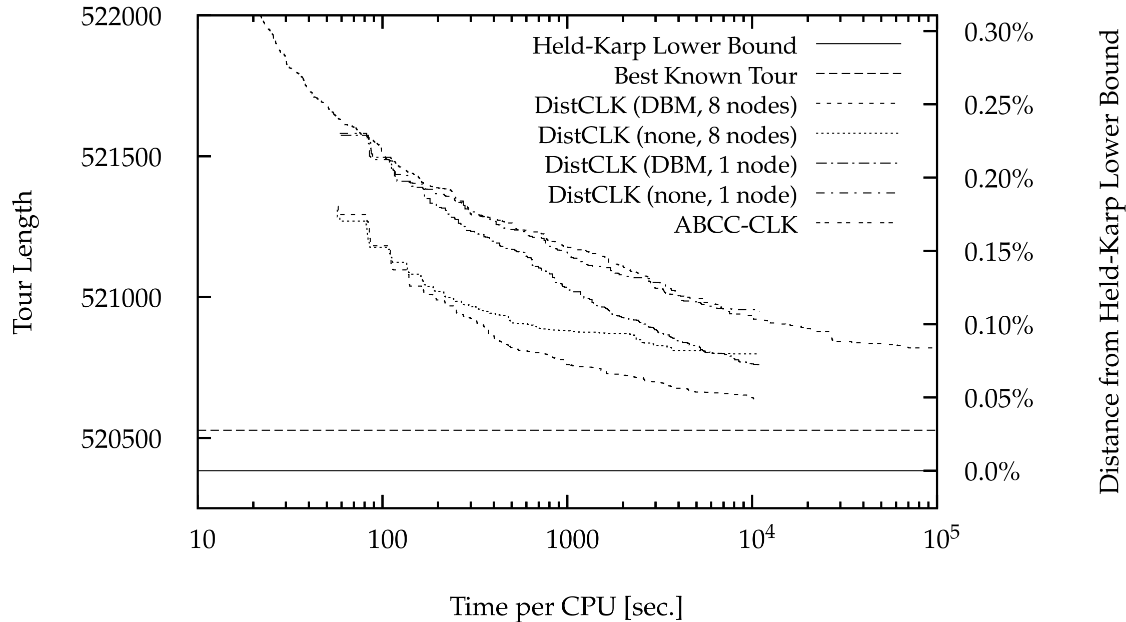


Figure 8: Effects of parallelization and variation running the distributed algorithms on a different number of nodes and optional perturbation for instance `fi10639`. Plots of the distributed algorithm ("DistCLK") marked with "DBM" used the variable strength double-bridge move variator as perturbation, plots marked with "none" were performed without perturbation.

Johnson and McGeoch compared [30] their own LK implementation (LK-JM) with Helgaun's LK. They report that LKH finds better tours than LK-JM for most instances in their testbed, but LKH requires significantly more time to reach these tour qualities.

Due to its long running times and good tour qualities LKH is an adequate choice to compare it with the distributed algorithm presented in this paper.

For the data used here, LKH's MAX_TRIALS parameter was set to 1.

Walshaw's Multi-Level LK Walshaw presented [41] a multi-level approach to solve TSP problems that embeds the Chained Lin-Kernighan algorithm, too (see section 1.3 on page 4), which is the reason for including it into the following comparison.

For the data used here, the number of iterations of the CLK algorithm was set to $10N$ (MLC^{N}LK), were N is the number of cities in the instance.

Tour Merging Cook and Seymour improve in their tour merging algorithm [18] results from independent runs of a underlying TSP solver (such as CLK or LKH) by merging the edges into a new graph and finding tours in this new graph. In an example using instance `rl5934` and LKH, 10 runs of LKH result in an average tour quality of 0.089% above the optimum and a best tour length of 0.006% above. The union of all 10 tours contains the optimal tour which is found by Cook and Seymour's algorithm within a very short period of time (compared to the time required for the LKH runs).

As this algorithm features both long running times (sum of several independent TSP solver) and good tour qualities, it has been chosen for comparison with the distributed algorithm.

For the data used here, the average over 5 runs of tour-merging using a branch decomposition with 10 CLK tours (12 quadrant neighbors, Don't-Look-Bits, N iterations and Geometric kicking strategy) was used.

Johnson & McGeoch's ILK In their comparison of ILK algorithms in [30] the authors use their own algorithm [29] as reference. Here, the data of a variant with $10N$ iterations, 20 quadrant neighbors, Don't-Look-Bits and maximum depth of 50 is compared to the results of the distributed algorithm.

This variant from the DIMACS challenge [1] is the one with the longest running time and the best tour qualities over all ILK variants by Johnson and McGeoch.

Exact Concorde To compare the results of the heuristics the results of the exact TSP solver from the concorde [2] package were included in table 8.

The computation time presented for the distributed algorithm (columns marked with "DistCLK" in table 8) are the average CPU times, multiplied by 8 (as results came from a system with 8 nodes) and scaled by a variable normalization factor to match computation times of a 500 MHz Alpha processor. This normalization is the same as used for the DIMACS challenge [1] and is generated by running a greedy algorithm on a testbed of random Euclidean instances. The normalization factor is calculated by comparing running times for the testbed instances to the known values for the Alpha machine. For instances that sizes are not covered by the testbed, the normalization factor is interpolated from surrounding instance sizes. The normalization factor for the instances used in table 8 ranges from 1.96 to 3.68.

For most of the compared instances, the distributed algorithm has on average better tours compared to the final tour quality levels of Helsgaun's LK (LKH) already after the first iteration, which is due to the underlying CLK algorithm. The current tour quality of a distributed algorithm is the best single node tour quality within the network. But for this initial tour quality, the distributed algorithm requires significantly more time than LKH to reach its final tour quality level for smaller instances (up to `usa13509`). For the two larger instances, however, less time is required. The ratio between the computation times for both algorithms shifts towards the distributed algorithm for increasing instances size: It grows from 0.13 for instance `fnl4461` and 0.50 for `usa13509` to 2.87 (instance `pla33810`) and 4.46 for instance `pla85900`.

Walshaw's Multi-level (MLC^{N}LK) approach's final tour qualities are worse compared to the tour qualities of the first iteration within the distributed algorithm, except for one case, but MLC^{N}LK requires significantly less time compared to the distributed algorithm for its first iteration. For one comparable case (instance `fl3795`), MLC^{N}LK requires only 26 normalized CPU seconds to find a tour that's quality is 0.54% above the optimum. The distributed variant, however, requires 938 seconds.

The tour merging (TM-CLK) by Cook and Seymour finds very good tour qualities for the instances that are used for comparison here. To gain this tour quality TM-CLK requires more time than the two heuristics above, but is still significantly faster than the distributed algorithm. E.g. for instance `pr2392`, TM-CLK requires only 93 seconds to find an optimal tour, contrary to the distributed algorithm that requires 7465. Currently, there is no data available for the other instances of this testbed. The distributed algorithm may performs better for larger instances when compared to the tour merging algorithm.

Compared to Johnson & McGeoch's Iterated LK the distributed algorithm performs better for most instances. Except for instances `pr2392` and `E1k.1` the distributed algorithm requires significantly less time, up to the factor of 4.5 for instance `pla33810`.

The exact tour lengths and the required computation times from concorde are available only for the four smallest instances of this testbed. For all these four instances concorde found the optimal solution in less time compared to the distributed algorithm. The advantage of concorde scales from the factor 1.8 (instance `C1k.1`) to 63.9 for instance `pr2392`. Again, as these instance sizes are quite small, better ratios can be expected for larger instances for the distributed algorithm.

Finally, the last block of table 8 contains the distributed algorithm's best results out of 10 runs and the normalized CPU time until the first occurrence of this result. For instances where the known optimal tour quality was not found the algorithm continued its search, so the CPU time until termination may be higher in some cases.

4 Conclusion

The proposed distributed algorithm improves the quality and performance of the original CLK algorithm in different ways. It allows to run the algorithm on multiple machines without loosing to much efficiency. Distributed computation is common for exact algorithms. Using the approach proposed here, heuristic algorithms can profit from large computer clusters, too. By exchanging tour between nodes, nodes with worse tours can leave their neighborhood to enter more promising areas of the search space. This strategy alone might degenerate as all nodes got stuck in a local optimum. To circumvent this flaw a perturbation move with variable strength was introduced. As simulations show this variation step improves the results compared to runs without variation.

This variator can even be applied to the single node variant. Applegate *et al.*'s original CLK algorithm gets stuck in local optima when solving instances of certain size, because its own perturbation moves are not strong enough. The distributed algorithm's variator can adapt its strength and help the CLK algorithm to leave its local optimum.

Instance	Helsgaun LK Dist	Helsgaun LK LKH	Helsgaun LK DistCLK	Walshaw Multi-Level CLK Dist	Walshaw Multi-Level CLK MLC^{N}LK	Walshaw Multi-Level CLK DistCLK	Cook&Seymour Tour Merging Dist	Cook&Seymour Tour Merging TM-CLK	Cook&Seymour Tour Merging DistCLK
`C1k.1`	0.12%	8.89	< 944.43	0.03%	11.96	< 944.43	0.00%	105.06	944.42
`E1k.1`	0.08%	9.78	< 9059.94	0.20%	4.35	< 9059.94	0.01%	31.02	< 9059.94
`pr2392`	0.24%	34.87	< 205.37	0.52%	8.29	< 205.37	0.00%	92.50	7465.24
`fl3795`	6.73%	74.06	< 914.73	0.54%	26.03	937.62	0.06%	509.69	16402.12
`fnl4461`	0.07%	129.23	978.12	0.20%	22.38	< 584.41			
`usa13509`	0.21%	1133.81	< 2272.18	0.19%	148.49	< 2272.18			
`pla33810`	0.96%	7982.09	< 2785.89	1.08%	294.81	< 2785.89			
`pla85900`	1.25%	48173.84	< 9350.55	0.75%	1092.51	< 9350.55			

Instance	Johnson&McGeoch ILK Dist	Johnson&McGeoch ILK ILK-JM-10N	Johnson&McGeoch ILK DistCLK	Exact Concode Dist	Exact Concode Opt-ABCC	Exact Concode DistCLK	Best of 10 runs Dist	Best of 10 runs DistCLK
`C1k.1`	0.00%	1292.40	944.43	0.00%	533.64	944.43	0.000%	198.29
`E1k.1`	0.05%	65.14	< 9059.94	0.00%	3854.50	9059.94	0.000%	65.07
`pr2392`	0.05%	220.54	681.95	0.00%	116.86	7465.24	0.000%	575.90
`fl3795`	0.00%	20597.78	16402.12	0.00%	6986.48	16402.12	0.000%	4283.36
`fnl4461`	0.11%	722.42	674.93				0.000%	14536.58
`usa13509`	0.11%	8640.36	5418.11				0.008%	179213.99
`pla33810`	0.68%	47599.30	10662.38				0.561%	171839.09
`pla85900`							0.468%	189023.53

Table 8: Normalized computation time compared with other algorithms. "Dist" is the distance to the optimum or Held-Karp Lower Bound (for instances `pla33810` and `pla85900`) as listed for the corresponding instance in the DIMACS challenge [1]. The two columns next to the distance are the CPU times for the two algorithms mention in the columns' header. For cells marked with "<", the distributed algorithm's intermediate results included only tours of better quality, so the value given is the point of time when an average value was available for the first time.

The comparison with other heuristic TSP solvers indicate that the distributed variant is only suitable for large instances. Due to the fact that 8 machines were running in parallel the *absolute* time to find a good solution makes the distributed algorithm competitive to existing heuristics for real-world applications.

There are different aspects of the distributed system that could be changed or improved in future versions.

- On the evolutionary side, sophisticated methods like conditionally tour acceptance upon receiving or a tour-merging algorithm known from evolutionary algorithms are enhancements to be considered for further studies. The high-quality results of Cook & Seymour suggest to incorporate their tour-merging algorithm into the distributed algorithm. This tour-merging would replace the current replacing of existing tours upon receiving better tours.
- Helsgaun's LK algorithm could be included as a node's local search algorithm to replace the CLK that was used for this system.
- Although the random double-bridge variator performs well, other perturbation steps (e.g. random swap) might be evaluated.
- Further work may include refactoring of the distributed algorithm's architecture resulting in a system that does not require a hub during its initial phase.

Acknowledgments

Thanks to David Applegate, Robert Bixby, Vašek Chvátal and William Cook for providing their Chained Lin-Kernighan algorithm to the TSP community.

References

[1] 8th DIMACS Implementation Challenge: The Traveling Salesman Problem. `http://www.research.att.com/~dsj/chtsp/`.

[2] Concorde TSP Solver. `http://www.tsp.gatech.edu/concorde.html`.

[3] distributed.net. http://www.distributed.net/.

[4] Gnutella. http://www.gnutella.com/.

[5] National Traveling Salesman Problems. http://www.tsp.gatech.edu/world/countries.html.

[6] SETI@home. http://setiathome.ssl.berkeley.edu/.

[7] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Finding Cuts in the TSP (a Preliminary Report). Technical report, Rutgers University, Piscataway NJ, 1995.

[8] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Finding tours in the TSP. Technical Report 99885, Forschungsinstitut für Diskrete Mathematik, Universität Bonn, 1999.

[9] David Applegate, William Cook, and André Rohe. Chained Lin-Kernighan for large traveling salesman problems. Technical Report 99887, Forschungsinstitut für Diskrete Mathematik, Universität Bonn, 1999.

[10] Maribel G. Arenas, Pierre Collet, Agoston E. Eiben, Márk Jelasity, Juan J. Merelo, Ben Paechter, Mike Preuß, and Marc Schoenauer. A framework for distributed evolutionary algorithms. In Juan Julián Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, José-Luis Fernández-Villacañas, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN VII, Proc. Seventh Int'l Conf., Granada*, volume 2439, pages 665–675, Berlin, 2002. Springer-Verlag.

[11] Sanjeev Arora. Polynomial Time Approximation Schemes for Euclidean TSP and other Geometric Problems. *Journal of the ACM*, 45(5):753–782, 1998.

[12] Achim Bachem and Michael Wottawa. Parallelisierung von Heuristiken für große Traveling-Salesman-Probleme. In Martin Baumann and Reinhard Grebe, editors, *Transputer-Anwender-Treffen*, Informatik Aktuell, pages 204–213. Springer, 1993.

[13] Ranieri Baraglia, José Ignacio Hidalgo, and Raffaele Perego. A Parallel Hybrid Heuristic for the TSP. In *Proceedings of EvoCOP2001, the First European Workshop on Evolutionary Computation in Combinatorial Optimization*, pages 193–202, 2001.

[14] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, 1975.

[15] Otakar Borůvka. On a certain minimal problem. *Praca Moravske Prirodovedecke Spolecnosti*, 3:37–58, 1926. in Czech.

[16] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. In J. F. Traub, editor, *Symposium on New Directions and Recent Results in Algorithms and Complexity*, page 441, Orlando, Florida, 1976. Academic Press.

[17] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science*, 2009:46, 2001.

[18] William Cook and Paul Seymour. Tour Merging via Branch-Decomposition. *INFORMS Journal on Computing*, 15(3):233–248, 2003.

[19] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[20] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.

[21] M. M. Flood. The Travelling Salesman Problem. *Operations Research*, 4:61–75, 1956.

[22] Martina Gorges-Schleuter. Asparagos96 and the Traveling Salesman Problem. In T. Bäck, Z. Michalewicz, and X. Yao, editors, *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation, Indianapolis, USA*, pages 171–174. IEEE Press, 1997.

[23] P. B. Grosso. *Computer Simulation of Genetic Adaptation: Parallel Subcomponent Interaction in a Multi-locus Model*. PhD thesis, University of Michigan, 1985.

[24] G. Harik, F. Lobo, and D. Goldberg. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3:287–297, 1999.

[25] M. Held and Richard M. Karp. The Travelling Salesman Problem and Minimum Spanning Trees. *Operations Research*, 18:1138–1162, 1970.

[26] M. Held and Richard M. Karp. The Travelling Salesman Problem and Minimum Spanning Trees: Part II. *Mathematical Programming*, 1(1):6–25, 1971.

[27] Keld Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[28] Nguyen Dinh Hung. *Hybrid Genetic Algorithms for Combinatorial Optimization Problems*. PhD thesis, Department of Information Systems Engineering, Graduate School of Engineering, University of Miyazaki, Miyazaki, Japan, March 2004.

[29] David S. Johnson and Lyle A. McGeoch. *Local Search in Combinatorial Optimization*, chapter The Traveling Salesman Problem: A Case Study in Local Optimization, pages 215–310. John Wiley and Sons, Ltd., 1997.

[30] David S. Johnson and Lyle A. McGeoch. Experimental Analysis of Heuristics for the STSP. In Gutin and Punnen, editors, *The Traveling Salesman Problem and its Variations*. Kluwer Academic Publishers, 2002.

[31] David S. Johnson, Lyle A. McGeoch, and E. E. Rothberg. Asymptotic Experimental Analysis for the Held-Karp Traveling Salesman Bound. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 341–350. Society for Industrial and Applied Mathematics, 1996.

[32] Richard M. Karp. Probabilistic Analysis of Partitioning Algorithms for the Traveling-Salesman Problem in the Plane. *Mathematics of Operations Research*, 2(3):209–244, 1977.

[33] Motoo Kimura. Stepping-stone model of population. *Annual Report of the National Institute of Genetics*, 3:62–63, 1953.

[34] E. L. Lawler and D. E. Wood. Branch-and-Bound Methods: A Survey. *Operations Research*, 14(4):699–719, 1966.

[35] S. Lin. Computer Solutions Of The Traveling Salesman Problem. *Bell System Technical Journal*, 44:2245–2269, 1965.

[36] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[37] Olivier Martin, Steve W. Otto, and Edward W. Felten. Large-Step Markov Chains for the Traveling Salesman Problem. *Complex Systems*, 5:299–326, 1991.

[38] Jaroslav Nesetril, Eva Milková, and Helena Nesetrilová. Otakar Borůvka on Minimum Spanning Tree Problem: Translation of Both the 1926 Papers, Comments, History. *DMATH: Discrete Mathematics*, 233, 2001.

[39] Ben Paechter, Thomas Bäck, Marc Schoenauer, Michael Sebag, A. E. Eiben, Juan J. Merelo, and Terry C. Fogarty. A distributed resource evolutionary algorithm machine (DREAM). In *Proceedings of the Congress on Evolutionary Computation 2000 (CEC 2000)*, pages 951–958. IEEE, IEEE Press, 2000.

[40] G. Reinelt. TSPLIB – A Traveling Salesman Problem Library. *ORSA Journal On Computing*, 3:376–384, 1991.

[41] Chris Walshaw. A Multilevel Approach to the Travelling Salesman Problem. Mathematics Research Report 00/IM/63, Computing and Mathematical Sciences, University of Greenwich, London SE10 9LS, UK, August 2000.

[42] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. SETI@home : Massively Distributed Computing for SETI. *IEEE Computing in Science and Engineering*, 3(1), 2001. http://www.computer.org/cise/articles/seti.htm.